University College
of Southeast Norway

www.usn.no

FMH606 Master's Thesis 2018

Compact temperature regulator for subsea sensor

Sondre Benjamin Mogård

Faculty of Technology, Natural sciences and Maritime Sciences
Campus Porsgrunn

University College of Southeast Norway

www.usn.no

Course: FMH606 Master's Thesis, 2018

Title: Compact temperature regulator for subsea sensor

Number of pages: 72

Keywords: PID, Software PID, Subsea, compact regulator

Student:	Sondre Benjamin Mogård
Supervisor:	Hans-Petter Halvorsen
External partner:	OCTIO
Availability:	Open

Summary:

In volume constrained subsea systems reducing size is beneficial for several reasons. Devices in subsea for often is enclosed in an air tight pressure vessel. Replacing industrial controllers and other off-the-shelf components with it a small system on a chip is of interest.

In this master thesis, design and implementation of a prototype subsea system with software PID control using a system on a chip device was developed.

Simulation of the system was built from a mathematical model describing the system and programmed in LabVIEW software.

Complete prototype of the system was developed and performance in temperature stability was tested.

Preface

This thesis is the final project as part of the master's degree in System and Control Engineering at University College of Southeast Norway (HSN). The thesis is written during the spring semester of 2018.

I wish to thank my supervisor Hans-Petter Halvorsen and Trond Arne Espedal for guidance and supervision throughout the project.

Bergen, 15/05-2018

Sondre Benjamin Mogård

Contents

1 Introduction

The projects main goal is to develop a prototype software Proportional-Integral-Derivative (PID) controller in a system on a chip (SOC) device, for temperature regulation in a subsea sensor system. This task is provided by OCTIO. OCTIO works with geophysical monitoring services for oil drilling operations, to ensure safe drilling without discharging into the environment or other accidents. The system in focus already have temperature control by using off-the-shelf industrial PID controllers. With a SOC device the size would deduct by 50%. Which is significant and of interest, since the system is often mobilized. Size and weight deduction isn't only beneficial for practical reasons, but also safety reasons. Detailed description of what the subsea sensor system is comprised of will not be mentioned, since it's a secret OCTIO wants to keep.

1.1 Background

Commercial PID controllers often includes display and/ or rack mounts. Each of the controllers need power modules. Furthermore, control and monitoring needs dedicated communication channels, often ethernet, RS-232 or RS-485, which may add the need of protocol converters. This is impractical for volume constrained subsea application, where everything must be fitted into pressure vessels. It also could create the issue of too much self heating from electronic components in warmer environments.

One example of a temperature regulated subsea sensor system is comprised by splitting the sensor system and the control systems into multiple units, as shown in Figure 1-1. The aspect of self-heating from electronic components, is handled this way. But, splitting the system into several units also adds on the risk faulty wire connections, by needing connection between the units as well as top side control system.

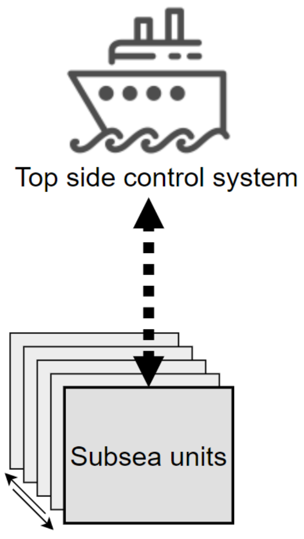


Figure 1-1: Subsea sensor system comprised of several units.

By implementing software PID controller the temperature regulated subsea sensor system can be reduced to one unit, as shown in Figure 1-2. Which is advantageous for several reasons, including the deduction of industrial components needed, but also transportation and easier to handle when lowering it down into the ocean and up again. Also, it will reduce power consumption and risk of wires to break.

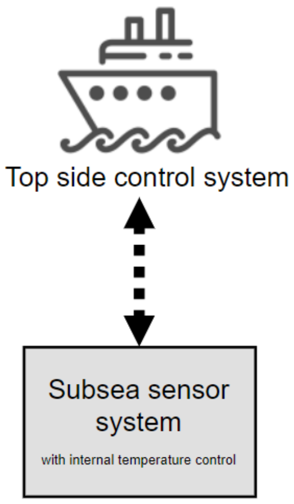


Figure 1-2: Subsea sensor system comprised of one unit.

1.2 Objectives

The task description specifies development and implementation of a versatile and compact PID temperature controller in a SOC device. With the following possible sub tasks gathered from the task description in Appendix A:

1. Study and gain experience of a commercial PID controller in a “sample temperature regulation system”.
2. Build a, somewhat simplified, mathematical model of the system, and use it to create a realistic simulator of the temperature regulation system. Behavior can be compared with data from real appliance.
3. Include a PID controller to the simulator.
4. Develop and implement a software PID controller on a microcontroller, with comparable performance as the reference temperature controller.
5. Develop and implement a simple register based system for parameter setting, control and monitoring.
6. Extend the system to an RTOS task (for integration into an existing software system).
7. Extend the system to comprise a multistage (cascade) PID controller.

1.3 Focus Areas

In dialogue with OCTIO, the tasks were prioritized as such; main goal is to develop a physical prototype with software PID on a SOC device. With performance in temperature stability verified for use in subsea sensor systems, +/- 0.1 °C. Extend it with a simple graphical user interface for adjusting parameters and monitoring. Build a simulator and extend it with PID control.

Programming language LabVIEW will be used for developing a simulation of the system and Python for development of a functioning prototype software for testing. The controller performance in temperature stability will be verified by testing performance in air and water, and while submerging into water. Necessary research regarding hardware is essential before any development of prototype can begin.

Challenges may be developing a simulator realistic enough to be able to use it for tuning PID purposes. Also finding good PID parameters, since the systems environment will change from air with temperatures up to around 26 °C to water with temperatures down to around -3 °C.

1.4 System description

System description is visualized in Figure 1-3, with functional and non-functional requirements listed beneath.

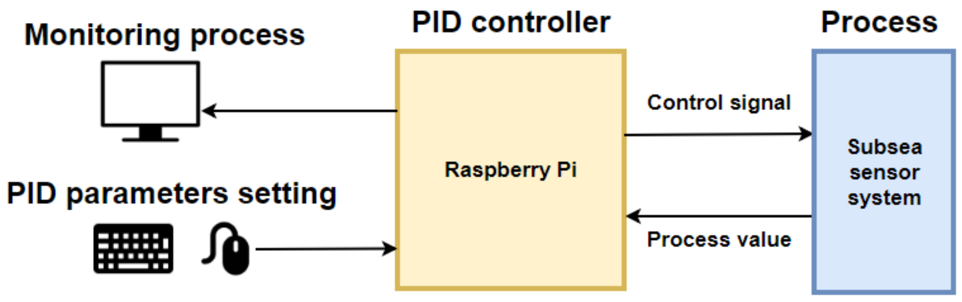


Figure 1-3: System description

Functional requirements:

- Read temperature.
- Compute control signal.
- Displaying system variables.
- Temperature logging.
- Parameter setting.

Non-functional requirements:

- Temperature stability +/- 0.1 °C.
- Pulse width modulation.
- Programming languages:
    - G (LabVIEW)
    - Python
- Hardware:
    - PT-100, 4- or 3-wire.
    - 12 V Heat pads
    - Solid-State Relay (for easier testing on real system)

1.5 Report structure

The report is divided into the following chapters:

- 1: Introduction
- 2: Research and theory
- 3: Simulation
- 4: Building Physical Prototype
- 5: Developing Software PID
- 6: Testing and Tuning Prototype
- 7: Conclusion and Future Works

2 Research and theory

This chapter is about theory used to solve the task. Including principals, components and programming languages in the field of computer science. As well as PID control theory and needed understanding of Thermodynamics.

2.1 Computer Science

The following subchapter presents the needed knowledge about electrical components, programming, communication protocols, signal modulation and process.

2.1.1 System on a Chip (SOC)

A system on a chip (SOC) is a small chip with all required electronic components and circuits in a given system, for example a computer. It often comprised of an processor, memory and some external interfaces for communication protocols, such as HDMI, USB and SPI. A difference between a traditional PC and a SOC device is that the CPU, memory and other components isn't replaceable. All the components are integrated into one chip. [1]

SOCs are commonly divided into 3 types:

- SOCs built around a microcontroller.
- SOCs built around a microprocessor.
- SOCs designed for a specific purpose, which doesn't fit into the 2 other categories.[1]

2.1.2 Python Programming Language

Python is an interpreted programming language where the syntax emphasizes readability. It will be used in this thesis to create the software PID controller. The Python interpreter and the standard library is open source software. In Python there is no compilation which makes the edit, test and debug faster. If a bad input or a bug appears the Python interpreter will raise an exception. Fallowed by printing its stack trace.[2]

2.1.3 LabVIEW

Laboratory Virtual Instrument Engineering Workbench (LabVIEW) is software for visual programming in the language G. Which is the main difference to most other development platforms. Predefined components can be added to the program using drag and drop from a list. G is a dataflow programming language, e.g. it changes outputs depending on variables in the program flow. The changes are reflected in real time. [3]

A LabVIEW program is called Virtual Instrument (VI) and a VI can be implemented inside another VI. Two windows are used when programming in LabVIEW, Front Panel and Block Diagram. The Front panel is the GUI, with components like indicators and controls, which is outputs and controllable inputs. The Block Diagram window is where the program flow is determined. An example of the two windows is shown in Figure 2-1, window to the left is the Front Panel and to the right is the Block Diagram. [3]

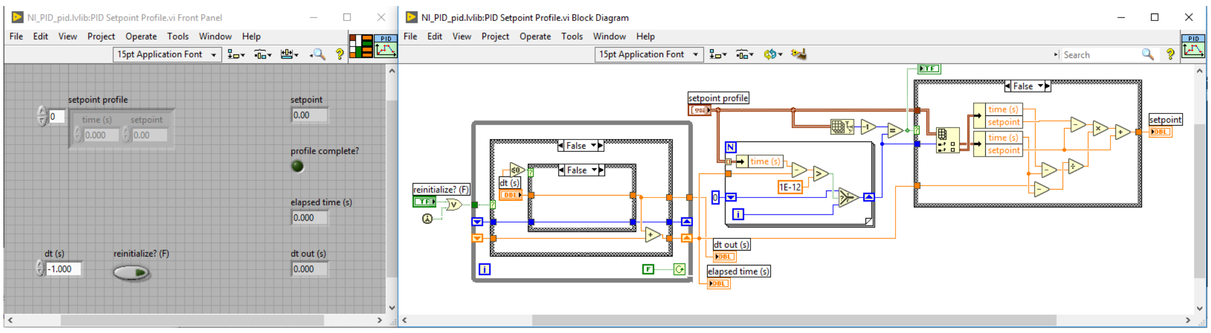

NI_PID_pid.lvlib:PID Setpoint Profile.vi Front Panel
File Edit View Project Operate Tools Window Help
setpoint profile
time (s)
setpoint
profile complete?
elapsed time (s)
dt (s)
reinitialize? (F)
dt out (s)
NI_PID_pid.lvlib:PID Setpoint Profile.vi Block Diagram
False
1E-12

Figure 2-1: The two LabVIEW windows Front Panel, to the left, and Block Diagram, to the right.

2.1.4 Pulse-width modulation (PWM)

Pulse-Width Modulation (PWM) is a modulation with the width of an impulse. Where the widths of the pulses correspond to specific data values. PWM operations is like a switch that constantly rotates on and off, illustrated in Figure 2-2.

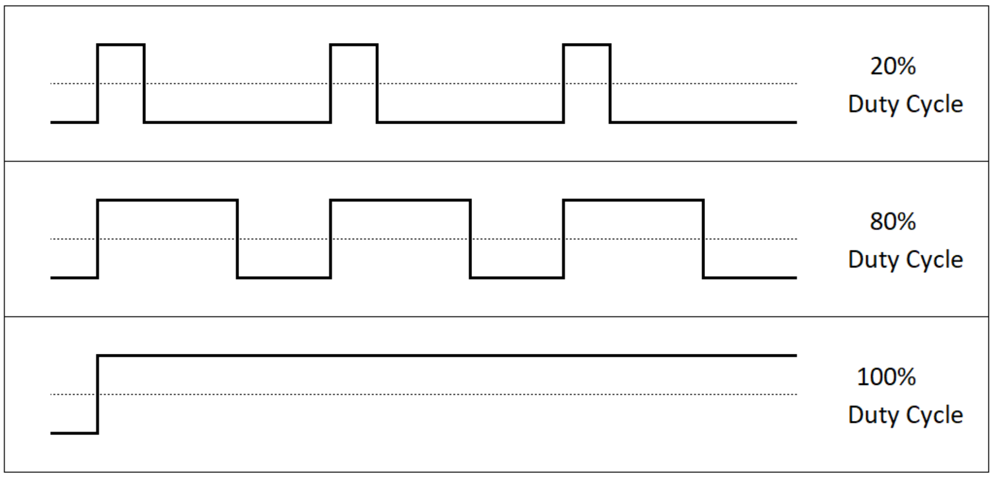


Figure 2-2: Three different pulse width signals.

2.1.5 Serial Peripheral Interface (SPI)

This subchapter is based on reference [4]. The SPI communication protocol will be used in this thesis, for retrieving measurement data from an RTD temperature sensor amplifier with ADC chip MAX31865. SPI is a synchronous serial communication interface for short distance communication. SPI is full duplex, meaning both receiving and sending simultaneously, using a master-slave architecture as shown in Figure 2-3.

