Department of Physics and Astronomy

University of Heidelberg

Master thesis

in Physics

submitted by

Arno Friedrich

born in Dresden

May 2020

Systems Development towards a

Grazing Incidence Helium Atom Scattering Experiment

This Master thesis has been carried out by

Arno Friedrich

at the

Physikalisches Institut

Ruprecht-Karls-Universität Heidelberg

under the supervision of

Herrn Priv.-Doz. Maarten DeKieviet

Abstract

The thesis reports on the development of subsystems to facilitate Grazing Incidence Helium Atom Scattering (GI-HAS).
As a basis, an experimental set up from Bell Labs, USA had been transported to CAM, Germany and is in the process of reconstruction and components upgrade. To that end, a new power control and monitoring system has been designed and implemented. The system is capable of autonomously controlling the experiment based on sensory input about its state. A sensor simulation environment was designed to validate the control logic prior to deployment.
In addition, a method has been developed for defining a central beam axis, with respect to which the components of the time-of-flight detector chamber have been aligned. The periphery of the experiment, such as cooling, forevacuum distribution and state sensors, were implemented, making the entire detector chamber ready for operation. The first vacuum tests can be carried out immediately after the lock-down regulations are loosened.
The power management system has been tested for its reliability and functionality.

Zusammenfassung

Die vorliegende Arbeit berichtet über die Entwicklung von Subsystemen für die Heliumatomstreuung bei streifendem Einfall (GI-HAS).
Als Basis diehnt ein Versuchsaufbau der von Bell Labs, USA, zum CAM, Deutschland transportiert wurde und der im Prozess der Instandsetzung und Aktualisierung seine Komponenten ist. Zu diesem Zweck wurde ein neues Leistungssteuerungs- und Überwachungs-system entwickelt und implementiert. Das System ist in der Lage, basierend auf sensorischen Eingaben über dessen Zustand, das Experiment autonom zu steuern. Eine Sensorsimulationsumgebung wurde dazu entworfen mit deren Hilfe eine Validierung der Leistungskontrolllogik durchgeführt wurde.
Darüber hinaus wurde ein Verfahren zum Definieren einer zentralen Strahlachse entwickelt, bezüglich der die Komponenten der Flugzeitdetektorkammer ausgerichtet wurden. Die Peripherie des Experiments, wie Kühlung, Vorvakuumverteilung und Zustandssensorik wurden implementiert, und die gesamte Detektorkammer damit betribesbereit gemacht. Erste Vakuumtests können somit nach Lockerung des lock-downs sofort erfolgen.
Das Leistungsverwaltungssystem wurde auf seine Zuverlässigkeit und Funktionalität getestet.

Contents

1. Motivation

In 1969 Cabrera, Celli and Manson published a theoretical study which led to the suggestion to utilize helium for the detection of surface phonons (*Cabrera et al.*, 1969). Until, then the predominant technique for bulk phonon detection was neutron scattering. This method, however, is of limited use for surface phonons because of the small scattering cross section of neutrons. Shortly thereafter, Helium Atom Scattering (HAS) was carried out for the first time in *Fisher and Bledsoe* (1971). While they could not resolve the dispersion relation of single surface phonons because of an insufficient velocity resolution ($\frac{\Delta v}{v} \geq 5\%$), they did however show a clear interaction between helium atoms and surface phonons (*Benedek and Toennies*, 2018, p. 36). From then on, many surface phenomena were discovered and advancements in Helium Atom Scattering Techniques were made - both of which continue to this day.

Previous experiments revealed, that the atomic structure at a surface differs significantly from not only the bulk material but also the theoretical predictions for an ideal surface. For example, analysis of metal surfaces exposed an unexpected soft surface phonon branch, which lies below the longitudinal acoustic branch (*Benedek and Toennies*, 2018, p. viii). When comparing the surface plane of a metal with an equivalent plane inside the bulk material, one finds differences in the structure of both planes. The redistribution of charges leads to a change in either the periodicity of the surface (reconstruction) or the interlayer spacing near the surface (relaxation).

Insulators do not experience the same kind of changes since charge redistribution mostly does not apply here. However, other variations of the surface structure can still appear. One example is surface roughening, which can be seen as the 2-D equivalent to amorphous bulk material. A probing method that is strictly surface sensitive can also be employed for the study of adsorbates or thin films on top of a surface.

Helium Atom Scattering was found to be well equipped for the analysis of exactly these surface phenomena and dynamics. The downside, however, are the significant technical challenges associated with creating a helium beam and detecting the scattered atoms.

The longstanding goal of measuring a complete single-phonon inelastic scattering dispersion curve was first achieved in *Brusdeylins et al.* (1980). They used a time-of-flight measurement technique to calculate the energy exchange between helium beam and sample surface. The essence of this technique is to evaluate the velocity distribution of the helium beam before and after scattering by measuring the flight-time of its atoms.

In *DeKieviet et al.* (1995) that velocity distribution was measured with a different technique, called ^{3}He Atomic Beam Spin Echo (ABSE). The fermionic Helium-3 isotope is used as a probe instead of Helium-4. This makes it possible to use the Lamor pression of the nuclear spin as an indicator for the time-of-flight of the beam. The method functions by spin polarizing the atoms and then manipulating their spin with a specific arrangement of magnetic fields along the beam line. This leads to an excellent energy resolution of $< 1\,\mathrm{neV}$ (*DeKieviet et al.*, 1995).

The goal of our research group is to develop an enhanced (transversal) version of the ABSE experiment. In addition, we want to combine it with a ^{4}He time-of-flight scattering setup. The combination of both techniques will hopefully open the door to new opportunities for analyzing structure dynamics of disordered surfaces on the smallest scales.

2. Context of this Work

The HAS experiment, originally developed and operated at AT&T Bell Labs in Murray Hill, USA, under the supervision of Prof. Dr. Bruce Doak, acts as the starting point for the combination of GI-HAS and transverse ABSE spectroscopy. After its discontinuation in 2010, the setup was transported to the Center for Advanced Materials in Heidelberg, Germany. Both the incompatible power requirements and different measuring conventions between the two countries pose some great technical challenges.
The overall goal of a combined GI-HAS and transversal ABSE experiment can be broken down into separate objectives. The first objective is to recreate and adapt the HAS machine as it was employed in the USA at AT&T labs and later on at Arizona State University Tempe. Subsequently, it is planned to upgrade parts of the experiment, e.g. crystal manipulator and load-lock system, to optimize resolution and efficiency of the setup. Finally, the necessary components for a transverse atom spin echo measurement will be implemented and tested. The start of the reconstruction process of the HAS experiment was reported in *Turczyk* (2018).
In this thesis, that endeavor is continued in two areas: The first part will cover the development and testing of the new power distribution and monitoring system. The second part describes the alignment of the detector vacuum chambers and the installation of its peripheral systems.

3. Theory

Helium has a number of properties which make it uniquely suited for the detection of long wavelength surface phonons. In the following, I will give a short overview of the theory behind helium atom scattering and the time-of-flight measurement technique.

3.1. Helium Atom Scattering

3.1.1. Single-phonon Inelastic Scattering

Single-phonon scattering is the method of choice for mapping the dispersion relation $\omega(\overline{Q})$ over the entire first Brillouin zone of a surface. Care must be taken to set up the experimental parameters, such as incident beam energy and surface temperature, in a way that no multi-phonon interactions appear.

An inelastic scattering process can be described by the conservation of both the total energy and the momentum parallel to the surface (*Benedek and Toennies*, 2018, p. 22):

$$\Delta E + \hbar\omega(\overline{Q}) = 0 \quad \text{with} \quad \Delta E = E_f - E_i = \frac{\hbar^2}{2m}\left(k_f^2 - k_i^2\right), \tag{1}$$

$$\Delta\overline{K} + \overline{Q} + \overline{G} = 0 \quad \text{with} \quad \Delta\overline{K} = \overline{K}_f - \overline{K}_i, \tag{2}$$

where $\omega(\overline{Q})$ describes the frequency of the scattered phonon in relation to its wave vector $\overline{Q}$ parallel to the surface. ΔE denotes the energy that is transferred between helium atom and phonon. $\overline{K}$ is the parallel wave vector of the atom, denoted with i or f for either "initial" (also "incident") and "final", respectively. $\overline{G}$ is the reciprocal lattice vector. In general, all capital letters describe vectors parallel to the surface plane.
Equations 1 and 2 can be satisfied by different combinations of wave vectors and ΔE:

- $k_f > k_i \iff \Delta E > 0$: Annihilation of phonon
- $k_f < k_i \iff \Delta E < 0$: Creation of phonon
- $\overline{K}_f > \overline{K}_i \iff G = 0$: Creation $\rightarrow$ backward travel, Annihilation $\rightarrow$ forward
- $\overline{K}_f < \overline{K}_i \iff G \neq 0$: Creation $\rightarrow$ forward travel, Annihilation $\rightarrow$ backward

This means, we effectively have two distinctions for the type of inelastic scattering. The first is the energy transfer. If ΔE is larger then zero, the helium atom gained energy from the interaction and destroyed a phonon in the process. Whereas if it is smaller than zero, the atom lost energy and created a new phonon.
The next distinction is the direction in which the phonon travels in relation to $\overline{K}_f$. If the scattering process results in a $\overline{K}_f$ which falls outside of the first Brillouin zone, the reciprocal lattice vector $\overline{G}$ is applied. This back scattering or "Umklapp" process results in a phonon wave vector $\overline{Q}$ which falls back into the first Brillouin zone. That is because the (reciprocal) lattice is by definition periodic. Therefore, each point on the K-space can be mathematically represented as a point in the first Brillouin zone by adding or subtracting $\overline{G}$. Physically, $\hbar\overline{G}$ is the momentum that the helium atom transfers onto the center of mass of the target. Depending on the direction of $\overline{Q}$, this leads to a forward or backward motion (with respect to $\overline{K}_f$) of the phonon.

Lastly, it is also possible that no energy transfer takes place at all during the atom-surface interaction ($\Delta E = 0$). This is called diffraction and is an elastic scattering process.
Both the inelastic and elastic scattering processes are used to determine different properties of a surface. All configurations of inelastic scattering can provide information about the dispersion relation $\omega(\overline{Q})$. Whereas, diffraction is employed to map out the unit cell structure of a surface.

3.1.2. Advantages for Surface Phonon Detection

Helium atoms offer significant benefits for probing surface phonons in comparison to other scattering beam particles like neutrons, electrons or photons.

For a successful detection of phonons, the interacting particles should have an similar momentum and kinetic energy. For example, electrons with $E_{kin} \approx 200\,\mathrm{meV}$ have enough momentum but the phonon energies, which have to be detected, are in the order of magnitudes lower with $\Delta E \approx 2 - 40\mathrm{meV}$. Thus, it requires an extremely high energy resolution. In addition, generating electron beams with such a low energy is very difficult due to Coulomb repulsion. In other words, at very low kinetic energies it becomes exceedingly hard to generate a focused particle beam because of the repulsive forces between electrons in the beam.
In contrast, it is possible to generate helium atom beams with a kinetic energy more in range with that of surface phonons. Most importantly however, the wave vector (which is proportional to the momentum) of helium $k \approx 2\pi$ Å^{-1}, when its kinetic energy is tuned to approximately $20\,\mathrm{meV}$. This is similar to the inverse of a typical crystal lattice spacing. Consequently, it is in the same range as the wave vector (and therefore momentum) of a surface phonon. The similar momentum and energy are the main reasons for the usage of helium as a surface probe.
It is possible to generate helium beams with a low kinetic energy ($E < 70\,\mathrm{meV}$), which makes its surface interactions entirely non-destructive (*Hulpke*, 1992, p. 265). Another key point is its large scattering cross section. Because of this, the classical turning point of helium is approximately 3 Å above any surface. And finally, HAS can be used to investigate metals, insulators and semiconductors alike because, it is neutral and inert and does not cause nor suffer from surface charging (*Benedek and Toennies*, 2018, p. 17).
Taken together, its momentum, beam energy, scattering cross section and inert nature makes helium at thermal energies, a clear favorite for the investigation of surface dynamics.

3.2. Time-of-Flight Spectroscopy

3.2.1. From ToF Spectrum to Dispersion Relation

The parameters from equation 1 and 2 that are measurable in the experiment are: ΔE and $\Delta\overline{K}$. For the former, the time-of-flight spectroscopy is used and the latter is calculated using ΔE and the scattering angles ϑ_i and ϑ_f.
We limit ourselves to planar scattering. In this case, $\overline{K}_i$, $\overline{K}_f$ and the surface normal $\overline{N}$ lie in the same plane, which has to be brought to align with a high symmetry direction of the crystal lattice.
Figure 1 depicts the general experiential set up. The helium source generates an atom beam with a low velocity spread ($\frac{\Delta v}{v} \approx 1\%$). The chopper is a high-frequency rotating

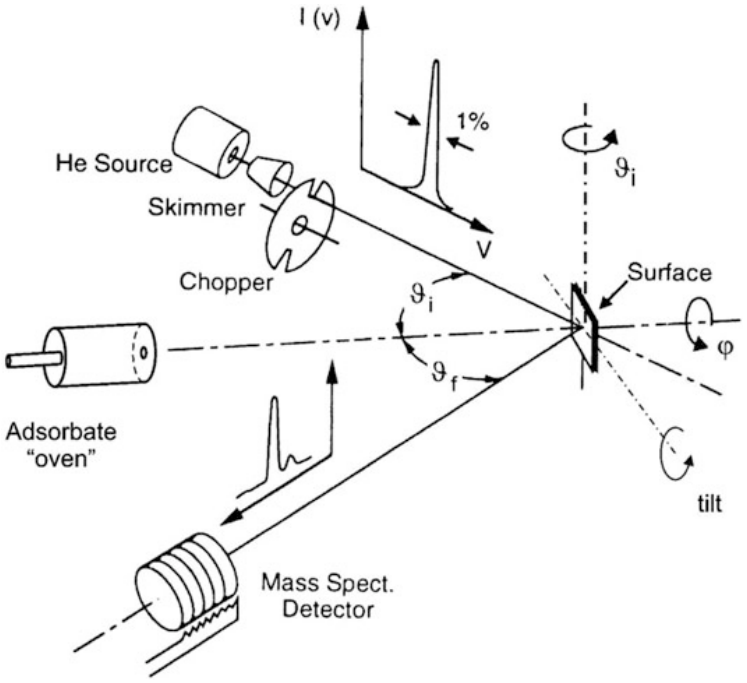


Figure 1: (*Benedek and Toennies*, 2018, pr. 23) Visualization of the time-of-flight spectroscopy with helium atoms. The helium source generates an atom beam with a low velocity spread ($\frac{\Delta v}{v} \approx 1\%$). The beam is chopped into discrete packages and scattered on the target. A mass spectrometer then generates time-of-flight spectra by counting the incoming atoms per unit time since they left the chopper. Intensity peaks from elastic and inelastic scattering are visibly in these spectra.

plate with slits to allow for chopping the beam into discrete packages. The beam then hits the target at an incident angle ϑ_i and interacts with the surface. The total scattering angle Θ_T is usually fixed, meaning $\Theta_T = \vartheta_i + \vartheta_f$, because of the bulky source and detector chambers. In this arrangement, only one of the scattering angles can be freely chosen by rotating the target surface. This is the case for our experiment as well, with the caveat that we plan to incorporate two beam sources.
The scattered helium atoms are then detected by a mass spectrometer. A timer is started whenever a beam package leaves the chopper (t_0). The detector then counts the number of received atoms per unit time since t_0. This way, a time-of-flight distribution (or histogram) is created in which the different scattering processes can be visible. Besides an intensity peak for elastic scattering, annihilation and creation of phonons will show pronounced peaks before and after the elastic scattering peak, respectively. This has to be combined with measurements of the Time-of-Flight (ToF) distribution of the incident beam to resolve the transferred energy ΔE. The transferred momentum $\Delta\overline{K}$ lies on a so called "scan curve". This describes all values of $\Delta\overline{K}$ and ΔE, which satisfy the conditions from equation 1 and 2. For a given incident beam energy E_i and scattering angles this relation is given by (*Benedek and Toennies*, 2018, p. 256):

$$\frac{\Delta E}{E_i} + 1 = \frac{\sin^2 \vartheta_i}{\sin^2 \vartheta_f} \left(1 + \frac{\Delta\overline{K}}{\overline{K}_i}\right)^2 . \tag{3}$$

In the case of planar scattering, we can simplify the wave vectors and write $\Delta\overline{K} = \Delta K$ and $\overline{K}_i = K_i$. K_i is also the parallel component of k_i in relation to the surface plane:

$$K_i = \sin(\vartheta_i) \cdot k_i \tag{4}$$

$$\implies \frac{\Delta E}{E_i} + 1 = \frac{1}{\sin^2(\vartheta_f)} \left(\sin(\vartheta_i) + \frac{\Delta K}{k_i} \right)^2 . \tag{5}$$

The relation between energy E_i and wave vector k_i follows from:

$$E_i = \frac{p^2}{2m} = \frac{\hbar^2}{2m} {k_i}^2 \quad \text{with} \quad p = \hbar k. \tag{6}$$

We can therefore calculate the momentum transfer ΔK from the energy transfer ΔE using only the incident energy E_i and the scattering angles.

The question remains how to determine ΔE and E_i from the time-of-flight spectrum. This can be done with the classic kinetic energy function:

$$E = \frac{m}{2} v^2. \tag{7}$$

The method, with which the speed of incident and final beam is derived, differs between HAS experiments. In essence, one has to make a precise distance measurement along the beam line and determine the time it takes for the particles to travel that distance. However, the exact point within the mass spectrometer at which the helium atoms are detected is hard to pin down. A precise measurement from that point to the scattering surface is an experimental and technical challenge
Dr. Bruce Doak's solution to this is a movable detector, mounted on linear rails in parallel to the beam axis. Two ToF spectra from the same surface are taken, while the detector is positioned at different distances from the target. The distance between these positions is easily and precisely measured using a standard laser interferometer. This way, a distinct speed value can be assigned to each peak on the ToF spectrum. Using equation 7, the energy E_f can be uniquely determined for every peak. Since $\Delta E = E_f - E_i$, the incident beam energy is the last missing parameter.
Specular reflection as an elastic scattering process where $\Delta E = 0$. This means, we can expect one intensity peak on the ToF spectrum for which $E_i = E_f$ must hold. Specular reflection becomes the most probable scattering process with the highest intensity peak when: $\vartheta_i = \vartheta_f$. Consequently, we can adjust the scattering angle by rotating the target until this condition is fulfilled and obtain the last required parameter E_i.
After this calibration procedure, it is possible to determine the dispersion relation $\omega(\overline{Q})$, which is proportional to $\Delta E(\Delta K)$, by measuring the time-of-flight spectrum of the scattered helium atom beam.

It is important to note that the "scan curves" mentioned above are in reality not precise lines but rather an envelope of possible values due to kinematic smearing. The atoms in the helium beam have both a velocity spread and a angular spread. Since velocity and angle are both used to calculate k_i , their uncertainties translate to the values of the scan curve as well. As it it turns out, these are almost never the limiting factor for the energy resolution (*Hulpke*, 1992, p. 12). It is the time spread, described in section 3.2.2, which usually limits the accuracy of the dispersion curve measurement.

3.2.2. Energy Resolution

The ability to resolve two distinct peaks in the ToF spectrum depends on the width of those peaks. The total time spread Δ_T of a ToF peak is influenced by 4 factors:

- Chopper pulse length
- Dispersion between chopper and target
- Dispersion between target and detector
- Finite length of ionizer within detector

The following calculations for the resulting effective energy spread are taken from *Hulpke* (1992, pp. 13).
The time spread for the chopper pulse length Δ_C describes the time in which an atom can pass the chopper opening:

$$\Delta_C = \frac{w_s N_s f_s}{2\pi r}, \tag{8}$$

where w_s is the larger one of either beam diameter or slit width, N_s describes the number of slits on radius r, and f_s is the chopper rotational frequency.
The dispersion of the atom over the distance between chopper and target is explained by:

$$\Delta_{ct} = \frac{-x_{ct}\Delta v_i}{{v_i}^2}, \tag{9}$$

with x_{ct} being the distance between chopper and target.
The time dispersion Δ_{td} after scattering and before detection of the atoms is due to the spread in velocities v_f of the atoms:

$$\Delta_{td} = -x_{td}\left(1 + \frac{\Delta E}{E_i}\right)^{-\frac{3}{2}} \Delta v_i, \tag{10}$$

while $x_t d$ is the distance between target and detector. And finally, the time spread Δ_D caused by the finite length of the ionizer L_D is simply the time it takes an atom to travel that length with a speed v_f. Of course, that velocity again has a certain spread:

$$\Delta_D = \frac{L_D}{v_f} \quad \text{with} \quad v_f = \left(1 + \frac{\Delta E}{E_i}\right)^{\frac{1}{2}} \tag{11}$$

All these contributions can be combined to the total time spread:

$$\Delta_T = \left({\Delta_C}^2 + {\Delta_{ct}}^2 + {\Delta_{td}}^2 + {\Delta_D}^2\right)^{1/2}. \tag{12}$$

This can be put in relation to the effective energy width $\delta(\Delta E)$ with:

$$\frac{\delta(\Delta E)}{E_i} = -2\left(1 + \frac{\Delta E}{E_i}\right)^{3/2} \frac{v_i}{x_{td}} \Delta_T. \tag{13}$$

This expression shows that phonon creation events, where $\Delta E < 0$, have a much better resolution compared to annihilation events with $\Delta E > 0$. This is especially true for high energy events, where $\Delta E \to \pm E_i$.

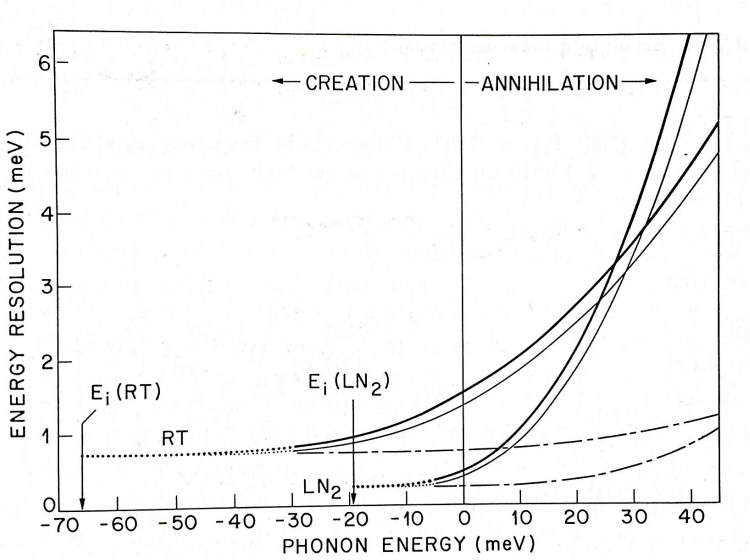


Figure 2: (*Hulpke*, 1992, p. 16) Calculated effective width (dark solid lines) of peak in energy spectra in relation to respective phonon energy. RT and LN_2 refer to room temperature and liquid nitrogen temperature, respectively. Different experimental improvements to resolution are shown: Lighter solid lines indicate moving chopper behind scattering target. Dotted-slashed line show descreasing L_D and Δ_C to 20 % of their previous value. Dotted lines indicate areas where measurements are difficult due to low speed of scattered helium beam.

However, phonon annihilation is often more relevant. For obvious reasons, no phonon creation can be triggered for $\Delta E > E_i$. Unfortunately, E_i can not be increased indefinitely, because multi-phonon scattering will overshadow the singe-phonon processes at some point. Therefore, annihilation becomes the only possibility when measuring high energy surface phonons in a regime where $\Delta E > E_i$.
In general, the effect of the time-of-flight spread on the energy resolution is also much greater than the contributions from kinematic smearing, mentioned in section 3.2.1. Equation 13 is used to visualize these effects on the energy resolution for different experimental parameters (figure 2). It shows the width of a peak in the energy spectrum plotted against the phonon energy ΔE for a number of variations of the experimental parameters. RT and LN_2 refer to room temperature and liquid nitrogen temperature of the beam nozzle, respectively. The most substantial gains in resolution can be achieved by decreasing the detector length L_D and the chopper shutter time spread Δ_C. The former has technical limitations on the minimal size of an ionizer. Whereas the latter is not only dependent on the chopper parameters, like slit width or chopper frequency, it also depends on the beam diameter. However, the beam diameter cannot be made arbitrarily small because it becomes very difficult to create, guide and detect the beam.
All in all, the time spread of the beam chopper, the length of the ionizer and the velocity spread of the helium beam require the primary focus, when aiming for an outstanding energy resolution.

3.2.3. Grazing Incident Helium Atom Scattering

The incident angle ϑ_i has a substantial effect on the entire measurement. By changing its value, we can adjust the parallel incident momentum K_i, which has implications for the scattering process. In many experimental set ups, the final angle ϑ_f is also tied to ϑ_i through: $\Theta_T = \vartheta_i + \vartheta_f$. In the experiment in the Center for Advanced Materials, we plan to employ two beam source chambers with $\Theta_T = 90°$ and $\Theta_T = 180°$, respectively. The second source will be used for grazing incident helium atom scattering (GI-HAS).
In section 3.1.1, I showed that only the parallel component of the complete incident wave vector $\overline{k_i}$ is considered in the momentum conservation. This is quite intuitive since the wave vector $\overline{Q}$ of a two-dimensional surface phonon should not have any component parallel to the surface normal (from here on defined as Z-direction). In grazing incidence we minimize the Z component of $\overline{k_i}$ (and thus maximize $\overline{K_i}$) because it does not contribute to the atom-phonon-interaction at all. Since $k_{i,z} = k_i \cdot \cos\vartheta_i$, we have to move the incident angle close to $90°$ and thus "graze" the surface with the atom beam. The closer ϑ_i gets to $\vartheta_i = 90°$,the higher the beam reflectivity. This is a strict consequence of the fact that the scattering process is quantum mechanical in nature.
Another point to consider is the larger effective surface A_{eff}, which is probed by the helium beam. Starting from the area of an ellipses: $A = \pi ab$, and defining b as the semi-major axis parallel to $\overline{K_i}$:

$$b = \frac{r_B}{\sin(90 - \vartheta_i)}\ ,\ a = r_B \implies A_{eff} = \frac{\pi {r_B}^2}{\sin(90 - \vartheta_i)}. \tag{14}$$

As expected, given a constant beam radius r_B, A_{eff} increases non-linearly with shallower incidence. Consequently, the goal is to minimize $90° - \vartheta_i$. At some point, however, A_{eff} will exceed the size of the sample surface. The disadvantage that a smaller percentage of helium atoms from the beam will scatter at the surface, is offset by the fact that the inelastic scattering happens with virtually no Z component of $\overline{k_i}$. The percentage of scattered particles at a given angle ϑ_i can be increased by making the beam as thin as possible.
Furthermore, the analysis of a grazing incident beam simplifies significantly. E_i can be derived directly by aiming the atom beam at the detector and measuring its velocity as described in section 3.2.1. This assumes that the path difference between scattered and unscattered atoms is negligible. Therefore, it is of vital importance to adjust the incident angle to as close to $90°$ as possible.
Finally, the scan curve calculated with equation 5 simplifies to:

$$\frac{\Delta E}{E_i} = \left(\frac{\Delta K}{k_i}\right)^2 + \frac{\Delta K}{k_i}. \tag{15}$$

3.2.4. Practical Consideration from Theory

From the above section, we summarize some considerations for the actual design of an HAS experiment:

Vacuum Requirements The aim of the entire experiment is to use a quadropole mass spectrometer to precisely measure the arrival times of single helium atoms. It is therefore important to have a low background noise at the detector chamber. Thus, after they have been measured, any remaining helium atoms pose a problem. The solution is to employ a pass through detector and collect the particles in a beam

dump behind the detector. In addition, a capable system of differential pump stages should be utilized to keep stray helium atoms from entering the detector area, and thus, creating the required vacuum conditions for precise measurements.

Linear Displacement The measurement principle relies on the fact, that the detector can be positioned at different distances from the target surface. Consequently, it has to be mounted on linear rails and care has to be taken to make sure the movement of the detector in parallel with the beam axis.

Pumping Power The helium beam is generated via free jet expansion. In broad terms, that means helium moves under high pressure through a nozzle and expands into an adjacent vacuum area. During the point-wise expansion, the helium's ambient temperature drops significantly, which leads to the desired sharp velocity distribution ($\frac{\Delta v}{v} \approx 1\ \%$) (*Benedek and Toennies*, 2018, p. 265). However, the pressure difference between nozzle and vacuum area has to be kept stable even though a significant amount of helium particles is constantly released into the vacuum. Thus, vacuum pumps with an extremely high throughput are necessary to ensure a continuous free jet expansion.

4. Implementations

In the above sections, I described the general principle of helium atom scattering and outlined the ideas behind time-of-flight measurements. This concluded with the considerations for our HAS implementation recounted above. Those considerations will act as guides for the design and implementation of power management and detector chambers, described in this section.

4.1. Power Management

The GIHAS experiment makes extensive use of diffusion pumps. It requires ultra high vacuum and particular strong pumping powers in selected areas. As explained in section 3.2.4, the helium beam is generated via pressure differences. That leads to the necessity to remove large amounts of excess helium, that is expanded but does not end up in the actual beam. This is done by powerful diffusion pumps that have a high power consumption. Therefore, a fail-safe method of distributing power and monitoring all pumps is needed.
The excess heat, primarily produced by the diffusion pumps, is removed via water cooling. To make sure in case of water leakage none of the electrical equipment gets damaged, an automatic way for detecting leaks and shutting down the water supply is also required.
The temperature and flow rate of the cooling water also needs to be tracked. So if any irregularity occurs all affected components can be shut down in a controlled manner.
Lastly in an event in which pumps break down, we want to protect as much of the vacuum as possible from being contaminated by the back flow of air through the fore- and UHV-pumps.
All together, the system requires:

- Automatic and constant **monitoring** of temperature, cooling water flow, possible leakages and vacuum pressure
- Automatic **power up and shutdown** of electrical consumers depending on the observed variables
- Optional **manual control** over all consumers in the experiment
- Easy **mapping** of observed variables and electrical consumers to facilitate upgrades of the experiment
- Continuous **logging** of the systems entire state
- Automatic **notification** to operators in case a malfunction occurs

4.1.1. General Design and Logic

The basic principle of this power unit is to map digital inputs (temperature sensors, water flow sensors, etc.) to digital outputs (relays connected to all power sockets). This mapping can be represented as a $n \times m$ - Matrix. Figure 3 depicts a flow chart of this normal mode of operation:
Each input has a rule, which defines the state in which the outputs should be if that particular input registers an error. Error means the input state changes and now differs from the normal operation mode value for this input. From now on this is called a sensor event.

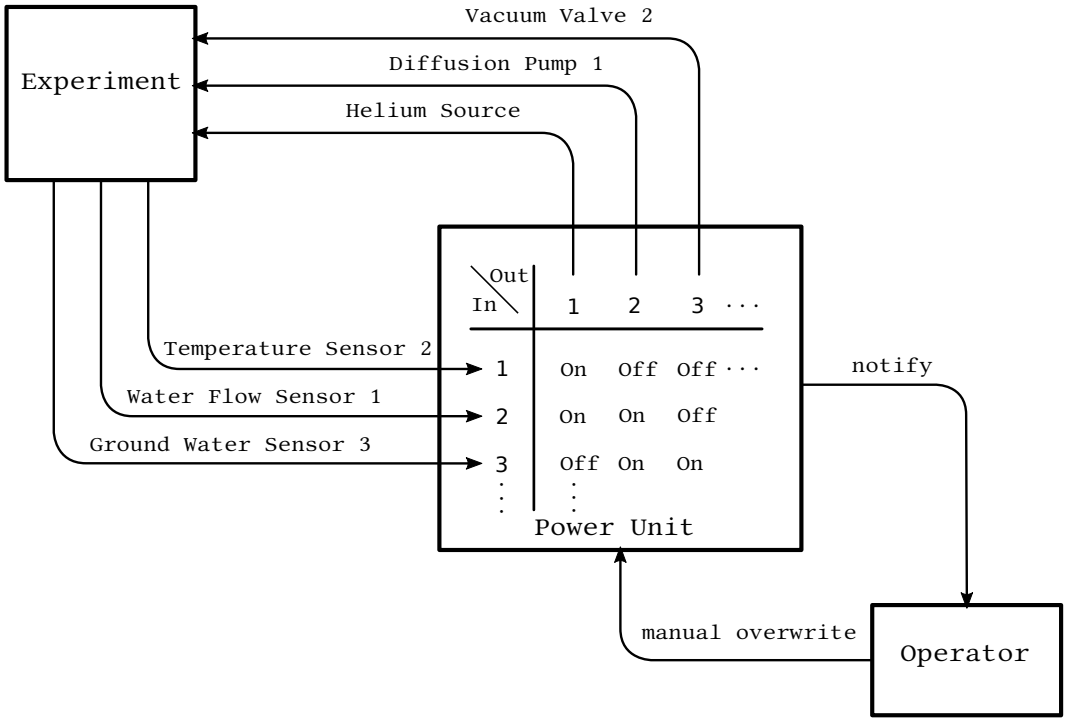


Figure 3: Control logic of power unit. A predefined matrix decides which outputs are disabled if a certain input registers a value which is different from its expected value.

Therewith, it is possible to define the behavior of all outputs depending on the inputs by altering the elements of the input/output matrix.
In our current implementation no outputs are ever activated as a response to a sensor event. This feature could easily be added, but will not be used until the normal power control has been tested for a substantial period of time.
In addition, the operator can manually override the state of all inputs and outputs. This ensures that the operator can always interfere in case a sensor or part of the machine breaks down. It also allows the operator to override the normal operation mode in case of a system malfunction or if the experiment is in an unexpected state during testing. The power unit displays both the actual and the automatic status through LEDs on its front side to the operator. They can then decide whether to allow the planned action or set the outputs manually. In case no personnel is in the laboratory during a sensor event, a report is also sent remotely.
In the following I will outline the general hardware setup of the control system.

4.1.2. Implementation of Hardware Components

The hardware for the control unit can be roughly split into high and low voltage components. The high voltage elements deal with the supply and distribution of three-phase and single phase AC current to the power sockets. They have been designed and built in *Paknejad* (2017). The low voltage side is responsible for control and monitoring of all input and output sockets.
Figure 4 shows the main components and the design of their connections. The represen-

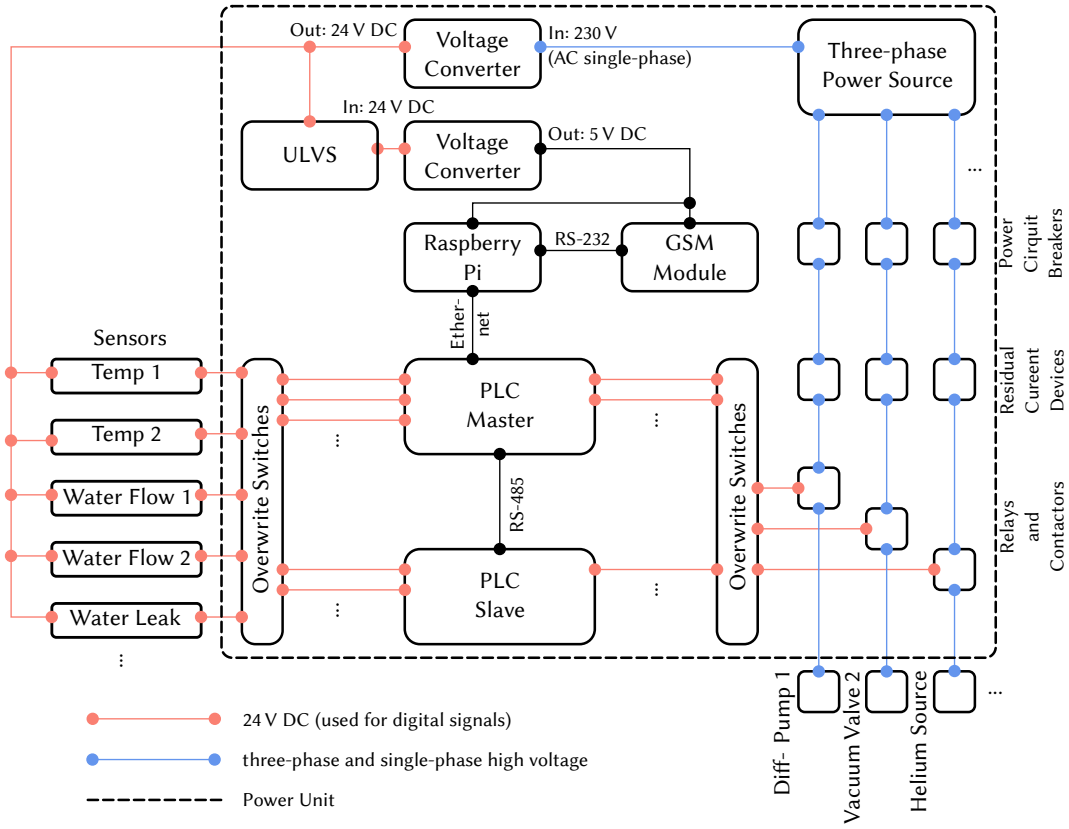


Figure 4: Implementation and connections of hardware components inside power unit. Some details such as safety fuses and power supply for PLCs and for the override switches have been omitted for a more concise representation. For the complete circuit layout please see appendix D.

tation focuses on the low voltage control logic of the power unit.
The power supply for all control elements is drawn from the two voltage converters that supply 24 V and 5 V. All sensors are powered from a 24 V line that leads out of the power unit (leftmost red connection). In essence, every sensor controls one relay. Their opening and closing encodes the signal, which feeds back to the power unit and enters either the master of the slave PLC. The whole mapping system outlined in section 4.1.1 is realized within the master PLC. After the mapping is carried out the PLC controls the power outputs via relays and contactors. Some implementation details, such as safety fuses are omitted in the figure. For a complete circuit diagram please see appendix D.
Apart from the override switches, which are explained in section 4.1.3, each component and its usage is described below:

High Voltage Components The power supply unit has 60 schuko-sockets of which 12 are secured with a 16 A fuse (blue casing) and the rest of them with a 6 A fuse (black casing). Furthermore, it is equipped with 18 three-phase sockets, which can put out a maximum of 16 A at 400 V (*Paknejad*, 2017, p.29). Two additional three-phase sockets can supply a maximum current of 35 A at 400 V and are reserved for the large diffusion pumps of the source chambers.