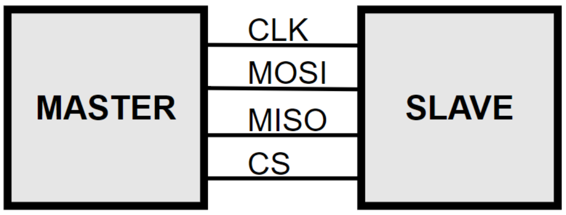


Figure 2-3: Serial Peripheral Interface communication example.

CLK is clock generated by the master. *MOSI* stands for Master Output Salve Input and is data output from the master. *MISO* is data output from the slave and stands for Master Input Slave Output. *CS* is Slave Select, used for slave selection, it could be left out in a single master to single slave circuit.

The master and slave uses shift registers to shift data in Serial In Serial Out (SISO) mode. Data transmission starts when the master configures the clock frequency. During each clock cycle the master sends a bit on the MOSI connection and the slave reads it. Simultaneously, the slave sends the master a bit over the MISO connection, as illustrated in Figure 2-4.

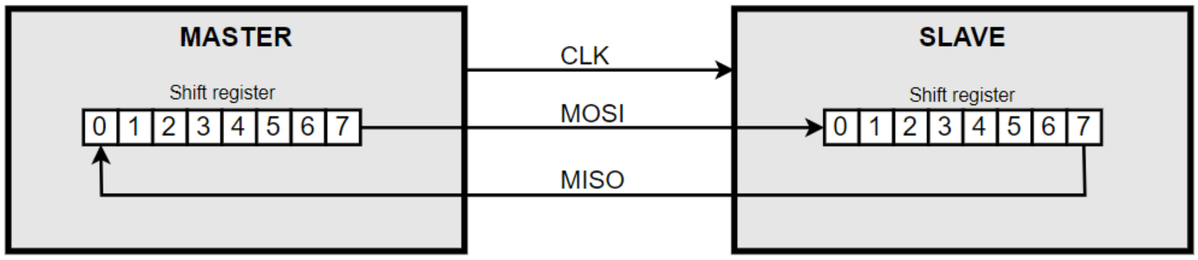


Figure 2-4: Two shift registers connected together forming a circular buffer.

The master sets the clock frequency for SPI data transfer, but clock polarity (CPOL) and clock phase (CPHA) must also match with the slave. CPHA defines when to transfer data and CPOL is the idle or active state of the clock. Clock Edge (CKE) is inverted clock phase. Figure 2-5 shows a timing diagram representing a data transfer in a time domain containing multiple rows.

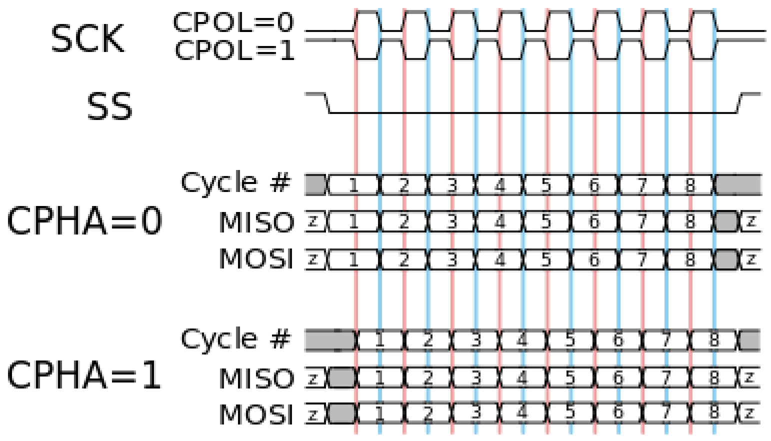


Figure 2-5: SPI timing diagram for clock polarity and phase.[4]

CPOL equal to 0 means that the base value of clock is 0, e.g. idle state is 0 and active state is 1. If CPHA also is equal to 0 data transmissions occurs during 0 to 1 transitions. Different SPI clock configurations are shown in Table 2.1.

Table 2.1: Different modes of SPI.

SPI MODE	CPOL	CPHA	CKE
0	0	0	1
1	0	1	0
2	1	0	1
3	1	1	0

2.1.6 Threads

A process can be divided into blocks of statements called threads, illustrated in Figure 2-6. It is the smallest sequence of programmed instructions for the processor registers. A thread is responsible for the execution of its block of statements. [5]

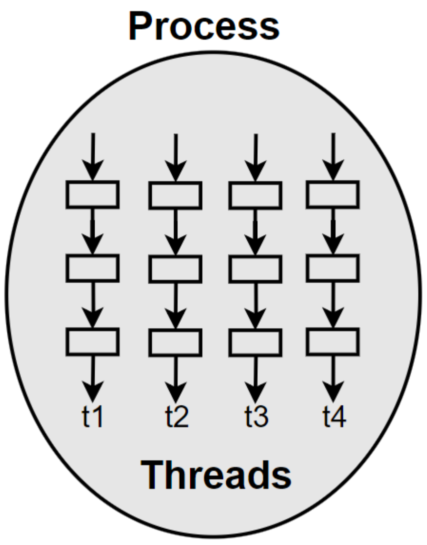


Figure 2-6: A process that is divided into 4 block statements called threads.

2.2 PID control

This chapter is about basic knowledge of PID control and based on reference [6]. Figure 2-7 shows how a closed loop control system uses feedback, to determine new output value. It calculates the error value between the measured feedback, process variable, and the set point, which is the desired value to achieve. Based on this error value the controller performs steps to bring the output value closer to the desired set point.

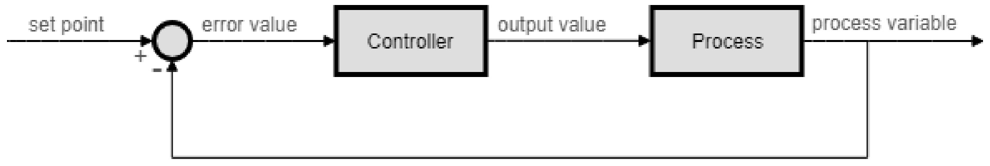


Figure 2-7: Closed loop control system using feedback to steer the output value to the desired process variable, which is the set point value.

PID controller is the most common control algorithm and is used in a closed control feedback loop to regulate a process, for example temperature in tank by controlling heat elements. PID stands for proportional-integral-derivative which is the three mathematical operations the regulator performs, these are shown in Figure 2-8 and explained in more depth in the following subchapters.

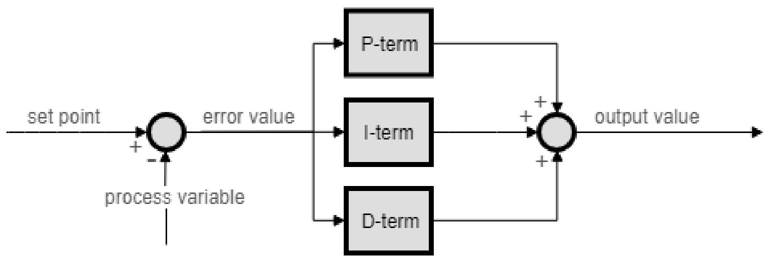


Figure 2-8: PID controller is comprised of 3 terms, proportional, integral and derivative term.

There are variations of the PID algorithm, but the one used in this project is shown in equation (2.1).

$$u(t) = K_p\, e(t) + K_i \int_0^t e(t)dt + K_d \frac{de(t)}{dt} \tag{2.1}$$

Where $e(t)$ is the error value, equation for calculating $e(t)$ shown in equation (2.2), and $u(t)$ is the control signal to the actuator, also known as output value. K_p, K_i and K_d is respectively the coefficients for the proportional, integral and derivative terms, often denoted P, I and D respectively. These coefficients, called gains, changes the error value through the controller and generates new output value to the actuator, to bring the process value near set point. The gain values are altered to find the optimal system response, this procedure is called tuning, more about this in chapter 2.2.5.

$$e(t) = r(t) - y(t) \tag{2.2}$$

Where the difference between set point, $r(t)$, and process variable, $y(t)$, accumulates to the error value, $e(t)$.

2.2.1 P-term

A controller with only P-term, commonly called P-controller, has output value proportional to the error value, since control value is error value multiplied by the proportional coefficient, as shown in equation (2.3). That means, larger error, or larger K_p value, gives larger output. This type of controller has some drawbacks, for example never reaching set point in steady state, because of small error value the controller output becomes negligible. E.g. even when reaching steady state there is still error, an offset from set point, commonly called steady-state error, shown in Figure 2-9. Another drawback is if K_p value is too high, that will lead to oscillation of process variable.

$$\text{P-term} = K_p\, e(t) \tag{2.3}$$

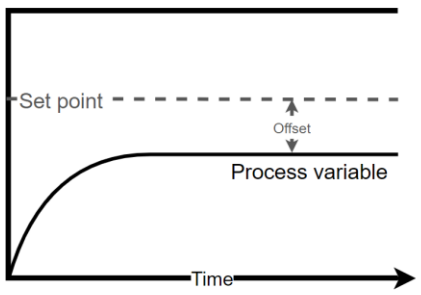


Figure 2-9: Steady-state error between set point and process variable, also called offset.

2.2.2 I-term

The I-term, shown in equation (2.4), eliminates offset. It accounts for past values of the error value by integrating them from startup t = 0 to present time. I-term seeks to remove the steady state error by adding control effect based on historic cumulative error value. I-term won't stop changing the output value until the error value is 0. When the error value is 0, the effect in output value is solely from the I-term.[1]

$$\text{I-term} = K_i \int_0^t e(t)dt \tag{2.4}$$

2.2.3 D-term

D-term, shown in equation (2.5), is a best estimate of the future trend of the error value, based on its current rate of change. Using the D-term could allow for higher values for K_p and K_i whilst the loop is still stable, giving faster response and better loop performance.[2] E.g. the main purpose of D-term is improving the transient response. Though a problem with the D-term is that it's sensitive to noise. With a noise spike in measurement of process variable the D-term will increase to compensate and furthermore make the control less stable.

$$\text{D-term} = K_d \frac{de(t)}{dt} \tag{2.5}$$

2.2.4 Anti Windup

Every actuator has a saturation limit, for example a valve cannot open more than 100% or close less than 0%. Under normal circumstances these limits shouldn't be reached. But say there is a disturbance and the error value grow beyond the normal circumstances and the output value maximum isn't enough to compensate for this disturbance. It will cause the I-term to accumulate over the time period to bring back process variable near to the set point by integrating up the deviation so that the output value will increase and increase. Therefor overshooting and continuing to increase as this accumulated error is growing, this is called integral windup, see Figure 2-10. As a result, the output value may exceed the saturation limits of the final control element, because of the I-term, and become unstable.

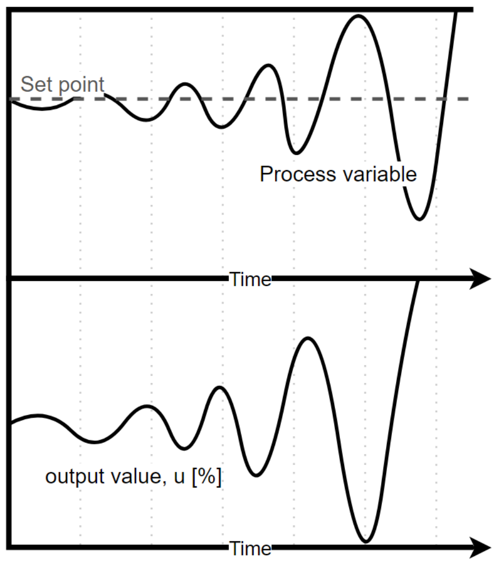


Figure 2-10: Example of how integral windup might affect the control system after a disturbance. With no limitations on the I-term the peaks grow, and the system becomes unstable.

[1] If it hasn't manually been added effect.

[2] Assuming good tuning for K_p, K_i and K_d.

Since windup is caused by the I-term, the solution is to clamp it. If only limiting output value, the I-term will still grow too large, or low, and the windup problem will still be present. There are different solutions to this problem, for example:

- setting error value equal to 0 when output value is at saturation limit.
- giving the I-term maximum and minimum values within saturation limits.

2.2.5 Tuning PID controller theory

Before tuning it is important to define performance requirements. This is often measured by applying a step change in set point and then measuring the response of the process variable. The response can then be evaluated by measuring defined waveform characteristics, as shown in Figure 2-11. The definition of these quantities can vary from industry to academic.