Each of these outputs is secured with a residual current protective device and a power circuit breaker to ensure save operation of all electrical devices. All of them can be controlled via contactors and relays. The details of their implementation can be found in *Paknejad* (2017).
All in all, 80 sockets can be operated by the power unit. To run all sockets simultaneously, the power unit is supplied by a three-phase $128\,\mathrm{A}$ at $400\,\mathrm{V}$ power line. (*Paknejad*, 2017, p. 29).

Programmable Logic Controller (PLC) This component responsible for the actual implementation of the logic described in section 4.1.1 is a *Controllino Mega.* An ATmega2560 microcontroller is used as a processor unit. The supply voltage is set to 24 V, since the relays managed by the Controllino require that level.
The PLC has 16 inputs ("Analog0" to "Analog15" in figure 5), which in this case will only be used as digital inputs. They interpret an incoming potential between 18 V and 26,4 V as a logical 1 and between 0 V and 7,2 V as a logical 0.
The voltage of the 36 outputs is set to 24 V as well. 16 of these outputs are carried out as relays that switch a 24 V source on or off ("Relay0" to "Relay15" in figure 5). The reset ("Digital0" to "Digital19") are digital outputs. Two Controllinos are employed in the power unit to have enough input and output terminals available for all sensors and power sockets. They arer set up in a master/slave configuration. The communication between two PLCs is possible via RS485. Alternatively, the unit connects via the Ethernet, as well. The maximum current of each digital output is limited to 2 A to protect against short circuits. The pins of the internal ATmega2560 are also directly accessible at the top panel and not shown in figure 5. This can be used to reboot the PLC by setting a 5 V potential to the internal ATmega2560 reset pin.

Raspberry Pi A small computer, model: *Raspberry Pi 3 B V1.2*, is connected to one of the Controllinos via its Ethernet connection. Its task is to continuously log the state of the experiments and to notify the operators if a sensor reports an anomaly.
It has multiple USB connections, an Ethernet port, WLAN module, and GPIO connectors. The same device will be used for a test bench to verify the power units internal logic.

GSM Module The device SIEMENS TC35 connects to the Raspberry Pi via RS232. Provided it has an activated SIM card attached, it can be used to send Text notifications if an anomaly is logged by the Raspberry Pi.

Temperature Sensor To make sure the pumping system is adequately cooled, temperature sensors are placed on the cooling system. For that purpose, a PT100 sensors together with digital meters from AOYI Electricity with the model number XMTD-2132 are used. These types of sensors are resistance temperature detectors. They work by measuring the resistance of a suitable material with a well-known resistance/temperature relationship - in this case, platinum with a linear temperature coefficient of $\alpha = 0.003\,85\,^{\circ}\mathrm{C}$ within the range from $0\,^{\circ}\mathrm{C}$ to $100\,^{\circ}\mathrm{C}$, where:

$$\alpha = \frac{R_{100} - R_0}{100\,^{\circ}\mathrm{C} \cdot R_{100}}. \tag{16}$$

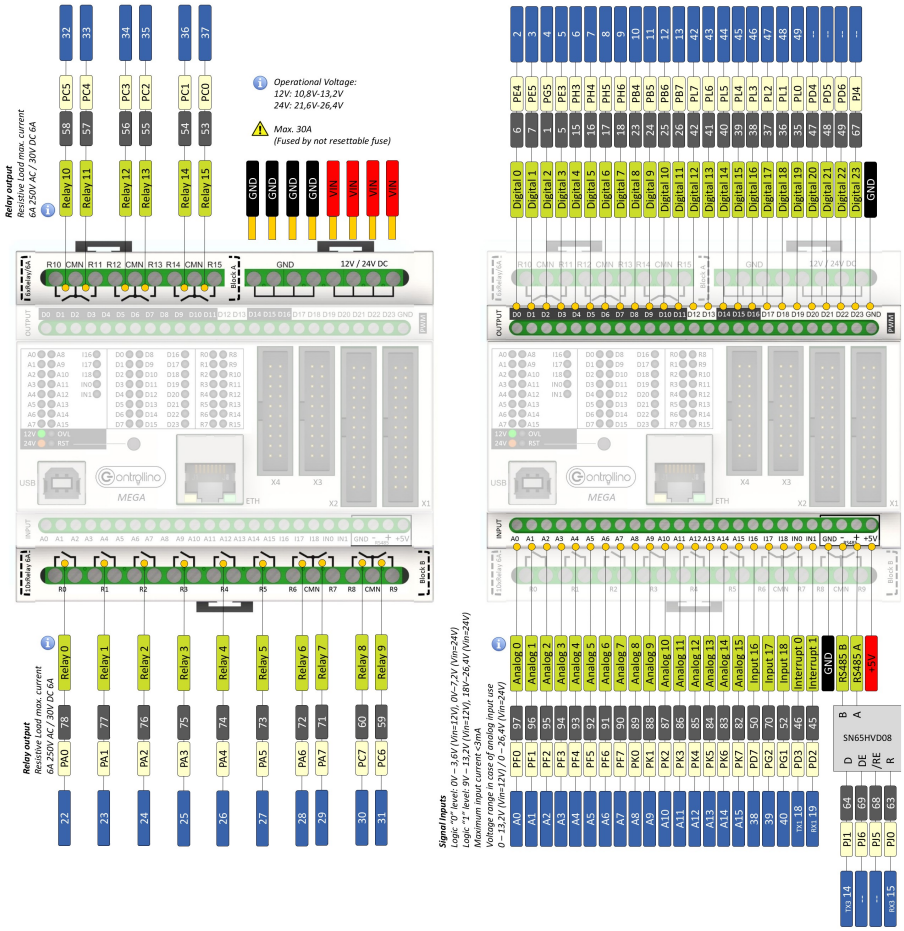


Figure 5: Pinout table of a PLC "Controllino Mega" (*CONELCOM*, 2018).

While R_0 and R_{100} denotes the resistance at 0 and 100 °C respectively.
In our case, the sensors only need to give out a notification if the temperature rises above predefined value. For this purpose the digital meter has a build in relay, which opens above an adjustable threshold. The details and manual for this device can be found in *Paggi* (2020) and *Willer* (2020).

Water Flow Sensor A reliable cooling system is paramount for any long term usage, since the experiment consumes a substantial amount of power. To make sure each vacuum pump is sufficiently supplied with water as a coolant, water flow sensors from *PKP*, model: DS52, are installed. They work passively by opening a switch if the water flow drops below an adjustable threshold.
They cannot be used to measure the water flow precisely, since they only have an accuracy of $\pm$ 10 %[1]. However, they are suited to detect whether ot not sufficient water flow is present at all times.

[1]see datasheet: `https://www.pkp.de/images/produkte/pdf_bed/ds52-bed-d.pdf`, accessed on 14.5.2020

Ground Water Sensor To make sure no water spillage remains undetected, ground water sensors of type: *Pollin Wassermelder* Version 1.0 and 1.1 are distributed across the laboratory floor. They work by measuring the resistance between two contacts. If these are connected by a body of water the resistance between them drops below a cetrain level. The sensor opens a relay, which is normally closed as long as the sensor is supplied with power. Out of the box, its logic is inverted. The relay would close if the resistance drops but we adapted the sensors wiring to get the inverted behavior.
Two versions of the sensor are used that have slightly different wiring schematics. Distinct changes were required for both versions. In Figure 6 the schematics for both types with the necessary adaptations (red) are shown. In both cases, one has to invert the inputs of one operational amplifier.
To accommodate multiple sensors within one 19-inch rack, a casing for five sensors has been designed and built by *Paggi* (2020) and *Willer* (2020).

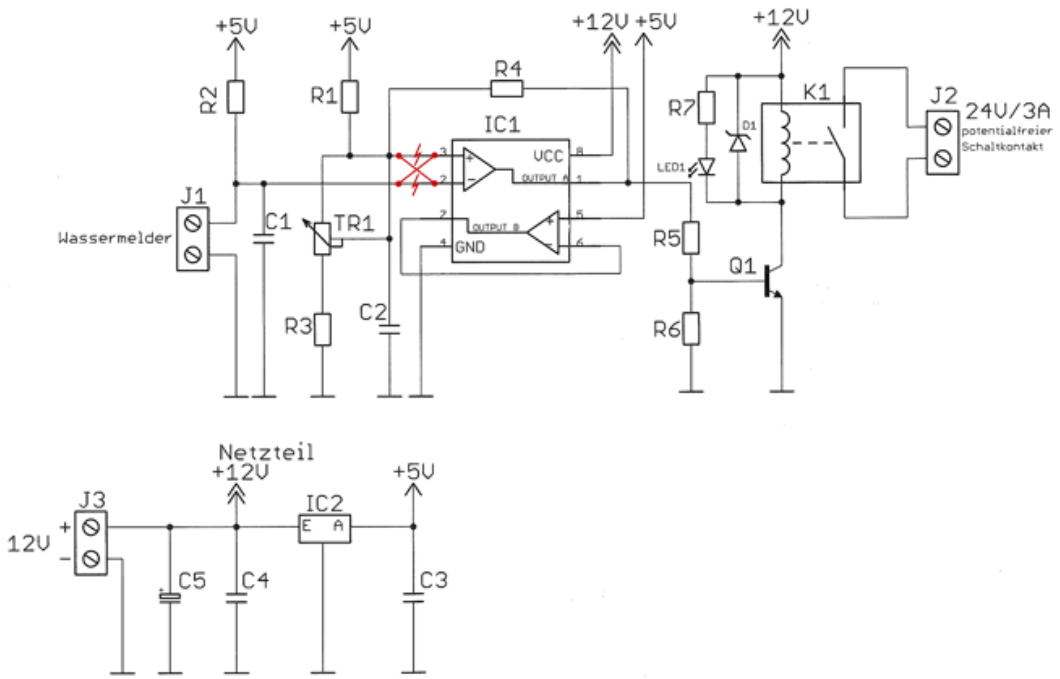
Uninterruptible Low Voltage Source (ULVS) In case of a power outage, it is required to continue recording the state of the experiment and simultaneously notify the responsible operator. The USV-module *SITOP 6EP1931-2DC21* from Siemens is used for that purpose. It generates a 24 V DC current. If the normal power fails a backup battery supplies limited emergency power. This can be used to power the raspberry pi and the GSM module.
Unfortunately the current USV-module does not provide enough power to additionally run all sensors and the PLCs as well. A second USV-module would be necessary to simultaneously supply all devices.

Voltage Converters Two voltage converters are used to get the low voltage necessary for most of the above components. One is the *SITOP PSU100S* from Siemens, which is backed up by the uninterruptible low voltage source. It provides 24 V DC with an emitted active power of 480 W.[2]
Another voltage transformer from 24 V to 5 V supplies the Raspberry Pi and the GSM module. We use the model *PYB20-Q24-S5-T* from CUI Inc. with an emitted active power of 20 W.[3]

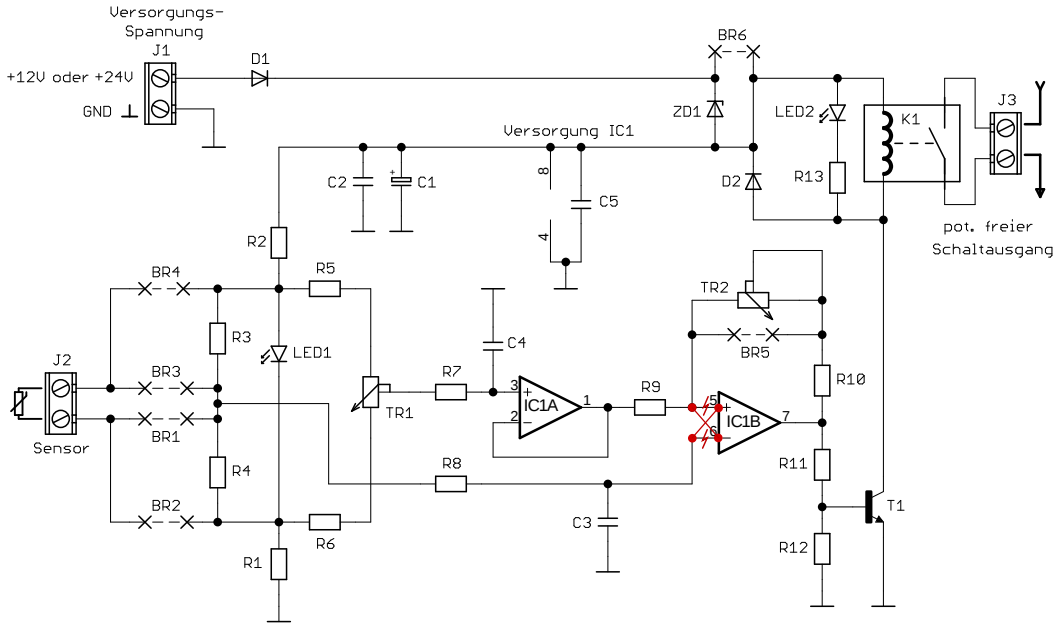
4.1.3. Setup of Manual Control

The manual control is realized with an override switch for every input and output pin. The connection of those switches are indicated in figure 4. They are positioned directly before and after relays and sensors, respectively. In both cases each switch is equipped with two indicator LEDs. The switch itself is a three-way rotary switch. Its positions and LEDs have slightly different meaning for input and output. However, the connection layout for both is similar and shown in figure 7. The switch consists of two separate two-way-levers. Both have a normally open (NO), a normally closed (NC) and a contact (C) terminal. Here "normally" refers to the middle position of the rotary switch. This means that the NC and C terminal of both levers are connected in that position. To build a three-way switch

[2] see datasheet: https://cache.industry.siemens.com/dl/files/474/67476474/att_81235/v1/PSU100S_Handbuch_de-DE.pdf, accessed on 15.5.2020
[3] see datasheet: https://www.mouser.de/datasheet/2/670/pyb20-t-1312599.pdf, accessed on 15.5.2020



(a) Schematics for version 1.0



(b) Schematics for version 1.1

Figure 6: Circuit diagram for two versions of the ground water sensor. Changes (shown in red) are necessary to enable the relay K1 to open when water is detected. Circuits are taken from a print out manual of *Pollin* ground water sensor. Purchased from: *https://www.pollin.de/p/bausatz-wassermelder-810141*, accessed on 14.5.2020

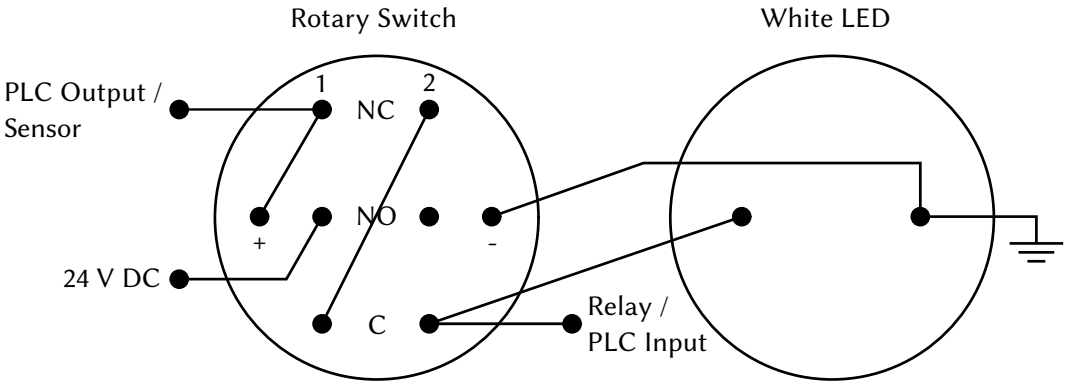


Figure 7: Backside connections of the rotary switch. Input and Output schematics are similar except for different connections on NC1 and C2. The switch consists of two separate two-way-levers (labeled with 1 and 2) which are interconnected between NC and C. The external LED has no predefined polarity since it uses a built-in rectifier.

Figure 8: Front and back view of rotary switch and status LED. Currently in "automatic" setting. Each component is labeled with a reference from the circuit diagram in section D.

from this set up, the C1 and NC2 terminals were soldered together. Now C2 is connected with either NO2, NO1 or NC1 depending on the switch position. The connections to those three terminals were defined in such a way that the middle position translated to an "automatic on/off" mode, counter clock wise flipping to "manually off" and clock wise flipping to "manually on". The "+" and "-" terminals are the contacts for an internal LED. In contrast the external LED uses a built-in rectifier and therefore has no defined polarity. Figure 8 shows the front and back side of one switch-LED-pair, mounted to the front of the power unit

After describing the general design of the manual controls I will now focus on the differences between the input and output side of the power unit.
The control panel, shown in figure 8, belongs to the input pin A0. Its wiring schematics is depicted in Figure 9a. B1 is a socket for an arbitrary sensor plugged into the power unit. Every sensor is expected to function as a normally closed relay. The settings of the switch are on, off and automatic. On and off means A0 is pulled to 24 V or 0 V, respectively. Automatic indicates that the state of the pin is controlled by the sensor. Two indicator LEDs are added to provide visual feedback to the operator. LED "17V1.1" shines blue and is built into the rotary switch itself. It shows the state the sensor is reporting. "17V1.2" is a separate, white glowing LED. It signals the actual state of the PLC's input pin at any given moment. Figure 9b shows the wiring schematics for output pin D0. The three switch settings are the same as for input but their meanings are slightly different. On and off refer to closing and opening the relay (20K4) of the power socket. Automatic means the PLC controls the relay. 21V1.1 is the integrated, blue LED and indicates whether the PLC would enable the socket. The separate, white LED (21V1.2) signals the actual state of the power socket. In general one can think of the blue LED as an indicator for the state in automatic setting and the white for the actual state of the input pin or output relay.
The schematics also explain why the PLCs and sensors cannot be connected to the emergency power supply described in section 4.1.2. When the automatic setting is chosen all input LEDs are powered by the incoming connection from the sensors. On the output side the PLC has to power all LEDs and relays of each socket which is set to automatic. The power consumption of all those devices combined is too high for the uninterruptible low voltage source.

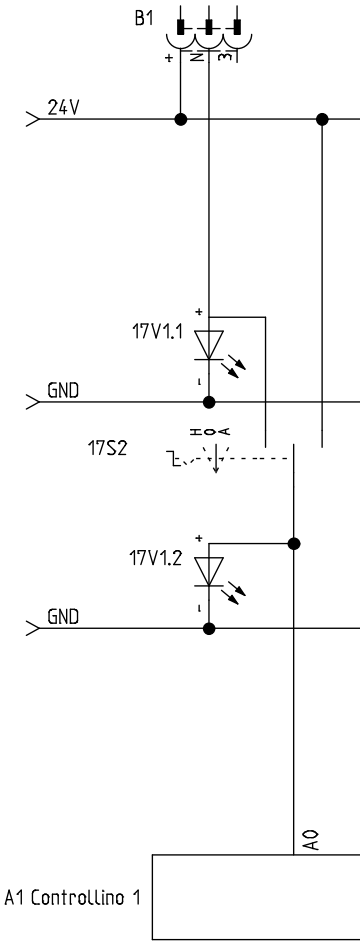
The power unit can monitor a maximum of 36 inputs and manage 80 outputs. All those manual controls have to be placed on the front side of the power unit. To make the controls more concise and intuitive, a layout has been designed which places most outputs in the order in which their respective consumers appear in the experiment (figure 10). The layout is realized by cutting a hole grid pattern into the front panels. The diameter is chosen as such that the rotary switches and external LEDs fit exactly. The output controls are divided into three groups.

Fixed Outputs which have a predefined electrical consumer according to figure 10.

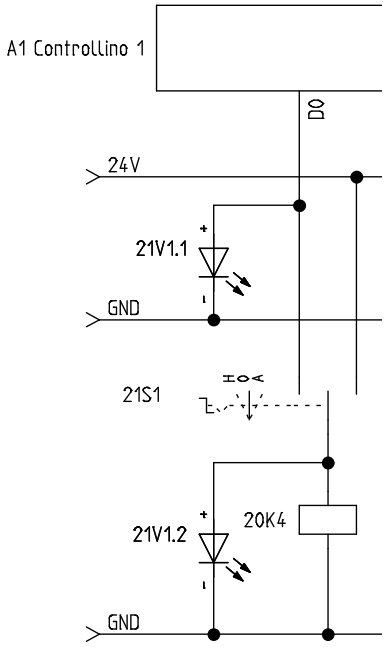
General General purpose outputs controllable by the PLC but for no specific consumers.

Uncontrolled Outputs with no specific consumer and not controllable by the PLC.

Table 1 concludes this section by indicating the PLC connections, usage parameters and switch positions for all output sockets.



(a) Input LED-Switch pair schematics. The LED 17V1.1 shows the state of the sensor output. LED 17V1.2 indicates the actual state of the corresponding PLC input. Automatic setting means state of sensor gets passed on to PLC.



(b) Output LED-Switch pair schematics. The LED 21V1.1 shows the state of the PLC output. LED 21V1.2 indicates the actual state of the corresponding output relay. Automatic setting means PLC controls output relay.

Figure 9: Circuit diagram of the manual control for input and output of the PLCs. A three-way rotary switch is used to choose between on, off and automatic. Two LEDs indicate the actual state and the chosen state in automatic setting. B1 is the socket for an arbitrary sensor. A0 is the input terminal of the PLC A1. 17S2 and 21S1 denote three-way rotary switches. D0 is the output terminal of the PLC A1.

Table 1: Connection table for all output sockets. Pos indicates the position of its override switch on the front, right hand panel of the power unit. The lowest leftmost hole is defined as 1,1. 3-P indicates a three-phase socket.

PLC Terminal	Output Socket	Intended Use	Power	Line	Pos [x, y]
D0	G20	HS-20, Source 1	30A	3-P	2,6
D1	G19	Roots 1, Source 1	16A	3-P	2,4
D2	G18	Roots 2, Source 1	16A	3-P	2,3
D3	G21	2W Valve, Source 1	6A	L1	2,2
D4	G17	Pre-pump, Source 1	16A	3-P	2,1
D5	G63	Varian 0184, Source 1	16A	L1	4,6
D6	G22	2W Valve, Chop 1	6A	L1	4,2
D7	G23	Pre-pump, Chop 1	6A	L2	4,1
D8	G16	HS-20, Source 2	30A	3-P	6,6
D9	G15	Roots 1, Source 2	16A	3-P	6,4
D10	G14	Roots2, Source 2	16A	3-P	6,3
D11	G24	2W Valve, Source 2	6A	L2	6,2
D12	G13	Pre-pump, Source 2	16A	3-P	6,1
D13	G65	Varian 0183, Chop 2	16A	L2	8,6
D14	G25	2W Valve, Chop 2	6A	L3	8,2
D15	G26	Pre-pump, Chop 2	6A	L3	8,1
D16	G28	Tubo pump 1, Scatt	6A	L1	10,12
D17	G28	Turbo pump 2, Scatt	6A	L1	10,11
D18	G66	Ed 100, Scatt	16A	L2	10,10
D19	G77	Ed 63, Scatt	16A	L2	10,6
R0	G29	2W Valve, Scatt	6A	L2	10,2
R1	G30	Pre-pump, Scatt	6A	L2	10,1
R2	G31	Turbo pump, Pitot	6A	L3	12,7
R3	G67	2xEd 100 1, ToF	16A	L3	13,6
R4	G32	2W Valve, ToF	6A	L3	16,2
R5	G33	Pre-pump, ToF	6A	L2	16,1
R6	G68	2xEd 100 2, ToF	16A	L3	17,6
R7	G34	Turbo pump, Det	6A	L1	20,7
R8	G64	Ed 63, Det	16A	L1	20,3
R9	G35	2W Valve, Det	6A	L2	20,2
R10	G36	Pre-pump, Det	6A	L2	20,1
R11	G42	-	6A	L2	14,29
R12	G43	-	6A	L3	14,27
R13	G44	-	6A	L3	14,25
R14	G70	-	16A	L1	14,23
R15	G80	-	16A	L3	14,21
S:D0	G45	-	6A	L1	14,19
S:D1	G46	-	6A	L1	14,17
S:D2	G47	-	6A	L2	14,15
-	G39	-	6A	L1	20,29
-	G40	-	6A	L1	20,27
-	G41	-	6A	L2	20,25
-	G37	-	6A	L3	20,23
-	G38	-	6A	L3	20,21
-	G69	-	16A	L1	20,19
-	G78	-	16A	L2	20,17
-	G79	-	16A	L3	20,15

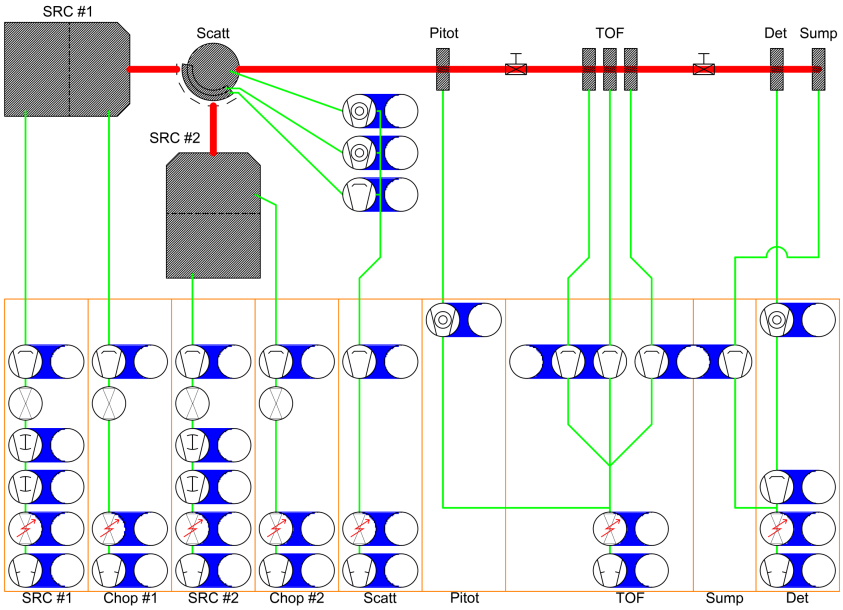


Figure 10: Layout of predefined outputs on the front panel of the power unit. The position of the electrical consumers in this layout corresponds to the position of their respective LED-switch pairs on the right hand side front panel. Designed by Tom Turczyk, permission to publish given on 06.05.2020

4.1.4. Testbench for the PLC Program

Since the power unit is in charge of continuously monitoring the experiment with minimal supervision, it is essential that the logic behind it performs as intended. That is why, a test bench for the PLCs was build and a test suite developed.
In principle, a Raspberry Pi changes the state of the input pins of the PLCs to simulate a sensor reporting an anomaly. The PLC is connected via Ethernet to the Raspberry Pi. The information which is normally send for logging purposes is now evaluated by our test suite. Each test simulates a certain sensor event and expects a predefined response message from the Controllino. It counts as passed, if the correct response arrives within a 10 s time frame.
Care has been taken to design the test suite in a way that it is easy to incorporate new test conditions. To archive that, the suite is implemented in Python using the *pytest*[4] library. For a detailed usage guide please see appendix A.3.

Components:

The Hardware for this test bench is shown in figure 11. It consists of the following components:

2 × Controllino Mega Programmable Logic Controllers which are loaded with the software that is to be tested. They have to be configured in the same way the PLCs inside the power unit would be.

[4] taken from `https://docs.pytest.org`, accessed on 12.8.2019

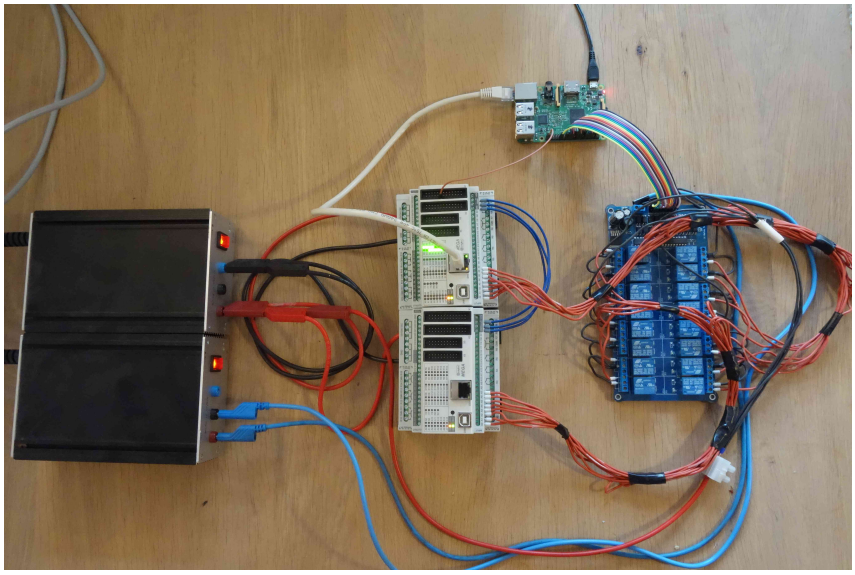


Figure 11: Test bench to simulate inputs for master and slave PLC. From left to right: Power sources (black), master (upper) and slave (lower) PLC, Raspberry Pi (green), Relayboard (blue).

Raspberry Pi Model 3 B Installed with Raspbian Stretch Lite. This component controls the input states of the PLCs and evaluates their outputs.

SainSmart 16 Channel Relay Board Adjusts the voltage coming from the GPIO pins of the Raspberry Pi so it can be registered by the Controllinos.

12 Volt DC Power Source Used to power the relay board.

24 Volt DC Power Source Provides power to PLCs and simulated inputs.

Micro-B-USB power connector Power source for Raspberry Pi.

Set Up:

1. Connect GPIO Pin of Raspberry Pi with input side of relay board using the connections scheme described in table 2 via a ribbon connector
2. Use single core $0.75\ mm^2$ electric wire to connect 24 Volt power supply to middle terminal of each relay on relay board.
3. Connect relay board channels with PLCs according to table 2. Use the ¸right hand terminal of each relay and use the same wire as described in the previous step.
4. Connect channel K9 to K16 of relay board with input terminals D0 to D7 of slave PLC in the same manner as in previous step.
5. Use Cat5 patch cable to connect Ethernet sockets of master PLC and Raspberry Pi.
6. Connect GPIO pin number 21 with reset pin[5] of master PLC.

[5]for exact pin position please see: `https://controllino.biz/wp-content/uploads/2018/10/CONTROLLINO-MEGA-Pinout.pfd`, accessed on 02.03.2020

Table 2: Connections between the GPIO pins of Raspberry Pi, channel on relay board and input pins of both PLCs.

(a) Master PLC connections

RPi GPIO	Relay Board	PLC master
02	K01	A00
03	K02	A01
04	K03	A02
14	K04	A03
15	K05	A04
18	K06	A05
17	K07	A06
27	K08	A07

(b) Slave PLC connections

RPi GPIO	Relay Board	PLC slave
22	K09	A00
23	K10	A01
24	K11	A02
10	K12	A03
09	K13	A04
25	K14	A05
08	K15	A06
07	K16	A07

7. Use single core $0.75\ mm^2$ electric wire to connect RS-485 interface of master and slave PLC as shown in figure 14.

8. Connect relay board with 12 Volt power supply using single core wire with banana plug.

9. Connect power supply terminals shown in figure 5 to the same 24 Volt power source as in step 2 using similar electric wire as in previous step.

10. Connect Micro-USB power supply to Raspberry Pi.

Test Logic:

Figure 12 describes the execution order and tasks of each method of the test program. The *pytest* library searches for all tests, which can be recognized by its file, class and function name. Each name has to start with "test". Every detected test is then carried out even if a previous test failed. In addition, *pytest* has the ability to encapsulate each test with a setup and tear down method. This means, the test bench is reset to a default configuration before each run. Similar functions exist for setup and tear down of an entire test class. I chose this encapsulated approach to ensure an easy way of implementing new test scenarios for future expansions of the PLC program. The entire test program can be found in appendix C.3.

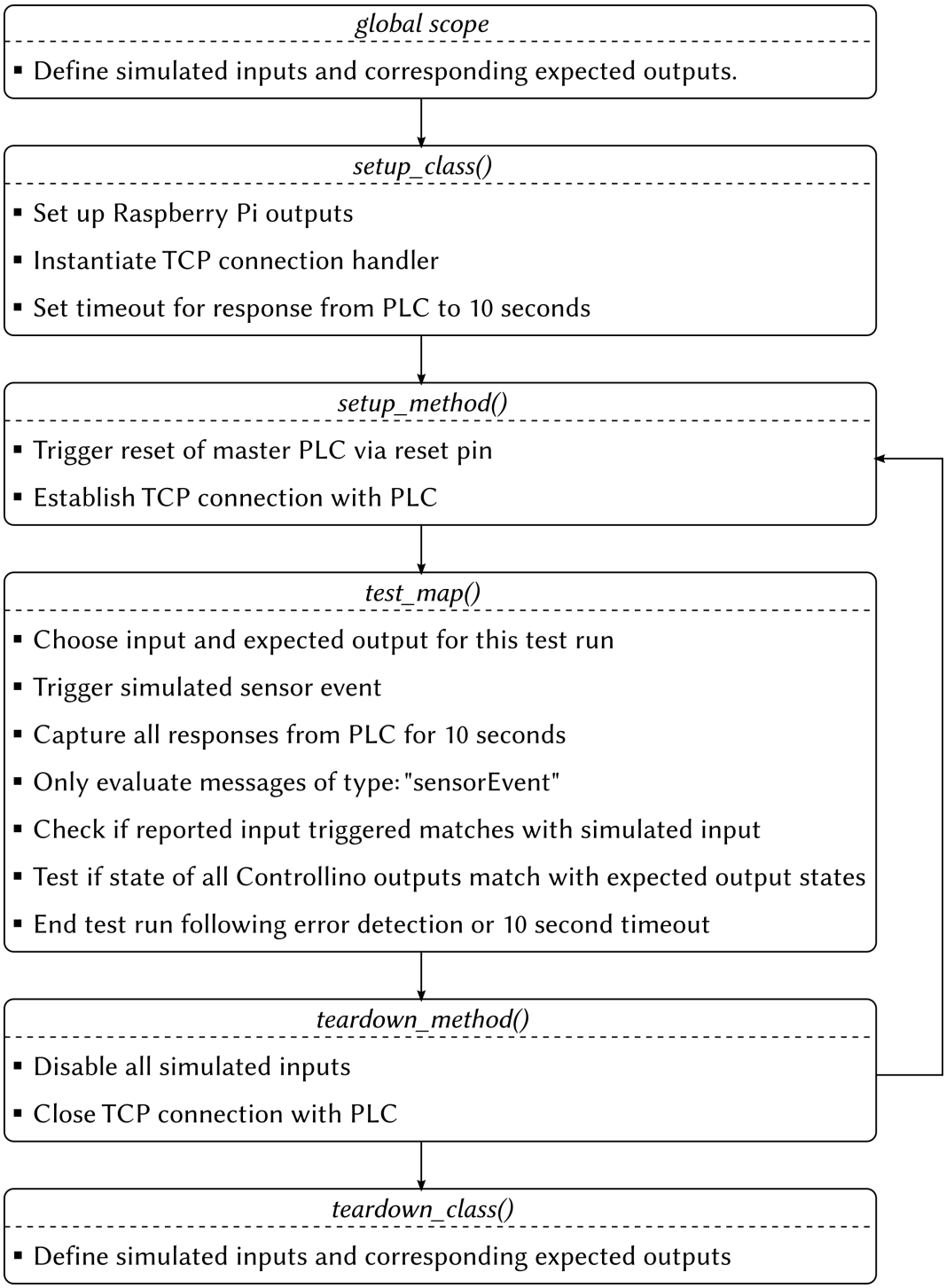


Figure 12: Flow chart of the test bench software. Arrows indicate the execution order. The tasks of each method are listed below its name. The related scripts can be found in appendix C.