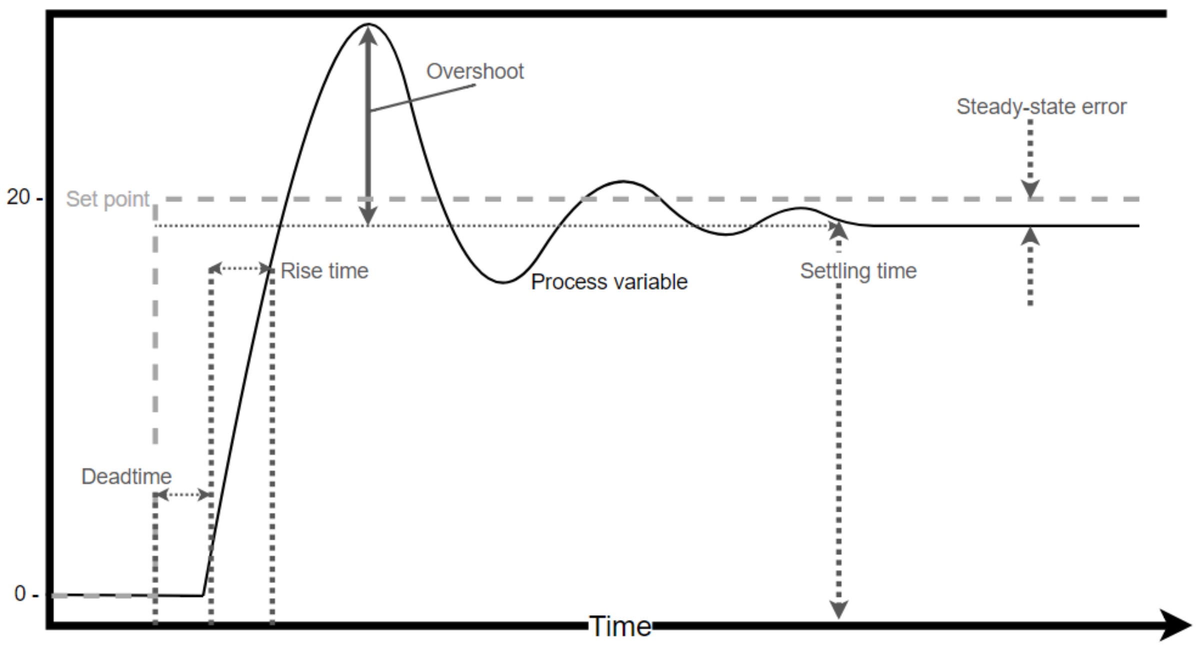


Figure 2-11: Example of a PID-controlled closed loop response to a change in set point.

Figure 2-11 shows the names of the different waveform characteristics. Rise time is the time from process variable goes from 10% to 90% of steady state. Overshoot is the amount the process variable goes over, in positive y-direction, the process variable value in steady state. The time required for the process variable to settle within bounds of the steady state process variable value, for example +/- 5%, is called settling time. As mentioned in chapter 2.2.1, the steady-state error is the offset between process variable and set point in steady state. Deadtime is the delay from when the step change happens to when the process variable change can be observed.

There are several methods created for tuning PID controller, these will not be discussed here. Instead the characteristics of typical system response while tuning and the effects from changing the different PID gains will be in focus. Table 2.2 shows trade-offs of how individually changing the different PID coefficients typically affect the system response. These correlations aren't always accurate since each gain has an effect on each other, for best performance all the coefficients must be tuned jointly.

Table 2.2 Possible effects of independently tuning the PID gains on a closed loop response.

	Rise time	**Overshoot**	**Settling time**	**Steady-state error**	**Stability**

Increasing K_p	Decrease	Increase	Small increase	Decrease	Degrade
Increasing K_i	Small decrease	Increase	Increase	Large decrease	Degrade
Increasing K_d	Small decrease	Decrease	Decrease	Minor change	Improve

Some typical damped system responses are unstable, critically damped, under-damped and over-damped, these are shown in Figure 2-12. Unstable systems produce an oscillation diverging step response, this kind of system won't settle down. An under-damped system has an oscillating response as well, but in this case, it eventually dampens out, unlike the unstable system. Over-damped system is characterized by long rise and settling time, but no overshoot. A critically damped system response is a compromise of response time and damping effects. It has shorter rise and settling time compared to over-damped system, but longer peak time in comparison with under-damped system.

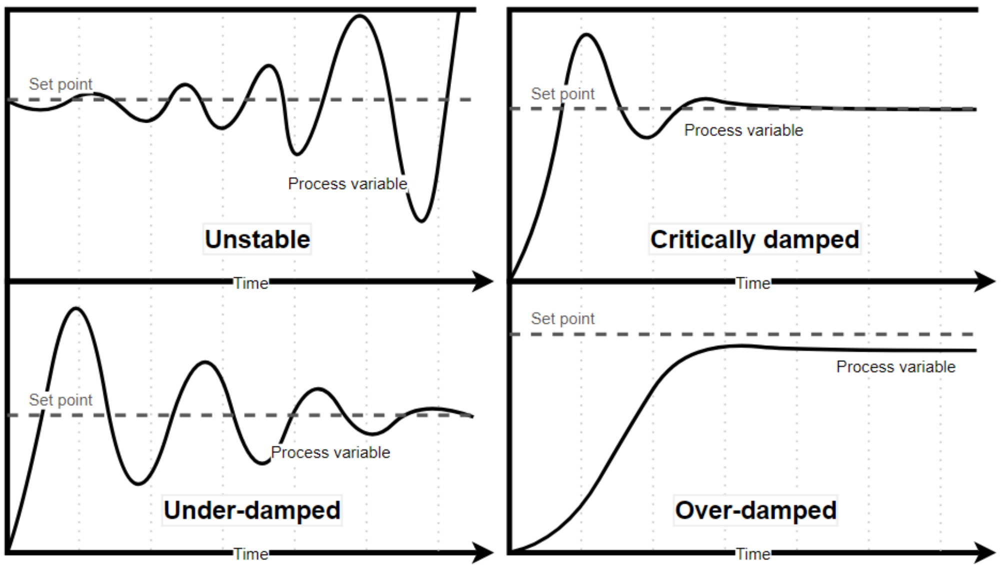


Figure 2-12: System damping examples.

Loop cycle is also an important parameter of a closed system. It is the time interval between each call to the PID-controller, also sometimes called cycle time or sample time. This depends on the system to be controlled, if it changes quickly a fast loop cycle is required, or the system will become unstable. If the loop cycle is very fast, it isn't usually a problem, but it could make the D-term be essentially just noise.

3 Simulation

This chapter is developing a simulation of the temperature behavior in the subsea sensor system. By analyzing the system, building a mathematical model, calculating constants, creating simulation software in LabVIEW and comparing with the real systems behavior. The calculated values describing the system will not be presented, since OCTIO do not wish to have them openly available. However formulas used to find these values will be mentioned.

3.1 Analyze

The system, shown in Figure 3-1, has a cylindrical shaped pressure vessel defining the boundaries. Inside there are multiple layers of solids and fluids before reaching the inner area. The system is completely sealed, so that the system isn't affected by pressure changes in surroundings. However, what does affect the system is environment temperature, which is the main disturbance and is why a controller is needed. The compartment connected to the heating element is the actuator for the controller and is denoted heating compartment.

The subsea sensor system, in addition to have changes in environment temperature, have change in environment medium. It will go between air and water, which has different heat transfer coefficients. Water transfers heat faster than air. For the environment this system is used the water environment will usually also have lower temperatures than when in air surroundings. For the real system, internal temperature will stabilize around 8.7 °C for environment water temperature of 4 °C. While the internal temperature would rise to around 30 °C in environment air of 25 °C, which means it isn't linear [7].

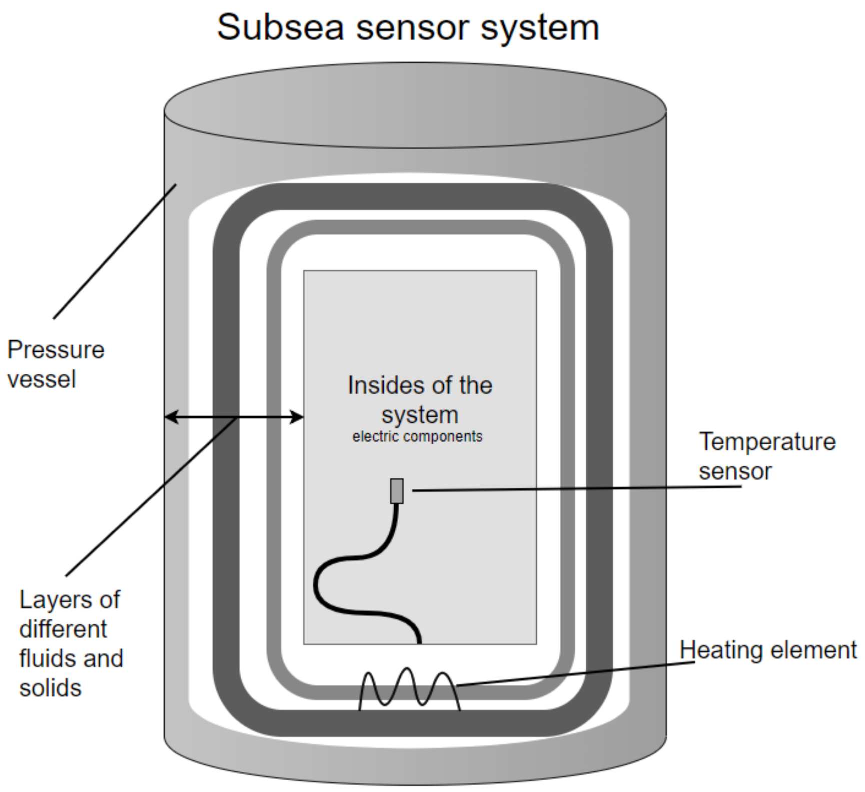


Figure 3-1: Cross-section of the subsea sensor system.

3.2 Building the Mathematical Model

This subchapter is based on reference [8]. The goal is to create a differential equation which calculates the temperature inside the system, based on disturbance from environment temperature and medium, as well as effect from heating compartment and self heating from electric components. E.g. the energy balance for thermodynamic systems is a good starting point, which is elaborated in following subchapter.

3.2.1 Energy Balance

Balance law for thermodynamic systems gives the energy balance shown in equation (3.1). Time argument is excluded in this subchapter for simplicity reasons.

$$\frac{dE}{dt} = \sum Q_{in} - \sum Q_{out} + \sum Q_{generated} \tag{3.1}$$

The energy balance deriving will be based on assumptions and equations representing energy inputs and outputs presented in following subchapter.

3.2.2 Assumptions about the System

Energy is assumed to be proportional with temperature and volume, as seen in equation (3.2).

$$E = c\rho VT \tag{3.2}$$

Where c is specific heat capacity of the system [J/kg K], ρ is the density of the system [kg/m^3], V is volume of the system [m^2] and T is temperature inside the system [K].

Q_{in} in the energy balance is representing the added effect. This can be divided into effect from electrical components, denoted P_c, and effect added from heating compartment, when temperature control is implemented, denoted P_{HP}, as shown in equation (3.3).

Also assuming, that there is no storage of energy in the heating element, which means the effect added to heat element is added directly to the system. Therefor it isn't necessary to derive an own energy balance for the heat pads.

$$Q_{in} = P_c + P_{HP} = P_{in} \tag{3.3}$$

Where P_{in} is the accumulated heat effect added. Heat loss is denoted Q_{out}, as shown in equation (3.4).

$$Q_{out} = U[T_{env} - T] \tag{3.4}$$

Where U is the overall heat transfer [J/s m^{2}K] or [W/m^{2}K]. Expression for U can be derived from equation for thermal resistance shown in equation (3.5).

$$R_{tot} = \frac{1}{U} \tag{3.5}$$

By multiplying with U on both sides and then dividing by R_{tot}, U is alone on left-hand side, shown in equation (3.6).

$$U = \frac{1}{R_{tot}} \tag{3.6}$$

Inserting equation (3.6) into equation (3.4) gives equation (3.7) to describe heat loss.

$$Q_{out} = \frac{T_{env} - T}{R_{tot}} \tag{3.7}$$

There is no energy generation, no chemical reaction etc. In other words, $Q_{generated}$ can be neglected from the energy balance, as shown in equation (3.8).

$$\frac{dE}{dt} = \sum Q_{in} - \sum Q_{out} \tag{3.8}$$

3.2.3 Differential equation

The energy balance can now be written as shown in equation (3.9), based on assumptions and expressions from previous subchapter.

$$\frac{d(c\rho VT)}{dt} = P_{in} - \frac{(T - T_{env})}{R_{tot}} \tag{3.9}$$

Assuming constant volume, density and specific heat capacity they can be moved out of the derivation as shown in equation (3.10).

$$c\rho V \frac{dT}{dt} = P_{in} - \frac{(T - T_{env})}{R_{tot}} \tag{3.10}$$

Which leads to a differential equation, shown in equation (3.11), for temperature inside the subsea sensor system, which can be used for simulation.

$$\frac{dT}{dt} = \frac{1}{c\rho V}\left[P_{in} - \frac{1}{R_{tot}}(T - T_{env})\right] \tag{3.11}$$

The differential equation has three time dependent parts:

- the internal temperature, T.
- the environmental temperature, T_{env}.
- added power, P_{in}.