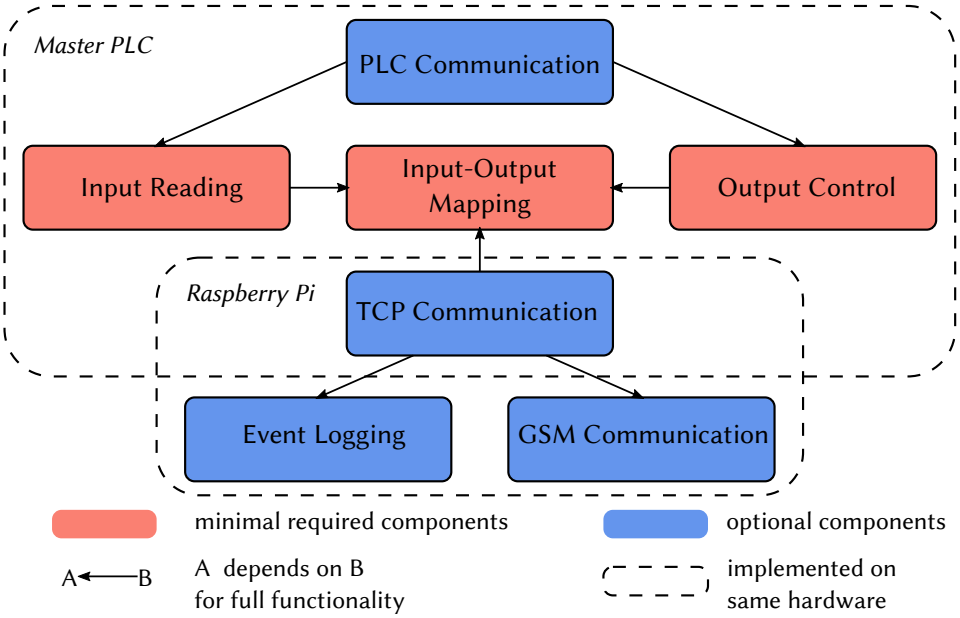


Figure 13: Overview of modules required for a working power unit control.

4.1.5. Control Software Development

The two primary goals for building the control software of the power unit were easy usage and fast expandability. With that in mind, the control logic is separated into seven components (see figure 13):

Input-Output-Mapping Encapsulated in a class named: *Machine*, this module is responsible for input-output-mapping, power unit start up, and TCP communication.

Input Reading This component is responsible for storing the input states and scanning each for state changes. Its logic is defined in the class: *Inputs*.

Output Control The module stores all output states and is responsible for setting each output to its respective state. Its logic is defined in the class: *Outputs*.

PLC Communication The RS-485 serial standard is used along with the Modbus RTU protocol. This establishes a master - slave communication structure between the two PLCs.

TCP Communication The TCP/IP software stack is used for communication between Raspberry Pi and master PLC. It employs a server (Raspberry Pi) - client (master PLC) architecture. The client will never expect a response from the server to minimize its dependencies.

GSM Communication The GSM module communicates with the Raspberry Pi via the RS-232 serial standard.

Event Logging The *pylogging*[6] library is used for recording all information sent from the master PLC.

[6] taken from: `https://pypi.org/project/pylogging/`, version 1.0.2, accessed on 3.5.2019

The minimal required components (shown in red) were developed first, establishing a basic working prototype. From then on, every software component was built via a test-driven development process. This entailed that the tests for each new component were written before its actual software, thereby establishing fixed milestones for every software iteration.

In the following sections, I will describe the details of each component. For a more concise representation, I will omit various parts of the code, such as constructor methods, and focus on the core functionality of each element. Here, I will assume basic knowledge of C++ and its usage of classes. For the complete program, please see appendix B.

4.1.6. Input Reading

The *Inputs* class has 3 main members.: *inputData*, *normalInput* and *inputRepr*.

```
1 class Inputs {
2     private:
3         Pin inputData[MAX_INPUT_SIZE];
4         byte normalInput[MAX_INPUT_SIZE];
5 
6     public:
7         byte inputRepr[MAX_INPUT_SIZE] = {};
8 }
```

inputData can be seen as the single source of truth. It always represents the current state of all inputs. For that purpose it consists of elements of a custom structure called *Pin*.

```
1 struct Pin {
2     String pinName;
3     byte pinNumber;
4     byte pinState ;
5 }
```

Here *pinNumber* refers to the value assigned to that physical pin by the arduino library. In figure 5, they can be recognized as the integer values with the blue background.
pinState is the digital state of that pin, which can be either 0 or 1. There are no guards in place to protect against adding other values for two reasons. Firstly, a user can hardly input wrong values because the member will only be referred to by methods from this class. And secondly, it makes it easier to expand this class to analog inputs in the future. For example, analog inputs could be utilized to analyze values from pressure gauges or temperature readings without intermediate controllers.
pinName is a name given to that pin. Importantly it has to start with either "M:" or "S:" to allow differentiation between inputs that belong to the master or slave PLC respectively.

normalInput is a byte array which holds the expected values for all inputs. This means, if a sensor is operating normally it should always transmit this value to the PLC. The order of *normalInput* needs to be the same as the order of *inputData*, since the index is the only way to refer to one specific input.

inputRepr is the public representation of all input states. It can be accessed from outside of the class and is used to transmit the input state to the Raspberry Pi. It is not strictly

necessary for the normal operation of the program and could be removed if the status reports do not longer rely on it.

The main objective of this class is to read and update all inputs and notify the *machine* class of any anomalies. In the following, if an input state deviates from its expected value it will be called a sensor event.
To archive that goal, multiple member functions are defined. At the lowest level stands: *readInput*.

```
1  int readInput(int inputNumber) {
2
3      Pin mpin = inputData[inputNumber];
4      if (mpin.pinName[0] == 'M') {
5          return digitalRead (inputData[inputNumber].pinNumber);
6      }
7      else if (mpin.pinName[0] == 'S') {
8          // send request for input value to slave PLC
9          // wait for response and return it
10     }
11 }
```

This function returns the current value for the input with index *inputNumber*. Most importantly, it checks whether the input is part of the master or slave PLC. In the latter case, a request is sent towards the second PLC. The details of this communication are described below in section 4.1.10.

The core method of this class is *getChanges()*, which checks whether any input has changed its state to something other than its expected value. It returns the index of that input, i.e. its position in *inputData*, if that is the case. It does so by looping over all elements of *inputData* and calling *readInput()* on each of them.

```
1  int getChanges() {
2      for (int i=0; i<MAX_INPUT_SIZE; i++) {
3          int tempInput = readInput(i);
4          if (inputData[i]. pinState != tempInput) {
5              delay (200);
6              if (inputData[i]. pinState == readInput(i)){
7                  continue;
8              }
9              inputRepr[i] = inputData[i]. pinState = tempInput;
10             if (tempInput!=normalInput[i]) {
11                 return i;
12             }
13         }
14     }
15     return -1;
16 }
```

The function also makes sure no false readings are mistaken for real ones. When ever it detects a sensor event, it will wait for 200 ms and reevaluate that input. On Line 11 it will then return the index of the rechecked input if its state differs from the expected value.
This guards against multiple triggering while a sensor is in the process of changing its

value. Additionally, it makes it possible to switch between manual and automatic input mode without inadvertently triggering a sensor event. The details of that switching operation are described in section 4.1.3.
Lastly, the function returns -1 if no sensor event was detected.

The *Inputs* class also comprises one method which checks if all sensors report their expected value.

```
1 int checknormalInput(){
2     for (int i=0; i<MAX_INPUT_SIZE; i++) {
3         if (readInput(i) != normalInput[i]){
4             return i;
5         }
6     }
7     return -1;
8 }
```

This is realized by looping over all elements of *inputData* in Line 2 and comparing the readings with their corresponding expected values. I use this function to make sure that the experiment is in working order before powering up any outputs.

4.1.7. Output Control

The *Outputs* class is similarly setup as *Inputs*. Again, it has the three core members:

outputData Represents the state of all outputs. Please find details on the *Pin* structure in the above section 4.1.6.

normalOutput Byte array of the states of all outputs if no input detects any errors.

outputRepr Public representation of all output states. It is not strictly needed for the functionality of the PLCs since it is only used for logging purposes.

The main tasks of this class is to write all outputs correctly and check whether an output state is already set beforehand. To do that at the lowest level, the method *writeOutput* finds out if a requested output is part of the master or the slave PLC and sets its required state.

```
1  void writeOutput(int outputNumber, byte outputState) {
2      Pin mpin = outputData[outputNumber];
3      if (mpin.pinName[0] == 'M') {
4          digitalWrite(mpin.pinNumber, outputState);
5      }
6      else if (mpin.pinName[0] == 'S') {
7          // send request to change output state
8          // wait for response
9      }
10 }
```

Again the details for communication between master and slave PLC will be explained below in section 4.1.10. As it was the case with input names, we can see here why it is important to start all names with either "M:" or "S:" for master and slave respectively. Otherwise the setting of that pin will be ignored entirely.

The next method, called *setOutput()*, is responsible for changing the state of the entire output array.

```
1 void setOutput(byte (&configArray)[MAX_OUTPUT_SIZE]) {
2     for (int i = 0; i < MAX_OUTPUT_SIZE; i++) {
3         if (configArray[i] < 2) {
4             writeOutput(i, configArray[i]);
5             outputRepr[i] = outputData[i].pinState = configArray[i];
6         }
7     }
8     // log changing of states
9 }
```

The input parameter *configArray* is a reference to a byte array with the same length as the number of outputs. Each element represents the requested state of the output with the same index in *outputData*. Therefore their order is important and has to be the same as in *outputData*.
It is often desirable to only change some of all outputs. For that purpose one can define 3 values for each element: "0": pulling the output to 0 V, "1": pulling the output to 24 V and "2": do not change this output at all. In practice, this is realized in Line 3 by ignoring all values of *configArray* which are higher than 1.

To check whether a certain output state is already in place, the method: *isAllreadySet()* can be used.

```
1  bool isAlreadySet(byte (&toBeChecked)[MAX_OUTPUT_SIZE]) {
2      bool isSet = false;
3
4      for (int i = 0; i < MAX_OUTPUT_SIZE; i++) {
5          if (toBeChecked[i] == 2) {
6              continue;
7          }
8          if (toBeChecked[i] != outputData[i].pinState) {
9              return false;
10         } else {
11             isSet = true;
12         }
13     }
14     return isSet;
15 }
```

Similar to the previous function, the parameter *toBeChecked* is a byte array with the same length as *outputData* and has three possible elements: 0, 1 or 2. Here the value 2 means: do not check the state of that output at all.
This means one always has to insert an array which will be checked against the entire output list but you can choose to ignore certain outputs. The method returns true if *toBeChecked* is the same as *outputData* on all positions with values of either 0 or 1. Otherwise it returns, which also includes the case that *toBeChecked* only consists of twos.

4.1.8. Input-Output Mapping

The main logic of the entire program is wrapped in the *Machine* class. It decides which outputs to disable and holds all information about the entire state of the experiment. For that reason its two primary member variables are *input* and *output*. Both are instances of the *Inputs* and *Outputs* classes respectively. Together with a two-dimensional byte array called *mmapping*, which holds the mapping matrix, it knows the state of the experiment at every point in time.

The mapping of input to output states is the core of this power unit. It can be seen as a matrix multiplication with a unit vector of the $\mathbb{R}^n$ space where n is the number of inputs:

$$\underbrace{\vec{e}_1^{\,T}}_{\text{input state}} \cdot \underbrace{\begin{pmatrix} 1 & 1 & 0 & \cdots \\ 0 & 1 & 1 & \\ 1 & 0 & 2 & \\ \vdots & & & \ddots \end{pmatrix}}_{\text{mapping matrix}} = \underbrace{\begin{pmatrix} 1 & 1 & 0 & \cdots \end{pmatrix}}_{\text{output state}} \cdot$$

Here the input state describes a registered state change to something other than the expected value of one input. With this mapping, it is possibly to register and respond to single sensor events. That means, the mapping can only be used to change the states of all outputs if one sensor is triggered.

The main method of this class is *runAllMappings()*. It uses the methods described in section 4.1.6 and 4.1.7 to make the required changes to output if a sensor event is detected.

```
1 void runAllMappings() {
2     int changedIndex = input.getChanges();
3     if (changedIndex != -1) {
4         if (!output.isAllreadySet(mmapping[changedIndex])){
5             output.setOutput(mmapping[changedIndex]);
6         }
7         // log sensor event and response
8     }
9 }
```

This function runs inside the main loop of the PLC. Therefore, it continuously checks on Line 2 and 3 whether a sensor event occurred. Should this be the case, it verifies that the required outputs from the mapping matrix are not already set. If so, it sets the new output states and logs the sensor event and its response.

The next method of the *Machine* class is responsible for turning on all outputs during the start up of the power unit and is called *checkAndEnableNormalOutput()*.

```
1 void checkAndEnableNormalOutput() {
2     int index = input.checknormalInput();
3     if (index == -1) {
4         output.setNormalOutput();
5     } else {
```

```
6           // log startup error and index variable
7       }
8   }
```

It uses the method *checknormalInput()* described in section 4.1.6 to test whether all inputs report their expected values. Tf that is the case, the normal output values can be set. The method realizing this simply calls *setOutput()* with the *Outputs* class variable *normalOutput* as the argument.

Another functionality of this class is to establish an Ethernet connection with the Raspberry Pi. We defined the *checkConnection()* method for that purpose. As the name suggests, it tests whether a connection exists and tries to establish one if that is not the case. It has a timer variable which can be adjusted to change to rate at which the connectivity is checked. The details of that communication is explained in section 4.1.11.

4.1.9. Setup and Control

Until now I described three different classes which are needed for operating the power unit. In the following, I will explain how exactly these classes are initialized and used to control all aspects of the system.
Since the *Controllino* is based on the Arduino infrastructure, its usage is quite similar to any other Arduino program. In general, the Arduino library takes care of all low level communication with the central ATmega2650 micro controller. Among others it comprises methods such as *digitalRead(pin)* or *digitalWrite(pin, value)* which we use to read and set the state of each input or output pin.
The language used for programming is at its core C++. But the Arduino compiler *WinAVR* has some convenience additions to make writing code more accessible, for example: automatic forward declaration of functions. Usually in C++ functions just like variables, structures or types have to be declared before they are called. The Arduino compiler, however, automatically inserts the prototype for all top-level functions at the beginning of a script. This enables the user to call functions which might be defined at the very end of the respective program. The removal of the normally required *main()* function is another simplification. It is replaced with *setup()* and *loop()*. In the background, these two functions will be converted to *main()* but they make the transition into embedded programming easier. I will explain the details of both function and how they are used below. Again, the following code listings only serve as a simplified visualization. The actual working program can be found in section B.

setup() This function is run once when the PLC powers on. All initial configurations like enabling communications or setting the modes of all pins correctly is done at this point. The definition of all input and output connections as well as the mapping between them is also defined here. Essentially this means under normal conditions the entire power unit can be configured within this function.

```
1   void setup () {
2
3       Pin inputPinLayout[] = {
4           {F("M:B01"), CONTROLLINO_A0, 1}, //H2O flow sensor
5           {F("M:B02"), CONTROLLINO_A1, 1}, //H2O flow sensor
```

```
6         {F("M:B03"), CONTROLLINO_A2, 1}, //Temperature sensor
7         {F("M:B04"), CONTROLLINO_A3, 1}, //H2O ground sensor
8     };
9
10    Pin outputPinLayout[] = {
11        {F("M:G31"), CONTROLLINO_R2, 1},
12        {F("M:G67"), CONTROLLINO_R3, 1},
13        {F("M:G32"), CONTROLLINO_R4, 1},
14        {F("M:G33"), CONTROLLINO_R5, 1},
15        {F("M:G68"), CONTROLLINO_R6, 1},
16    };
17
18    byte mapping[MAX_INPUT_SIZE][MAX_OUTPUT_SIZE] = {
19        //R02,R03,R04,R05,R06
20        {0 , 0 , 0 , 0 , 2}, // CONTROLLINO_A0
21        {0 , 0 , 0 , 0 , 2}, // CONTROLLINO_A1
22        {2 , 0 , 2 , 2 , 0}, // CONTROLLINO_A2
23        {0 , 0 , 2 , 2 , 0}, // CONTROLLINO_A3
24    };
25
26    const int interruptPin = CONTROLLINO_IN0;
27    pinMode(interruptPin, INPUT);
28     attachInterrupt ( digitalPinToInterrupt ( interruptPin ),
29    resetWrapper, RISING);
30
31    // start Serial communication for debugging
32
33    // start Ethernet connection
34
35    // enable connection with server over port 4000
36
37    // setup rs485 query and connection
38
39    controller .begin(inputPinLayout, outputPinLayout, mapping);
40
41    controller .checkAndEnableNormalOutput();
42 }
```

Line 3 to 16 define the input and output layouts. *Pin* is a structure with three variables, as it was described in section 4.1.6. In this case, the last variable is used slightly different. The first and second refer to the name and pin number of that particular input or output. Ih contrast to its previous usage, the last variable describes the values for *normalInput* and *normalOutput* of *Inputs* and *Outputs*, respectively. The *CONTROLLINO_X* notations are defined keywords from the Controllino library. They denote the values which represent the specific pins of the ATmega2560 micro controller.
On Line 18 to 24, the mapping matrix is defined. For safety reasons, no outputs will ever be enabled as a result of a sensor event. That is why matrix elements are only 0 or 2. But in principle, an automatic start up of systems depending on sensor input is also possible.
It is important to remember that the static variable *MAX_INPUT_SIZE* and *MAX_OUTPUT_SIZE* have to be manually adjusted at the very beginning of the script. This limitation stems from the fact that adjustable length arrays, i.e. vectors, do not exist in C++ without the standard library and recreating them would have caused more prob-

lems than it solved.

The next section enables the PLC to rerun *checkAndEnableNormalOutput()*. This functionality is important if we want to fix an error in the experiment without having to shut down all electrical devices. In case of a complete shutdown, the Controllino takes about 10 s to reboot. During that time all outputs are disabled. This is not ideal since some outputs may have to keep running after a sensor event. Therefore, it was necessary to develop an option for restarting all outputs without rebooting the PLC. The solution is to define an interrupt routine. The PLC has dedicated interrupt pins. Their purpose is to immediately stop the main program and run some predetermined function when they register a signal. I chose the rising flank of the applied voltage as that signal. But it can also be defined differently if required. The interrupt pin is attached to a standard switch which supplies 24 V to the pin if pressed. The function that runs as a consequence (line 28) is not as expected *checkAndEnableNormalOutput()* but *resetWrapper()*. This is necessary because interrupt routines have to be static functions without any arguments. But since non-static class methods, such as *checkAndEnableNormalOutput()*, always have a pointer, pointing to its own instance, they cannot be used as interrupts. That is why *resetWrapper()* is simply a top-level function which calls *checkAndEnableNormalOutput()*.
In line 31 to 37, the different communication systems are setup. Their details will be explained in separate sections and therefore are only shortly mentioned here.

In the following command, the *Machine* instance *controller* calls its *begin()* method. This can be understood as the constructor method for the *Machine* class. *controller* has to be a top-level variable because it stores the state of the experiment. It would be deleted once the program steps out of *setup()*, if it were to be initialized within that scope. On the other hand, an actual constructor method for *Machine* cannot run outside of *setup()* because the mode setting of pins is not allowed there by the Arduino framework. This is why the method *begin()* had to be defined. It does everything a normal constructor would, but can be called after initializing an instance. For the same reason *Inputs* and *Outputs* also have this kind of constructor methods.
Lastly, *checkAndEnableNormalOutput()* is called to start up the necessary outputs if all sensors are reporting no errors (line 41).

loop() The *loop()* function holds the main logic of the program. The PLC will continuously execute its body in a loop. It does not have the ability for threading. Therefore, it is important to avoid extensive blocking of the main thread It is possible, however, to run multiple functions in a pseudo-parallel mode with the use of timer variables. This is how the Controllino rechecks for an Ethernet connection with a set frequency without blocking the sensor readings.

```
1 void loop() {
2     controller.runAllMappings();
3     controller.checkConnection();
4 }
```

As one can see, the main loop only calls the *Machine* class methods: *runAllMappings()* and *checkConnection()*. Both are explained in section 4.1.8.

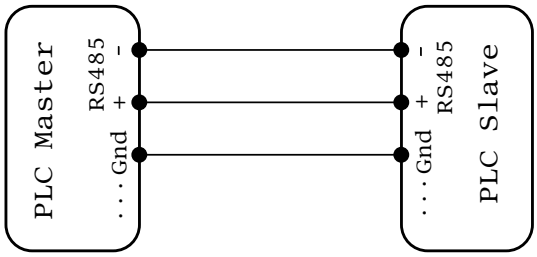


Figure 14: Wiring schematics for RS-485 connection between two Controllino Mega.

4.1.10. PLC Communication

Since the power unit has more sensor inputs and power outputs than one Controllino can support a second PLC had to be added. But an input from the first PLC might have in impact on the outputs of the second. That is why they need to communicate with each other.
To realize this, a master/slave system was chosen. The master PLC handles all communications, mappings and chooses the entire output state. It can send requests for either reading or writing to the slave, which continuously listens for them and only replies once a request has been received.
The Controllino Mega has an RS485 serial interface which we will use for this communication. Both PLCs are connected to each other using the wiring scheme shown in figure 14. The ModbusRTU protocol is applied on top of that for the actual packet sending. The RTU-master sends out a request, which is forwarded to all slaves that are physically connected. Only one slave device is used in this case but the system can easily be expanded if required. The Modbus request sent out from master consist of 4 parts[7]:

Adress Static slave address. Always set to 1 for our use case.

Function Code Indicates whether the slave device should read or write to a register.

Data Actual information being transmitted. Length depends on function code.

CRC Cyclic Redundancy Check. Used to verify that all data was received correctly.

In the *Arduino ModbusRTU*[8] library a slave device holds a number of registers as a byte array. The master device only has access to this array. The function code is used to define the number of registers to read or write. For this system, I only use value 3 for reading of a single register and 6 for writing one.
In the Arduino *ModbusRTU* library the transmission of information works by having a copy of the slave registers on the master PLC. The master device only accesses its own copy. It then updates the respective register position when it makes a request and receives a response. The slave device sends out this response after it received a request (also called query). The response has the same structure as the request. Below is a step-by-step description of the entire Modbus communication.

1. Define Slave and Master register arrays

[7] for details please see `https://www.modbustools.com/modbus.html`, accessed on 25.4.2020
[8] taken from: `https://github.com/smarmengol/Modbus-Master-Slave-for-Arduino`, accessed on 27.8.2019

2. Initialize Modbus master and slave main objects
3. Define reverse pin lookup array in master
4. Initialize query object and set static parameters like slave adress
5. Set Baudrate and start communication
6. Use pin lookup array to get slave register index from pin number
7. Set function code, slave register index and pointer to master register array in query
8. For changing output pin state: set value in master register
9. Send query to slave
10. Update or read from slave register when query is received
11. Continuously match pin states and register values
12. Send response with updated register value to master
13. Listen for response in master and update master register

The code in listing 1 and 2 is annotated with these steps at the beginning of each line. Step 6 requires further explanation. To send out a request for a certain pin the master PLC has to know at which position that pin stores its values in the slave register array. But the only information it has, is its pin number (the numbers with blue background in figure 5). I therefore defined a lookup array which links the pin number and the index of the respective pin in the slave register array. This array is used in reverse, which might appear counter intuitive at the first glance. The pin number is the index of the lookup array, whereas the positions in the slave register array are its elements. This is done for performance reasons. Accessing one element from an array by indexing is faster than looking for a certain element and returning its index.

A simplified version of the code for the slave PLC is shown below.

Listing 1: Shortend slave PLC logic. The numbers at the beginning of some lines correspond to steps in the list above.

```
1  (1)    uint16_t ModbusSlaveRegisters[4];
2
3  (2)    Modbus ControllinoModbusSlave(SlaveModbusAdd, RS485Serial, 0);
4
5         void setup() {
6           pinMode(CONTROLLINO_A0, INPUT);
7           pinMode(CONTROLLINO_A1, INPUT);
8           pinMode(CONTROLLINO_D0, OUTPUT);
9           pinMode(CONTROLLINO_D1, OUTPUT);
10
11 (5)       ControllinoModbusSlave.begin(19200);
12        }
13
14        void loop() {
```

```
(10)      ControllinoModbusSlave.poll(ModbusSlaveRegisters, 4);

(11)      ModbusSlaveRegisters[0] = digitalRead(CONTROLLINO_A0);
(11)      ModbusSlaveRegisters[1] = digitalRead(CONTROLLINO_A1);
(11)       digitalWrite(CONTROLLINO_D0, ModbusSlaveRegisters[2]);
(11)       digitalWrite(CONTROLLINO_D1, ModbusSlaveRegisters[3]);
      }
```

As we can see here, the main tasks are executed by the *poll()* method from the *Modbus* class. It listens for any requests, updates the slave register values if necessary and sends back a response.

The code for the master PLC is a bit more involved. In the following, only parts of the program, which directly refer to the modbus communication, are shown. The entire program can be found in section B.1.

Listing 2: Parts of the master PLC program that refer to the modbus communication. The numbers at the beginning of some lines correspond to steps in the list above.

```
(1)   uint16_t ModbusSlaveRegisters[52];
(2)   Modbus ControllinoModbusMaster(0, 3, 0);
(3)   int slavePinLookup[69] = {-1, 16, 17, 18, ... };
(4)   modbus_t query;
(4)   query.u8id = 1;
(4)   query.u16CoilsNo = 1;

      void setup() {
(5)     ControllinoModbusMaster.begin(19200);
        ControllinoModbusMaster.setTimeOut(5000);
      }

      // Inputs class method
      int readInput(int inputNumber){
        if (pin is in slave) {
(6)       int registerIndex = slavePinLookup[mpin.pinNumber-1];
          if ( registerIndex == -1)
            // log lookup error and return

(7)       query.u8fct = 3;
(7)       query.u16RegAdd = registerIndex;
(7)       query.au16reg = ModbusSlaveRegisters + registerIndex ;
(9)       ControllinoModbusMaster.query(query);

          while (communication not idle and timeout not reached)
(13)        ControllinoModbusMaster.poll ();
          return ModbusSlaveRegisters[ registerIndex ];
        }
      }

      // Outputs class method
      void writeOutput(int outputNumber, byte outputState) {
        if (pin is in slave) {
(6)       int registerIndex = slavePinLookup[mpin.pinNumber - 1];
```

```
35                  if ( registerIndex == -1)
36                      // log lookup error and return
37
38  (7)             query.u8fct = 6;
39  (7)             query.u16RegAdd = registerIndex;
40  (8)             ModbusSlaveRegisters[ registerIndex ] = outputState;
41  (7)             query.au16reg = ModbusSlaveRegisters + registerIndex;
42  (9)             ControllinoModbusMaster.query(query);
43
44                  while(communication not idle and timeout not reached)
45  (13)                ControllinoModbusMaster.poll ();
46              }
47          }
```

There is a downside to sending a request for each slave pin individually as it is realized in this project. The sending and receiving takes some time. *readInput()* is also called most frequently in the entire program. Effectively, this means it takes between one and two seconds to register a sensor event from the slave PLC. However, this is not an issue because no sensors which require a response within that time frame are utilized.

4.1.11. TCP Communication

The PLC needs to communicate with the Raspberry Pi. This is important for logging and notifying the operators. The Controllino Mega has an Ethernet connection, which we use for that purpose.

TCP Protocol We employ the TCP/IP protocol stack to send data packets over the Ethernet connection. It has a client/server architecture. In this project, the PLC is the client that sends information to the server, which is the Raspberry Pi. The main design goal of this system is to always prioritize the PLC's normal working mode and to never interrupt it by any other operation. It is not ideal if a message is not transmitted but it would be disastrous if waiting for a response blocks the actual handling of the power outputs. That is why the PLC never expects a response from the server.
To use this protocol, the arduino library *Ethernet*[9] is imported into the program. The enumeration below describes the normal flow of this communication.

1. Initialize Ethernet client object.
2. Define IP and MAC addresses.
3. Initialize Ethernet object with addresses.
4. Establish connection with server over defined port.
5. Try to reconnect if connection failed after a predefined time span.
6. Server waits for any connection to accept.
7. Server continuously listens for data.
8. Check for an End-of-Line (EOL) character in received data as end of transmission.

[9] taken from: `https://github.com/arduino-libraries/Ethernet`, accessed on 23.4.2019

9. If none are found append next packet to previous one until an EOL character is present.

10. Log finished transmission.

This method ensures that the PLC runs normally even if the Raspberry Pi shuts down. The server can also be rebooted without restarting the PLC because it continuously rechecks for a new connection if it lost the original one.

The client side implementation is fairly short since it does not have to listen for any return messages. Definitions of IP, MAC addresses and EthernetClient object are done at the beginning of the script. Step 3 and 4 are carried out in *setup()*. Step 5 is realized as a *Machine* class method with the name *checkConnection()*, which is called inside *loop()*.

```
1 void checkConnection() {
2     if (! client.connected() && millis() - startTimeReconnect >= waitReconnect) {
3         startTimeReconnect = millis();
4         client.stop();
5         client.connect(server, 4000);
6     }
7 }
```

waitReconnect is the duration the program waits before trying to reconnect in milliseconds. The usage of *millis()* as timer has to be handled carefully. *millis())* returns an unsigned long integer value showing the milliseconds passed since the PLC was started. That means the value will overflow and move back to zero after approximately 49.7 days. This would lead to run time errors like failing to reconnect or significantly slowing down the program. This can be avoided by comparing durations instead of time-stamps. The duration *millis() - startTimeReconnect* on line 3 is always a positive integer. It can be seen as a modular arithmetic operation because both operands are unsigned values. Therefore at worst the rechecking can start to early while *millis()* rolls over.
The same approach is used in section 4.1.10 while the master PLC waits for a response from the slave PLC. It is even more important in this case that the execution is never interrupted when the timer function rolls over.

Step 6 to 10 in the above enumeration describes the server side of the TCP communication. I chose the programming language Python for the server logic because of its ease of use and wide range of available libraries. The entire script can be found in section C.1.
The server logic is encapsulated inside the *Handler* class. In the following, I will give a short overview over each of its methods.

__init__(server, port, log_file_path, sms_sender_func = None)
: This is a python-specific method. It is automatically called during instantiation of an object of the given class. Its usage here is to setup the logging logic and define a socket. The socket acts as the communication end point for the TCP protocol. The two most important member variables which are defined by this method are: *root_log* and *sock*. The former logs messages and can be used in the following: *self.root_log.info(msg)*. The latter can be bound to the Raspberry Pi's server IP address and its open port. Its task is to listen for incoming connections. The function arguments have the following meanings:

server: the ip address of the Raspberry Pi

port: the port which should be used for communication, e.g.: 4000

log_file_path: the full path of the file for all log messages,

sms_sender_func: a function which will be executed whenever a message of the type: *sensorEvent* is received. The message is also passed to that function as an argument so it can be send out via SMS. Per default an empty filler method is called here.

__call__(connection)
This is another python-specific method. When a class contains this function, their instances can be called as if they were normal functions and this method is executed instead. In this class, it starts to continuously listen for incoming messages from a specified connection. It also logs all received messages. They then get passed to the SMS sender function, which was set during *__init__()*.

bind_socket()
This function binds socket and IP address together. It retries the binding for 20 seconds, if that operation fails. This makes starting up the power unit more robust because the binding only works if the PLC is powered on. Therefore, the server has to wait until this is achieved. The function is called within *__init__()*.

get_connection()
When this method is called, the socket starts listening for any incoming connection. Once a connection is found, it accepts it and returns a connection object, which enables us to read and write to the client. Importantly, the connection is expected to stay active indefinitely. If it drops for any reason the server script has to be restarted. Normally, a server would ask its client periodically whether it is still alive. But that option could not be used here because one design goal was to never expect any messages from the server.

To actually run the server logic the *Handler* class has to be instantiated. This is done at the top-level of the *rpi_server.py* script from section C.1. One has to define the proper parameters for *__init__()*. Then the script tries to find a connection and starts listening for messages from it. All failed attempts at connecting, binding and translating a message are logged.
The entire logic is intentionally kept simple and short to minimize the error potential. But this also implies that it is not foolproof against all edge cases and exceptions. At some point during normal operation, one will likely have to access the Raspberry Pi for maintenance. That is why one should make sure to follow the instructions in section A.2 for a wireless connection.

JSON Data Format The actual data packets are sent in the form of Json documents. This makes them equally readable to humans and machines. I use the *ArduinoJson*[10] package, version 5.13.2, for this purpose. Generally, Json documents are similarly structured as dictionaries. An arbitrary amount of key value pairs can be added to the documents. The key "type" is always present in our transmissions. The server uses it to determine the type

[10] taken from `https://github.com/bblanchon/ArduinoJson`, version: 5.13.2, accessed on 13.12.2018

Type	Added Keys	Meaning
setAllOutputs	changedOut	A number of outputs have been changed. changedOut holds array of required output changes
startupError	changedIn	Sensor event during *checkAndEnableNormalOutput()* detected. changedIn gives the index of the triggered input.
sensorEvent	changedIn, totalOut	Sensor event during normal operation detected and output was changed accordingly.
error	errorMessage	Critical error in program execution. errorMessage describes details.

Table 4: Meaning of all defined message types the PLC sends to the server. "type" key is always present in transmitted json document.

to which the message belongs to. Its options are explained in table 4.

The PLC program has three top-level functions to build, manipulate and send the Json documents. The method *jsonBuild()* is used to instantiate a Json object:

```
1 const size_t capacity = 650;
2 JsonObject & jsonBuild (String type){
3     StaticJsonBuffer <capacity> jsonBuffer ;
4     JsonObject& root = jsonBuffer . createObject ();
5     root["type"] = type;
6     return root;
7 }
```

The capacity parameter on line 2 is just big enough to hold a message of type: "sensorEvent" with 72 outputs. This is the largest message the system sends out. The Json document does not know its required size when it is created. That is why, it can always hold the largest possible message and I took care to make that message as small as possible. In general, space is one of the main constraints on the Controllino Mega. Thus, if a new message type is defined, one has to make sure the Json document has enough space to hold it and that is does not exceed the PLC's capacity.
The function then creates and returns *root*. This is the Json object to which key value pairs can be added. The first pair is always the type. It is already added inside *jsonBuild()* on line 5.
The next function, named *jsonAddArray()* is used to add a byte array of arbitrary length to the Json object.

```
1 template <size_t N> void jsonAddArray(
2     JsonObject & someRoot, String name, byte (&ar)[N]) {
3     JsonArray& someArray = someRoot.createNestedArray(name);
4     for (int i=0; i<N; i++) {
5         someArray.add(ar[i ]);
6     }
7 }
```

Command	Response	Meaning
AT+CMGF=1	OK	Set human readable SMS text mode.
AT+CPMS="SM","SM","SM"	OK	Set SIM card as primary storage space.
AT+CMGS="[number]"	>	Prepare sending of SMS to phone number.
\x1a	+CMGS [Ref]	Signify end of SMS message.

Table 5: AT commands necessary for sending message as SMS using the GSM module. Response describes the expected return value if the operation was successful. Message has to be transmitted after third command.

Importantly, the function header of any top-level method has to stand on a single line. This also includes any template instructions. Otherwise, the Arduino compiler parses it incorrectly. In above listing it is separated for better visibility.
The final Json method sends the message over Ethernet to the server. It is called *jsonSend()* and takes the prepared Json document as its only parameter. After sending the actual message it also transmits an EOL character to signal the end of that data packet.
All in all, a message is sent by creating the Json document with *buildJson()*, append an arbitrary amount of key value paris to it and send it along using *sendJson()*. The used Json object have to be destroy successfully afterwards to avoid a potential memory leak. Th this end, these objects always have to exist in the scope of a function. Thus, they are destroyed once the program steps out of that function scope. This does not apply to *loop()* since its function body is never escaped.

4.1.12. GSM Communication

The main advantage of transmitting all important events the PLC encounters to the Raspberry Pi is the ability to send out SMS warnings in case of an experiment malfunction. This is achieved using the GSM module *Siemens TC35*. It utilizes a RS-232 connection, which can either be accessed via USB-RS-232-Adapter or directly with the GPIO pins of the Raspberry Pi. I favored this adapter approach a direct one because the GPIO pins operate at $3.3\,\mathrm{V}$ but the GSM module uses $5\,\mathrm{V}$.
The TC35 accepts commands via the *Hayes Command Set*. This is a set of strings which can be sent through the serial connections to instruct the GSM module. In this script, only a small subset of these commands are needed to facilitate the sending of SMS messages. These commands and their meaning are explained in table 5.
The logic for this communication is realized as a Python module. The details of which can be found in section C.2. I will give a step-by-step description of this logic below.

Done once after start up of GSM module:

1. Open serial port to GSM module and choose its timeout.
2. Set human readable SMS text mode.
3. Define the SIM card as primary storage device for SMS

Done with each SMS sending:

4. Send phone number of receiver to module and prepare for SMS sending.

5. Transmit message.

6. Send [Ctrl + Z] to indicate end of message

7. Listen for successful response

Step 5 to 8 are realized within one function. Its argument is the message to be transmitted. This method can be passed to the *Handler* class explained in section 4.1.11.

4.1.13. Software Summary

In the sections above, I described the details of implementing and testing the control logic for the power unit. Figure 15 summarizes the normal control flow for the entire program. The figure omits the TCP or RS-485 communication but focuses one the calling structure for mapping in- and outputs. Solid arrows indicate that the function calls the following function. Dotted arrows show the order in which functions are executed when they are called in the same scope. Conditions may be written on these arrows which have to be fulfilled for the execution to take place.
Taken together, the current software implementation fully meets the requirements, provided in section 4.1.1. Through testing in section 5.1.1 I could verify that continuous monitoring of all inputs is possible and the mapping of output state to changes of those inputs performs correctly. The communication with a Raspberry Pi is realized through the TCP protocol and messages are transmitted as JSON documents. This ensures a reliable way of logging all actions the power unit takes. The GSM module, which is connected to the Raspberry Pi, allows the immediate notification of the operator in case of a major malfunction of the power unit.
The control logic does have some disadvantageous, however, with the major one regarding its memory usage. The program in its current state uses almost the complete storage space the Controllino Mega offers. In particular, the static JSON object, which is created for every message, always takes up 650 Bytes of the approximately 8 KB because the program does not know how large the message is going to be beforehand. If longer messages are added to the program or memory is used in other ways, one will have to keep track of the remaining space on the PLC.
Another downside of this implementation is its speed. Since none of the responses (i.e. the shutdown of elctrical consumers) require a reaction time below one second, the program was not written with a focus on execution speed. The more inputs are checked by the PLC the slower it will run. The read out of the maximum number of 72 inputs from both Controllinos takes roughly 4 seconds. That means a response to some sensor event can require similarly up to 4 seconds. The main reason for this is the communication between master and slave PLC. As explained in section 4.1.10, each slave pin is queried individually, which is a simple yet time-inefficient solution. The combined pin query requires a more complex implementation, which exceeds the scope of this project. If the need for a higher execution speed arises in the future, I suggest to focus on improvements of the RS-485 communication.

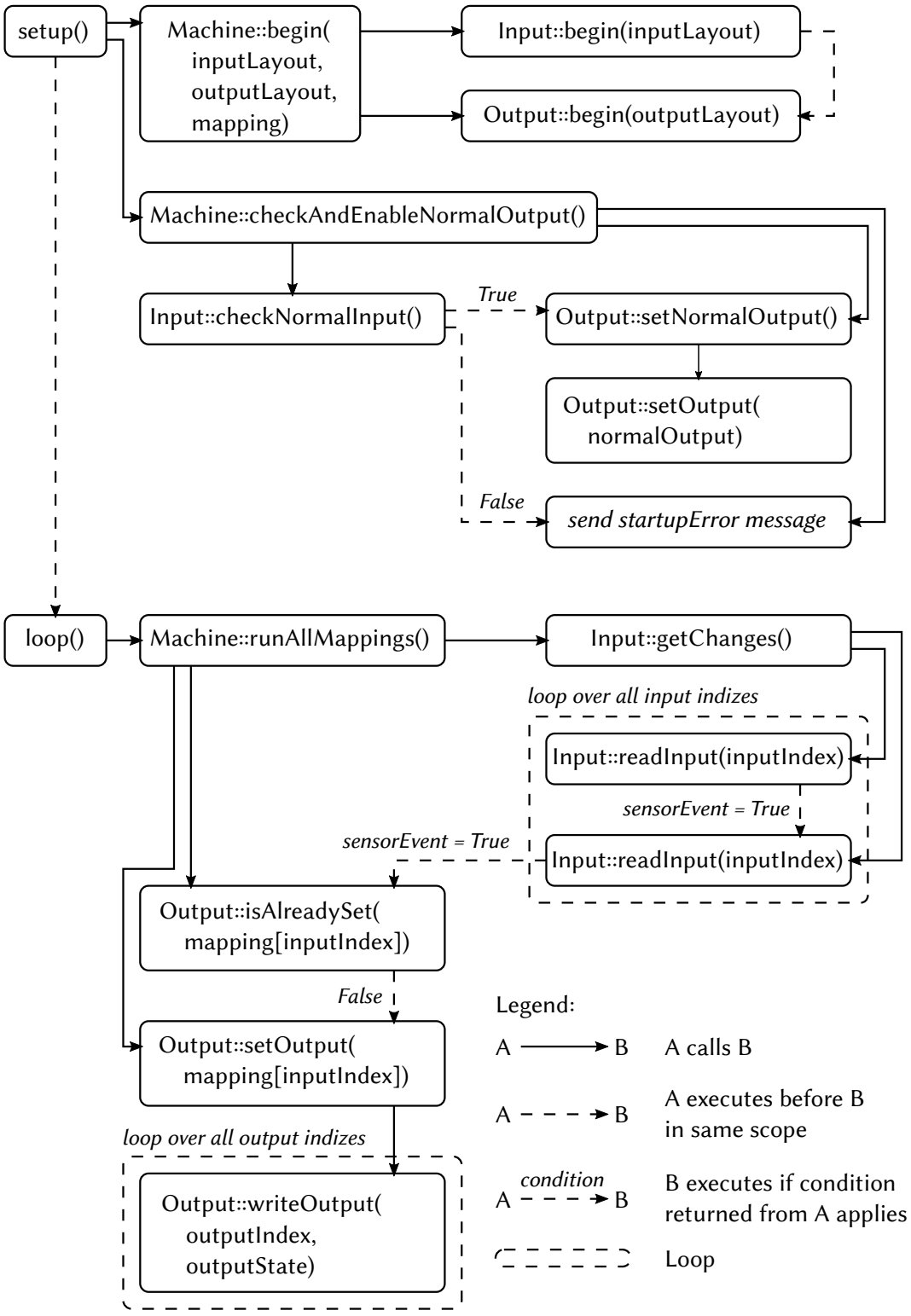


Figure 15: Call and execution structure of the mapping logic within the master PLC. Logging and communication function calls are omitted.

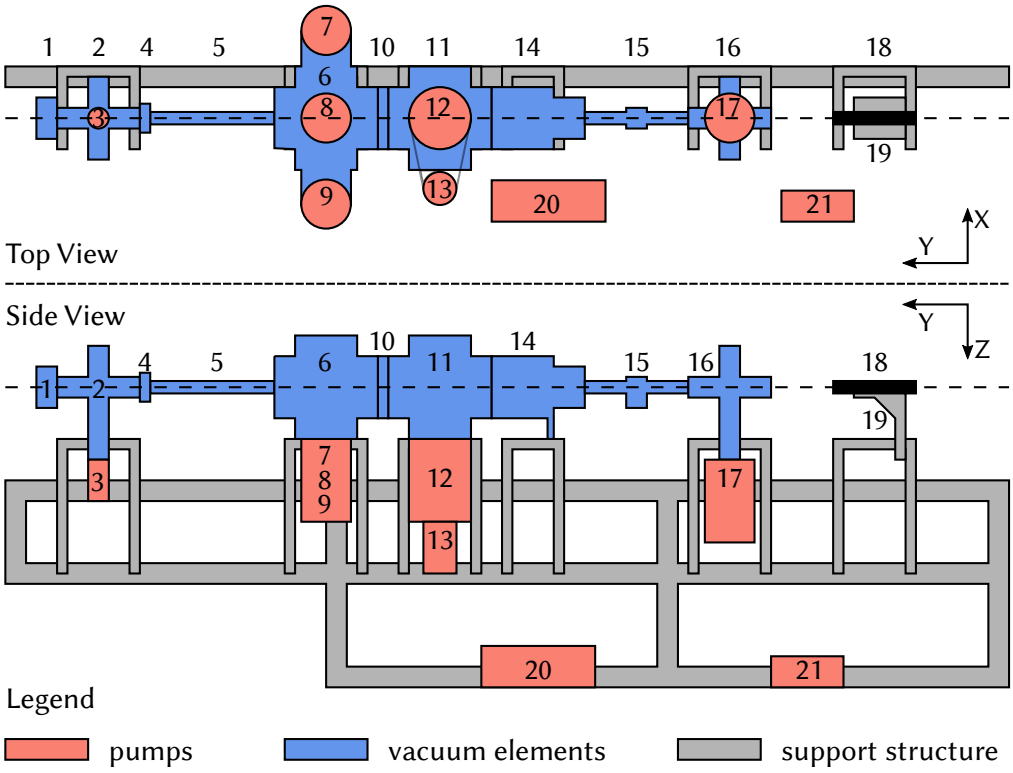


Figure 16: Time-of-Flight detector chamber. The scattering chamber connects to the left hand side. A telescope (black) is used to define and align the beam axis. Annotations can be found in table 6.

4.2. Time-of-Flight Detector Chamber

The second part of the thesis deals with the alignment and set up of the time-of-flight detector chamber. Because of its elongated appearance it will also be called ToF arm, in the following. A multitude of pumping techniques and differential pump stages are applied to make sure the lowest pressures can be reached directly at the detector.

4.2.1. General Design

In section 3.2.1, it was mentioned that the time-of-flight measurement works by calculating the speed and therefore kinetic energy of the helium particles before and after collision with the target. This calculation is realized by measuring the time the atom beam takes to travel different distances from source to detector, hence the name Time-of-Flight measurement. In this particular machine, the distance between detector and source is adjustable during operation. The entire detector chamber is therefore placed on linear rails.
Figure 16 shows an overview of the detector vacuum chamber on those rails. The annotated components are described in table 6 and explained further below.

manual shut-off valve When no measurements are running and no atom beam is present the valve will be closed off. This ensured that in case of a malfunction only the ToF arm can be shut down without affecting the other vacuum chambers.

Table 6: Time-of-Flight detector chamber components. Numbers referenced in figure 16.

Num	Description	Num	Description
1	manual shut-of valve	12	detector molecular turbo pump
2	six-way cross junction	13	fore-vacuum diffusion pump
3	molecular turbo pump	14	quadrupole mass spectrometer
4	two-way adjustable aperture	15	manual shut-off valve
5	bellow	16	beam dump cross junction
6	differential pump stage	17	beam dump diffusion pump
7	diffusion pump for back chamber	18	telescope
8	diffusion pump for middle chamber	19	adjustable mount for 18
9	diffusion pump for front chamber	20	fore-vacuum pump
10	four-way adjustable aperture	21	fore-vacuum pump
11	six-way cross junction		

cross junctions All cross junctions are used as connection points and differential pumping stages and house vacuum gauges and the detector parts.

molecular turbo pump This pump regulates the vacuum pressure in 2. A higher number of particles are scattered into this area, because of the following aperture (Num: 4).

adjustable apertures As opposed to photon beams, atom beam have an angular divergence. Therefore, Component 4 in conjunction with component 10 are used to decrease the beam cross section.

bellow Since the detector has to travel along a linear rail during operation a flexible bellow is used.

differential pump stage This component has 3 vacuum areas which are connected only through narrow apertures of fixed width. The goal is to ensure the lowest possible pressure levels directly at the detector. Each vacuum area is individually evacuated with a separate diffusion pump (Num: 7, 8, 9).

detector molecular turbo pump This pump is responsible for evacuating the detector area. It has to supply the lowest pressure levels of the entire ToF detector chamber.

fore-vacuum diffusion pump Because of the strict pressure requirements for the above turbo pump (Num: 12), its fore-vacuum is generated by a separate diffusion pump.

quadrupole mass spectrometer The core of the ToF arm. The mass spectrometer will be tuned to detect helium particles and register each particle along with its timing. The helium atoms are not stopped when they are detected.

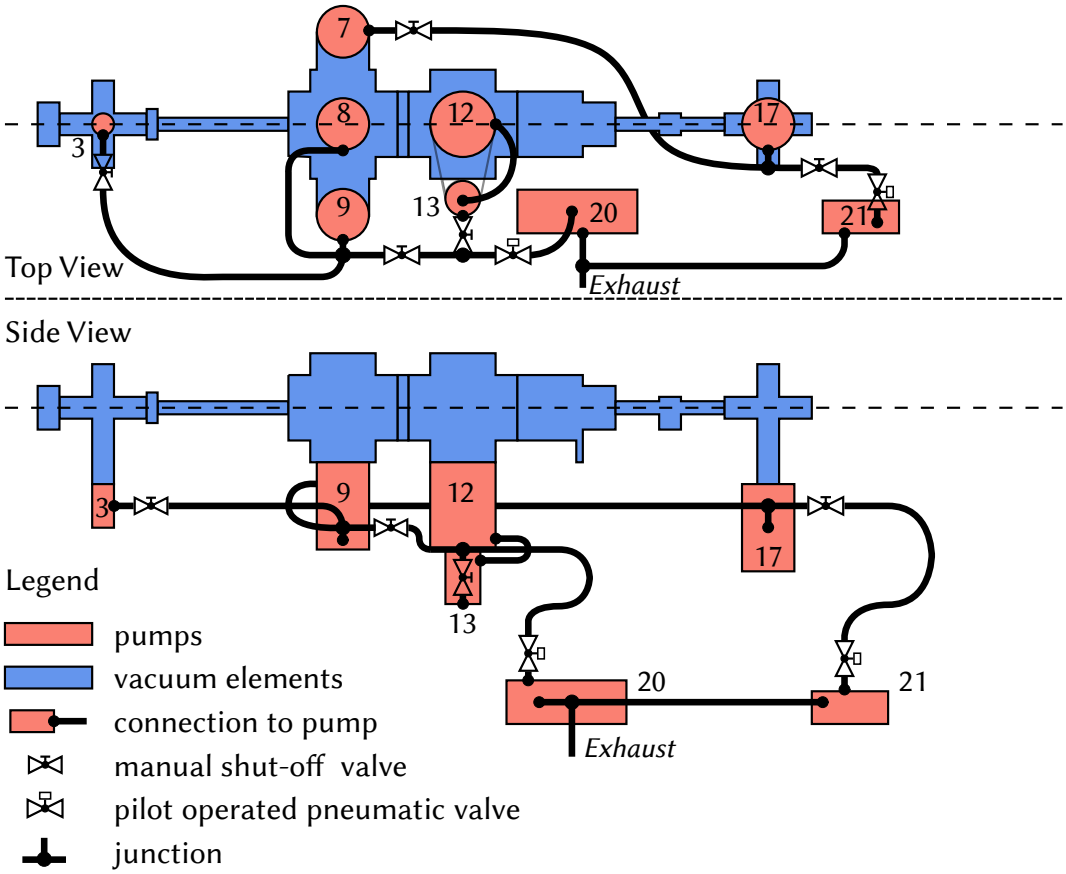


Figure 17: Fore-vacuum configuration for all ToF arm vacuum pumps. Support structure is not shown for conciseness. Annotations can be found in table 6.

beam dump diffusion pump Component 16 and 17 are used to capture and dispose the helium particle beam after the measurement.

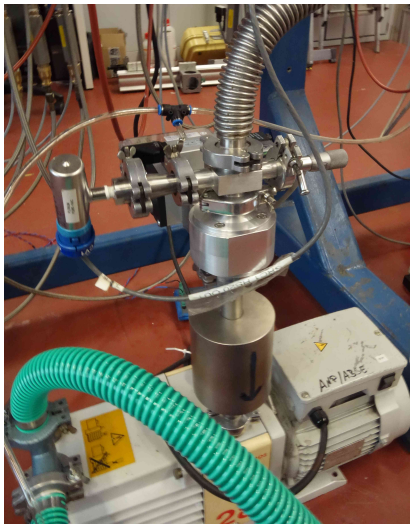
telescope The alignment of all components for the Time-of-Flight measurement is done using a telescope. Its view axis defines the beam path.

adjustable mount By changing the position and orientation of the telescope its view axis can be adjusted to be parallel with the linear rails.

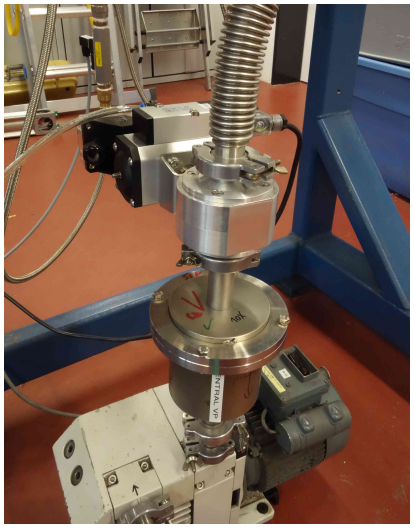
for-vacuum pumps Component 21 and 21 are connected with each UHV pump through a system of bellows. The for-vacuum has to be established before the UHV pumps can be started.

4.2.2. Fore-vacuum Implementation

All pumps necessary for the vacuum generation inside the ToF arm require a fore-vacuum to function. Two pumps (20 and 21 in figure 16) are used for that purpose. They can generate a vacuum pressure between 10^{-3} and 10^{-4} mbar. Figure 17 describes the connection layout between those two and all UHV pumps.
When choosing the connection scheme, an effort was made to ensure that two UHV pumps



(a) Pump: *Edwards E2M28* (20 of table 6)



(b) Pump: *Trivac D4B 112 45* (21 of table 6)

Figure 18: fore-vacuum pumps for ToF arm.

with significantly different mass flow rates should not be connected to the same fore-vacuum pump. Here, the mass flow rate is proportional to the throughput at constant temperature. As can be seen in figure 10, it was originally intended to connect detector pump 13 with the beam dump pump 17. However, the beam dump has to remove all remaining helium atoms which passed the detector. The mass spectrometer detector has a low ionization efficacy of approximately 10^{-6}. Therefore, the beam dump has to collect and remove almost the entire helium beam. In contrast, the detector diffusion pump should have a relatively low throughput. For this reason we decided to connect pump 17 with pump 7 instead. Diffusion pump 7 evacuates the area directly in front of aperture 10, which has with $50\,\mu\mathrm{m}$ the smallest diameter of the entire ToF arm. It can therefore be expected, that a significant portion of the helium atoms will be reflected here. Those atoms are removed by pump 7 making its throughput better comparable with that of pump 17. In addition, both pumps are electrical connected in series (*Turczyk*, 2018, p.22), since they were the only pumps, that could not be modified from a 110 V to a 230 V power system. Hence, they can only ever work in union.
All remaining UHV pumps are connected to fore-vacuum pump 20. We presume that these vacuum areas require a higher throughput because of the numerous apertures enclosed in them. In consequence pump 20 was chosen, because it has the higher pumping speed. However, since both pumps are encapsulated with shut-off valves, it will be trivial to switch them if the throughput of the UHV pumps differs by too much.

Pumps and Attachments The two pump models in use are: *Edwards E2M28* and *Trivac D4B 112 45* (figure 18a and 18b respectively). The former can reach an ultimate nominal pressure of $10^{-3}\,\mathrm{mbar}$ with a nominal pumping speed of $28\,\mathrm{m^2\,h^{-1}}$.[11] While the later can supply a ultimate nominal pressure of $10^{-4}\,\mathrm{mbar}$ but has a nominal pumping speed of only

[11] see data sheet: `https://www.idealvac.com/files/ManualsII/Edwards_E2M28_to_E2M30_Users_Instruction_Manual.pdf`, accessed on 10.5.2020

$4.8\,\mathrm{m}^2\,\mathrm{h}^{-1}$.[12] A residue oil catch connect on top of each pump. This simply consists of a cylindrical area filled with steel wool. A medium pressure gauge and a ventilation valve are attached to the oil catch of pump 20. Pump 21 is equipped with neither. Consequently, the entire ToF detector chamber is vented using the valve from pump 20. As long as both adjustable apertures are opened to their maximum size, this should not give rise to major pressure differences and thus avoids damages while venting.

Valves I affixed one pilot operated pneumatic valve above each fore-vacuum pump. They are in a normally-closed configuration. Hence, they close automatically in case of a power outage. The power unit controls both. Depending on whether the fore-vacuum pumps should continue to work during a malfunction, the valve are closed or left open to preserve the fore-vacuum pressure as much as possible. The working fluid is compressed air, which is generated by the institute's compressor. A reservoir was added as a buffer, to take over the air supply, in case the compressor fails. Its distribution system was built in cooperation with Marko Kohler (*Kohler*, 2020) and Tom Turczyk.
In addition to the pilot operated valves, I installed a number of manual shut-off valves at strategic locations (see figure 17). Some of them may seem redundant but they were included for future maintenance purposes and because they would have been left over otherwise. For example, by closing off the manual valve adjacent to pump 17, one could replace the pneumatic valve without having to vent the entire ToF arm.

Bellows and Connections The connection and distribution between the fore-vacuum pumps and all other UHV pumps was done using flexible bellows. The have the advantage to be malleable. This is important because the vacuum elements have to change their position by approximately 40 cm during operation while pump 20 and 21 are stationary. That is also the reason why the bellows leading away from them feature a pronounced S shape.
The entire fore-vacuum system is connected with KF flanges.

Pressure Gauges All UHV pumps of the ToF arm can only be started below a certain fore-vacuum pressure. For this reason, it is especially important to have a good understanding of the current pressure level. Consequently, I installed a pressure gauge of type *Granville-Phillips Convectron Vacuum Gauge Series 275*[13] directly adjacent to each UHV pump (seen on the left in blue in figure 18a). Since most of these gauges have been in use for some time, they should be inspected, calibrated and their reading compared to each other before relying on them as indicators.
The UHV vacuum can be monitored using ionization gauges.

Gauge Controllers The fore-vacuum pressure gauges connect to *Granville-Phillps* controllers, series 375B532 via RS-232 cables. Two Controllers are places inside a 19 inch rack, dedicated for sensor and turbo pump electronics. In addition, two *Delock* 4 port RS-232 multiplexer were built in *Kohler* (2020). This allows us to connect all fore-vacuum gauges simultaneously and cycle through them as needed.

[12] see data sheet: `https://www.leyboldproducts.de/media/pdf/e3/be/64/171_83_01_TRIVAC_B_DE.pdf`, accessed on 10.5.2020

[13] see datasheet: `https://www.idealvac.com/files/ManualsII/GP275_DigitalManual.pdf`, accessed on 10.5.2020

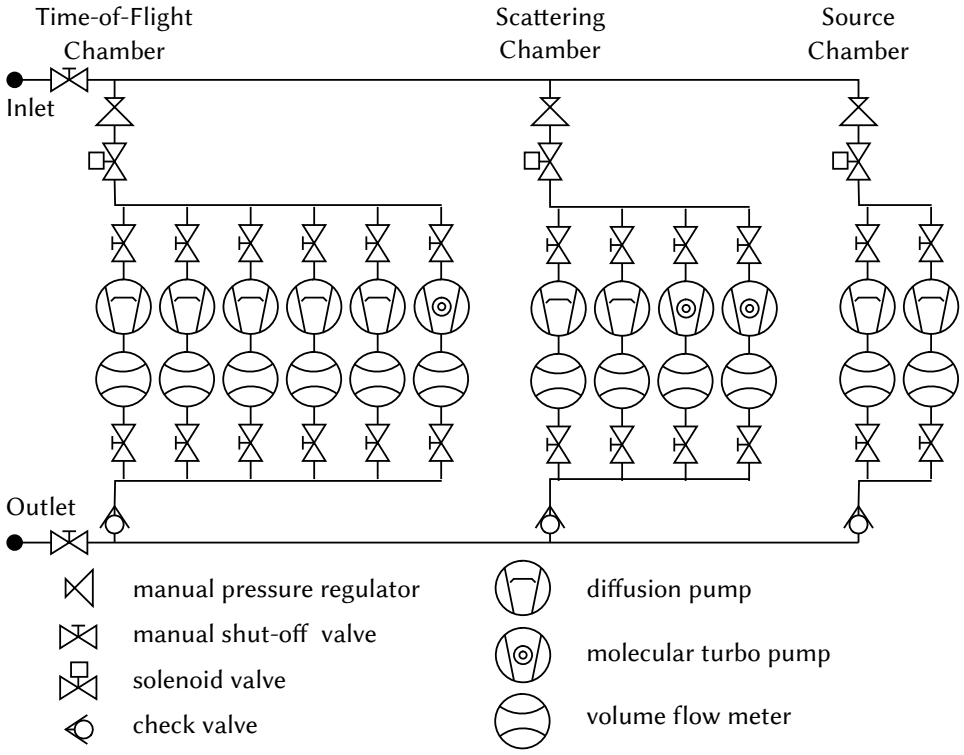


Figure 19: Schematics of the cooling system for GI-HAS experiment. Initial concept was designed by Schko Sabir as part of his master thesis but was not published there. Solenoid valves are operated by power unit. The three joint in and outlets of the chambers come down from the laboratory ceiling.

The UHV ionization gauges can be controlled by four *Granville-Phillips*, series 350 ionization gauge controllers installed in the same rack.

All in all, the fore-vacuum system as it stands is ready to use and awaits initial tests and validation of pressure levels. Due to the unfortunate nationwide lab closing in March and April of 2020, these tests could not be carried out as of yet.

4.2.3. Cooling System

Besides fore-vacuum pressure and a power connection, each UHV pump also requires sufficient cooling. This is especially important for the diffusion pumps. They rely on a cooled outer shell in their working principle and overheat quickly without it.
The initial design of the cooling system had been done by Schko Sabir as part of his master thesis but was not published there. I altered that design slightly to fit our updated requirements (figure 19) and installed it at the 3 main components of the experiment with support from Marko Kohler as part of his bachelor thesis. All but one UHV pump of the ToF arm have a dedicated cooling supply with a manual shut-off valve at the in- and outlet. The molecular turbo pump 3 from table 6 only requires a cooling fan. Figure 20 shows that cooling system. The pressure regulator, solenoid valves, and check valves are not seen here and attach near the main water inlet at the ceiling. The upper pipe is connected to the main inlet and the lower pipe attached to the main outlet. While joining the twelve in-

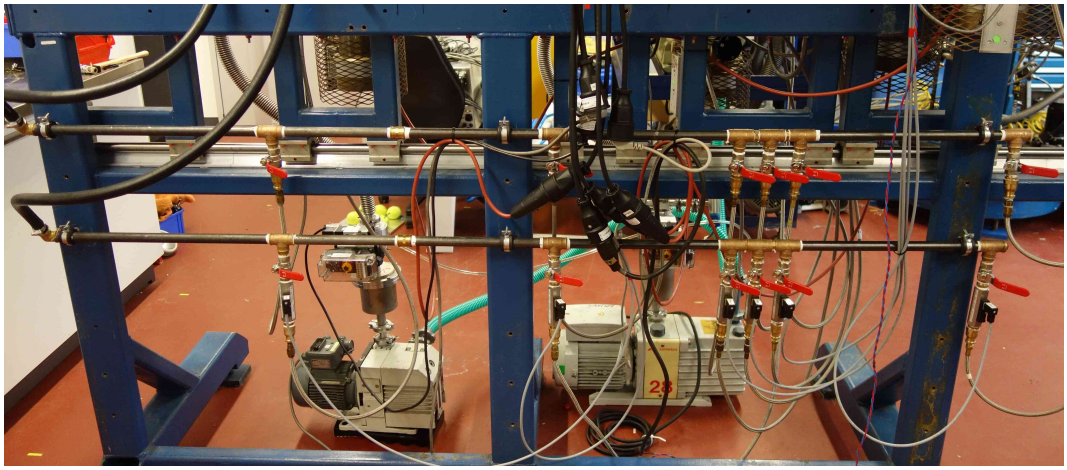


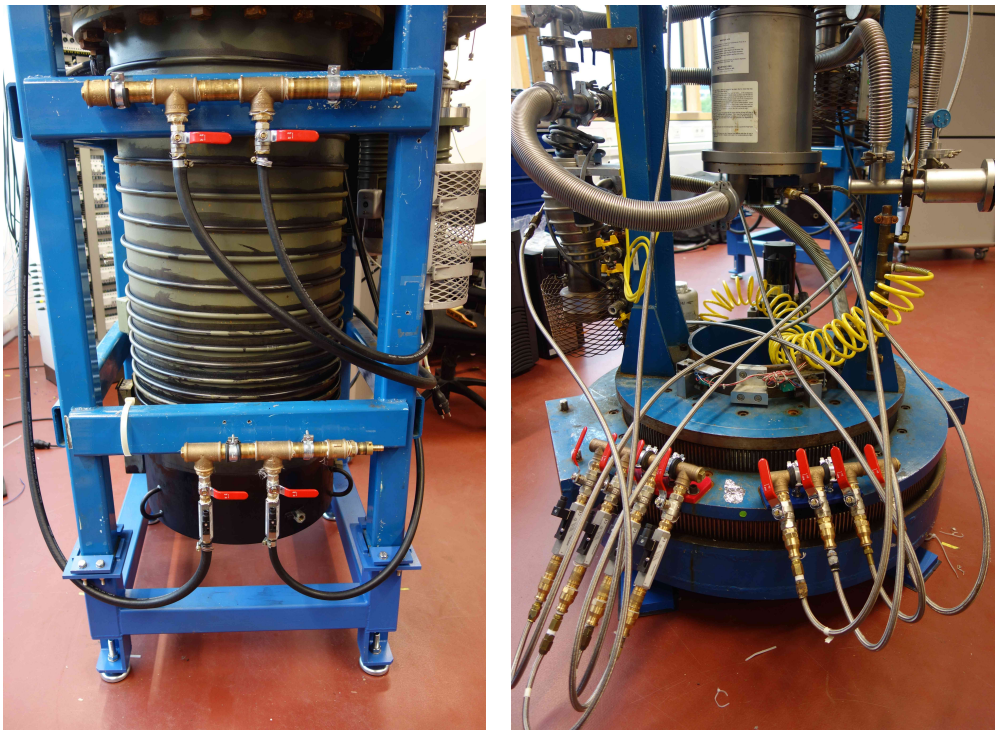
Figure 20: Cooling system for the ToF arm. Inlet is connected to the upper pipe and the outlet affixed to the lower pipe. Solenoid and check valves as well as pressure regulator are attached at the ceiling and not pictured here.

dividual in- and out-lets, I paid special attention to the pump connections. Since the entire vacuum area has to travel a significant distance on its support structure, the connections have to be both flexible and long enough. Otherwise they could entangle themselves in the support frames. In addition I attached a water flow sensor at the outlet of each pump. Their working principle has been described in section 4.1.2.

The cooling system for the other two main components of the experiment can be seen in figure 21. As with the ToF cooling system the pressure regulator, solenoid and check valves are affixed near the ceiling. As of this writing, both subsystems have not been tested because the chambers are not yet placed at their final positions and could therefore not be connected to the main inlet.

4.2.4. Alignment of Vacuum Components

The helium particle beam has to travel uninterrupted from the target to the detector. For that purpose, each vacuum element is made individually adjustable in X and Z direction (direction reference in figure 16) on the ToF arm. Before they can be aligned however, the general beam axis has to be chosen. The entire measurement principle hinges on the fact that the detector can be placed at different distances from the target during operation. For that reason all vacuum components of the ToF arm are placed on a linear rail. This includes apertures down to a diameter of $74\,\mu m$. Therefore, the chosen beam axis has to be precisely parallel to the linear rail. Otherwise the central axis of the detector would move away from the previously defined beam line when its position on the linear rail is changed. Figure 22 illustrates this issue.
As a result of this the first task is to choose an appropriate beam axis. In general, the alignment is done by placing a telescope at the end of the ToF arm and a light source at the front. The back of the beam dump cross junction has a window, which allows us to use the view axis of the telescope as our defined beam axis. This means, the telescope has to be positioned in such a way that its view axis is parallel to the motion of the detector. It also requires an appropriate X-Z-Position since the entire experiment, including source



(a) (b)

Figure 21: Cooling system for source chamber (a) and scattering chamber (b). System for source chamber was built in ?. The main connections for in- and outlets are not attached.

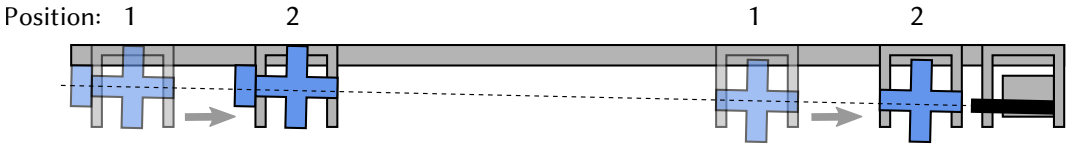


Figure 22: Illustration of alignment issue. The chosen beam line has to be parallel to the movement of the detector. The vacuum components are aligned to the view axis of the telescope (black box) in position 1. Their central axis moves away from the previously defined beam line, if it is not parallel to the linear rail. For conciseness only the first and last vacuum component is shown. Not to scale.

Figure 23: Target for aligning telescope view axis in parallel to linear rails. Green protrusions at the middle and end are the contact points for the linear rails. The target center can be adjusted with a movable aperture seen in the lower left corner.

and scattering chamber, will be aligned to its axis.
In *Turczyk* (2018) the support structure and linear rails were put in place. The upper linear rail was adjusted to be horizontal within an uncertainty of $10\,\mu m$ and the distance between upper and lower rail set to be constant within $50\,\mu m$ (*Turczyk*, 2018, p. 18f.).
In the following sections vacuum components will be denoted with their respective number in table 6.

Definition of Beam Axis

In order to set the beam line and therefore view axis of the telescope in parallel to the linear rails, a movable target was constructed[14] (figure 23). It can be placed on the linear rails in a similar manner to the vacuum supports structures. It features an aperture with adjustable diameter which can be moved in the X-Z-plane.
The telescope itself is secured to its movable mount via clamps (see figure 25a). The goal is to find the correct orientation and position of said telescope. In figure 24 step 10 to 16 of the following description are visually represented:

Material:

A Telescope, type: *Brunson Instruments Model Number: 81-1*

B Movable target described above (figure 23)

C Spirit level

D Feeler gauge blades

Procedure:

1. Secure telescope on mount as shown in figure 25a.

[14] developed and built by Tom Turczyk, permission to publish given on 06.05.2020

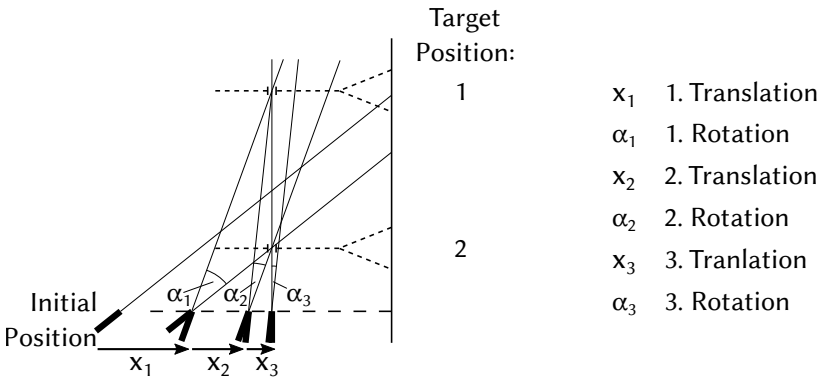


Figure 24: Representation of step 10 to 16 of alignment of the telescope axis to the detector translation rails axis. The enumeration on the right shows the necessary movement order. The translation x_3 shows the slight overshoot necessary to perfectly center the view axis in the last orientation change α_3.