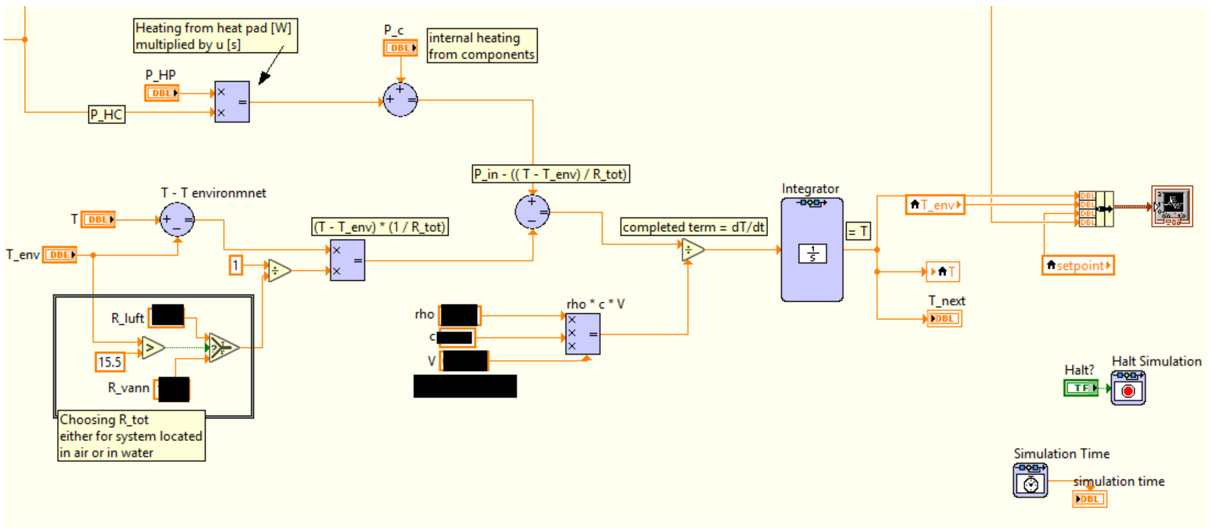
3.3 Developing Simulation in LabVIEW

This chapter is about developing the LabVIEW simulation of the subsea sensor. It will include code and functionality.

3.3.1 LabVIEW Code for Simulating

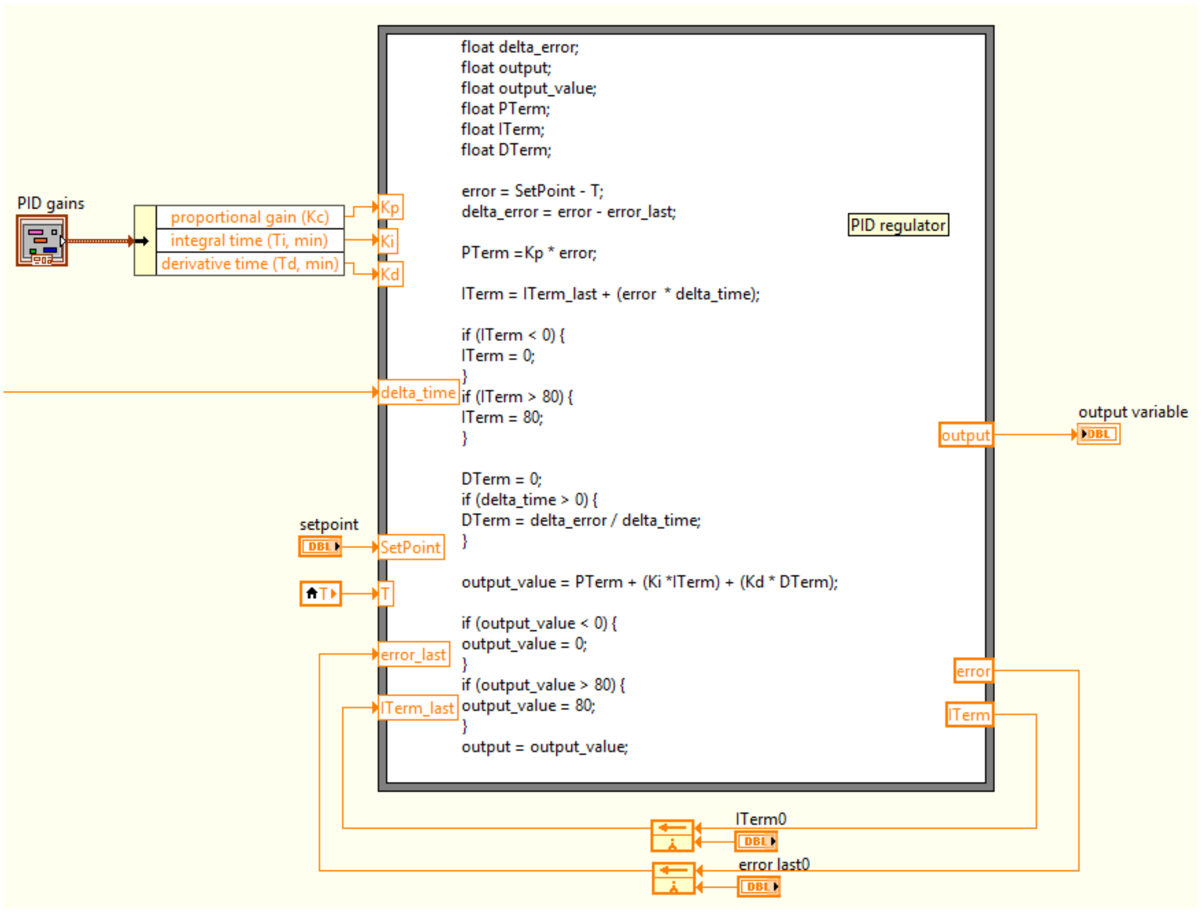
Figure 3-1 shows how the differential equation is comprised in the Block Diagram window. Inside the frame in the lower left corner its created a logic to use either R_{Tot} for water environment or air. This is automatically chosen based on environment temperature. If environment temperature is over 15.5 °C it will use R_{Tot} for air.

In Figure 3-2 the PID algorithm made in a Formula Node is shown. The output of the PID controller is scaled to a PWM signal using logic shown in Figure 3-3. How the whole program is connected inside a Control and Simulation loop can be seen in Figure 3-4.



Heating from heat pad [W] multiplied by u [s]
P_c
internal heating from components
P_HP
P_HC
P_in - ((T - T_env) / R_tot)
T - T environmnet
T
T_env
(T - T_env) * (1 / R_tot)
1
Integrator
= T
completed term = dT/dt
T_env
setpoint
T
T_next
R_luft
15.5
R_vann
Choosing R_tot either for system located in air or in water
rho
c
V
rho * c * V
Halt?
Halt Simulation
Simulation Time
simulation time

Figure 3.1: LabVIEW code of the differential equation.



```
float delta_error;
float output;
float output_value;
float PTerm;
float ITerm;
float DTerm;

error = SetPoint - T;
delta_error = error - error_last;

PTerm =Kp * error;

ITerm = ITerm_last + (error  * delta_time);

if (ITerm < 0) {
ITerm = 0;
}
if (ITerm > 80) {
ITerm = 80;
}

DTerm = 0;
if (delta_time > 0) {
DTerm = delta_error / delta_time;
}

output_value = PTerm + (Ki *ITerm) + (Kd * DTerm);

if (output_value < 0) {
output_value = 0;
}
if (output_value > 80) {
output_value = 80;
}
output = output_value;
```

Figure 3-2: PID controller made using Formula Node in LabVIEW.

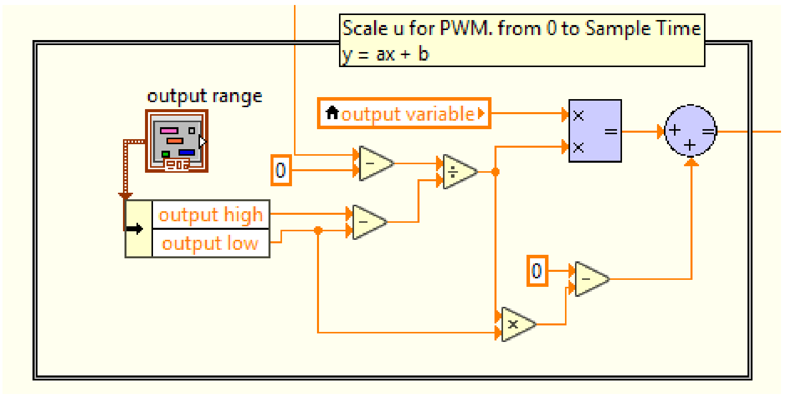


Figure 3-3: Scaling output of the PID to a PWM signal.

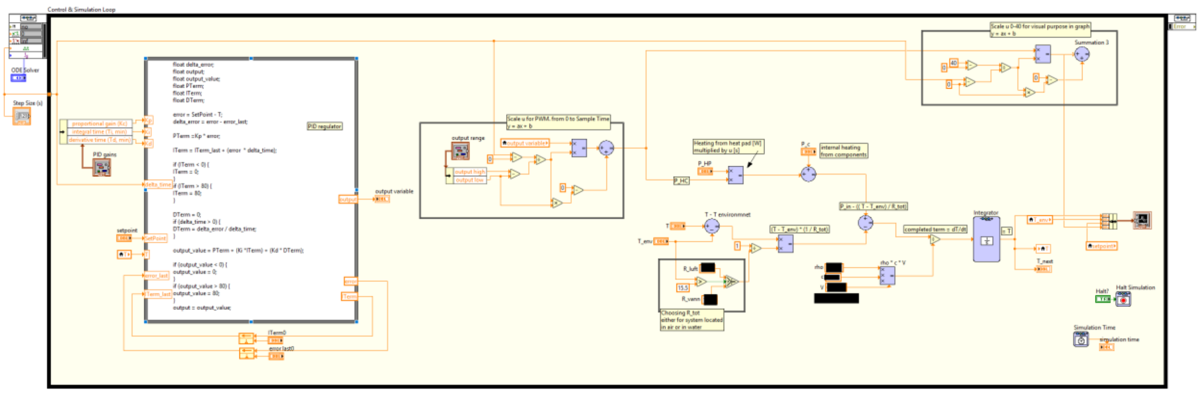


Figure 3-4: The complete Block Diagram view of the LabVIEW program.

3.4 Comparing Simulator to the Subsea Sensor System

This chapter is about comparing simulator to the real subsea sensor systems temperature behavior with respect to environment temperature. The values representing the real subsea sensor was approved for sharing by OCTIO.

Figure 3-5 shows the simulator responding to step response in environment temperature. The first step is from 25 °C to 4 °C, the second is from 4 °C back to 25 °C. From the graph it can be read that temperature inside the system in 25 °C environment air stabilizes on 30 °C. It can also be read that it stabilizes on 8.7 °C in environment of 4 °C.

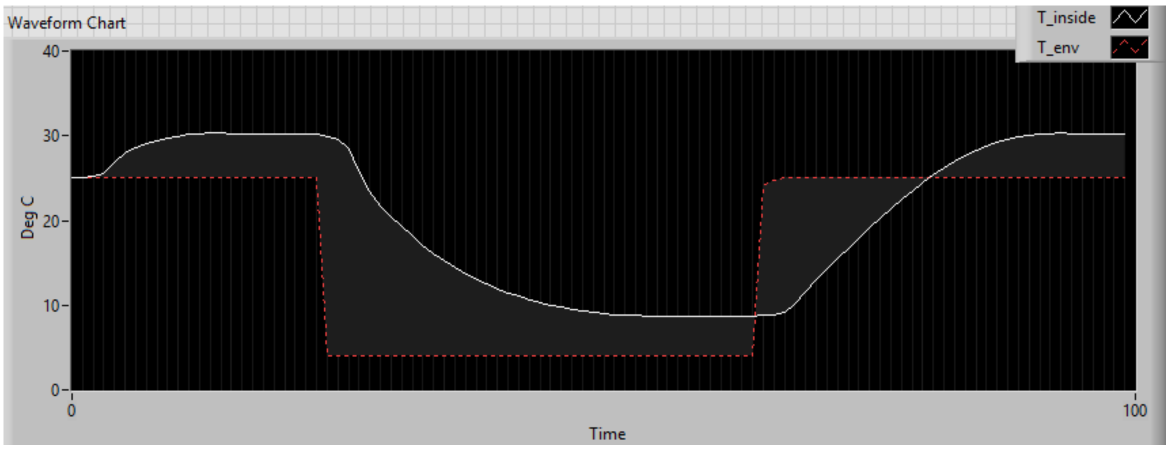


Figure 3-5: Simulator responding to step response in environment temperature.

The simulated temperature response curve is also of similar shape the real system. For example, with the step response from 25 °C to 4, the curve slowly decreases in the beginning. Then it decreases very fast until slowly converging into steady-state.

4 Building Physical Prototype

This chapter is about designing and building a prototype subsea sensor system with control system on a Raspberry Pi. It is comprised of materials lying around and commercial available electronic components. The software PID controller is developed using Python programming language.

4.1 Design Specifications

Figure 3-1 shows the subsea sensor system that will be made a prototype of. By using materials at hand combined with available industrial electronic components. For water proof housing it was found a spare pressure vessel, of different dimensions than the real appliance. Which is ideal since it will have similar heat transfer values. The heating compartment will be represented by a soda can, as shown in Figure 4-1, with three available heat pads glued to it. Temperature sensor will be placed in the middle of the soda can. The different layers between pressure vessel and heating compartment will be represented by a layer of insulation and air. The internal heating from electrical components will not be represented in the prototype.

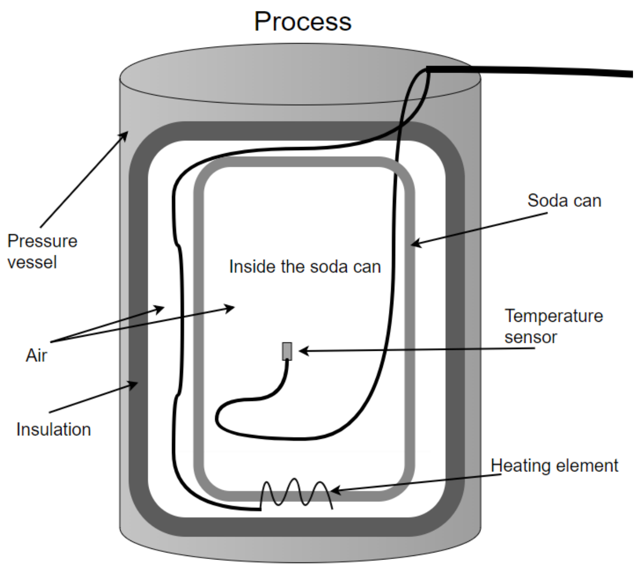


Figure 4-1: Prototype of subsea sensor system.