2. Adjust azimuth so A is roughly in line with central axis of vacuum elements.

3. Use spirit level and feeler gauge blades to adjust telescope horizontally.
   a) Place C on top of A.
   b) Add D between front of telescope bracket and mount (see figure 25a).
   c) When set up is horizontal: Secure bracket with clamps onto mount.
   Note: Tightening the clamps leads to a slight change in inclination which has to be accounted for.

4. Make sure all vacuum elements can be sufficiently adjusted in the X-Z-plane. If not move all elements until that is the case.

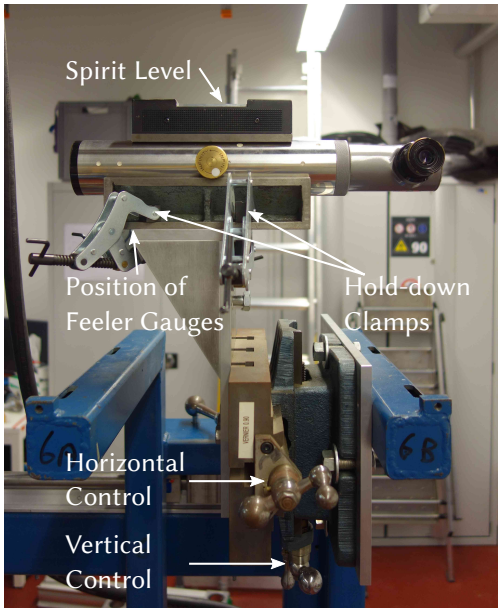
5. Open all shut-off valves of the TOo arm.

6. Set both apertures in ToF arm to the largest possible diameter.
   Note: Do not use the "window" option of aperture 10 and 4 from figure 16 as that places a glass window into the beam line which distortes the view due to light diffraction.

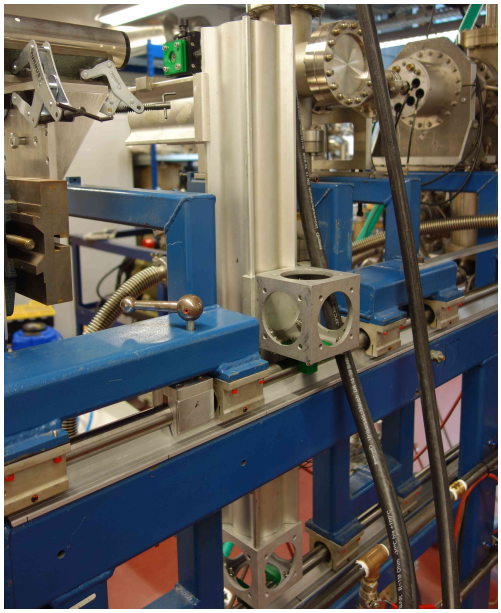
7. Make sure you can see through the entire length of the ToF arm with the telescope:
   a) Place any object in front of last vacuum element (Num: 1 in figure 16) e.g. some string or paper.
   b) Change the focal length of the telescope gradually until you can see that object.
   c) If it does not appear change X and Z position of telescope mount until it does.
   d) If not successful use the guide in subsection below to adjust the vacuum elements until it is fully visible.

8. Place B as close as possible to the telescope as shown in figure 25b (in the following called position 1).

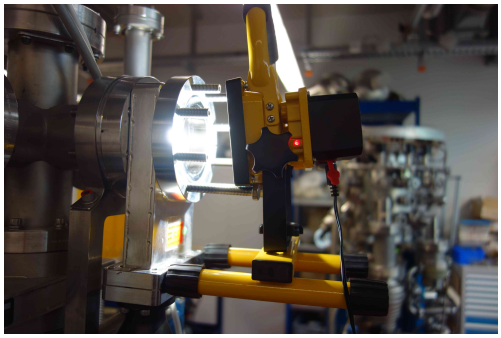
9. Adjust diameter of aperture so it aligns with circular marks in telescope.



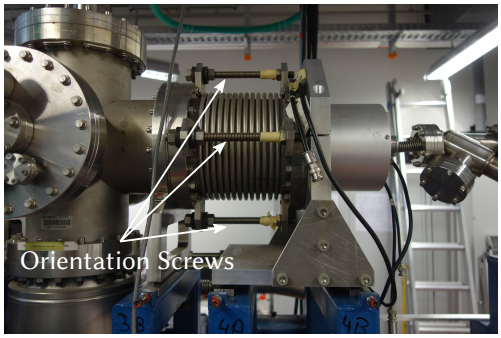
(a) Telescope mount and positioning



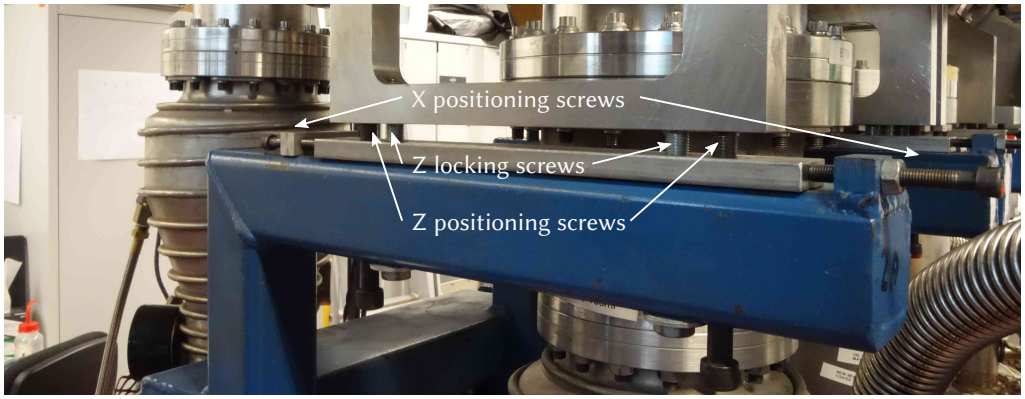
(b) Alignment target in position 1



(c) Placement of light source in front of opened valve (component 1).



(d) Detector with 4 orientation screws. Placed point symmetrical around view axis.



(e) Z and X positioning screws. Each frame has two adjustable mount points.

Figure 25: Illustrations of equipment placement for aligning beam axis and centering vacuum elements.

10. Adjust the X and Z position of the target's aperture until it is in line with the view axis of A.
    Important: This will define the height of the entire beam line so make sure the source and scattering chambers can be positioned at that height as well.

11. Place B furthest away from A (in the following called position 2).

12. Change focal length until you can clearly see B again.

13. Change orientation of A until target is again centered on view axis:
    Note: Inclination should be fairly close to optimal. Focus on azimuth first.
    a) Lightly decrease pressure of hold down clamps of A.
    b) Use a mallet to lightly tab the back of telescope bracket for a precise adjustment to the azimuth.
    c) If precision is not high enough balance tightness of clamps and strength of tabs.
    d) If vacuum elements obstruct view on target move all elements until it is fully visible using steps described in the subsection below.
    Note: No fine tuning is required here. The target should only be visible and not precisely centered.
    e) In case inclination is not yet optimal change pressure of hold down clamps for adjustment.
    f) If no centering can be archieved this way change number of feeler gauge blades beneath telescope bracket and restart from step 13a.
    Note: Keeping the view axis horizontal is not important anymore and was only used to have an initial inclination.

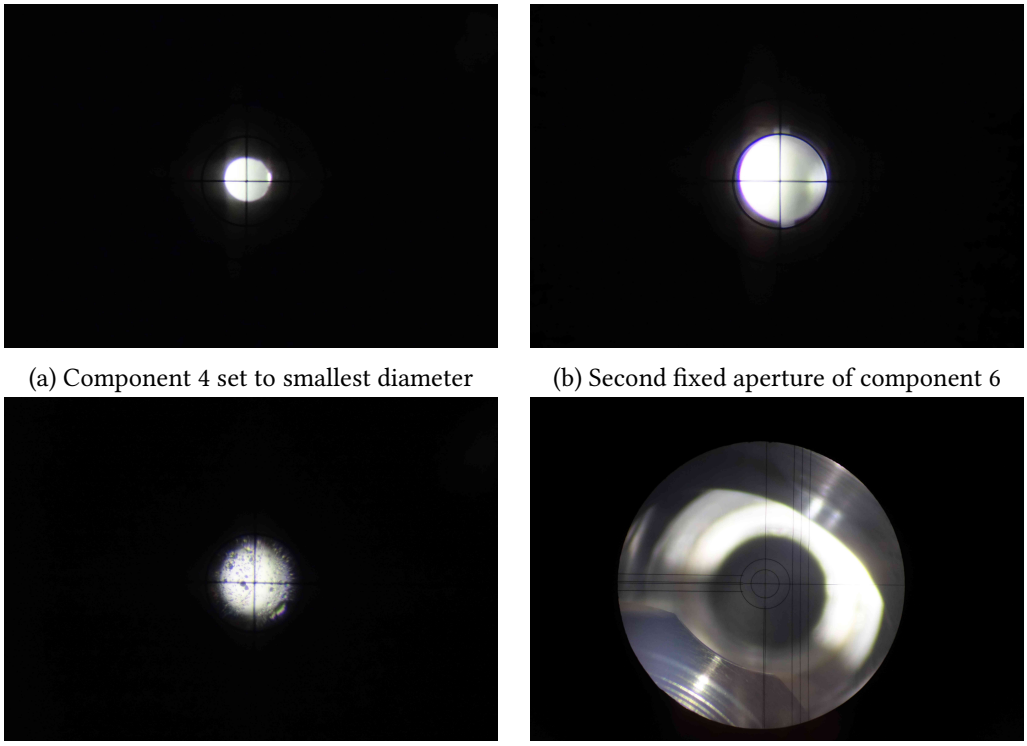
14. Reset B to position 1 and refocus A to B.

15. Adjust position of A until view axis is again centered on target.

16. Jump back to step 11 and repeat the process until the target is centered in both positions without adjustments. A representation of this process can be seen in figure 24
    Note: For a perfect alignment, it is necessary to "overshoot" the transversal position change of the last repetition slightly. The last movement x_3 in figure 24 illustrates this. However, it can also cause an overcompensation into the opposite direction.

Alignment of Vacuum Components

After setting the view axis parallel to the linear rails and to the appropriate X and Z position, all components of the ToF arm have to be centered onto that axis. Each of the six support frames shown in figure 16 have positioning screws for the X-Z-pane (labeled in figure 25e). Component 5 and 15 from figure 16 both have flexible bellows. In contrast component 6, 10, 11 and 14 are rigidly connected via CF flanges. It follows that these four elements, which are positioned on three different support frames, have to be aligned together. The other two can be aligned separately. The smallest and therefore most crucial apertures of the three rigidly connected components are two openings inside the differential pump stage and the adjustable aperture (component 10). Those three apertures define



(a) Component 4 set to smallest diameter

(b) Second fixed aperture of component 6

(c) Component 10 in "window" configuration

(d) Component 15 illuminated from back side

Figure 26: Views through the telescope with focal length adjusted to each element in view.

the orientation and position of the detector components. Since the opening diameter of component 10 is adjustable, the first focus is to align the differential pump stage with the view axis.

Material:

A Telescope, type *Brunson Instruments 81-1*, aligned via description in subsection 4.2.4

B Light source

C ToF arm, in a state in which procedure from above subsection was just completed

Procedure:

1. Place B in front of opening of component 1 as shown in figure 25c.
2. Switch component 10 to "window" option
3. Starting from the shortest focal length of A adjust focus until you make out the outlines of the second fixed aperture of the differential pump stage.
   Note: It has two fixed apertures in short succession. Those have to be centered first.
   a) Increase the focal length until the window of component 10 is sharply focused (see figure 26c).
      Note: It is easily recognizable because vacuum oil is deposited on it.

   b) Slowly increase focal length.
   c) After the first fixed aperture comes in and out of focus, stop when rounded edge of second aperture is sharply visible (see figure 26b).

4. Loosen all securing screws from all vacuum components.
5. Use front and rear positioning screws of component 6 for X direction, shown in figure 25e, to center second fixed aperture laterally.
   Note: Make sure all other lateral positioning screws of component 6, 11 and 14 are loose initially. Lightly tighten X direction screws of component 11 and 14 as well to follow movement of component 6.
6. Center second aperture longitudinally onto view axis. Use all positioning screws of component 6, 11 and 14 somewhat simultaneously.
7. Move focus back to first fixed aperture.
8. Laterally center that aperture.
   a) Try to use X direction screws of component 11 and 14 more than of component 6 to minimize misalignment of second fixed aperture.
   b) Periodically check alignment of both apertures while doing this.
9. Longitudinally center first aperture. Minimize usage of front X direction screws of component 6 while doing so.
10. Decrease adjustable aperture diameter of component 10 by one step.
11. Decrease focal length of A to focus aperture of component 10.
12. Repeat alignment of first lateral and then longitudinal position while making sure not to misalign the previous apertures.
    Note: Component 6, 11 and 14 together have 12 positioning screws for the X and Z direction each. At this point every adjustment has to be done using all 12 positioning screws in unison. This is necessary to avoid misaligning one aperture while centering the other. Due to the number of screws with varying reachability working on this procedure with two people saves a lot of time.
13. Lower aperture diameter of component 10.
14. Repeat step 12 until smallest diameter is reached.
15. Decrease focal length of A to focus inner elements of component 14.
    Note: Those reach into component 11 but are not attached to it.
16. Use orientation screws shown in figure 25d to center detector elements.
17. Follow movement of detector with front and back positioning screws to distribute loads evenly on support frames.
18. Verify by decreasing the vocal length further that the view axis has sufficient distance from all vacuum walls.

19. Use positioning screws on component 16 to increase distance from view axis where necessary. Perfect alignment is not needed here.
    Note: Closing the shut-off valve of component 15 by half provides a good visual reference for all edges found in the adjacent vacuum area (see figure 26d).

20. Focus aperture of component 4.

21. Close aperture to smalles possible diameter (see figure 26a).

22. Center aperture to view axis using the positioning screws of component 2.

With that the ToF detector chamber should be fully aligned with the linear rails and all components to each other. When the telescope is now moved in the Y direction no aperture should make any apparent movement away from the view axis of the telescope.

5. Testing and Current State

In this section, the implemented components from section 4 will be tested. In addition, an overview of their completed state will be presented.

5.1. Power Management

5.1.1. Test Bench Results

The software described in the previous sections was examined using the test bench described in section 4.1.4. A detailed description on how to use the test bench is provided in section A.3.
Various mapping configurations have been tested during development of the control logic. In particular, mappings which rely on the communication between master and slave PLC have been evaluated on their reliability. As of this writing, no mapping variation could be found which led to a test failing or any unexpected behavior.
The final evaluated mapping is explained in table 7. It is intended for the initial vacuum testing of the flight detector chamber. The table also describes the potential consequences of each sensor event and the general response of the power unit.
The underlying idea of the specific responses to the two types of sensors is evident: In case of the water flow sensor, the pump which lost cooling has to be disabled to avoid irreparable damage. That leads to pollution build up inside the affected pump. This contamination would be spread into the adjacent vacuum chamber by the other connected pumps. Therefore, all pumps attached to the affected vacuum area always have to shut down in unison.
If a ground water sensor triggers an alarm, one can expect a significant leak in the cooling water distribution. This requires the main water valve to be disabled. Because of that and the aforementioned reason, all pumps which lost cooling and are affixed to the same vacuum area have to be shut down. However, the fore-vacuum pumps should stay online in an effort to minimize the contamination and heating of the entire vacuum chamber.
After successfully testing this response configuration on the test bench, it was subsequently programmed onto the finished power unit using the procedure described in section A.1.

All in all, the current implementation of the PLC logic passes all tests without errors. This shows that the program can reliably be employed to monitor and saveguard the experiment.
The main strength of this test bench, however, is its possibility for continuous testing. Whenever a new mapping is implemented into the PLCs, its correct behavior can be tested beforehand. Also if new features are added to the program one can easily test if the original mappings still work as intended. For each new iteration of the program new tests can be appended to evaluate the new features. At the same time, one can be sure the old behavior still works as long as the respective tests still pass.
A potential improvement of the testing procedure is an automatic logging of the actual state of the PLCs' pins. It relies in its current state on the accuracy of the reported output via the TCP connection. The user can additionally check the actual state by visual inspection. For each output pin an LED is built into the top cover of both PLC, which lights up as soon as $24\,\mathrm{V}$ is applied to the corresponding pin. When a test runs one can check if the

Table 7: Mapping configuration for initial vacuum testing of detector chamber. Temperature sensors were not yet activated.

Sensor	Input	Consequences	Response
H_2O Flow Sensor 1	B1	Break down of the cooling water flow leads to overheating and damage to the pumps and contamination of the entire vacuum by evaporating oil used in diffusion pump.	Shut down all pumps connected to same vacuum as affected pump and close valves adjacent to pre pumps.
H_2O Flow Sensor 2	B2		
H_2O Flow Sensor 3	B3		
H_2O Flow Sensor 4	B4		
H_2O Flow Sensor 5	B5		
H_2O Flow Sensor 6	B6		
Ground H_2O Sensor 1	B7	Significant amounts of water on floor lead to obvious dangers for electrical equipment and personnel.	Shut off main cooling water valves. Power down pumps which lost cooling. Keep prepumps running and their valves open.
Ground H_2O Sensor 2	B8		

expected output LEDs light up. Yet, this is clearly not a fail-save method of testing. That is why extra care should be taken whenever the logic responsible for setting outputs states inside the *Outputs* class is altered.

5.1.2. Overview of Finished Power Unit

Control Logic Figure 27 and 28 show the complete implementation of the power unit with its main components annotated. In the following, I will shortly go through each element in the order they are used when an error is detected.
Sensors are connected via a two-phase cable with the sockets shown in figure 28a. If, by any reason, the sensor opens its relay, the applied voltage on the input socket drops to 0 V. Subsequently, the manual control annotated with 1 in figure 27 is used to decide whether the signal should be overwritten. If the automatic mode is chosen, the signal is transmitted to the input pin of component 8, the PLCs. They are powered by the uninterruptible low voltage source, denoted with 7, which in turn is connected to the 230 V to 24 V converter, with index 6. The master PLC queries the slave PLC and uses its predefined mapping to decide if any outputs have to be turned off. At the same time, it notifies component 9, the Raspberry Pi, about the sensor event. It in turn logs that message and prompts the GSM module, labeled with 10, to send the notification to a predefined cell phone number. Both the Raspberry Pi and the GSM module are powered by element 5, which is the 24 V to 5 V converter. If the master PLC decides certain outputs have to be shut down, it pulls their respective pins to low. The output controls in component 3 and 4 can be used to overwrite that decision. Next, the relays and contactors in 11 and 12 are opened depending on the outputs that are chosen to be disabled. This leads the sockets on the back side, shown in figure 28b, to loose connection to a power source, which in the end turns off their respective electrical consumers.

Figure 27: Complete view of power unit with main components annotated. 1: input overwrite switches, 2: uncontrolled output switches, 3: general output switches, 4: fixed output switches, 5: 24 V to 5 V converter, 6: 230 V to 24 V converter, 7: uninterruptible low voltage source, 8: master and slave PLC, 9: Raspberry Pi, 10: GSM module, 11: single phase high voltage breakers and relays, 12: three-phase high voltage breakers and relays. The high voltage site was designed and build in *Paknejad* (2017).

(a) Input sensor sockets. (b) Power sockets.

Figure 28: Side and back panel of the power unit.

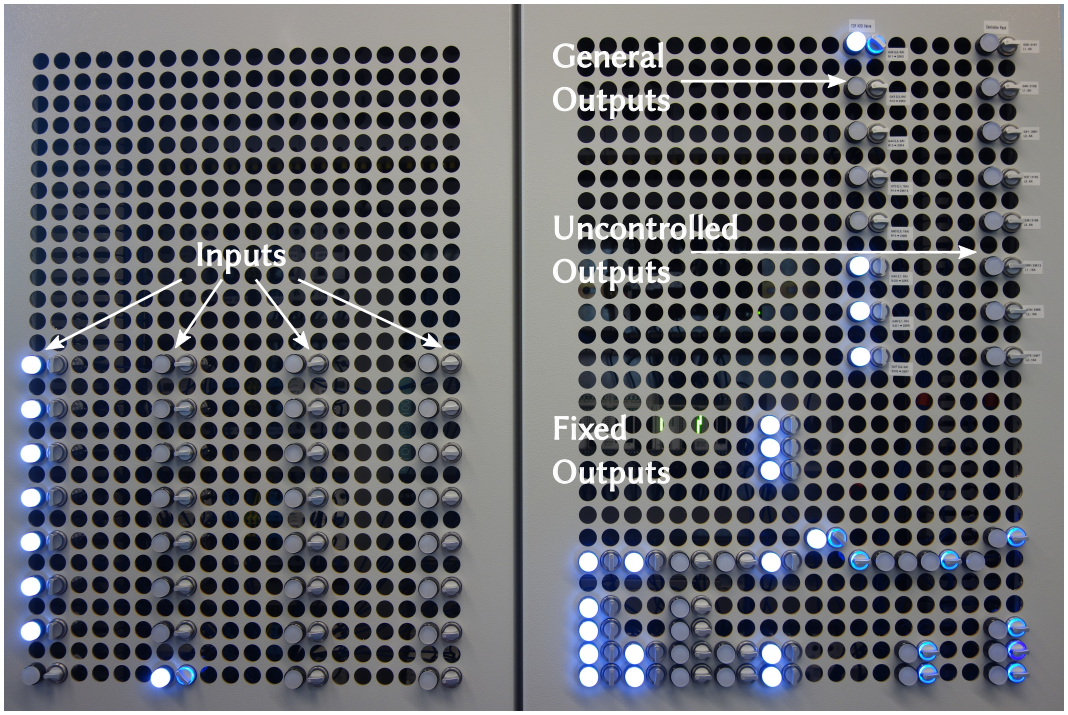


Figure 29: Front cover of the power unit. Each rotary switch has an internal blue LED indicating the state if automatic mode was chosen. White external LED shows the actual state. Horizontal position means: off, diagonal: automatic, vertical: on.

Manual Control The finished manual control possibilities are shown in figure 29. All input controls are placed on the left side. The inputs are consecutively numbered by column. It starts with B1 at the top left, continues to B8 at the bottom left and finishes with B32 at the bottom right.
On the other side of the front cover, three groups of output controls can be seen. "Fixed Outputs" are the controls for all main electrical consumers of the experiment. All of them can be autonomously managed by the PLC. They are ordered according to the layout from figure 10. "General Outputs" describe 8 control switches which have an automatic setting but are not assigned to any specific consumer. They are intended for smaller devices, which have to be monitored nevertheless, such as various water and vacuum valves. "Uncontrolled Outputs" represent the last group of controllers which do not have an "automatic" mode. They are not connected to the PLCs and can only be manually controlled. Their intended purpose is for control and measurement equipment, which should not be depended on any sensory input of the power unit.
One might notice that the total number of outputs is not 80 but only 47. The unaccounted outputs are intended for the planned combination of ToF GI-HAS and transverse ABSE experiments. Since the exact layout (and implementation) of the new components are not determined yet, it was reasonable to omit these output switches at the current state of the experiment.

Usage The practical application of this power unit is intended to be as straightforward as possible. Sensors can be plugged into any input socket. In principle, an electrical consumer which has to be monitored could be connected to any output from table 1 as well. However, the manual switches were designed and set up with particular electrical consumers in mind. Therefore, it makes sense to adhere to the intended use cases described in table 1. The suggested configuration and start up of the power unit is done as follows:

1. Shut down power unit via main connector located on the wall behind it.
2. Connect any desired sensors and outputs.
3. Program master PLC using the guide provided in section A.1.
4. For immediate start up set input overwrite switches according to their expected input states.
5. Switch on power circuit breakers of all used sockets.
6. Switch on central circuit breaker (15F1) for low voltage components.
7. Switch on main power connector of power unit.
8. Verify that all sensors show normal values.
9. Switch on outputs in an appropriate order.
10. After startup of the entire experiment switch inputs and outputs over to automatic control.

5.2. Time-of-Flight Detector Chamber

The second part of the thesis dealt with the configuration of one of the main vacuum components of the experiment. Multiple support systems for the ToF arm were build: a fore-vacuum system, cooling water distribution, pressurized air distribution, as well as various sensors and its controls. Lastly, an appropriate beam line was defined and all vacuum elements were aligned to that axis.

5.2.1. Estimation of Cooling Power

The multitude of diffusion pumps employed in the experiment require an extensive cooling system, which I described in section 4.2.3. Now, the cooling power available to the ToF arm will be approximated to make sure the system is capable of dissipating the generated heat.

The water flow rate Q for each UHV pump of the ToF arm was individually estimated to validate no obstructions exist within their cooling channels. This was done by measuring the weight of the water flown through one pump within a certain time frame. A scale with an accuracy of $\approx 2\ \%$ was used for this.[15] The water running from the pump outlet was funneled into a container placed onto the scale. The chosen time frame was $3\,\mathrm{min}$ and was measured with a manual stop watch. The water density is taken to be $1\ Kg/m^3$.

[15] see datasheet: `fig:water_tof`, accessed on 05.05.2020

Table 8: Estimate of water flow rate through ToF pumps. Measurement time was 3 min. Pump 1 refers to pump connected to leftmost cooling valves in figure 20 and pump 6 to rightmost. The water pressure changed between measurements and was therefore recorded. Accuracy of both P and Q is estimated to be ± 10 %. Sources of error are: human reaction time, inaccurate pressure gauge with analog scale, time delay between closing valve and actual stop of water flow.

	Pump 1	Pump 2	Pump 3	Pump 4	Pump 5	Pump 6
$P\ [bar]$	3.9	3.6	3.8	3.6	3.6	3.6
$Q\ [\frac{l}{min}]$	3.5	2.9	2.8	3.2	3.4	3.0
$Q_{2bar}\ [\frac{l}{min}]$	1.8	1.6	1.5	1.8	1.9	1.7

All in all, this is a rough estimation of the general cooling water flow and should not be taken as a precise measurement. The intention of this measurement is to understand whether the cooling system exceeds its requirements by a margin of at least 2. This is necessary because the institute-wide water pressure can fluctuate. These fluctuation ΔP should not pose any risks to the experiments due to insufficient water flow and a large margin of error should be incorporated into these results. Therefore, a rough estimation of the cooling power is adequate for this use case.
Table 8 summarizes the results. A contributing factor to the accuracy of these results, apart from the scale precision, is that the water flow cannot be started and stopped immediately but with a time delay of up to 3 seconds. I will forgo a precise error propagation at this point and estimate the combined error from scale accuracy and time delay for ± 5 %. This is warranted, since ΔP has a much larger effect on the accuracy than all other error sources.
If we assume laminar flow then Q is proportional depended on P. Taking this into account, the values were normalized for $P = 2\ \mathrm{bar}$, which is roughly the pressure that the institute wide cooling water system can supply. This can lead to issues with the flow sensors, because then $\overline{Q} \approx 1.7\ \frac{\mathrm{l}}{\mathrm{min}}$, which is on the lowest end of detectable values.
In fact, their nominal operating range has a minimum of $2\ \frac{\mathrm{l}}{\mathrm{min}}$.[16] However, the sensor threshold can be adjusted to be lower than that and in consequence they can still be used as intended. But this fact should be kept in mind if, for what ever reason, the water pressure drops even lower.

Using the measured flow rate, we can also put an upper limit on the cooling power the system can provide. I assume the water is at room temperature initially and should not heat up above $50\,^{\circ}\mathrm{C}$. Since the water pipes run unprotected across the experiment, higher temperatures would be dangerous to equipment and personnel. The specific heat c of water is $4.187 \frac{\mathrm{kJ}}{\mathrm{kg\,K}}$. The power P which could be dissipated by the cooling water within those bounds is thereby:

$$P = Q \cdot c \cdot \Delta T = \frac{1.7\,\mathrm{kg}}{60\,\mathrm{s}} \cdot \frac{4.187\,\mathrm{kJ}}{\mathrm{kg\,K}} \cdot 25\,\mathrm{K} \approx 2.97\ \left[\frac{\mathrm{kJ}}{\mathrm{s}} = \mathrm{kW}\right]$$

[16] see datasheet: `https://pkp.de/images/produkte/pdf_dat/ds52-d.pdf`, accessed at 25.4.2020

Turczyk (2018, p. 22) states that the biggest diffusion pumps (type *Edwards 100*) on the ToF arm require $0.65\,\mathrm{kW}$ to operate. Obviously far less then the $2.97\,\mathrm{kW}$ calculated above. Even accounting for the fact that the heat flux within the pumps is probably the limiting factor, the power circuit breakers will have already engaged before any kind of over heating would be possible.
However, the biggest diffusion pumps of the experiment which are attached to the source chamber have a nominal power consumption of $11\,\mathrm{kW}$ (*Turczyk*, 2018, p. 22). Their cooling pipes have a larger diameter and thus Q will be higher. Yet one can expect that the cooling water will heat up significantly more in these pumps. That is why, their water flow rate should be individually measured before any long duration tests are conducted. This will validate whether the system meets the cooling requirements for the source chamber as well.

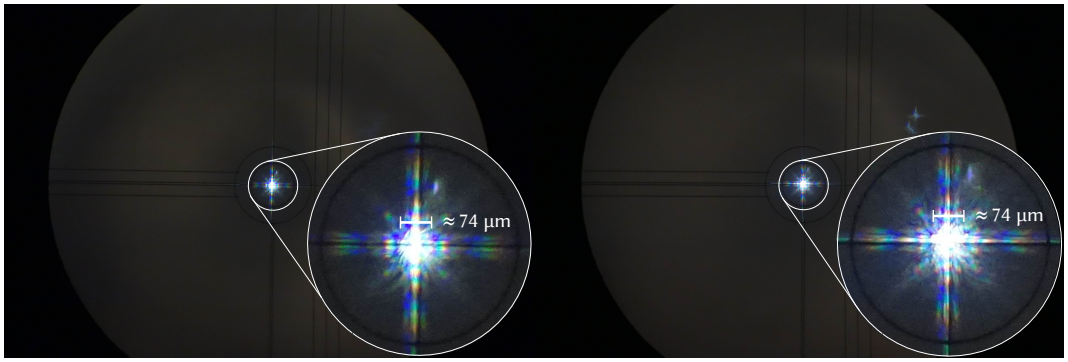
Focusing again on the ToF arm, we can also ask what the maximum temperature change of its cooling system will be:

$$\Delta T = \frac{P}{Q \cdot e} = \frac{0.65\,\mathrm{kW}}{\frac{1.7\,\mathrm{kg}}{60\,\mathrm{s}} \cdot \frac{4.187\ \mathrm{kJ}}{\mathrm{kg\,K}}} \approx 5.5\ \mathrm{K}$$

We can therefore expect the heating to be no higher than $5.5\,\mathrm{K}$. As of this writing, the temperature sensors are not yet installed on the cooling pipes. Once they are however, this value in addition to an uncertainty of approximately $\pm 15\%$ can be used as a threshold for their sensitivity. Sources of errors are: measurement accuracy of Q, actual power consumption of pumps and precision of the employed PT100 temperature gauges. Of course, tests with the sensors attached will have to verify this, because changes in the actual room temperature may also have an effect.
All in all, I expect the time-of-flight chamber to be sufficiently cooled and ready for its first vacuum tests.

5.2.2. Alignment Result

In section 4.2.4, the alignment of the beam axis and subsequently of the vacuum components with respect to the beam axis was described. Figure 30 shows the result of that alignment.
What can be seen is a view through the telescope focused on aperture 10 (from table 6) set to a diameter of $74\,\mu\mathrm{m}$. This is the smallest opening of the ToF arm and therefore well suited to validate the alignment. A camera was placed flat against the view port of the telescope with its maximum zoom level chosen and the same settings configured for both pictures.
Figure 30a shows a view with the detector positioned at a maximum distance from the scattering chamber. Whereas, figure 30b shows the detector placed as close to the scattering chamber as possible. In both figures, an enlarged version of the central area is presented in the lower right-hand corner. The path difference between the two positions was approximately $32\,\mathrm{cm}$.
The exact position of the telescope center can not be seen due to over saturation of the camera. However, from the cross hair we can determine, that the center lies within the aperture area in both detector positions. The helium beam has a minimum diameter larger than the aperture diameter. Consequently, the aperture will be fully covered by the helium



(a) Detector placed at maximum distance from scattering chamber.

(b) Detector placed at minimum distance to scattering chamber.

Figure 30: View through telescope focused on smallest aperture (component 10), set to a diameter of $74\,\mu\mathrm{m}$, at different positions of ToF detector. Path difference between positions is approximately $32\,\mathrm{cm}$. Lower right corner shows an enlargement of the center. As a reference, the approximate size of the aperture has been inscribed. The center of the view axis shows no recognizable drift in respect to the aperture.

beam irregardless of detector position, as long as the helium source chamber is correctly aligned with the view axis. We can thus summarize, the alignment was successful and a helium atom packet will be detectable irregardless of the motion of the mass spectrometer in Y direction.

5.2.3. Overview of the Completed ToF Arm

Figure 31 shows the finished ToF Arm. The components are annotated in the same manner as in figure 16. In the following, I will give an overview the central components in the order in which they interact with the helium atom beam.
If a measurement is conducted, valve 1 is opened. Otherwise it is left shut to separate and protect the vacuum areas. Aperture 4 is used to remove any cross scattered particles. It has 2 aperture positions: handle in: 3.175 mm, hadle out: glass window. To evacuated that area the molecular turbo pump 3 is attached. Since the aperture has a relatively large diameter compared to the helium beam, I do not expect a significant throughput at this point. Consequently, the smallest pump of the entire ToF arm is used here.
Bellow 5 is necessary to linear movement the detector. Together with the travel of a bellow attached to the scattering chamber, a path difference of approximately $40\,\mathrm{cm}$ is possible.
Next, the helium atoms travel through the differential pump stage 6. Its goal is to ensure a high quality vacuum at the detector. For the same reason aperture 10 can be adjusted to the smallest diameter of the entire experiment. It has 4 configurations: 1: $9.53\,\mathrm{mm}$ (handle fully in), 2: $74\,\mu\mathrm{m}$, 3: $3.175\,\mathrm{mm}$, 4: glass window.
The beam then reaches the quadrupole mass spectrometer (14). This has to be a flow-through detector as otherwise the background levels of helium within the detector area would be too high. Lastly, that makes it necessary to incorporate a beam dump (17) at the end of the beam line to collect and remove the measured helium atom packets.
Through the entire UHV area there are multiple ionization pressure gauges attached (com-

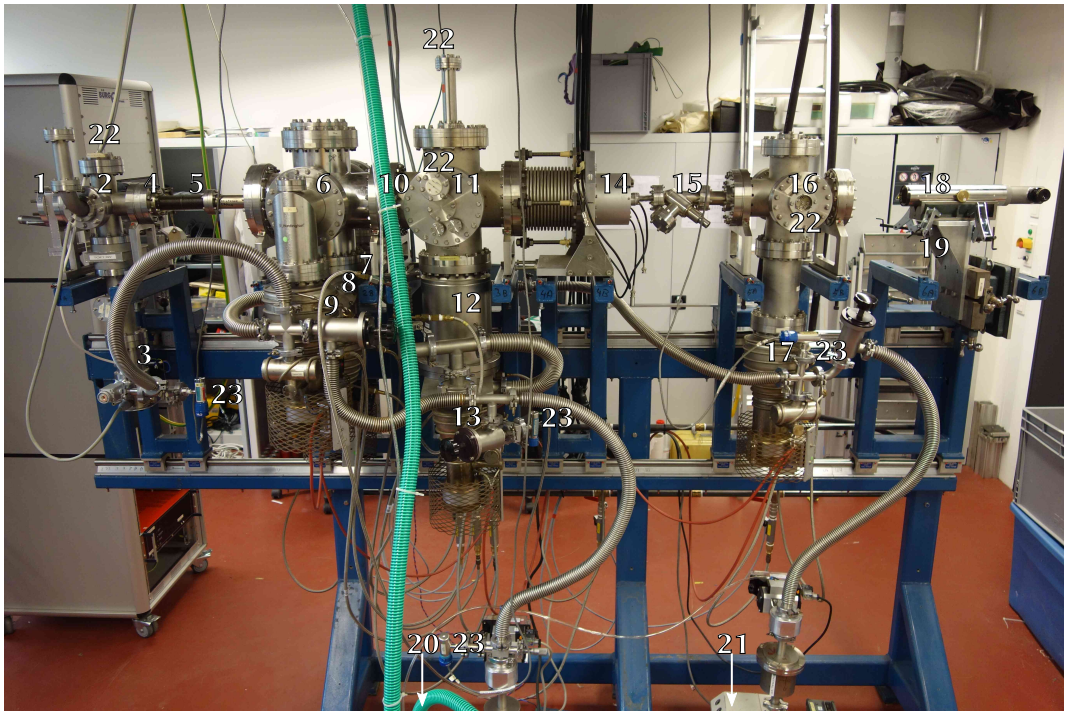
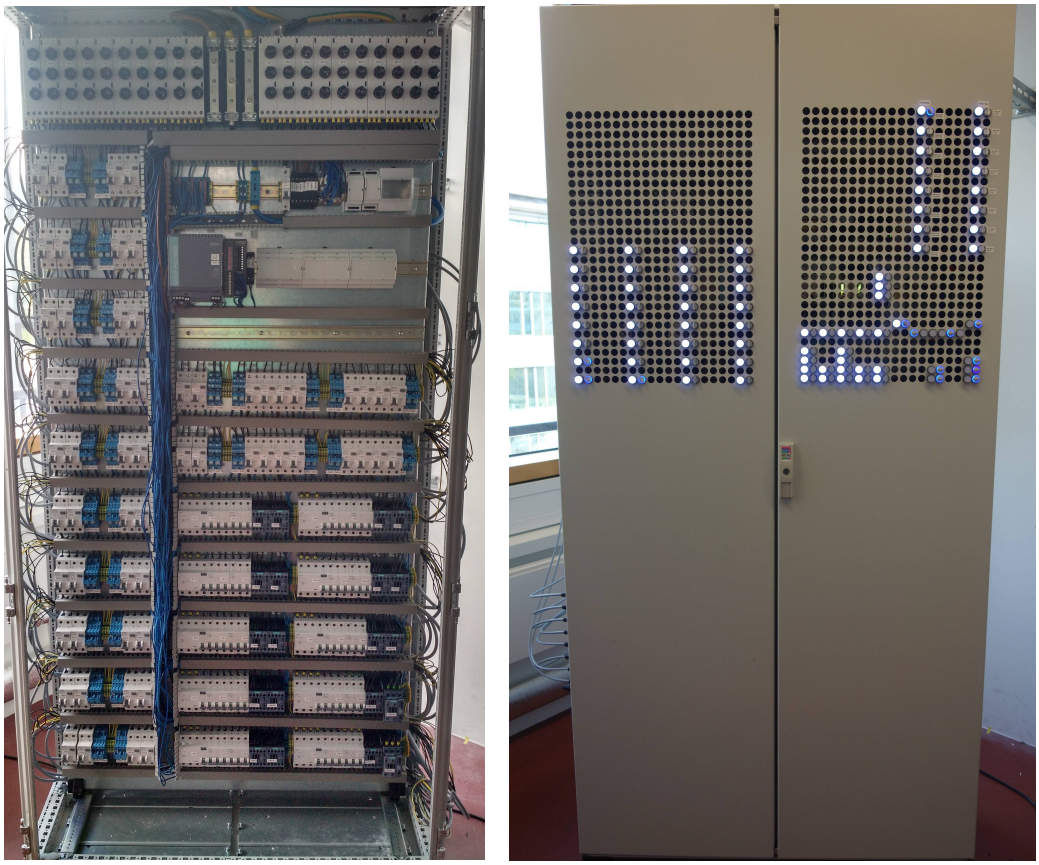


Figure 31: time-of-flight detector chamber in its completed state. Enumerations refer to table 6. In addition, label 22 annotates ionization pressure gauges and label 23 indicates fore-vacuum pressure gauges.

ponents 22). All of those have to be handles with care since they only work below pressure levels of approximately 10^{-3} mbar. In addition sensors for measuring the fore-vacuum pressure (23), the cooling water flow rate and ground water detectors (both on backside of support frame) are installed and connected to the power unit for monitoring.

As it stands the time-of-flight detector chamber is completed, has all periphery systems connected and awaits its first vacuum tests.

6. Summary



(a) Initial state at the start of this thesis.

(b) Final state at the end of this thesis.

Figure 32: Power control unit for the GI-HAS experiment. Initial state (a) was built in *Paknejad* (2017). Addition of outer shell, connection to main power supply, integration of control hardware and override switches, set up of input sensors, and programming of control logic was done to arrive at the finished power control unit (b).

In this thesis, I reported on the progress of two different parts of the HAS experiment. In the first part, the development of a power management system for the GI-HAS experiment was presented. I designed and implemented an automated control system, which enables the continuous, secure operation and monitoring of an ultra high vacuum experiment with minimal operator interaction. The comparison in figure 32 portrays the progress that was made during this process. Using the initial design for the high voltage, high current part from *Paknejad* (2017) as a starting point (figure 32a), the development concluded with the finished power distribution and monitoring system (figure 32b). The control hardware, along with the outer shell of the power unit, including the required state sensors, was built and installed as described in section 4.1.2. The main challenge was the development of a reliable and above all user-friendly control software. That was achieved using a test-driven design process on which I reported in section 4.1.5. For this purpose, I created and employed a test bench for the development of each software component. It

(a) Initial state at the start of this thesis.

(b) Result of this thesis.

Figure 33: Time-of-flight detector chamber of the GI-HAS experiment. Figure 33a taken from *Turczyk* (2018, p. 16). The addition of: pump cooling system, fore-vacuum distribution, compressed air reservoir, water flow, ground water and pressure sensors, controller for all pumps and sensors, along with the definition of a central beam axis, and alignment of detector components led to the completed ToF arm (b).

features an extendable design, so it may be used in the future to validate different input-output-mappings or new control systems prior to deployment on the power unit.
Considering the power management implementation, all initial requirements, as outlined in section 4.1, have been fulfilled. I demonstrated a robust automatic mapping of sensory inputs to power outputs. All input and output sockets can be overruled manually as described in section 4.1.3. These override switches as well as the entire software have a modular design. In the future, this will be beneficial for the transverse spin echo experiment.
Finally, the use of a logging server and GSM module allows for continuous tracking of the state of the experiment and for immediate notifications in case of an emergency. This will ensure a faster response to malfunctions and will help us finding the cause of it more readily.

The second part of this thesis focused on the set up and alignment of the time-of-flight detector chamber.
Starting from the general placement of the ToF arm from *Turczyk* (2018) (figure 33a), multiple auxiliary systems were developed to ensure stable UHV conditions at the mass spectrometer. This included a fore-vacuum distribution system as reported in section 4.2.2. In addition, I implemented a cooling system for the entire GI-HAS experiment. The maximum cooling power for the detector chamber pumps was evaluated to be $\approx 3\text{kW}$, which exceeds the requirements. However, the cooling capability for the source chamber will have to be tested in the future.
A key requirement for the measurement of helium atom scattering is the accurate align-

ment of the movable mass spectrometer along the beam axis. To that end, I developed a system for positioning a central view axis parallel to the linear motion of the detector in section 4.2.4. Subsequently, all components of the ToF arm were centered with respect to that axis. In section 5.2.2, it was shown that detection of a scattered helium beam will be possible at every detector position. The developments in this part culminated in the aligned time-of-flight detector chamber with all of its periphery systems connected and ready for operation (figure 31).

7. Outlook

At the end of this thesis the setup of the ToF detector chamber is completed and ready for vacuum operation. It was my intention to conclude this thesis with actual measurement and evaluation of the vacuum levels achieved at the detector. The measured pressure values could be used to estimate the background noise expected theoretically. Unfortunately, the university-wide lab closings in March and April of 2020 made it impossible to realize these measurements. Therefore, the immediate next steps will be the testing of fore-vacuum and, subsequently, ultra high vacuum in the detector arm.

In addition, several further systems have to be installed before first time-of-flight spectra can be measured. In the following, I will describe the most important and still missing components in the order in which they will interact with the helium beam.
The source chamber has already been positioned in line with the defined beam axis. Its fore-vacuum system, along with the needed pressure and temperature sensors have yet to be installed. In addition, the helium nozzle and chopper have to be affixed at their respective positions in the source chamber. The control electronics for the chopper and the quadropole mass spectrometer detector have to be put in place, and the electronic logics connecting them have to be installed.
The scattering chamber requires a fore-vacuum system, along with new sensors and an updated compressed air distribution for operating its various vacuum shut-off valves. Furthermore, it has to be aligned with the beam axis and connected to both the source and detector chamber.
The sample manipulator is the last missing piece before HAS experiments can begin. Its purpose is to handle the precise positioning of the target surface in relation to the helium beam inside the scattering chamber. The existing manipulator has to be repaired to regain its motion in all degrees of freedom. Finally, after positioning the manipulator in the scattering chamber, the experiment is ready for initial energy calibrations of the helium beam.

In section 2, I outlined the necessary steps to realize the combination of ToF GI-HAS and transverse ABSE spectroscopy at the Center for Advanced Materials in Heidelberg. The transverse ABSE extension is actively being worked on in parallel and reported in multiple theses such as *Lang* (2019), *Willer* (2020) and *Carvalho* (2020), whereas the considerable progress towards time-of-flight GI-HAS spectroscopy was described in this thesis.

A. Usage Procedures

A.1. Programming Pin Layout and Mapping onto PLC

Material

A PC with *Arduino IDE* version 1.8.0 or later installed.

B Master PLC program from section B.1

C USB A-to-B cable

Method

Done once during initial configuration of A:

1. Power up A
2. Download or clone git repository to your preferred directory from: `https://github.com/ArnossArnossi/GIHAS_control`. Two options exist:
   a) Using git: Open Terminal, navigate to any preferred directory and enter: "`git clone https://github.com/ArnossArnossi/GIHAS_control.git`"
   b) Download zip package directly from mentioned website and extract it.
3. Open *Arduino IDE* on A.
4. Go to File → Preferences.
5. Change "Sketchbook location" to root directory of downloaded git repository.
6. Enter the following in "Additional Boards Manager URLs": "`https://raw.githubusercontent.com/CONTROLLINO-PLC/CONTROLLINO_Library/master/Boards/package_ControllinoHardware_index.json`"
7. Go to Tools → Board → Boards Manager
8. Search for "Controllino".
9. Install "CONTROLLINO Boards", version 3.1.0.
10. Open file: "power_management.ino" and click on "Verify" to check whether everything is set up correctly.
    Note: If the necessary libraries cannot be found try to manually install them again under Sketch → Include Library → Manage Libraries.

Done for every reprogramming of the master PLC:

1. Open B with *Arduino IDE* on A.
2. Disable outputs of power unit using the overwrite switches.
   Note: Unless special circumstances require otherwise the entire power unit should also be shut down.

3. Connect A to master PLC inside power unit using C.
4. Go to Tools → Board and choose "CONTROLLINO MEGA".
5. Go to Tools → Port and choose correct port.
   Note: The naming varies depending on PC but the port is usually easily recognizable.
6. Go to top-level function named *setup()* in B. The entire configuration is done within this method.
7. Setup input pin layout correctly:
   - First of three parameter is name of input. Can be freely chosen but has to start with "M:" or "S:" to denote master or slave PLC.
   - Input sockets B1 to B16 connect to master PLC input A_0 to A_15 and sockets B17 to B36 connect to slave PLC in the same manner.
   - Second parameter refers to input pin of arduino depending on chosen input socket.
   - Third parameter is expected state of that input. A sensor event is triggered if the measured state deviates from it.
8. Setup output pin layout accordingly:
   - First parameter has same meaning and conditions as for input.
   - Choose output pin of arduino as second parameter according to table 1.
   - Third parameter refers to state of output socket under normal conditions: 1 for on, 0 for off.
9. Update number of inputs and outputs with the variables MAX_INPUT_SIZE and MAX_OUTPUT_SIZE, respectively, at beginning of this script.
10. Setup mapping matrix:
    - Rows corresponds to inputs. Columns correspond to outputs. Eg: If input with index 1 in *inputLayout* registers a sensor event, the output states in row 1 from *mapping* get applied to the output pins.

      $\Longrightarrow$ Order of inputLayout has to correspond to mapping rows. Order of outputLayout has to correspond to mapping columns.
    - Permissible elements are: 0 for disable output, 1 for enable output, 2 for do not change output.
11. Click upload button on upper left corner in *Arduino IDE* to program script onto master PLC.
    Note: Slave PLC should not have to be reprogrammed unless changes to the slave logic from section B.2 were made.

A.2. Configuring Wi-fi and Ethernet Connection

Static IP addresses for the Raspberry Pi server and PLC have to be set for establishing TCP communication. Furthermore, the WLAN module should be used to gain remote access to the Raspberry Pi for maintenance and testing.

Material

A Raspberry Pi 3 Model B with Raspbian Stretch Lite installed

B Micro-B-USB power connector

C HDMI capable monitor

D USB-A connectable keyboard

Method

1. Connect C and D to A.
2. Connect A to standard power outlet using B. $\Longrightarrow$ A powers up.
3. Login to account. Default user is *pi* and default password is *raspberry*.
4. Enter: "`sudo raspi-config`"
5. Go to: Network Options $\rightarrow$ Wi-fi.
6. Enter your Wifi Name and password.
7. Go to: Interfacing Options $\rightarrow$ SSH and choose "Yes".
8. Exit config editor by choosing "Finish".
9. Enter: "`sudo vim /etc/dhcpcd.conf`" or use any text editor of your liking.
10. Append the following lines to that file and save it:

```
interface eth0
static ip_address =192.168.2.10/24
interface wlan0
static ip_address=[local free ip address e.g. 192.168.0.11/24]
static routers=[ip address of your router]
static damain_name_servers=[usually same as above line]
```

    Note: Do not use IP Address 192.168.0.10. It is already given to master PLC.

11. Make sure the chosen wlan0 IP address is available in your router and will not be assigned differently. Or assign static IP address for wlan0 inside your router.
12. Enter: "`sudo reboot now`" in console to reboot the Raspberry Pi.

A.3. Usage of Test Bench

The test bench described in section 4.1.4 should be able to test all current mapping variations for the power unit. Moreover, because of its extendable framework it will be easy to implement new test variations when the power unit gains more features. In the following a step-by-step explanation is given to run the test procedure when the current control software is used but a different mapping should be tested. However, the usage of the test bench should remain roughly the same, even if new features and therefore new tests are added.

Material

A Test bench as described in section 4.1.4

B Computer with connection to local router. In this case using unix based Fedora 31 as operating system. Any OS will be sufficent but precise commands will vary on non-unix systems.

C Test script from section C.3

D Micro-USB-B power supply

Method

1. Enable router connection of Raspberry Pi and setup LAN connection to PLC by following instructions in section A.2.

2. Connect D to Raspberry Pi $\implies$ Wait for power up

3. Gain ssh access to Raspberry Pi by entering the following in the terminal: "`ssh pi@[RPi IP address, e.g. 192.168.0.11]`"

4. With connection to internet install *pytest*. Enter: "`sudo pip3 install pytest`"

5. Open D in text editor of your liking on B. Alternatively adapt test script directly on Raspberry Pi an skip step 8 and 3.

6. Copy mapping which is to be tested from PLC control software into D. Replace thereby the parameter called *mapping* in D.

7. Make sure order of parameter: *sim_inputs* is the same as in mapping parameter. I.e.: The correst simulated inputs refer to the correct expected mapping of outputs.

8. Copy adapted test script to Raspberry Pi:

   a) Open Terminal on B

   b) Navigate to directory of test script.

   c) Enter: "`scp ./test_mapping.py pi@[RPi IP Address]: /home/pi/test_mapping.py`"

9. Copy server script: *rpi_server.py* from section C.1 to the same directory as test script

10. Access Raspberry Pi again using SSH as in step 3

11. Enter: "`pytest`" to start the testing protocol.
    Note: Always issue this command in the same directory as the test script

B. Programmable Logic Controler Scripts

The main logic of the power unit can be found in these scripts. They are written in a variation of C++. The main logic resides in a single file for the master and slave device each and is not built into a separate library. The strategy ensures a usage that is as easy as possible. The required third-party libraries are: *Controllino*, version:1.1.2, *ArduinoJson*, version: 15.3.2, *ModbusRtu*, version: initial upload, *Ethernet*, version: 2.0.0.
All files including the necessary library versions are bundled in a git repository and can be downloaded at:
`https://github.com/ArnossArnossi/GIHAS_control`.

B.1. Master PLC Script

```
#include "Controllino.h"
#include "ArduinoJson.h"
#include "ModbusRtu.h"
#include <Ethernet.h>


const int MAX_INPUT_SIZE = 16; // number of input elements
const int MAX_OUTPUT_SIZE = 10; // number of outputs elements


byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED }; // MAC address for controllino.
IPAddress ip(192, 168, 2, 5); // the static IP address of controllino
IPAddress server (192,168,2,10) ; // numeric IP of the server i.e. rpi
EthernetClient client; // start ethernet client


// array in which the digital states of all inputs and outputs of the slave
    controllino are stored
// Modbus sends and receives any data to the slave only from this array
uint16_t ModbusSlaveRegisters[52];
Modbus ControllinoModbusMaster(0, 3, 0);
modbus_t query;
// receiving (slave) adress
query.u8id = 1;
// always only read one register at a time
query.u16CoilsNo = 1;

// lookup array for index of pin in slave.
int slavePinLookup[69] = {-1, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, -1,
    -1, -1, -1, -1, -1, -1,
                          -1, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48,
                              49, 50, 51, -1, -1, -1,
                          -1, 28, 29, 30, 31, 32, 33, 34, 35, -1, -1, -1, -1, 0,
                              1, 2, 3, 4, 5, 6,
                           7, 8, 9, 10, 11, 12, 13, 14, 15
                          };


// enough for 72 bytes from output array with
```

```
36 // "changedIndex":4 and "Type": "sensorevent" but not for totalIn
37 const size_t capacity = 650;
38 JsonObject & jsonBuild (String type){
39   // the returned object should never live in the global scope
40   // or else it wont be destroyed by garbage collection => memory leak
41   // Parameter: type of json message
42   // Return: JSON Object where data can be appended to
43
44    StaticJsonBuffer <capacity> jsonBuffer ;
45   JsonObject& root = jsonBuffer . createObject () ;
46   root["type"] = type;
47   return root;
48 }
49
50
51 void jsonSend(JsonObject & root){
52   // if the JSONbuffer is to small the program hangs at printTo ()
53   root . printTo ( client ) ;
54    client . println () ;
55 }
56
57
58 template <size_t N> void jsonAddArray(JsonObject & someRoot, String name, byte (&ar)
       [N]) {
59   // Add an int array to a json object .
60   // Parameters: someRoot, json object
61   //             name, name of the array
62   //             ar, reference to in array to be stored
63   JsonArray& someArray = someRoot.createNestedArray(name);
64   for (int i=0; i<N; i++) {
65     someArray.add(ar[i]) ;
66   }
67 }
68
69
70 struct Pin {
71   String pinName;
72   byte pinNumber;
73   byte pinState ;
74 };
75
76
77 class Inputs {
78
79   private:
80     Pin inputData[MAX_INPUT_SIZE];
81     // Timeout for waiting for an answer from slave
82     int slaveInputTimeout = 1000;
83     // expected values for input. Will be set at initiation of Input
84     byte normalInput[MAX_INPUT_SIZE];
85
86
87   public:
```

```
88      // should be initialized to array of zeros.
89      byte inputRepr[MAX_INPUT_SIZE] = {};
90
91
92      Inputs() {}
93
94
95      void begin(Pin pinLayout[]) {
96        // constructor method with different name to call in setup()
97        for (int i = 0; i < MAX_INPUT_SIZE; i++) {
98          if (pinLayout[i].pinName[0] == 'M') {
99            pinMode(pinLayout[i].pinNumber, INPUT);
100         }
101         inputData[i] = pinLayout[i];
102         normalInput[i] = pinLayout[i].pinState;
103         inputRepr[i] = inputData[i].pinState;
104       }
105     }
106
107
108     int readInput(int inputNumber) {
109       // Read state of input at postion inputNumber in inputLayout,
110       // check if request has to be send via rs485 first
111       // Return -1 if error is detected.
112       Pin mpin = inputData[inputNumber];
113       if (mpin.pinName[0] == 'M') {
114         return digitalRead(inputData[inputNumber].pinNumber);
115       }
116       else if (mpin.pinName[0] == 'S') {
117         int registerIndex = slavePinLookup[mpin.pinNumber - 1];
118
119         if (registerIndex == -1) {
120           String errorMessage = F("Slave has no pin with this pin number. Ignoring
                  this input.");
121           JsonObject &root = jsonBuild(F("error"));
122           root[F("error")] = errorMessage;
123           root[F("pinNumber")] = mpin.pinNumber;
124           jsonSend(root);
125           return -1;
126         }
127
128         query.u8fct = 3;
129         query.u16RegAdd = registerIndex;
130         query.au16reg = ModbusSlaveRegisters + registerIndex;
131
132         ControllinoModbusMaster.query(query);
133
134         unsigned long startTime = millis();
135         while (ControllinoModbusMaster.getState() != COM_IDLE && millis() -
                  startTime < slaveInputTimeout) {
136           ControllinoModbusMaster.poll();
137         }
138         return ModbusSlaveRegisters[registerIndex];
```

```
139       }
140       else  {
141           String errorMessage = F("Name not starting with M or S. Ignoring this
                input");
142           JsonObject &root = jsonBuild(F("error"));
143           root[F("error")] = errorMessage;
144           root[F("pinNumber")] = mpin.pinName;
145           jsonSend(root);
146           return -1;
147       }
148     }
149
150
151     void update() {
152       // Update the states of all pins in inputData to current value.
153
154       for (int i = 0; i < MAX_INPUT_SIZE; i++){
155         inputRepr[i] = inputData[i].pinState = readInput(i);
156       }
157     }
158
159
160     int getChanges() {
161       // Check if any input state has changed to something different than the
162       // expected input. If so return the index of that input pin
163       // if input changed: wait x ms and recheck the reading. If it changed again,
            Dont do anything
164       // this allows for flipping of the switches from manual to auto without
            triggering something
165
166       // Returns: int, index of changed input pin or -1 if no input has changed.
167
168       for (int i=0; i<MAX_INPUT_SIZE; i++){
169         int tempInput = readInput(i);
170         if (inputData[i].pinState != tempInput) {
171           delay(200);
172           if (inputData[i].pinState == readInput(i)){
173             continue;
174           }
175           inputRepr[i] = inputData[i].pinState = tempInput;
176           if (tempInput!=normalInput[i]) {
177             return i;
178           }
179         }
180       }
181       return -1;
182     }
183
184
185     int checknormalInput(){
186       // Check if input is the same as normalInput defined in setup()
187       // if any input is not as expected send index of that input otherwise send -1
188
```

```
189         for (int i=0; i<MAX_INPUT_SIZE; i++) {
190           if (readInput(i) != normalInput[i]) {
191             return i;
192           }
193         }
194         return -1;
195       }
196   };
197
198
199   class Outputs {
200
201     private:
202       Pin outputData[MAX_OUTPUT_SIZE];
203       byte normalOutput[MAX_OUTPUT_SIZE];
204       // timeout length before response from slave is ignored
205       int slaveOutputTimeout = 1000;
206
207
208     public:
209       // should be initialized to array of zeros.
210       byte outputRepr[MAX_OUTPUT_SIZE] = {};
211
212
213       Outputs() {}
214
215
216       void begin(Pin pinLayout[]) {
217         // constructor method with different name to call at a later time
218         for (int i = 0; i < MAX_OUTPUT_SIZE; i++) {
219           outputData[i] = pinLayout[i];
220           normalOutput[i] = pinLayout[i].pinState;
221
222
223           outputRepr[i] = outputData[i].pinState = 0; // make sure all outputs are off
                  initially
224           if (pinLayout[i].pinName[0] == 'M') {
225             pinMode(pinLayout[i].pinNumber, OUTPUT);
226             digitalWrite(outputData[i].pinNumber, 0);
227           }
228         }
229       }
230
231
232       void writeOutput(int outputNumber, byte outputState) {
233         // Check if output on postion outputnumber in outputPinLayout
234         // is in master or slave, write data or send query accordingly
235         Pin mpin = outputData[outputNumber];
236         if (mpin.pinName[0] == 'M') {
237           digitalWrite(mpin.pinNumber, outputState);
238         }
239         else if (mpin.pinName[0] == 'S') {
240           int registerIndex = slavePinLookup[mpin.pinNumber - 1];
```

```
241
242         if ( registerIndex == -1) {
243           String errorMessage = "Slave has no pin with pin number: " + String(mpin.
                pinNumber)
244                                 + "Ignoring this query.";
245           JsonObject & root = jsonBuild("error");
246           root["error"] = errorMessage;
247           jsonSend(root);
248           return;
249         }
250
251         query.u8fct = 6;
252         query.u16RegAdd = registerIndex;
253         ModbusSlaveRegisters[registerIndex] = outputState;
254         query.au16reg = ModbusSlaveRegisters + registerIndex;
255
256         ControllinoModbusMaster.query(query);
257
258         unsigned long startTime = millis();
259         while(ControllinoModbusMaster.getState() != COM_IDLE && millis() -
              startTime < slaveOutputTimeout) {
260           ControllinoModbusMaster.poll();
261         }
262       }
263       else {
264         String errorMessage = F("Name not starting with M or S. Ignoring this input"
              );
265         JsonObject & root = jsonBuild(F("error"));
266         root[F("error")] = errorMessage;
267         root[F("pinNumber")] = mpin.pinName;
268         jsonSend(root);
269         return;
270       }
271     }
272
273
274     void setOutput(byte (&configArray)[MAX_OUTPUT_SIZE]) {
275       // Allow setting the entire output state with one array.
276       // Must have same size AND ORDER as outputPinLayout.
277       // Elements can be 0: set outputpin with same index to 0
278       //                 1: set outputpin with same index to 1
279       //                 2: dont do anything with that pin
280       // Parameters: configArray: reference to int array with
281       //                          MAX_OUTPUT_SIZE as size
282
283       for (int i = 0; i < MAX_OUTPUT_SIZE; i++) {
284         if (configArray[i] < 2) {
285           writeOutput(i, configArray[i]);
286           outputData[i].pinState = configArray[i];
287           outputRepr[i] = configArray[i];
288         }
289       }
290       JsonObject & root = jsonBuild("setAllOutputs");
```

```
291         jsonAddArray(root, "changedOut", configArray);
292         jsonSend(root);
293       }
294
295
296       void setNormalOutput() {
297         setOutput(normalOutput);
298       }
299
300
301       bool isAlreadySet(byte (&toBeChecked)[MAX_OUTPUT_SIZE]) {
302         // Check if the desired output array "toBeChecked" is allready
303         // set in the current output.
304         // Parameters: toBeChecked: ref to int array of MAX_OUTPUT_SIZE in length
305         //                          can only have elements: 0: pin set to 0
306         //                                                  1: pin set to 1
307         //                                                  2: do not check this pin
308         // Returns: false: if output is different or if only 2s are in toBeChecked
309         //                 true: if output is same at positions with 0s and 1s,
310
311         bool isSet = false;
312
313         for (int i = 0; i < MAX_OUTPUT_SIZE; i++) {
314           if (toBeChecked[i] == 2) {
315             continue;
316           }
317           if (toBeChecked[i] != outputData[i].pinState) {
318             return false;
319           } else {
320             isSet = true;
321           }
322         }
323         return isSet;
324       }
325 };
326
327 class Machine {
328
329   private:
330     Inputs input;
331     Outputs output;
332     byte mmapping[MAX_INPUT_SIZE][MAX_OUTPUT_SIZE];
333
334
335   public:
336     // timing start for rechecking of ethernet connection in milliseconds
337     unsigned long startTimeReconnect = 0;
338     // time between rechekings of ethernet connection in milliseconds
339     int waitReconnect = 20000;
340
341
342     Machine() {}
343
```

```
344
345     void begin(Pin inputPinLayout[],
346                Pin outputPinLayout[],
347                byte mapping[MAX_INPUT_SIZE][MAX_OUTPUT_SIZE]) {
348       // constructor method with different name to call in setup()
349       input.begin(inputPinLayout);
350       output.begin(outputPinLayout);
351
352       for (int i=0; i<MAX_INPUT_SIZE; i++) {
353         for (int j=0; j<MAX_OUTPUT_SIZE; j++) {
354           mmapping[i][j] = mapping[i][j];
355         }
356       }
357     }
358
359
360     void checkAndEnableNormalOutput() {
361       // Check if input is different from normalInput, i.e. the expected input
              states.
362       int index = input.checknormalInput();
363       if (index == -1) {
364         output.setNormalOutput();
365       } else {
366         JsonObject & root = jsonBuild(F("startupError"));
367         root[F("changedIn")] = index;
368         jsonSend(root);
369       }
370     }
371
372
373     void runAllMappings() {
374       // Check if input has changed. If so set output to defined output in mapping.
375       int changedIndex = input.getChanges();
376       if (changedIndex != -1) {
377         if (!output.isAlreadySet(mmapping[changedIndex])){
378           output.setOutput(mmapping[changedIndex]);
379         }
380
381         Serial.println("sensorEvent on Pin: " + changedIndex);
382         JsonObject & root = jsonBuild(F("sensorEvent"));
383         root[F("changedIn")] = changedIndex;
384         jsonAddArray(root,F("totalOut"), output.outputRepr);
385         jsonSend(root);
386       }
387     }
388
389
390     void checkConnection() {
391       // Check if ethernet is connected.
392       // If not and timer for retry has run down: try to reconnect
393
394       if (!client.connected() && millis() - startTimeReconnect >= waitReconnect) {
395         startTimeReconnect = millis();
```

```
396             Serial . println (F("Trying to reconnect ... "));
397             client . stop () ;
398             client . connect( server , 4000);
399             Serial . print (F("Connectionstatus: "));
400             Serial . println ( client . connected());
401           }
402       }
403 };
404
405
406 Machine controller ;
407
408
409 void resetWrapper(){
410   // for some reason non static member function cannot be used as interrupt routines
411    controller . checkAndEnableNormalOutput();
412 }
413
414
415 void setup () {
416
417   // Setup the whole machine. All pin configurations go here!
418
419   // The in/out-putPinLayout describes the normal (i.e. expected) input and output
          state
420   // Remember to set MAX_INPUT_SIZE and MAX_OUTPUT_SIZE at the beginning of this
421   // script correctly , i.e. length of input and output layout respectively .
422   // Pin Name has to start with M or S depending on master or slave device
423   // or pin will be ignored otherwise .
424   Pin inputPinLayout[] = {
425     {F("M:B01"), CONTROLLINO_A0, 0},
426     {F("M:B02"), CONTROLLINO_A1, 0},
427     {F("M:B03"), CONTROLLINO_A2, 0},
428     {F("M:B04"), CONTROLLINO_A3, 0},
429     {F("M:B05"), CONTROLLINO_A4, 0},
430     {F("M:B06"), CONTROLLINO_A5, 0},
431     {F("M:B07"), CONTROLLINO_A6, 0},
432     {F("M:B07"), CONTROLLINO_A7, 0},
433     {F("S:B17"), CONTROLLINO_A0, 0},
434     {F("S:B18"), CONTROLLINO_A1, 0},
435     {F("S:B19"), CONTROLLINO_A2, 0},
436     {F("S:B20"), CONTROLLINO_A3, 0},
437     {F("S:B21"), CONTROLLINO_A4, 0},
438     {F("S:B22"), CONTROLLINO_A5, 0},
439     {F("S:B23"), CONTROLLINO_A6, 0},
440     {F("S:B24"), CONTROLLINO_A7, 0},
441   };
442
443   Pin outputPinLayout[] = {
444     {F("M:G31"), CONTROLLINO_R2, 1},
445     {F("M:G67"), CONTROLLINO_R3, 1},
446     {F("M:G32"), CONTROLLINO_R4, 1},
447     {F("M:G33"), CONTROLLINO_R5, 1},
```

```
448       {F("M:G68"), CONTROLLINO_R6, 1},
449       {F("M:G34"), CONTROLLINO_R7, 1},
450       {F("M:G64"), CONTROLLINO_R8, 1},
451       {F("M:G35"), CONTROLLINO_R9, 1},
452       {F("M:G36"), CONTROLLINO_R10, 1},
453       {F("M:G42"), CONTROLLINO_R11, 1},
454     };
455
456   // Each row coresponds to input with same index in inputLayout.
457   // And order of Elements in one row has to be the same as outputLayout.
458   // Note: Remember to update MAX_INPUT_SIZE and MAX_OUTPUT_SIZE at BEGINNING
          of script
459   byte mapping[MAX_INPUT_SIZE][MAX_OUTPUT_SIZE] = {
460   // Input\Output :R02,R03,R04,R05,R06,R07,R08,R09,R10,R11
461                       {0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 2}, // CONTROLLINO_A0,
462                       {0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 2}, // CONTROLLINO_A1,
463                       {0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 2}, // CONTROLLINO_A2,
464                       {0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 2}, // CONTROLLINO_A3,
465                       {0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 2}, // CONTROLLINO_A4,
466                       {0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 2}, // CONTROLLINO_A5,
467                       {0 , 0 , 2 , 2 , 0 , 0 , 0 , 2 , 2 , 0}, // CONTROLLINO_A6,
468                       {0 , 0 , 2 , 2 , 0 , 0 , 0 , 2 , 2 , 0}, // CONTROLLINO_A7,
469                       {0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 2}, // CONTROLLINO_A0, slave
                              inputs
470                       {0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 2}, // CONTROLLINO_A1,
471                       {0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 2}, // CONTROLLINO_A2,
472                       {0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 2}, // CONTROLLINO_A3,
473                       {0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 2}, // CONTROLLINO_A4,
474                       {0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 2}, // CONTROLLINO_A5,
475                       {0 , 0 , 2 , 2 , 0 , 0 , 0 , 2 , 2 , 0}, // CONTROLLINO_A6,
476                       {0 , 0 , 2 , 2 , 0 , 0 , 0 , 2 , 2 , 0}, // CONTROLLINO_A7,
477   };
478
479
480    // interrupt routine for restarting all outputs after sensorevent has been fixed
481     const int interruptPin = CONTROLLINO_IN0;
482     pinMode(interruptPin, INPUT);
483      attachInterrupt ( digitalPinToInterrupt ( interruptPin ), resetWrapper, RISING);
484
485
486      // start Serial and check if its connected correctly
487      Serial .begin(9600) ;
488     int startTime = millis () ;
489     while(! Serial ) {
490       if ( millis () - startTime >= 5000) {
491         break;
492       }
493     }
494
495
496      // start the Ethernet connection
497     Ethernet.begin(mac, ip) ;
498      // give the Ethernet a second to initialize
```

```
499     delay(1000);
500
501
502     // setup rs485 query and connection
503     ControllinoModbusMaster.begin(19200); // baud-rate at 19200
504     ControllinoModbusMaster.setTimeOut(5000);
505
506
507     // enable connection with server (rpi) over port 4000
508     Serial.println(F("Start connecting with ethernet ..."));
509     if (client.connect(server, 4000)) {
510       //TODO: change this to log message
511       Serial.println(F("Connected."));
512     } else {
513       Serial.println(F("Connection failed."));
514       controller.startTimeReconnect = millis();
515     }
516
517
518     // initilize controller with input and output pins
519     // set inputs to expected inputs tempoarily, set outputs to 0
520     controller.begin(inputPinLayout, outputPinLayout, mapping);
521     // Wait a bit. seems to be needed to set everything up correctly
522     delay(500);
523
524     // Check if inputs do not show an error before enableing normal operation
525     controller.checkAndEnableNormalOutput();
526 }
527
528
529 void loop() {
530
531     controller.runAllMappings();
532
533     controller.checkConnection();
534
535 }
```