The available heat pads are 12 V with 7.5 Watts (W) and will need 1.875 Ampere (A) when connected in parallel, as shown in equation (4.1). Which is more than the Raspberry Pi can provide with its outputs for 3.3 V and 5 V. To generate this signal, it will be utilized a laboratory power supply. The voltage will be sent through a relay which is to be controlled by the Raspberry Pi. An analogue to digital converter (ADC) is needed to read the temperature sensor using Raspberry Pi. Overview of prototype with subsea sensor system and control system can be seen in Figure 4-2.

$$I = \frac{V}{P} \rightarrow I = \frac{12\ V}{7.5 + 7.5 + 7.5\ W} = 1.875\ A \tag{4.1}$$

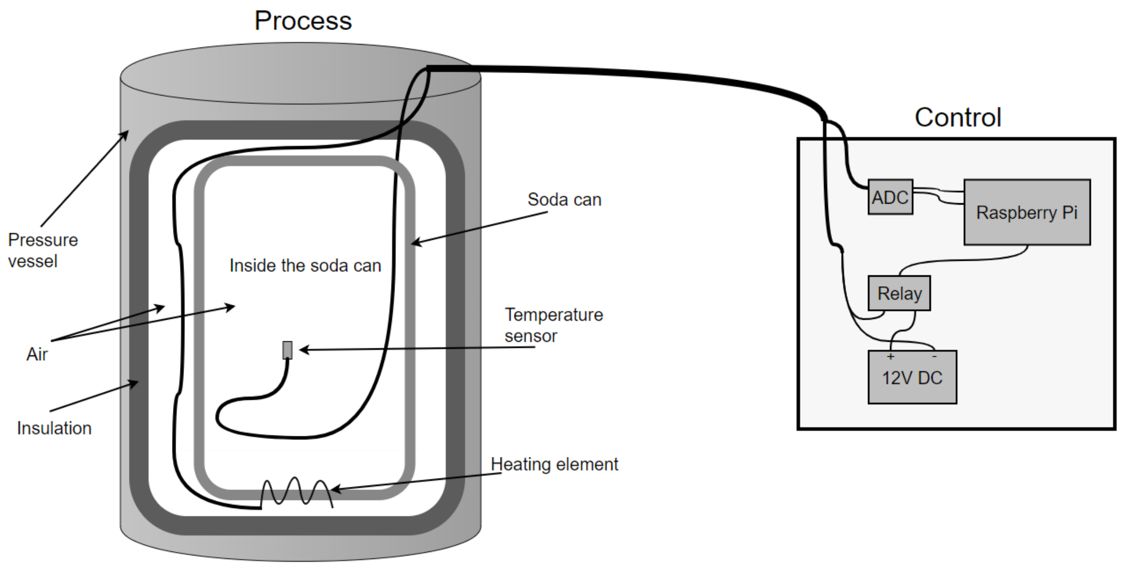


Figure 4-2: Prototype of subsea sensor system with control system.

4.2 Chosen Hardware

This subchapter is about hardware chosen to be used in the prototype system.

4.2.1 Raspberry Pi

Raspberry Pi, shown in Figure 4-3, is a single-board computer capable of running different types of OS. It has a Broadcom SOC, including an ARM compatible central processing unit (CPU) and graphics-processing unit (GPU). All storage is on an external micro SD card. [9]

Figure 4-3: Raspberry Pi. [10]

It's chosen to use the third generation Raspberry Pi, Raspberry Pi 3 Model B, which has WI-FI and Bluetooth module integrated. It has 40 General-purpose input/output (GPIO), UART, I2C and SPI support. Specifications are shown in Table 4.1.

Table 4.1: Raspberry Pi 3 Model B technical specifications. [10]

SOC	Broadcom BCM2837
CPU	64-bit quad-core, 1.2 GHz ARM Cortex-A53
GPU	Dual core Videocore IV® Multimedia co-processor
RAM	1 GB LPDDR2
Wireless connection	802.11 b/g/n 2.4 GHz Wireless LAN and Bluetooth 4.1

GPIO	40-pins
Ports	1 x 10/100 Ethernet port, 1 x HDMI video/audio connector, 1 x RCA video/audio connector, 1 x CSI camera connector, 4 x USB 2.0 ports
Storage	Micro SD card
Power	Micro USB connector for 2.5 A power supply

4.2.2 Temperature sensor

The PT-100 temperature sensor chosen is shown in Figure 4-4, it’s a 4-wire platinum resistance temperature detector (RTD). PT-100 also comes with 2- or 3-wires which has less accuracy, since the 4-wire connection eliminates the influence of the connection lead on the measuring result [11]. It is class A in accuracy, meaning it will be +/- 0.23 °C in temperature range -40 to 40 °C, compared to class B RTDs which is +/- 0.5 °C in the same range [12]. For more information about the sensor, see reference [13].

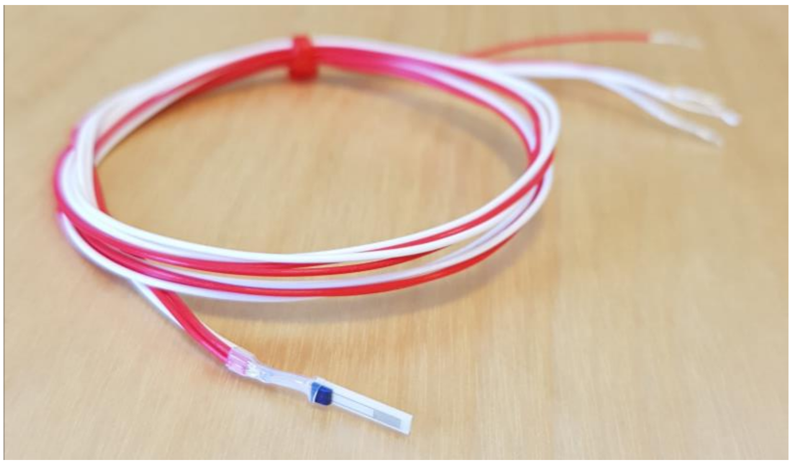


Figure 4-4: RS Pro PT100 Sensor

4.2.3 Analogue to Digital Converter

Instead of a regular ADC it was chosen to use PT100 RTD amplifier, shown in Figure 4-5, with internal ADC. Adafruit PT100 RTD Temperature Sensor Amplifier, MAX31865, is designed to read the low resistance and automatically adjust and compensate for the resistance of the connecting wires, which gives higher accuracy for temperature readings. [14]

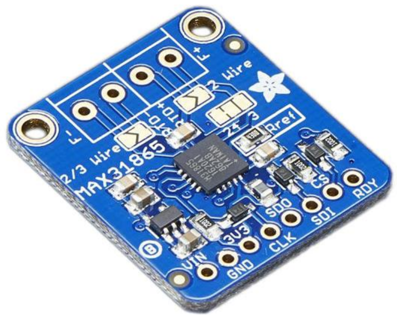


Figure 4-5: Adafruit PT100 RTD Temperature Sensor Amplifier MAX31865. [15]

MAX31865 can be read using SPI connection. It could be connected to Raspberry Pi is shown in Figure 4-6. ADC resolution for MAX31685 is 15 bit, which in nominal temperature resolution is 0.03125 °C.[16] For more information about MAX31865 see datasheet in reference [16]. MAX31865 will from here be referred to as ADC.

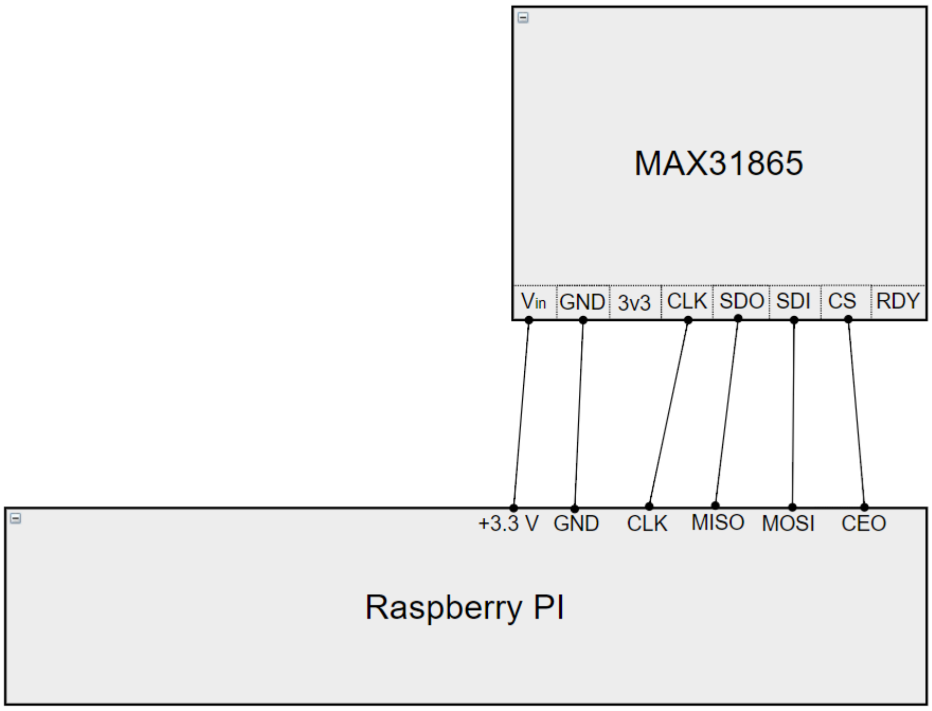


Figure 4-6: Connection between MAX31865 and Raspberry Pi.

4.2.4 Solid-State Relay and Transistor

The relay chosen to control the 12 V signal, Crydom Solid-State Relay (SSR) DC60S7, is shown in Figure 4-7. SSRs are fully electronic, there is no moving parts inside. They are fundamentally like other relays, there is complete electrical isolation between their low voltage input and, potentially high, voltage output contacts. The output is like an electrical switch, by having very high resistance when open, and a very low resistance when closed. Advantages with SSR is, among others, that it has fast response time and have high life-time expectancy.[17]

Figure 4-7: Crydom DC60S7 Solid-State Relay. [18]

The chosen SSR has minimum turn on voltage at 3.5 V[19]. The Raspberry Pi GPIOs output is 3.3 V, which isn't enough to turn on the SSR. Instead by using an NPN transistor, 1K Ω resistor, Raspberry Pi 5 V output DC power source and 3.3 V GPIO as shown in Figure 4-8, the control signal is 5 V. For more information about Crydom DC60S7 see datasheet in reference [19].

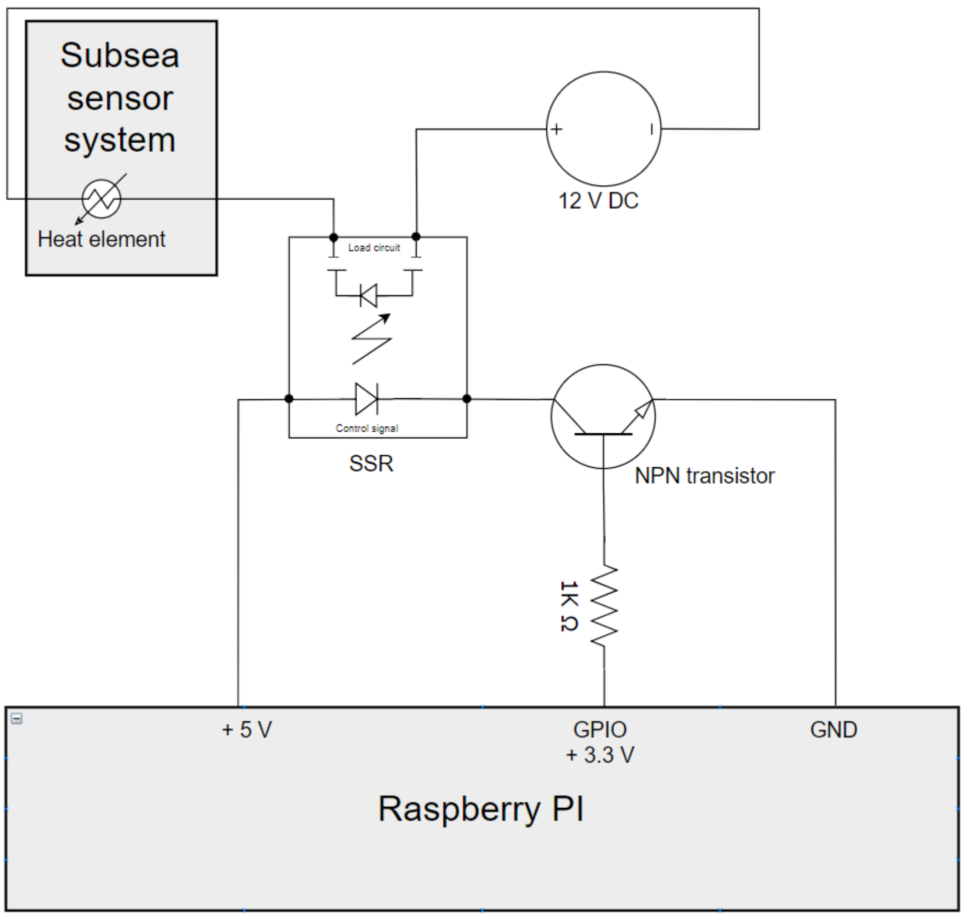


Figure 4-8: Wiring to generate a 5 volts control signal from Raspberry Pi, using NPN transistor and 1K Ω resistor.