B.2. Slave PLC Script

```
1  #include <Controllino.h>
2  #include "ModbusRtu.h"
3
4  #define SlaveModbusAdd 1
5
6  #define RS485Serial 3
7
8  Modbus ControllinoModbusSlave(SlaveModbusAdd, RS485Serial, 0);
9
10 // Specified internal registers in the Modbus slave device.
11 // Only these particular internal registers are available for Modbus master.
12 uint16_t ModbusSlaveRegisters[52];
13
```

```
14
15 void setup()
16 {
17     Serial.begin(9600);
18
19     // Inputs (index 0 to 15)
20     pinMode(CONTROLLINO_A0, INPUT);
21     pinMode(CONTROLLINO_A1, INPUT);
22     pinMode(CONTROLLINO_A2, INPUT);
23     pinMode(CONTROLLINO_A3, INPUT);
24     pinMode(CONTROLLINO_A4, INPUT);
25     pinMode(CONTROLLINO_A5, INPUT);
26     pinMode(CONTROLLINO_A6, INPUT);
27     pinMode(CONTROLLINO_A7, INPUT);
28     pinMode(CONTROLLINO_A8, INPUT);
29     pinMode(CONTROLLINO_A9, INPUT);
30     pinMode(CONTROLLINO_A10, INPUT);
31     pinMode(CONTROLLINO_A11, INPUT);
32     pinMode(CONTROLLINO_A12, INPUT);
33     pinMode(CONTROLLINO_A13, INPUT);
34     pinMode(CONTROLLINO_A14, INPUT);
35     pinMode(CONTROLLINO_A15, INPUT);
36
37
38     // Outputs (index 16 to 51, first relay index is 36)
39     pinMode(CONTROLLINO_D0, OUTPUT);
40     pinMode(CONTROLLINO_D1, OUTPUT);
41     pinMode(CONTROLLINO_D2, OUTPUT);
42     pinMode(CONTROLLINO_D3, OUTPUT);
43     pinMode(CONTROLLINO_D4, OUTPUT);
44     pinMode(CONTROLLINO_D5, OUTPUT);
45     pinMode(CONTROLLINO_D6, OUTPUT);
46     pinMode(CONTROLLINO_D7, OUTPUT);
47     pinMode(CONTROLLINO_D8, OUTPUT);
48     pinMode(CONTROLLINO_D9, OUTPUT);
49     pinMode(CONTROLLINO_D10, OUTPUT);
50     pinMode(CONTROLLINO_D11, OUTPUT);
51     pinMode(CONTROLLINO_D12, OUTPUT);
52     pinMode(CONTROLLINO_D13, OUTPUT);
53     pinMode(CONTROLLINO_D14, OUTPUT);
54     pinMode(CONTROLLINO_D15, OUTPUT);
55     pinMode(CONTROLLINO_D16, OUTPUT);
56     pinMode(CONTROLLINO_D17, OUTPUT);
57     pinMode(CONTROLLINO_D18, OUTPUT);
58     pinMode(CONTROLLINO_D19, OUTPUT);
59     pinMode(CONTROLLINO_R0, OUTPUT);
60     pinMode(CONTROLLINO_R1, OUTPUT);
61     pinMode(CONTROLLINO_R2, OUTPUT);
62     pinMode(CONTROLLINO_R3, OUTPUT);
63     pinMode(CONTROLLINO_R4, OUTPUT);
64     pinMode(CONTROLLINO_R5, OUTPUT);
65     pinMode(CONTROLLINO_R6, OUTPUT);
66     pinMode(CONTROLLINO_R7, OUTPUT);
```

```
67      pinMode(CONTROLLINO_R8, OUTPUT);
68      pinMode(CONTROLLINO_R9, OUTPUT);
69      pinMode(CONTROLLINO_R10, OUTPUT);
70      pinMode(CONTROLLINO_R11, OUTPUT);
71      pinMode(CONTROLLINO_R12, OUTPUT);
72      pinMode(CONTROLLINO_R13, OUTPUT);
73      pinMode(CONTROLLINO_R14, OUTPUT);
74      pinMode(CONTROLLINO_R15, OUTPUT);
75
76
77      ControllinoModbusSlave.begin(19200);
78  }
79
80  void loop()
81  {
82      // checks for incomming data
83      // if received frame is ok, write or read values from ModbusSlaveRegisters
84      ControllinoModbusSlave.poll(ModbusSlaveRegisters, 52);
85
86      // Used for debugging purposes
87      for (int i = 0; i<52; i++) {
88          Serial.print(ModbusSlaveRegisters[i]);
89          Serial.print(" ");
90      }
91      Serial.println();
92
93      ModbusSlaveRegisters[0] = digitalRead(CONTROLLINO_A0);
94      ModbusSlaveRegisters[1] = digitalRead(CONTROLLINO_A1);
95      ModbusSlaveRegisters[2] = digitalRead(CONTROLLINO_A2);
96      ModbusSlaveRegisters[3] = digitalRead(CONTROLLINO_A3);
97      ModbusSlaveRegisters[4] = digitalRead(CONTROLLINO_A4);
98      ModbusSlaveRegisters[5] = digitalRead(CONTROLLINO_A5);
99      ModbusSlaveRegisters[6] = digitalRead(CONTROLLINO_A6);
100     ModbusSlaveRegisters[7] = digitalRead(CONTROLLINO_A7);
101     ModbusSlaveRegisters[8] = digitalRead(CONTROLLINO_A8);
102     ModbusSlaveRegisters[9] = digitalRead(CONTROLLINO_A9);
103     ModbusSlaveRegisters[10] = digitalRead(CONTROLLINO_A10);
104     ModbusSlaveRegisters[11] = digitalRead(CONTROLLINO_A11);
105     ModbusSlaveRegisters[12] = digitalRead(CONTROLLINO_A12);
106     ModbusSlaveRegisters[13] = digitalRead(CONTROLLINO_A13);
107     ModbusSlaveRegisters[14] = digitalRead(CONTROLLINO_A14);
108     ModbusSlaveRegisters[15] = digitalRead(CONTROLLINO_A15);
109     digitalWrite(CONTROLLINO_D0, ModbusSlaveRegisters[16]);
110     digitalWrite(CONTROLLINO_D1, ModbusSlaveRegisters[17]);
111     digitalWrite(CONTROLLINO_D2, ModbusSlaveRegisters[18]);
112     digitalWrite(CONTROLLINO_D3, ModbusSlaveRegisters[19]);
113     digitalWrite(CONTROLLINO_D4, ModbusSlaveRegisters[20]);
114     digitalWrite(CONTROLLINO_D5, ModbusSlaveRegisters[21]);
115     digitalWrite(CONTROLLINO_D6, ModbusSlaveRegisters[22]);
116     digitalWrite(CONTROLLINO_D7, ModbusSlaveRegisters[23]);
117     digitalWrite(CONTROLLINO_D8, ModbusSlaveRegisters[24]);
118     digitalWrite(CONTROLLINO_D9, ModbusSlaveRegisters[25]);
119     digitalWrite(CONTROLLINO_D10, ModbusSlaveRegisters[26]);
```

```
120         digitalWrite (CONTROLLINO_D11, ModbusSlaveRegisters[27]);
121         digitalWrite (CONTROLLINO_D12, ModbusSlaveRegisters[28]);
122         digitalWrite (CONTROLLINO_D13, ModbusSlaveRegisters[29]);
123         digitalWrite (CONTROLLINO_D14, ModbusSlaveRegisters[30]);
124         digitalWrite (CONTROLLINO_D15, ModbusSlaveRegisters[31]);
125         digitalWrite (CONTROLLINO_D16, ModbusSlaveRegisters[32]);
126         digitalWrite (CONTROLLINO_D17, ModbusSlaveRegisters[33]);
127         digitalWrite (CONTROLLINO_D18, ModbusSlaveRegisters[34]);
128         digitalWrite (CONTROLLINO_D19, ModbusSlaveRegisters[35]);
129
130         digitalWrite (CONTROLLINO_R0, ModbusSlaveRegisters[36]);
131         digitalWrite (CONTROLLINO_R1, ModbusSlaveRegisters[37]);
132         digitalWrite (CONTROLLINO_R2, ModbusSlaveRegisters[38]);
133         digitalWrite (CONTROLLINO_R3, ModbusSlaveRegisters[39]);
134         digitalWrite (CONTROLLINO_R4, ModbusSlaveRegisters[40]);
135         digitalWrite (CONTROLLINO_R5, ModbusSlaveRegisters[41]);
136         digitalWrite (CONTROLLINO_R6, ModbusSlaveRegisters[42]);
137         digitalWrite (CONTROLLINO_R7, ModbusSlaveRegisters[43]);
138         digitalWrite (CONTROLLINO_R8, ModbusSlaveRegisters[44]);
139         digitalWrite (CONTROLLINO_R9, ModbusSlaveRegisters[45]);
140         digitalWrite (CONTROLLINO_R10, ModbusSlaveRegisters[46]);
141         digitalWrite (CONTROLLINO_R11, ModbusSlaveRegisters[47]);
142         digitalWrite (CONTROLLINO_R12, ModbusSlaveRegisters[48]);
143         digitalWrite (CONTROLLINO_R13, ModbusSlaveRegisters[49]);
144         digitalWrite (CONTROLLINO_R14, ModbusSlaveRegisters[50]);
145         digitalWrite (CONTROLLINO_R15, ModbusSlaveRegisters[51]);
146 }
```

C. Raspberry Pi Server Scripts

The server logic has to be run using Python version 3.0 or above. C.1 and C.2 have to reside in the same directory.

C.1. TCP Server Script

Python Script for Raspberry Pi server named: rpi_server.py.

```
1  import json
2  import socket
3  import sys
4  import time
5  import logging as lo
6  from gsm_handler import GSM
7
8
9  class Handler(object):
10
11     def __init__(
12         self, server, port, log_file_path,
13         sms_sender_func = None,
14         binding_timeout = 20):
15
16         # init logging
```

```
17          self.root_log = lo.getLogger()
18          formatter = lo.Formatter("%(asctime)s %(levelname)s %(message)s")
19          #logging to file
20          file_handler = lo.FileHandler( log_file_path )
21          file_handler.setFormatter(formatter)
22          self.root_log.addHandler( file_handler )
23          #logging to stdout
24          console_handler = lo.StreamHandler()
25          console_handler.setFormatter(formatter)
26          self.root_log.addHandler(console_handler)
27          #set log level
28          self.root_log.setLevel(lo.INFO)
29          self.root_log.info(" Initialized logging.")
30
31          self.binding_timeout = binding_timeout
32          self.server = server
33          self.port = port
34          self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
35          self.bind_socket()
36
37          if sms_sender_func is not None:
38              self.sms_func = sms_sender_func
39          else:
40              self.sms_func = self.default_sms_func
41
42      def __call__(self, conn):
43          with conn.makefile(mode="r") as f:
44          while True:
45              line = f.readline().strip()
46              self.root_log.info(line)
47                  try:
48                  msg = json.loads(line)
49              except (ValueError):
50                  self.root_log.warn("Could not translate message \
51  to json. Message: {}".format(line))
52                  msg = None
53              if msg is not None and msg["type"] == "sensorEvent":
54                  success = self.sms_func(line) # send entire json as sms
55                  self.root_log.info("Tried to send SMS. \
56  Success: {}".format(success))
57
58      def default_sms_func(self, message):
59          pass
60
61      def bind_socket(self):
62          self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
63          self.root_log.info("Trying to bind socket ... ")
64          for i in range(self.binding_timeout + 1):
65              try:
66                  self.sock.bind((self.server, self.port))
67                  self.root_log.info("Socket binding successfull with address: \
68  {} on port {}".format(self.server, self.port))
69                  break
```

```
70              except OSError:
71                  time.sleep(1)
72                  if i==self.binding_timeout:
73                      self.root_log.warn("Socket binding to adress: {} \
74 on port {} failed. Exiting programm..."
75 .format(self.server, self.port))
76                      sys.exit()
77
78      def get_connection(self):
79          self.sock.listen(1)
80          self.root_log.info("Waiting for incoming connection ...")
81          self.sock.settimeout(40)
82          conn, addr = self.sock.accept()
83          self.root_log.info("Connected address is: {}".format(addr))
84          self.sock.settimeout(None)
85          return conn
86
87
88 if __name__=="__main__":
89
90      gsm_handler= GSM("/dev/ttyAMA0", "+4917256187925")
91      gihas_handler = Handler(
92          "192.168.2.10", 4000,
93          "/home/pi/gihas_control.log",
94          sms_sender_func=gsm_handler.send_sms)
95
96      try:
97          conn = gihas_handler.get_connection()
98      except socket.timeout:
99          gihas_handler.root_log.warn("Could not find connection with client \
100 after {} seconds. Exiting program...".format(
101 gihas_handler.sock.gettimeout()))
102         sys.exit()
103
104     with conn:
105         gihas_handler(conn)
```

C.2. GSM Handling Module

Module for sending SMS messages. File is called: *gsm_handler.py*. Its class is used within *rpi_server.py*

```
1  import serial
2  import time
3
4
5  class GSM:
6      def __init__(
7          self, port_path, receiving_number,
8          baud_rate=9600, timeout=5):
9          """
10         Parameters:
11             port_path: (string) full file path to active
```

```
12              serial connection, e.g.: "/dev/ttyAMA0"
13          receiving_number: (string) number to send to
14              in international format, e.g.: +4917212312312
15          baud_rate: (int) baudrate of gsm module
16          timeout: (int) time (in seconds) handler waits for a response
17      """
18
19      self.ser = serial.Serial(
20          port_path, baudrate = baud_rate,
21          timeout=timeout)
22      time.sleep(1)
23      self.receiving_number = receiving_number
24
25      # set gsm module to sms text mode
26      self.write('AT+CMGF=1')
27
28      # choose sim card as primary sms storage space
29      self.write('AT+CPMS="SM","SM","SM"')
30
31  def read_response(self):
32      """
33      Listen for response from GSM module.
34      Only expect one response per given command.
35      readline() is a blocking call.
36      readline timeout is set during __init__.
37      Returns "" if timeout was reached.
38      """
39      return self.ser.readline()
40
41  def write(self, command):
42      self.ser.write(bytes(command, "utf-8") + b'\r')
43      # wait for module to process command
44      time.sleep(3)
45
46  def send_sms(self, message):
47      """
48      Send message to predefined number as sms.
49      Returns true if successfull, false otherwise.
50      """
51      # prepare sms
52      self.write('AT+CMGS="{}"'.format(self.receiving_number))
53      self.write(message)
54      # end sms
55      self.ser.write(b'\x1a')
56      return self.read_response().startswith("+CMGS")
```

C.3. Test Program for PLC Logic

Test script used to verify the logic of the PLC. It is called: *test_mapping.py*. Its name has to start with "test" so that the *pytest* library can recognize it. *rpi_server.py* script from section C.1 and this one have to reside in the same directory.

```
1 from .rpi_server import Handler
```

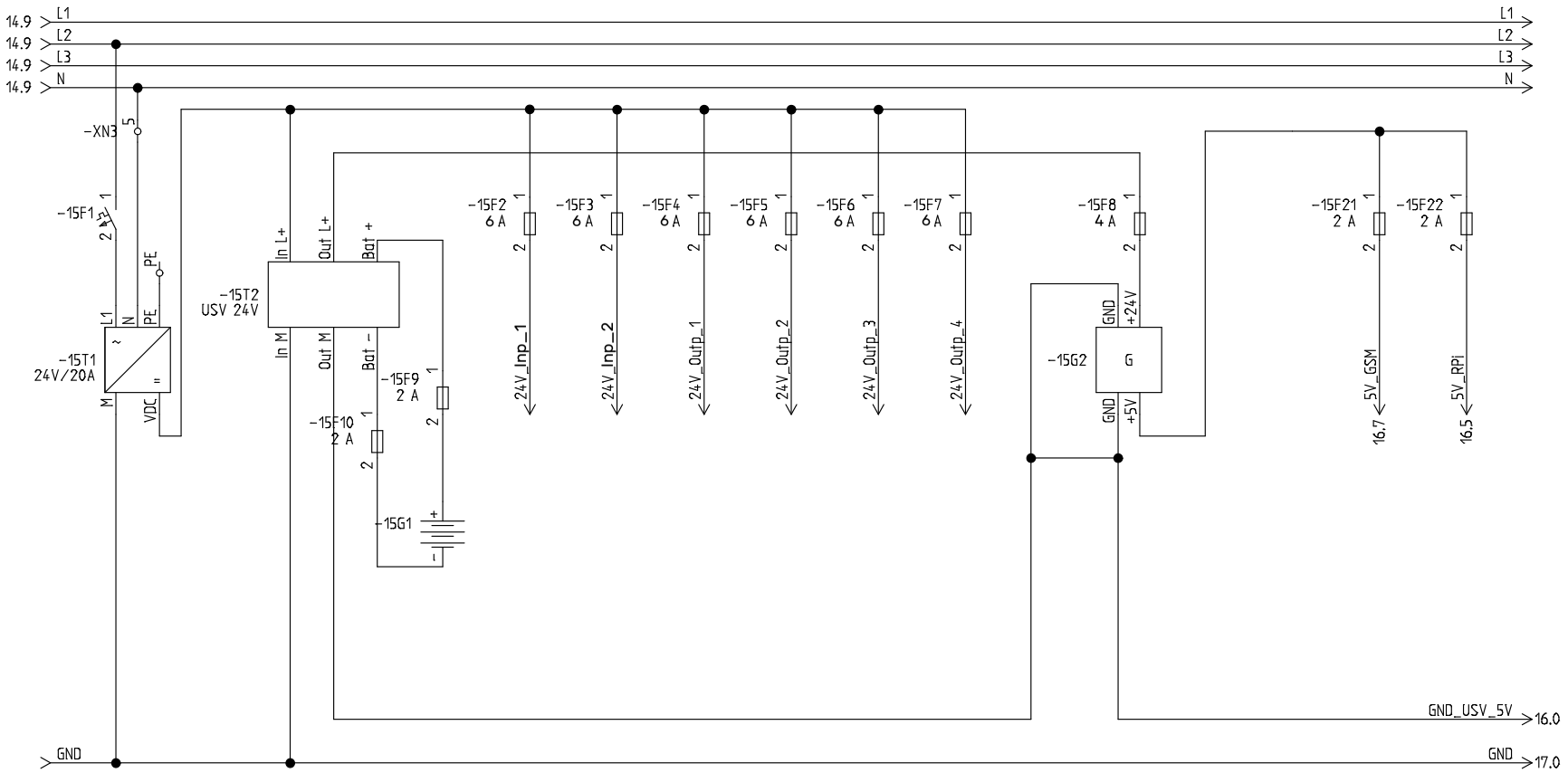
```
2  import gpiozero as go
3  import pytest
4  import socket
5  import sys
6  import time
7  import json
8  import signal
9
10
11 # setup timeout for test loop
12 class TimeoutException(Exception):
13     pass
14
15 def timeout_handler(signum, frame):    # Custom signal handler
16     raise TimeoutException
17
18 signal.signal(signal.SIGALRM, timeout_handler)
19
20
21 sim_inputs = [
22 # GPIO pin numbers of RPi
23 # order of elements correpsonds to
24 # order of inputs in mapping matrix
25     2, # M0
26     3, # M1
27     4, # M2
28     14, # M3
29     15, # M4
30     18, # M5
31     17, # M6
32     27, # M7
33     22, # S0
34     23, # S1
35     24, # S2
36     10, # S3
37     9, # S4
38     25, # S5
39     8, # S6
40     7 # S7
41 ]
42
43 mapping = [
44 [0, 0, 0, 0, 0, 0, 0, 0, 0, 2], # CONTROLLINO_A0
45 [0, 0, 0, 0, 0, 0, 0, 0, 0, 2], # CONTROLLINO_A1
46 [0, 0, 0, 0, 0, 0, 0, 0, 0, 2], # CONTROLLINO_A2
47 [0, 0, 0, 0, 0, 0, 0, 0, 0, 2], # CONTROLLINO_A3
48 [0, 0, 0, 0, 0, 0, 0, 0, 0, 2], # CONTROLLINO_A4
49 [0, 0, 0, 0, 0, 0, 0, 0, 0, 2], # CONTROLLINO_A5
50 [0, 0, 2, 2, 0, 0, 0, 2, 2, 0], # CONTROLLINO_A6
51 [0, 0, 2, 2, 0, 0, 0, 2, 2, 0], # CONTROLLINO_A7
52 [0, 0, 0, 0, 0, 0, 0, 0, 0, 2], # CONTROLLINO_A0,
53 [0, 0, 0, 0, 0, 0, 0, 0, 0, 2], # CONTROLLINO_A1
54 [0, 0, 0, 0, 0, 0, 0, 0, 0, 2], # CONTROLLINO_A2
```

```
55 [0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 2], # CONTROLLINO_A3
56 [0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 2], # CONTROLLINO_A4
57 [0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 2], # CONTROLLINO_A5
58 [0 , 0 , 2 , 2 , 0 , 0 , 0 , 2 , 2 , 0], # CONTROLLINO_A6
59 [0 , 0 , 2 , 2 , 0 , 0 , 0 , 2 , 2 , 0], # CONTROLLINO_A7
60 ]
61
62 reset_pin = go.DigitalOutputDevice(21, active_high=False)
63
64 def init_outputs():
65     for i, val in enumerate(sim_inputs):
66         sim_inputs[i] = go.DigitalOutputDevice(val, active_high=False)
67
68 def disable_outputs():
69     for val in sim_inputs:
70         val.off()
71
72 def close_outputs():
73     for val in sim_inputs:
74         val.close()
75
76 def reset_plc():
77     reset_pin.on()
78     time.sleep(.5)
79     reset_pin.off()
80
81
82 class TestMapping(object):
83
84     def setup_class(self):
85         init_outputs()
86         self.handler = Handler(
87             "192.168.2.10", 4000,
88             "/home/pi/testing_plc.log")
89         self.msg_timeout = 10
90
91     def teardown_class(self):
92         close_outputs()
93
94     def setup_method(self):
95         self.test_success = False
96         reset_plc()
97         try:
98             self.conn = self.handler.get_connection()
99         except socket.timeout:
100             self.handler.root_log.warn("Could not find connection with \
101 client after {} seconds. Exiting test ...".format(
102 self.handler.sock.gettimeout()))
103             sys.exit()
104         # wait for plc to finish starting up
105         time.sleep(5)
106
107     def teardown_method(self):
```

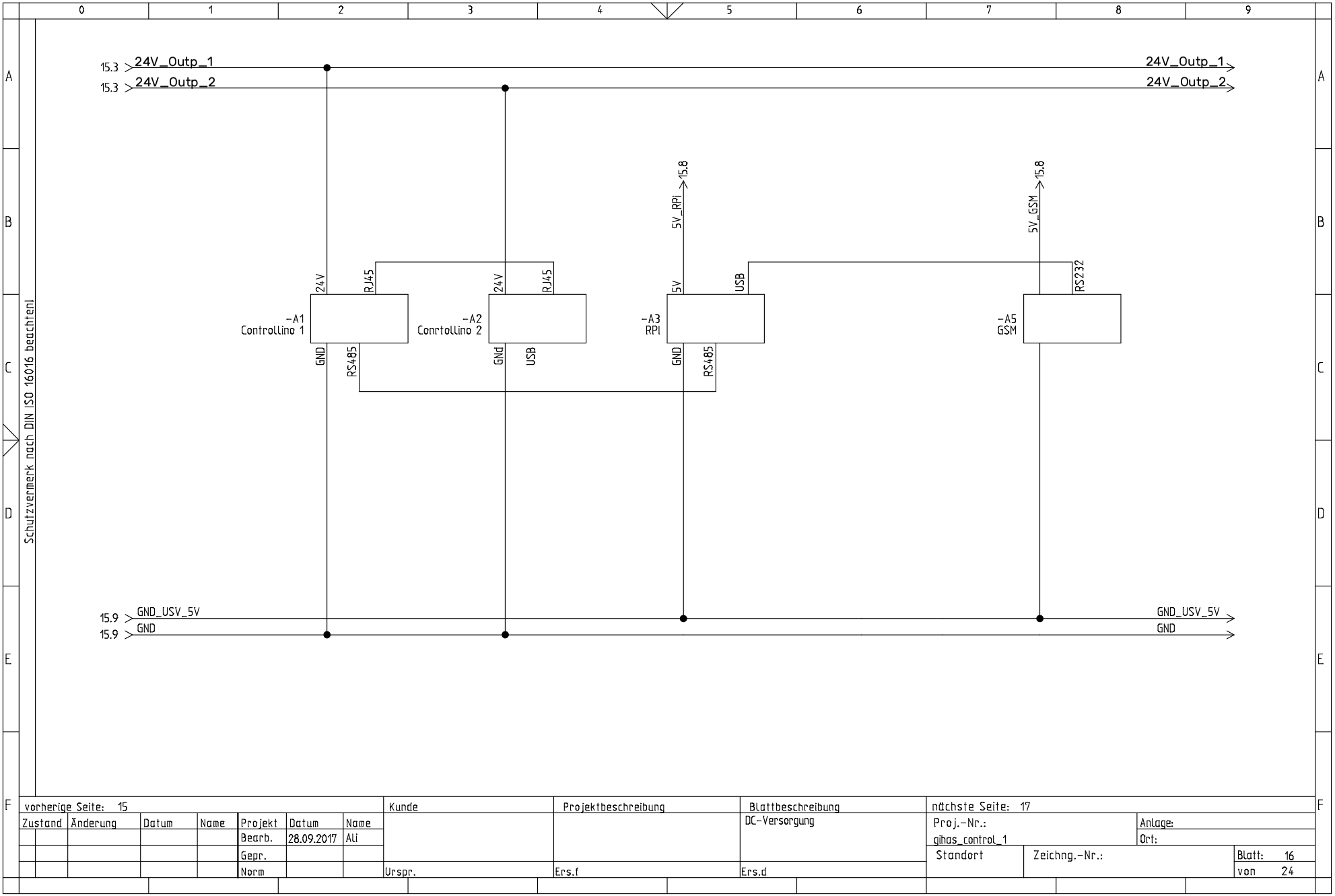
```
        disable_outputs()
        self.conn.close()

    @pytest.mark.parametrize("sensor_index, expected_output", [
    (0, mapping[0]),
    (1, mapping[1]),
    (2, mapping[2]),
    (3, mapping[3]),
    (4, mapping[4]),
    (5, mapping[5]),
    (6, mapping[6]),
    (7, mapping[7]),
    (8, mapping[8]),
    (9, mapping[9]),
    (10, mapping[10]),
    (11, mapping[11]),
    (12, mapping[12]),
    (13, mapping[13]),
    (14, mapping[14]),
    (15, mapping[15]),
    ])
    def test_map(self, sensor_index, expected_output):
        sim_inputs[sensor_index].on()
        with pytest.raises(TimeoutException):
            with self.conn.makefile(mode="r") as f:
                signal.alarm(self.msg_timeout)
                while True:

                    try:
                        line = f.readline().strip()
                        self.handler.root_log.info(line)
                    except TimeoutException:
                        if self.test_success:
                            raise TimeoutException
                        else:
                            raise AssertionError("No message type:sensorError \
received from PLC after {} sec.".format(self.msg_timeout))

                    try:
                        msg = json.loads(line)
                    except (ValueError):
                        self.handler.root_log.warn("Could not translate message \
to json for sensor: {}, message: {}".format(sensor_index, line))

                    if msg["type"] == "sensorEvent":
                        # the actual testing
                        assert msg["changedIn"] == sensor_index, "Wrong input \
has been triggered: {}".format(msg["changedIn"])
                        for i, val in enumerate(expected_output):
                            if val != 2:
                                assert val == msg["totalOut"][i], "Wrong output \
state on index: {}".format(i)
                        self.test_success = True
```

D. Circuit Layout of Control Logic

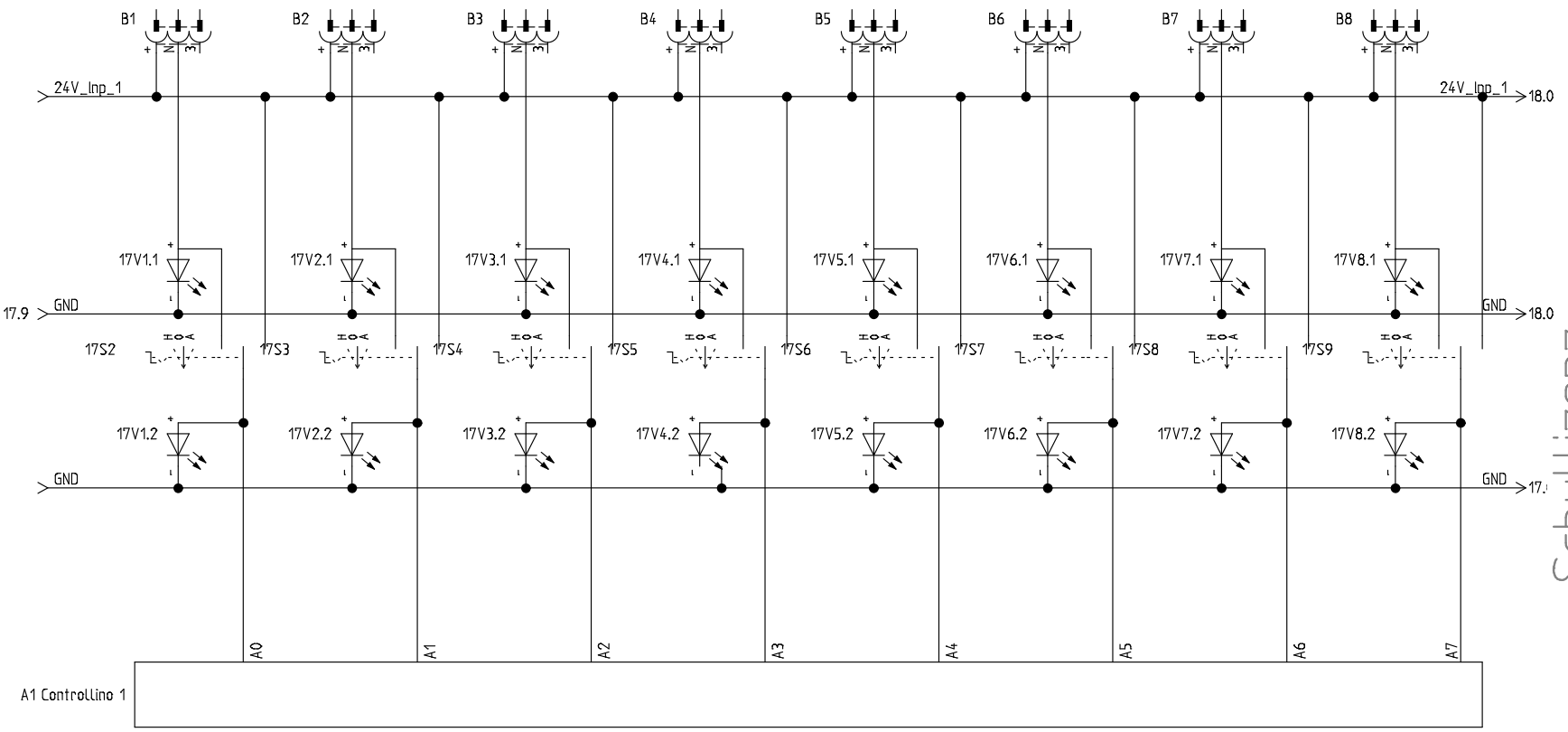
In this section, the circuit diagram for the entire low voltage side of the power unit is shown. It starts at page 15. The previous pages describe the high voltage side and can be found in *Paknejad* (2017). Page 15 and 16 have been modifed but their original design was taken from *Paknejad* (2017) as well. The last two page have a slightly different formating and naming convention. That is because I had to use a different version of the program *SeeElectrical*. It was used to design the high voltage circuit in *Paknejad* (2017) but the used educational version has a maximum amount of allowed pages, which I had to exceeded in this project.



vorherige Seite: 14							Kunde	Projektbeschreibung	Blattbeschreibung	nächste Seite: 16
Zustand	Änderung	Datum	Name	Projekt	Datum	Name			DC-Versorgung	Proj.-Nr.: gihas_control_1 / Anlage: / Ort:
				Bearb.	28.09.2017	Ali				Standort / Zeichng.-Nr.: / Blatt: 15 von 24
				Gepr.						
				Norm			Urspr.	Ers.f	Ers.d	



vorherige Seite: 15							Kunde	Projektbeschreibung	Blattbeschreibung	nächste Seite: 17
Zustand	Änderung	Datum	Name	Projekt	Datum	Name			DC-Versorgung	Proj.-Nr.: gihas_control_1 / Anlage: / Ort:
				Bearb.	28.09.2017	Ali				Standort / Zeichng.-Nr.: / Blatt: 16
				Gepr.						von 24
				Norm			Urspr.	Ers.f	Ers.d	

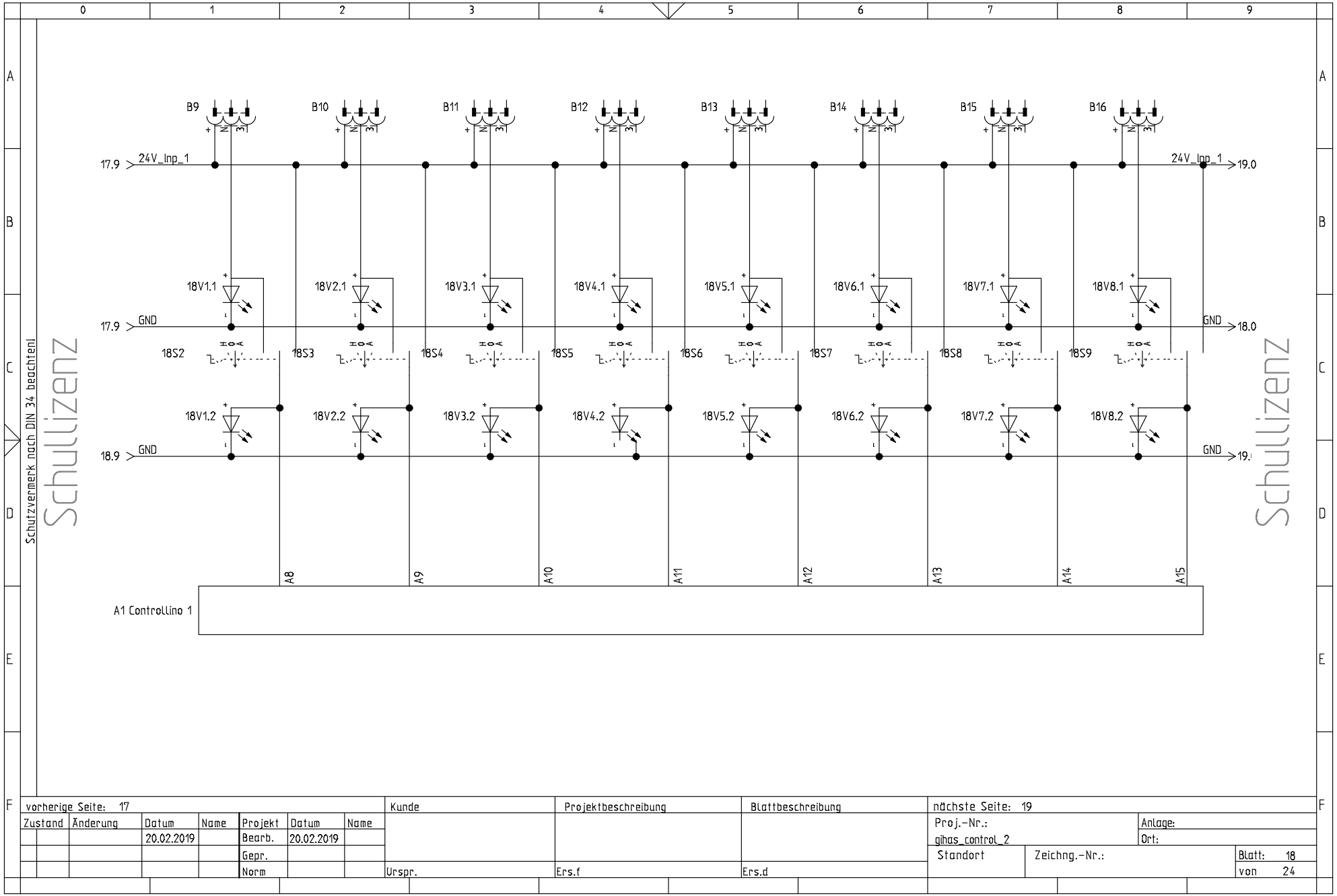


Schutzvermerk nach DIN 34 beachten!