4.3 Assembling the Prototype

This chapter is about assembling the physical prototype with the parts mentioned in previous subchapter. It includes soldering, electrical wiring, with testing of connections and circuit logic, as well as fitting it into the pressure vessel.

4.3.1 Soldering the ADC

The ADC came without the headers connected to the terminal block. To secure good connection these had to be soldered, as shown in Figure 4-9. The soldering where verified by connecting the amplifier to the temperature sensor and Raspberry Pi, as shown in Figure 4-10. Then run a test program on the Raspberry Pi to read the temperature. The temperature read was compared to another verified temperature sensor. This verified the soldered connections and that the amplifier and temperature sensor was working properly.

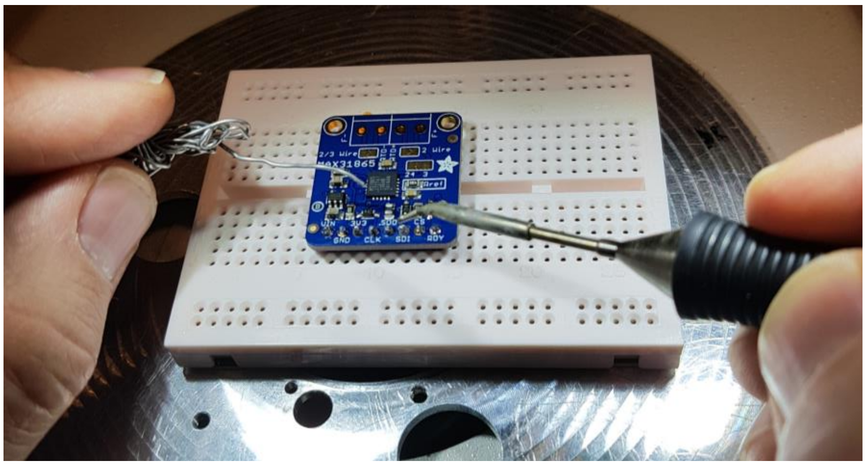


Figure 4-9. Soldering headers to the terminal block on the RTD temperature sensor amplifier.

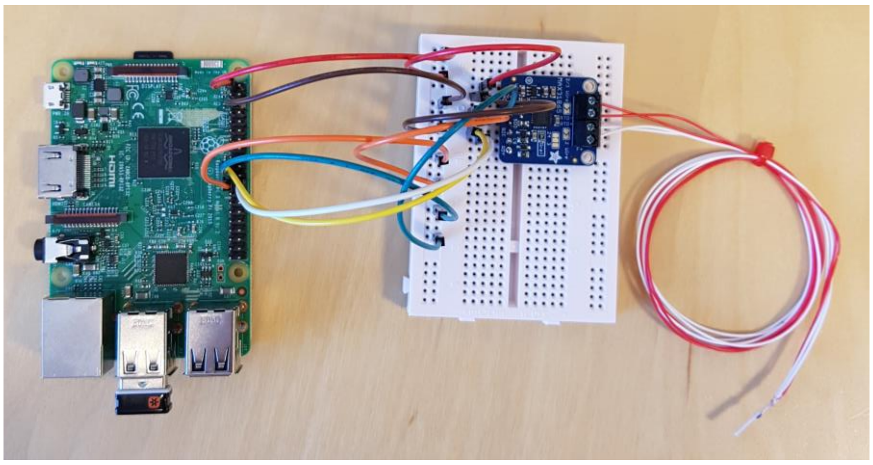


Figure 4-10: Raspberry Pi connected to the amplifier, which is attached to the PT100 temperature sensor.

4.3.2 Control Signal Circuit

The circuit for generating 5 V control signal using NPN transistor and a resistor was wired and was tested by measuring voltage across the SSR, as shown in Figure 4-11. The voltmeter reads 5.113 V when the GPIO pin is set high, as shown in Figure 4-12, and around 0 V when GPIO pin is set low. The circuit performance was concluded satisfactory, control signal of a little over 5 V isn't an issue when SSRs max control input is 32 V [19].

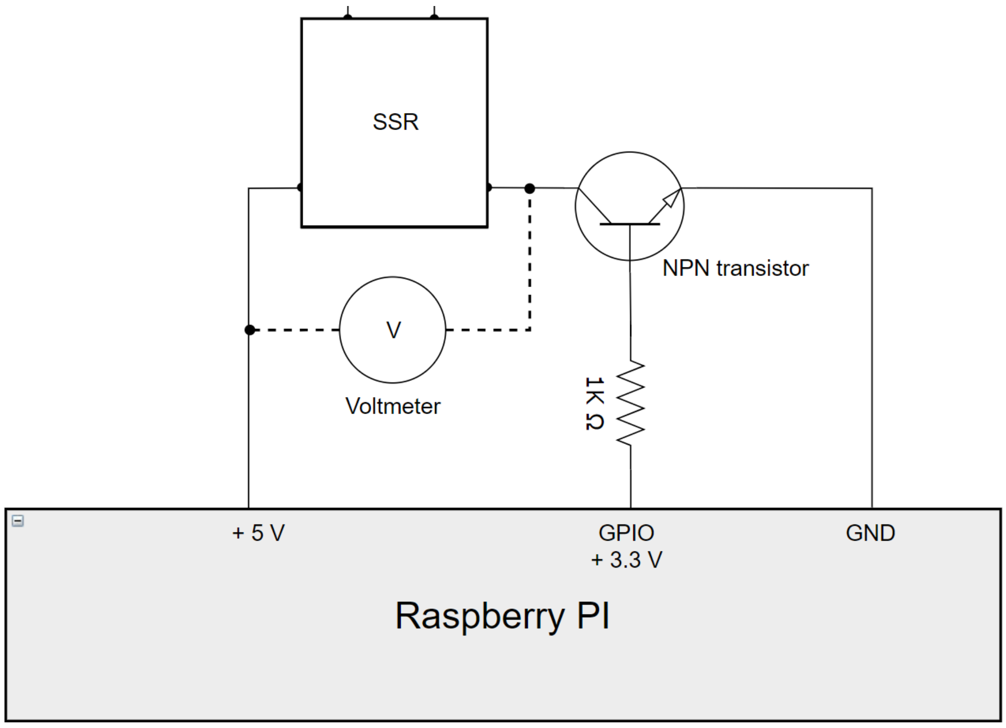


Figure 4-11: Electric circuit of how the voltmeter is connected to the control signal circuit, to read voltage across the SSR.

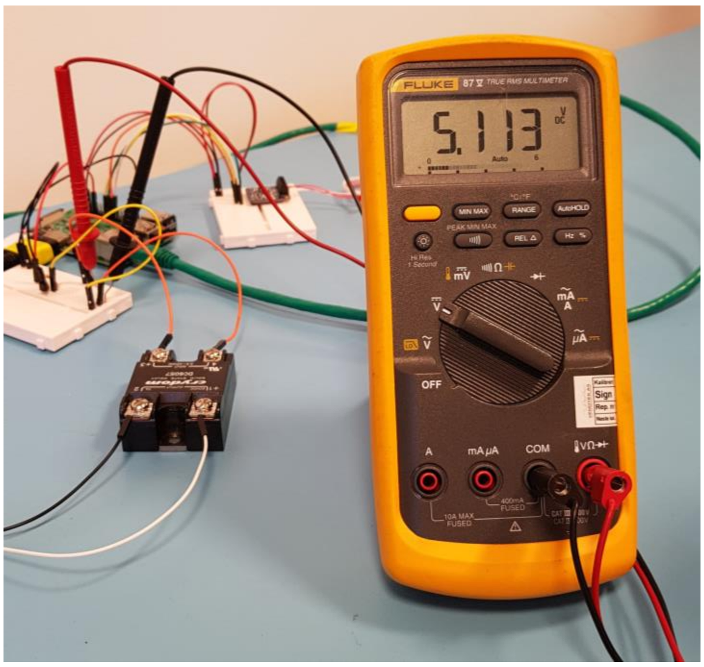


Figure 4-12: Voltmeter reading voltage across the SSR.

4.3.3 Heating compartment

The three available heat pads are silicone heater pads and will be glued to the soda can, around its circumference, to provide homogenous heating, as shown in Figure 4-13.

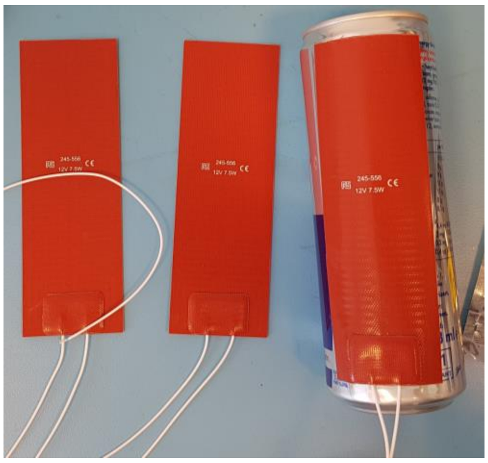


Figure 4-13: Three heat pads glued to a soda can.

There is used two 5-way conductors to connect the heat pads in parallel, shown in Figure 4-14. One of the conductors is connected to the output of the SSR, with the 12 V signal. The other is connected to ground, as shown in Figure 4-15. The circuit was verified by sending the 12 V through the SSR and then physically feeling the heating effect of each heat pad.

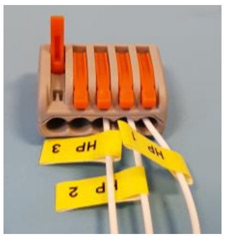


Figure 4-14: 5-way conductor connected to the three heat pads positive side.

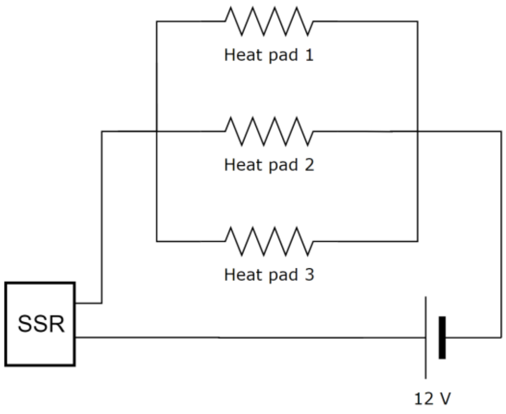


Figure 4-15: Three heat pads connected in parallel with a 12 V signal controller by a SSR.

4.3.4 Fitting Temperature Sensor

The temperature sensor will be located inside the soda can which will be placed inside the pressure vessel, as shown in Figure 4-1. The sensor is taped to wooden chopsticks to keep it approximately in the middle of the soda can, as shown in Figure 4-16.

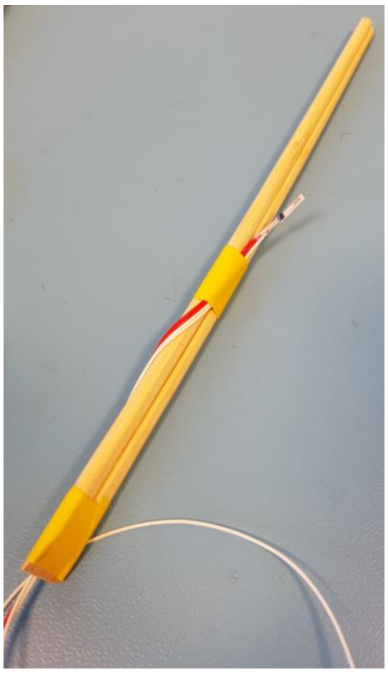


Figure 4-16: Temperature sensor taped to wooden chopsticks.

The chopsticks are put into the soda can through its opening, with a skew angle, all the way to the bottom. The chopsticks are taped to the soda can to secure the angle, as shown in Figure 4-17. Insulation material is then fitted into the soda cans opening, to ensure that heated air won’t escape easily, as shown in Figure 4-18.

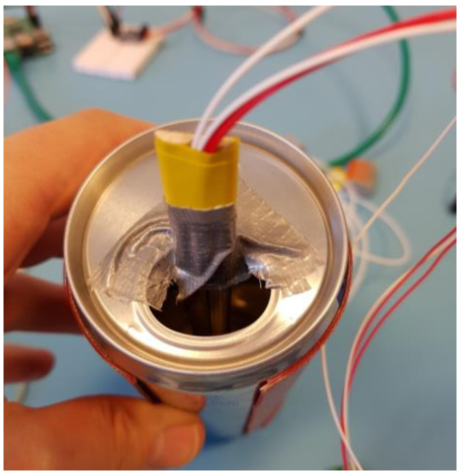


Figure 4-17: The chopsticks are taped to the soda can so it won’t move.

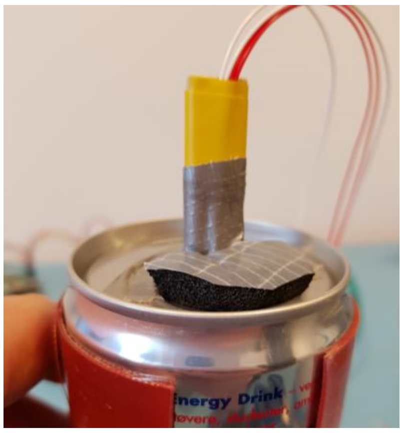


Figure 4-18: Insulation material is fitted into the soda cans opening, to ensure that heated air won’t escape easily.

4.3.5 Fitting into Pressure Vessel

The soda can, with heat pads and temperature sensor, is fasten to metal rods connected to the pressure vessel lid with bundling strap and tape, as shown in Figure 4-19. The cables for the heating circuit is strapped in the bottom, securing its location while going into the pressure vessel. All cables out of the pressure vessel needs to go through a, already existing, rubber tube in the lid, shown in Figure 4-20, to keep it waterproof.

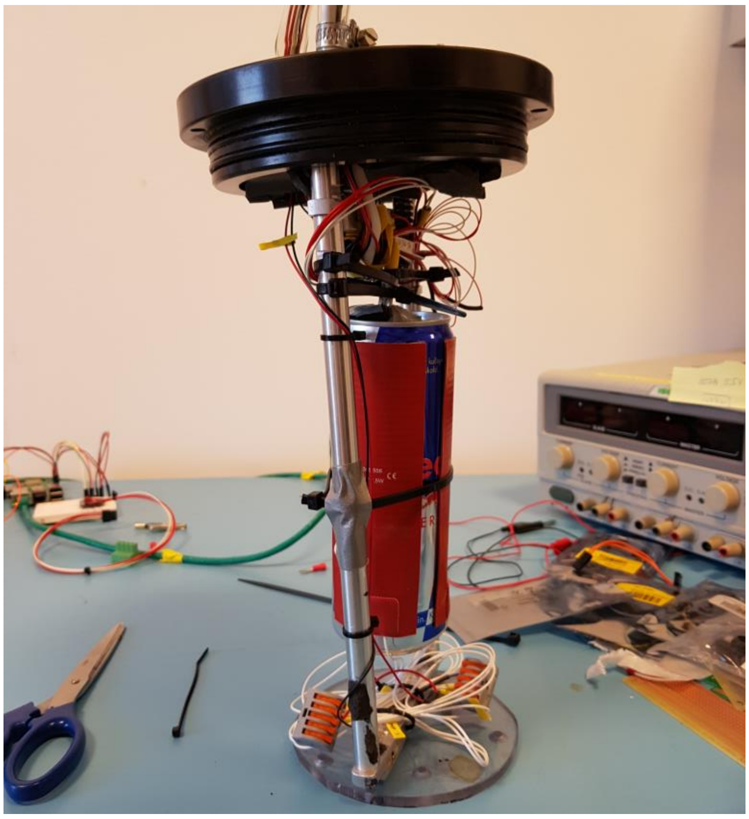


Figure 4-19: The insides of the physical prototype subsea sensor system.

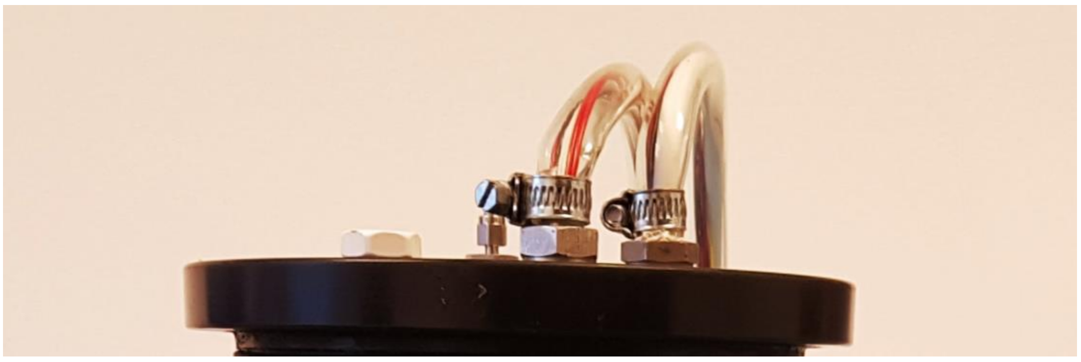


Figure 4-20: There are two rubber tubes in the lid of the pressure vessel where all the cables must go through, to keep it water proof.

Since the system will be moved around for testing in different environments, the temperature sensors cable is fitted with a connector, at outside end of the rubber tube, as shown in Figure 4-21. This way it can easily be detached from the amplifier and moved without unscrewing the cables from the terminal.

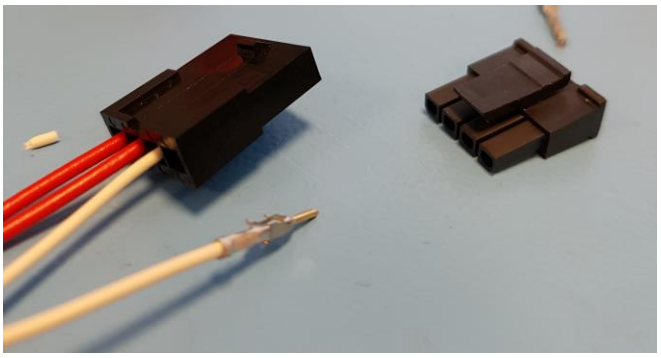


Figure 4-21: Fitting temperature sensor cables into a connector for easy detachment.

4.3.6 Connecting to Raspberry Pi

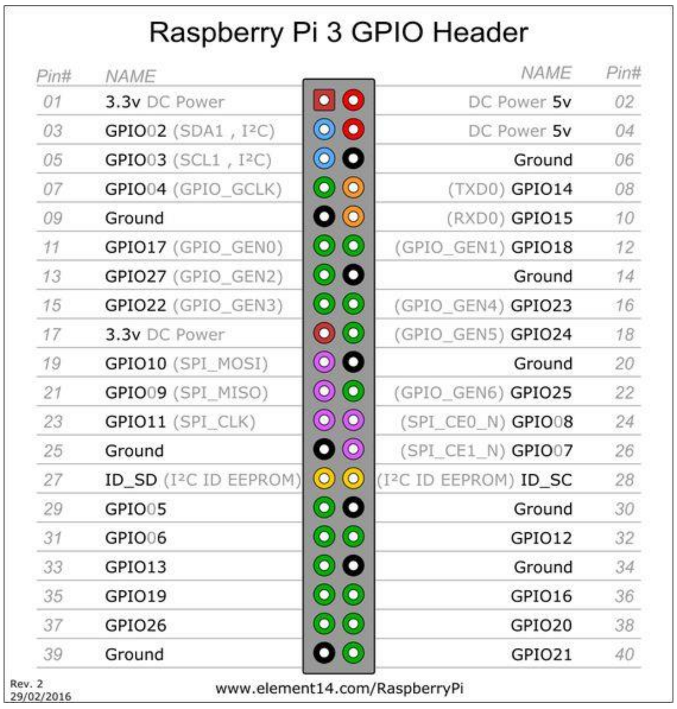
Table 4.2 and Table 4.3 presents how the Raspberry Pi is connected to RTD temperature sensor amplifier and the control signal circuit respectively, based on the Raspberry Pi pinout overview shown in Figure 4-22.

Table 4.2: Raspberry Pi pin connection to RTD temperature sensor amplifier MAX31685.

Raspberry Pi 3 Model B		**MAX31685**
Pin name:	**Pin number:**	**Name:**
3.3v DC Power	01	VIN
Ground	06	GND
GPIO11 (SPI_CLK)	23	CLK
GPIO09 (SPI_MISO)	21	SDO
GPIO10 (SPI_MOSI)	19	SDO
GPIO08 (SPI_CE0_N)	24	CS

Table 4.3: Raspberry Pi pin connection to the control signal circuit.

Raspberry Pi 3 Model B		**Component name:**	**Transistors connection side:**
Pin name:	**Pin number:**		
5v DC Power	02	SSR	Collector
GPIO17 (GPIO GEN0)	11	Resistor	Base
Ground	39		Emitter


Raspberry Pi 3 GPIO Header

Pin#	NAME	NAME	Pin#
01	3.3v DC Power	DC Power 5v	02
03	GPIO02 (SDA1 , I²C)	DC Power 5v	04
05	GPIO03 (SCL1 , I²C)	Ground	06
07	GPIO04 (GPIO_GCLK)	(TXD0) GPIO14	08
09	Ground	(RXD0) GPIO15	10
11	GPIO17 (GPIO_GEN0)	(GPIO_GEN1) GPIO18	12
13	GPIO27 (GPIO_GEN2)	Ground	14
15	GPIO22 (GPIO_GEN3)	(GPIO_GEN4) GPIO23	16
17	3.3v DC Power	(GPIO_GEN5) GPIO24	18
19	GPIO10 (SPI_MOSI)	Ground	20
21	GPIO09 (SPI_MISO)	(GPIO_GEN6) GPIO25	22
23	GPIO11 (SPI_CLK)	(SPI_CE0_N) GPIO08	24
25	Ground	(SPI_CE1_N) GPIO07	26
27	ID_SD (I²C ID EEPROM)	(I²C ID EEPROM) ID_SC	28
29	GPIO05	Ground	30
31	GPIO06	GPIO12	32
33	GPIO13	Ground	34
35	GPIO19	GPIO16	36
37	GPIO26	GPIO20	38
39	Ground	GPIO21	40

Rev. 2
29/02/2016
www.element14.com/RaspberryPi

Figure 4-22: Pinout of Raspberry Pi 3. [20]

4.3.7 Closing the Pressure Vessel

Then the lid, containing all the components, is put into the pressure vessel. The complete physical prototype of the subsea sensor system is shown in Figure 4-23.

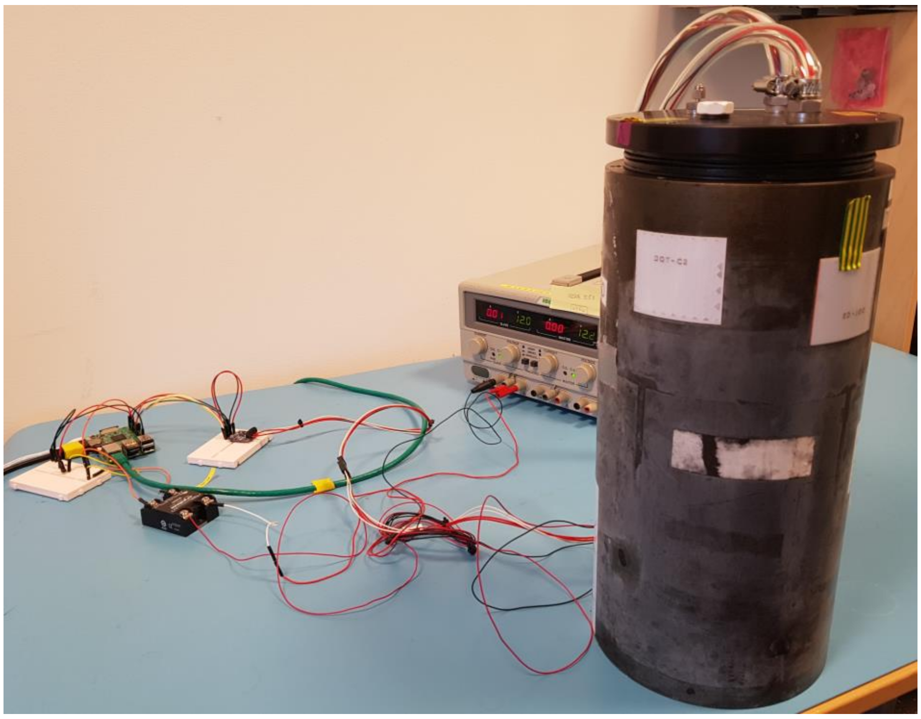


Figure 4-23: The physical prototype subsea sensor system.

5 Developing Software PID

This chapter is about the pre-preparation of the software development and the finished application. Presenting the planning techniques as well as snippets of the code.

5.1 Application planning

This chapter presents the basic ideas for the program and the following subchapters shows the planning drafts in the startup phase of software development. By using the planning tools FURPS and Use Case Diagram.

System overview is shown in Figure 5-1. The PID application retrieves parameters from the GUI and sends process status to it. It also reads information from the process and writes new control signal to the process.

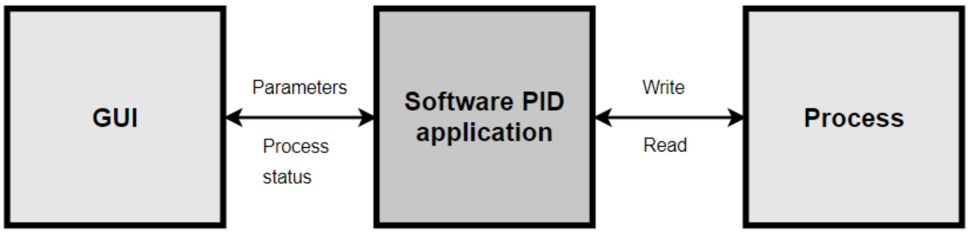


Figure 5-1: System overview from GUI to the application to the process.

Figure 5-2 explains the different components to be connected to the system. For information about the *Control signal circuit*, *Subsea sensor system* and *ADC* see chapter 4.