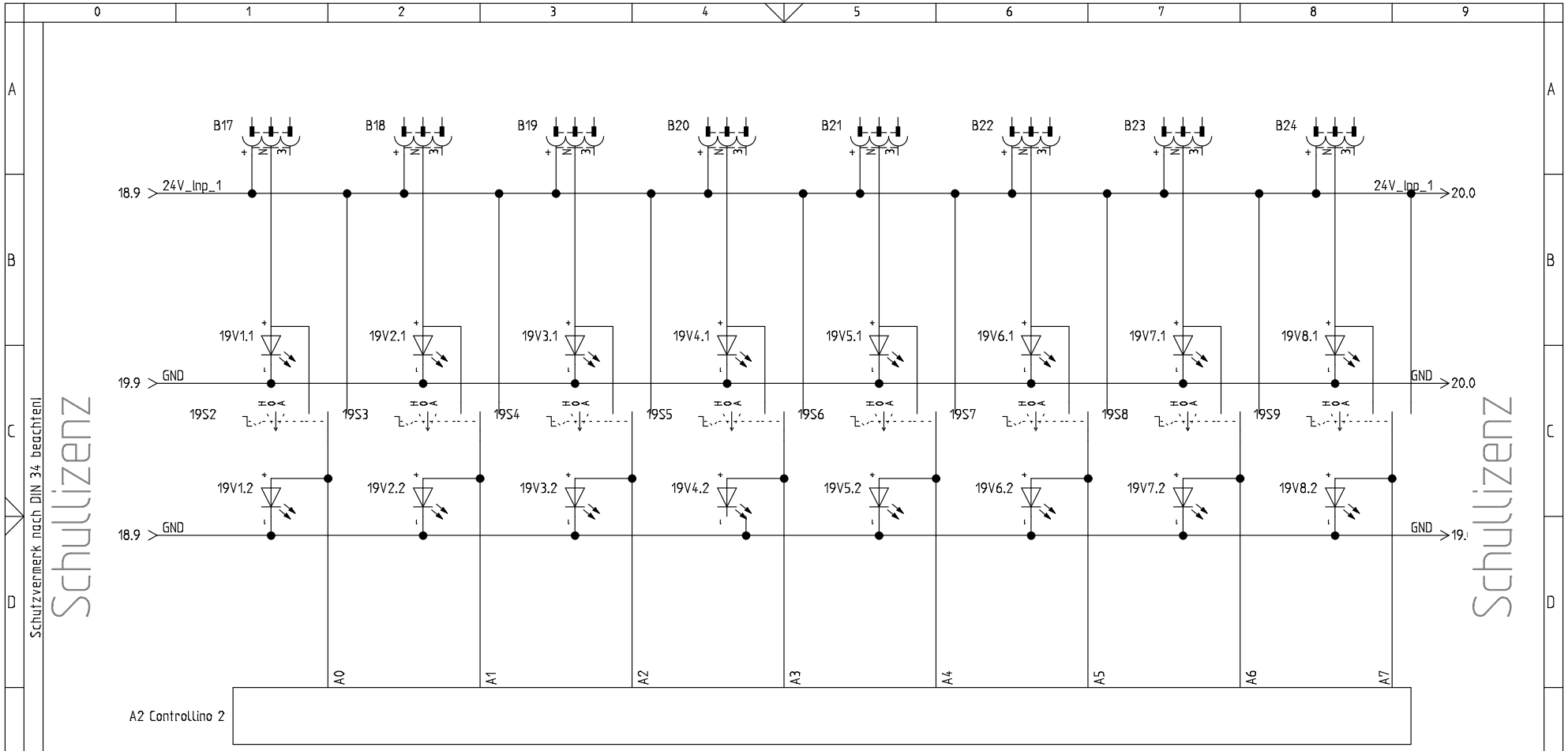
Schullizenz

Schullizenz

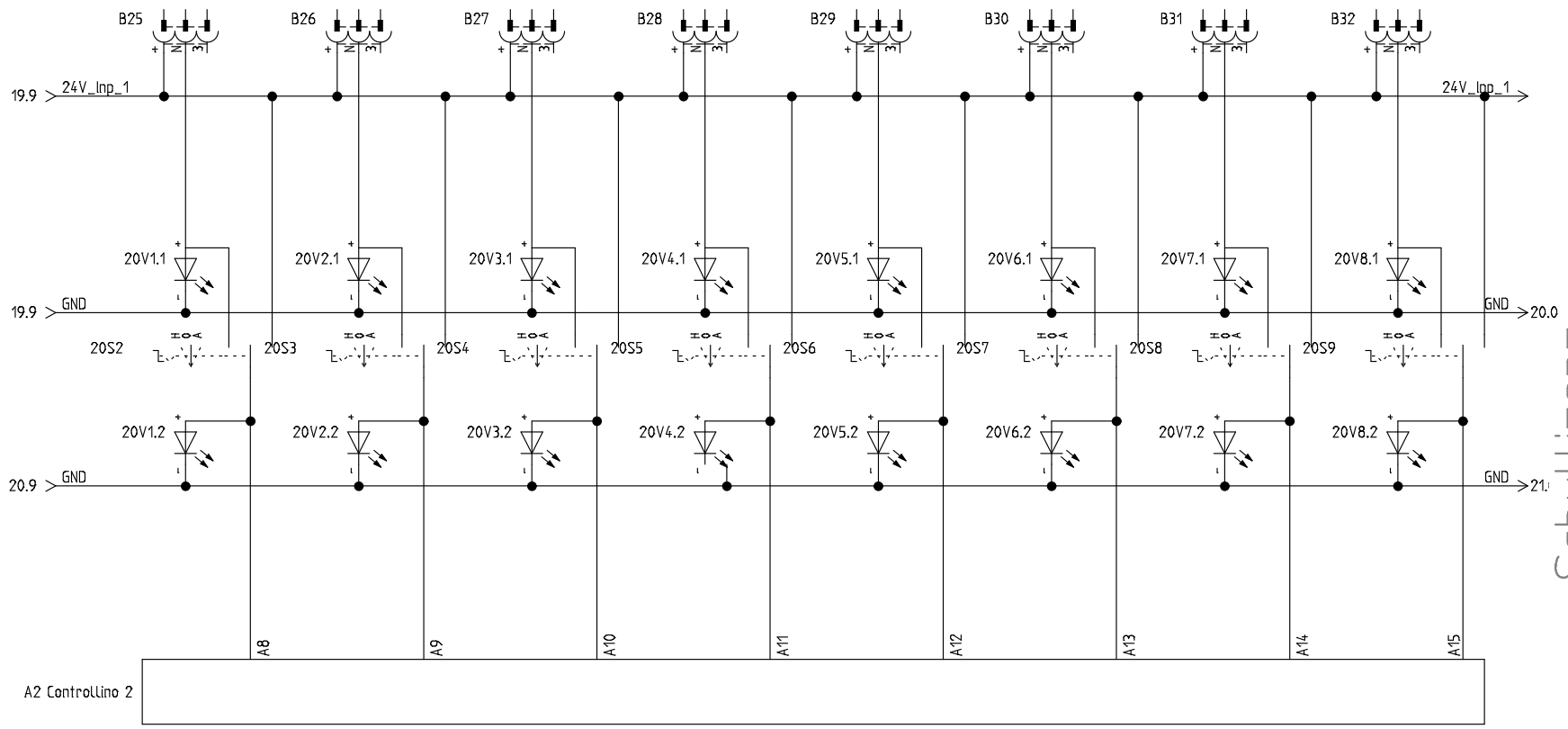
vorherige Seite: 16							Kunde	Projektbeschreibung	Blattbeschreibung	nächste Seite: 18
Zustand	Änderung	Datum	Name	Projekt	Datum	Name				Proj.-Nr.: / Anlage:
		20.02.2019		Bearb.	20.02.2019					gihas_control_2 / Ort:
				Gepr.						Standort / Zeichng.-Nr.: / Blatt: 17
				Norm			Urspr.	Ers.f	Ers.d	von 24



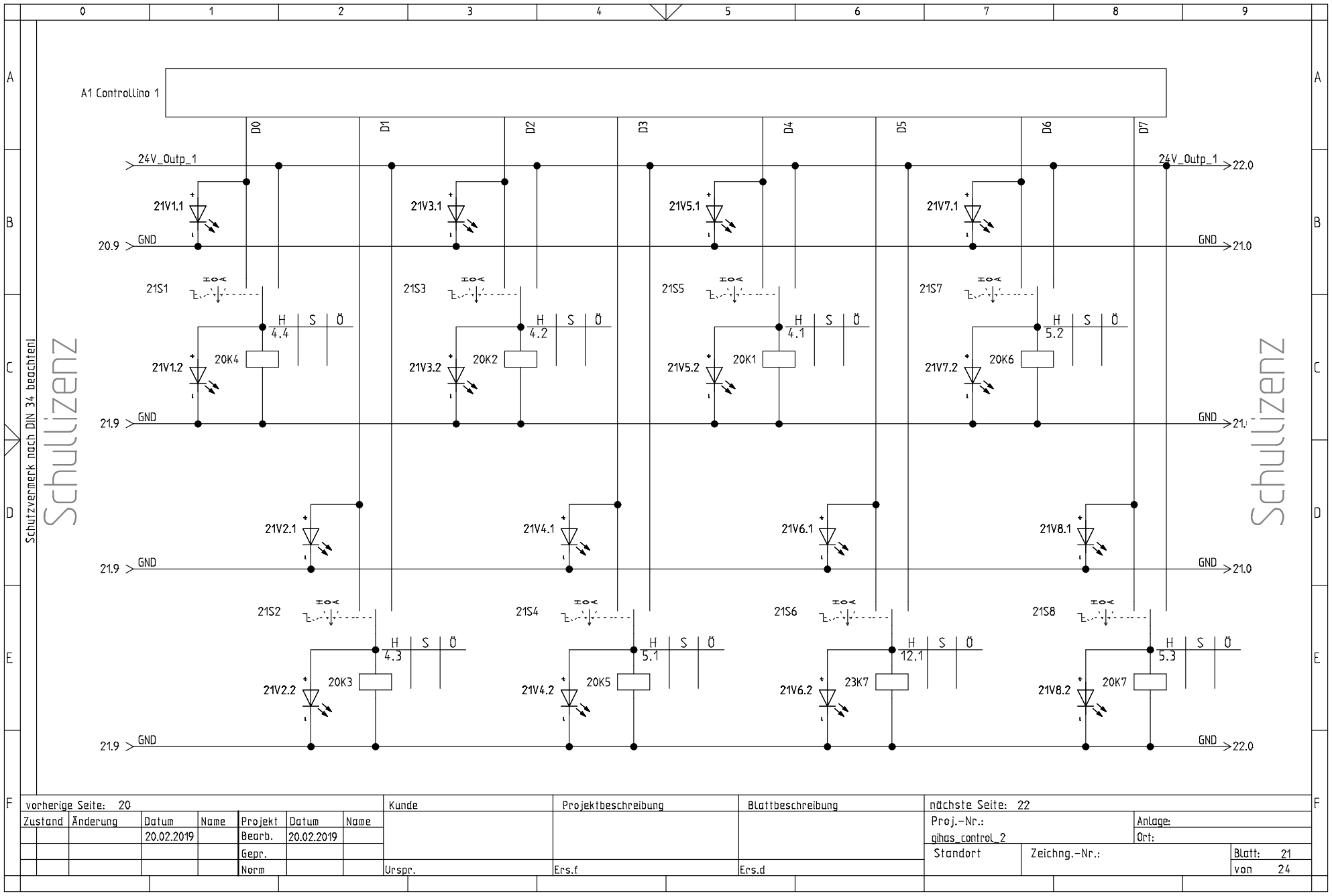
vorherige Seite: 17							Kunde	Projektbeschreibung	Blattbeschreibung	nächste Seite: 19
Zustand	Änderung	Datum	Name	Projekt	Datum	Name				Proj.-Nr.: / Anlage:
		20.02.2019		Bearb.	20.02.2019					gihas_control_2 / Ort:
				Gepr.						Standort / Zeichng.-Nr.: / Blatt: 18
				Norm			Urspr.	Ers.f	Ers.d	von 24



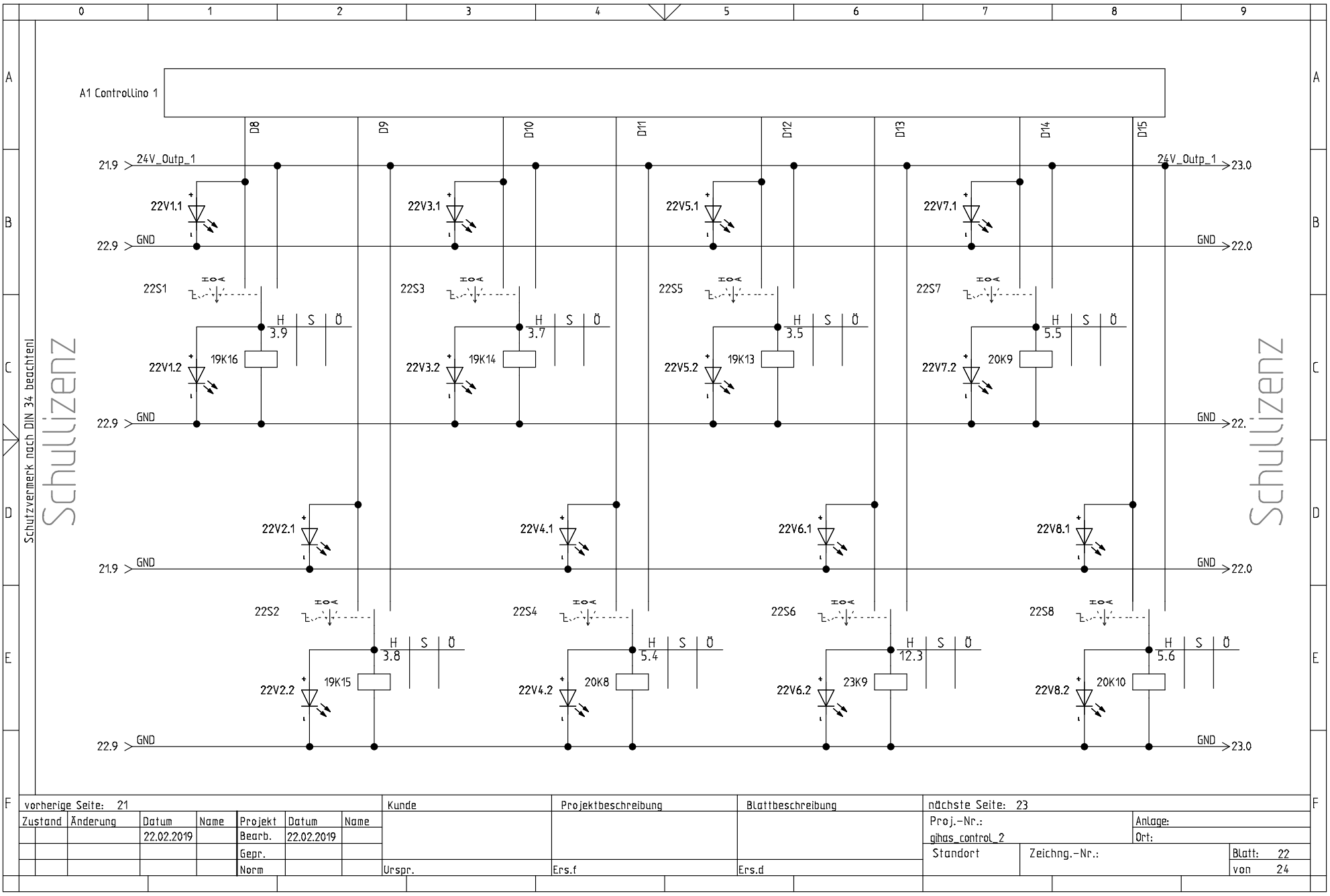
vorherige Seite: 18							Kunde	Projektbeschreibung	Blattbeschreibung	nächste Seite: 20
Zustand	Änderung	Datum	Name	Projekt	Datum	Name				Proj.-Nr.: / Anlage:
		20.02.2019		Bearb.	20.02.2019					gihas_control_2 / Ort:
				Gepr.						Standort / Zeichng.-Nr.: / Blatt: 19
				Norm			Urspr.	Ers.f	Ers.d	von 24



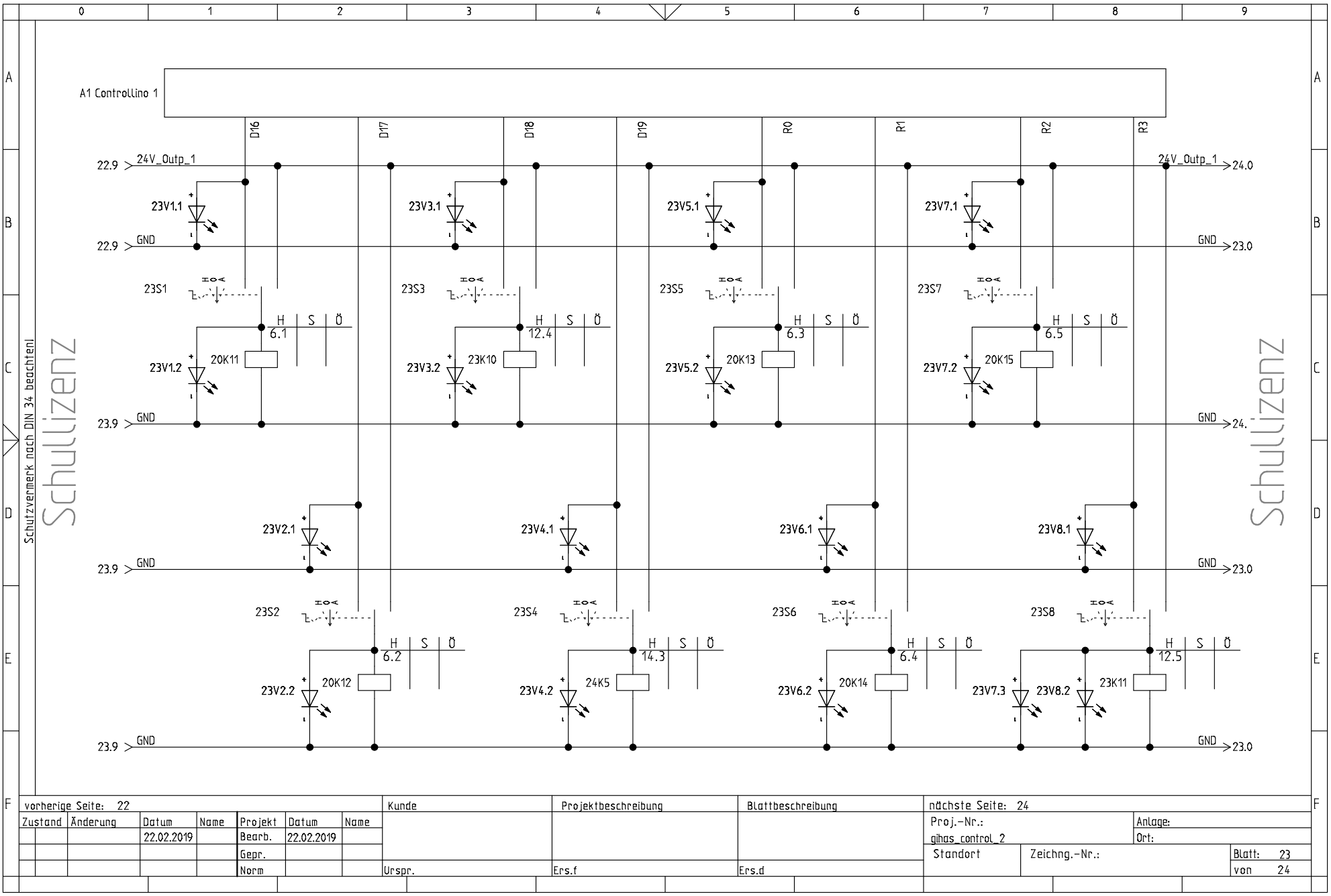
vorherige Seite: 19							Kunde	Projektbeschreibung	Blattbeschreibung	nächste Seite: 21
Zustand	Änderung	Datum	Name	Projekt	Datum	Name				Proj.-Nr.: / Anlage:
		20.02.2019		Bearb.	20.02.2019					gihas_control_2 / Ort:
				Gepr.						Standort / Zeichng.-Nr.: / Blatt: 20
				Norm			Urspr.	Ers.f	Ers.d	von 24



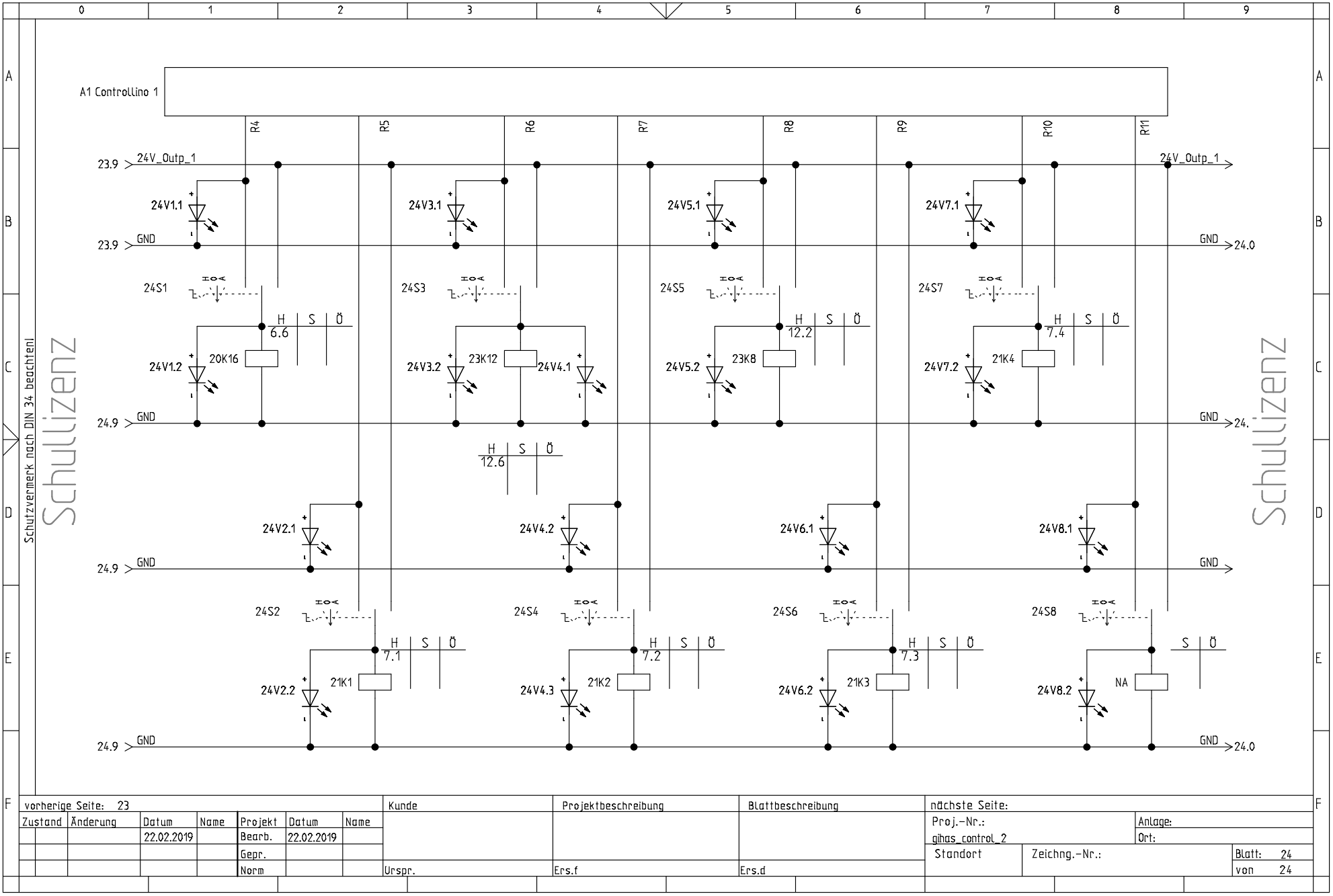
vorherige Seite: 20							Kunde	Projektbeschreibung	Blattbeschreibung	nächste Seite: 22
Zustand	Änderung	Datum	Name	Projekt	Datum	Name				Proj.-Nr.: gihas_control_2 / Anlage: / Ort:
		20.02.2019		Bearb.	20.02.2019					Standort / Zeichng.-Nr.: / Blatt: 21 von 24
				Gepr.						
				Norm			Urspr.	Ers.f	Ers.d	



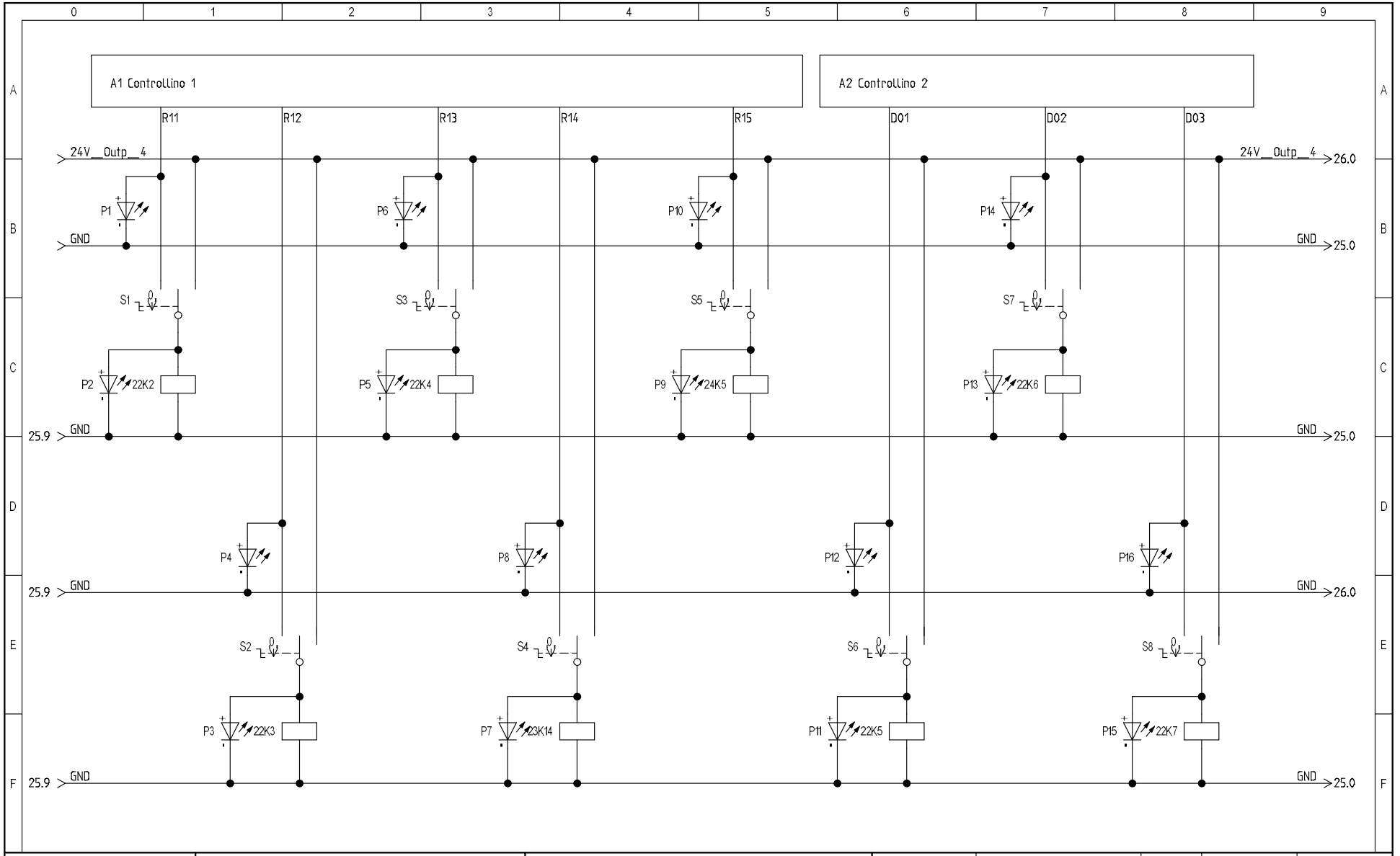
vorherige Seite: 21							Kunde	Projektbeschreibung	Blattbeschreibung	nächste Seite: 23
Zustand	Änderung	Datum	Name	Projekt	Datum	Name				Proj.-Nr.: gihas_control_2 / Anlage: / Ort:
		22.02.2019		Bearb.	22.02.2019					Standort / Zeichng.-Nr.: / Blatt: 22 von 24
				Gepr.						
				Norm			Urspr.	Ers.f	Ers.d	



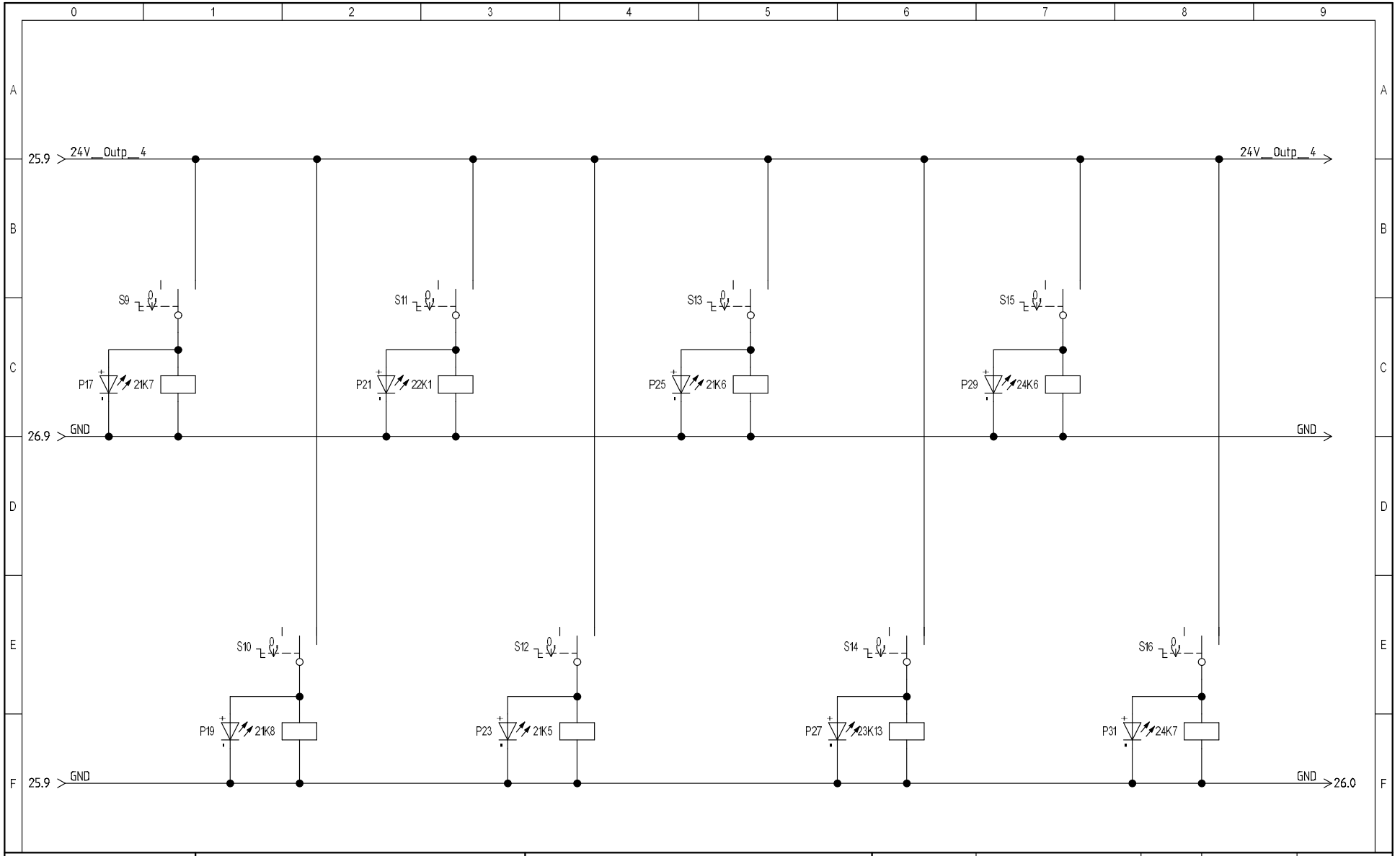
vorherige Seite: 22							Kunde	Projektbeschreibung	Blattbeschreibung	nächste Seite: 24
Zustand	Änderung	Datum	Name	Projekt	Datum	Name				Proj.-Nr.: gihas_control_2 / Anlage: / Ort:
		22.02.2019		Bearb.	22.02.2019					Standort / Zeichng.-Nr.: / Blatt: 23 von 24
				Gepr.						
				Norm			Urspr.	Ers.f	Ers.d	



vorherige Seite: 23							Kunde	Projektbeschreibung	Blattbeschreibung	nächste Seite:
Zustand	Änderung	Datum	Name	Projekt	Datum	Name				Proj.-Nr.: / Anlage:
		22.02.2019		Bearb.	22.02.2019					gihas_control_2 / Ort:
				Gepr.						Standort / Zeichng.-Nr.: / Blatt: 24
				Norm			Urspr.	Ers.f	Ers.d	von 24



IGE+XAO			Project: s26	Drawing no.:	Init.:	Rev.:	Sheet: 25
			Date: 01/05/2020	Function:	Location:	Total sheets: 2	Next sheet: 26



IGE+XAO

Project:	Drawing no.:	Init.:	Rev.:	Sheet:
s26				26
Date:	Function:	Location:	Total sheets:	Next sheet:
01/05/2020			2	

References

Benedek, G., and J. P. Toennies, *Atomic Scale Dynamics at Surfaces*, Springer-Verlag Berlin Heidelberg, 2018.

Brusdeylins, G., R. Doak, and J. Toennies, *Phys. Rev. Lett.*, *44*, 1417, 1980.

Cabrera, N., V. Celli, and J. Manson, *Phys. Rev. Lett.*, *22*, 346, 1969.

Carvalho, M. M., Bau und vermessung einer neuartigen trasversal-spineche-spule, Bachelor thesis (German), University of Heidelberg, 2020.

CONELCOM, Controllino mega pinout v1.1, `https://www.controllino.biz/wp-content/uploads/2018/10/CONTROLLINO-MEGA-Pinout.pdf`, accessed on 10.5.2020, 2018.

DeKieviet, M., D. Dubbers, C. Schmidt, D. Scholz, and U. Spinola, *Phys. Rev. Lett.*, *75*, 1919, 1995.

Fisher, S., and J. Bledsoe, *J. Vac. Sci. Technol.*, *9*, 814, 1971.

Hulpke, E., *Helium Atom Scattering from Surfaces*, Springer-Verlag Berlin Heidelberg, 1992.

Kohler, M., Onging bachelor thesis, Bachelor thesis (German), University of Heidelberg, 2020.

Lang, K., Design und planung zum bau der transversalen spinecho-spule für das gihas-experiment, Bachelor thesis (German), University of Heidelberg, 2019.

Paggi, S., Simulation of trajectories and cross sections of the 3he-abse, Bachelor thesis (German), University of Heidelberg, 2020.

Paknejad, A., Elektromagnetische symmetrie und eine gesicherte leistungsverteilung als programmierbare Überwachung für das ABSE-experiment, Bachelor thesis (German), University of Heidelberg, 2017.

Turczyk, T., Wiederaufbau einer maschine für heliumatomstreuung unter streifendem einfall (GIHAS) und die bewertung von spulen für die spätere erweiterung auf transversales spinecho, Diploma thesis (German), University of Heidelberg, 2018.

Willer, P., Entwicklung eines transfersystems for den probenwechsel im ultrahochvakuum, Bachelor thesis (German), University of Heidelberg, 2020.

Acknowledgment

First and foremost, I would like to thank Dr. Maarten DeKieviet for the opportunity to write this thesis. I am not only grateful for the tireless proof reading, but also for the continuous support over the entire time. The fact that his door was always open to me was an unexpected and an invaluable help.
My thanks goes out to my fellow former students Richard, Schko and Kevin (Let's go to Rewe!) for the fun times and great atmosphere. The same goes for Tom, Pascal, Marcia, Stefano and Marko with the addition of my huge appreciation for all the support in building the CAM experiment.
A big thank you to Nils and Manu for spotting and correcting an endless amount of mistakes in my drafts. My great appriciaction goes to my girlfriend Katrin as well, for not only proof reading but also for keeping my spirits high, in particular during the last weeks of writing.
I am extremly grateful to my parents Netti and Artur, for all the financial support but especially for each relaxing and fun day whenever I'm home or we're on vacation.

Most importantly, I would like to thank my sister Ulrike. Not just for correcting vast parts of this thesis, but more so for being there for me whenever I needed it.
Without you, this thesis would not have been possible.

Erklärung:

I hereby declare that I wrote the submitted thesis independently and I did not use any but the acknowledged sources and aids.

Ich versichere, dass ich diese Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den 25.5.2020