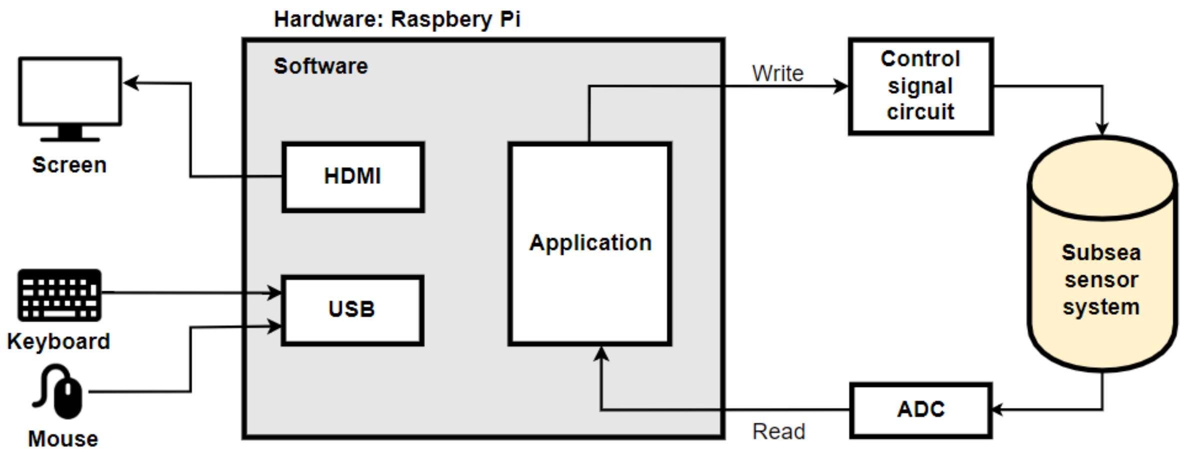


Figure 5-2: System overview of main components in the prototype subsea sensor software PID control system.

5.1.1 FURPS

FURPS is a model for classifying the functionality and requirements of a software program and is short for Functionality, Usability, Reliability, Performance and Supportability.[21] The FURPS is developed based on the task description and cooperation with external supervisors from OCTIO. The FURPS focus is on functionality and usability and it is shown in Table 5.1.

Table 5.1: FURPS

F:	Read Temperature: Establish connection to ADC. Retrieve temperature information from ADC using SPI.
	Activate Heating: Sets correct GPIO high, using GPIO library, to activate the SSR.
	PID controller: Regulating heating based on temperature readings.
	Monitor and Alter: Monitoring temperature and controller output and alter PID parameters.
U:	Application Language: English.
	System: Raspbian.
	Displaying current temperature, controllers output, sample time and PID parameters.
R:	Running for several days.
P:	Performance in temperature stability.
S:	-

5.1.2 Use Case Diagram

A Use Case Diagram, shown in Figure 5-3, is created to visualize the functionality described in the FURPS. It describes how the different system parts are connected.

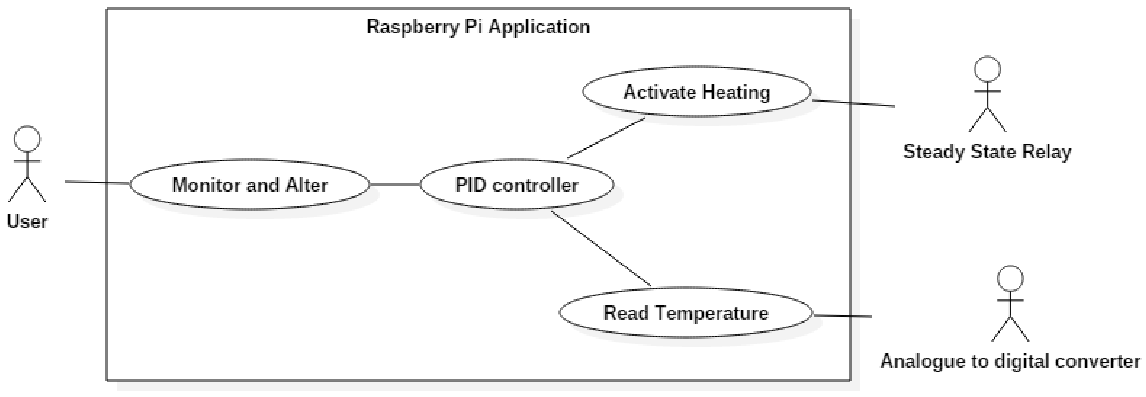


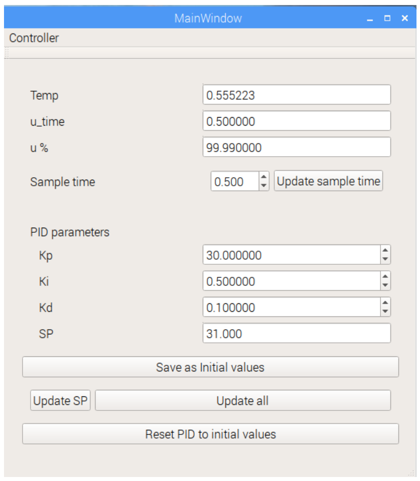
Figure 5-3: Use Case diagram for the PID controller software.

The *User* is a person with access to the system. *User* is connected to *Monitor and Alter* where it's possible to view current temperature and controller output. By receiving those values from the *PID controller* connection to *Monitor and Alter*. This connection also gives the *User* possibility to adjust the PID parameters.

PID controller is connected to *Read Temperature* where temperature information is retrieved from the ADC using SPI. It's also connected to *Activate Heating*, which turns on the SSR controlling the power to the heat pads.

5.2 Graphical User Interface (GUI)

The GUIs main purpose is give an easy way to adjust control parameters and to monitor the process. Instead of changing the code between tests and reading temperature values from the Terminal window. All the applications functionalities are gathered on one page, shown in Figure 5-4.



Temp 0.555223
u_time 0.500000
u % 99.990000
Sample time 0.500
Kp 30.000000
Ki 0.500000
Kd 0.100000
SP 31.000

Figure 5-4: The applications Graphical User Interphase.

The three topmost text fields, *Temp*, *u_time* and *u %*, are read only showing temperature, controller output in time [s] and percent respectively. They are from here denoted text output fields. The other text fields are for adjusting PID parameters and are denoted text input fields. They are updated during the startup procedure with values from a Comma Separated Values (CSV) config file containing initial values. Except from at startup, they are solely change according to user input. The functionality of the buttons is explained in Table 5.2. By clicking the *x* in the topmost right corner, the application shuts down securely with a shutdown procedure.

Table 5.2: Functionality of the buttons in the GUI.

Name:	**Functionality:**
Update sample time	Reads the text input field *Sample time* and sets the systems sample time variable equal to this value.
Save as initial values	Reads all text input fields and saves these to a CSV config file.
Update SP	Reads the text input field *SP* and sets the systems set point variable equal to this value.
Update all	Reads all text input fields and sets the systems respective variable equal to its counterpart text field value.
Reset PID to initial values	Updates all text input fields with values from a CSV config file.

5.3 Program Functionality

This chapter will explain the main functionalities of the application and includes some snippets of the code. For full code see APPENDIX

5.3.1 The Control Loop

The control loop is started in a separate thread, before initializing the GUI. The control loop functionality is explained in Figure 5-5. The control loop is a while loop inside a method called *control_loop()*. It has its own start up procedure, initializing variables and creating new CSV file for measurement data. With a unique filename of the current date and time, securing no overwrite of measurement data.

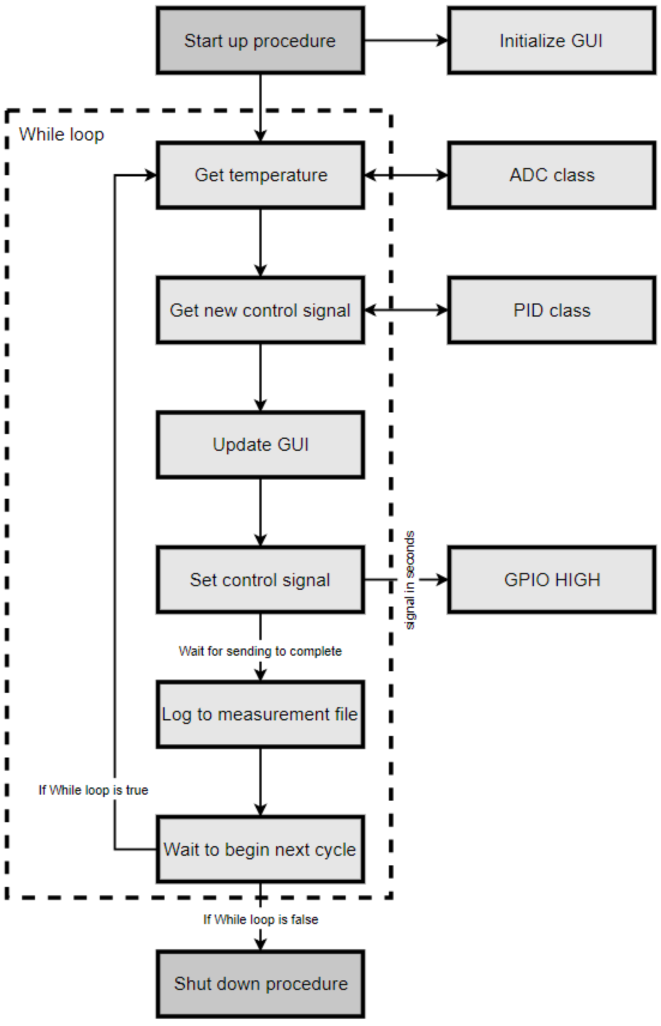
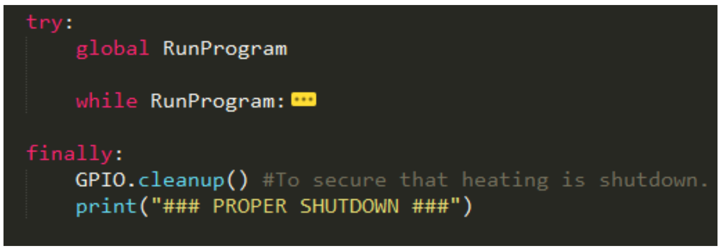


Figure 5-5: The functionality for the control loop.

The control loop is controlled by a global Boolean variable named *RunProgram*. As long as *RunProgram* is equal to TRUE the control loop continues its cycle. The control loop is enclosed by a try-finally statement, to secure proper shutdown, e.g. turning of heating. This is shown in Figure 5-6. A *finally* clause is always executed before leaving the *try* statement. To visually verify that the application is shut down properly it is printed “### PROPER SHUTDOWN ###” in the Raspberry Pi terminal.

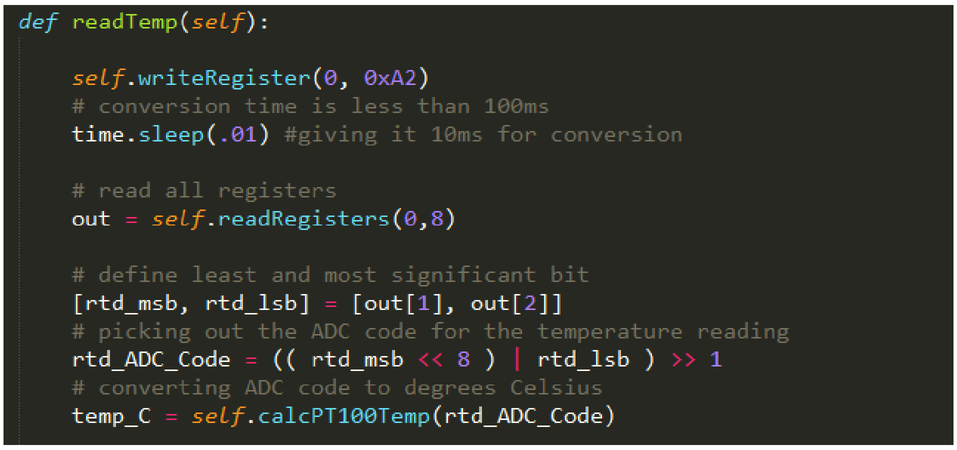


```
try:
    global RunProgram

    while RunProgram:...

finally:
    GPIO.cleanup() #To secure that heating is shutdown.
    print("### PROPER SHUTDOWN ###")
```

Figure 5-6: The try-finally statement that enclosuses the While Loop, to secure proper shutdown.

5.3.2 Reading Temperature from the ADC

For reading temperature there is created a class called ADC, this class is based on code found in reference [22]. A code snippet of the method, inside the ADC class, for reading temperature with return value is showed in Figure 5-7. The steps of this method is illustrated in Figure 5-8. In step 1 sets the clock frequency. Step 2 is the shift register data exchange. Step 3 is converting reading from Ω to °C, this formula is explained in the following subchapter. Last step, step 4 is returning the temperature value. If its detected cable faults when reading the temperature, an error message describing the cable fault will be printed in the Terminal before properly shutting down.



```
def readTemp(self):

    self.writeRegister(0, 0xA2)
    # conversion time is less than 100ms
    time.sleep(.01) #giving it 10ms for conversion

    # read all registers
    out = self.readRegisters(0,8)

    # define least and most significant bit
    [rtd_msb, rtd_lsb] = [out[1], out[2]]
    # picking out the ADC code for the temperature reading
    rtd_ADC_Code = (( rtd_msb << 8 ) | rtd_lsb ) >> 1
    # converting ADC code to degrees Celsius
    temp_C = self.calcPT100Temp(rtd_ADC_Code)
```

Figure 5-7: Snip of code from readTemp method in the ADC class.

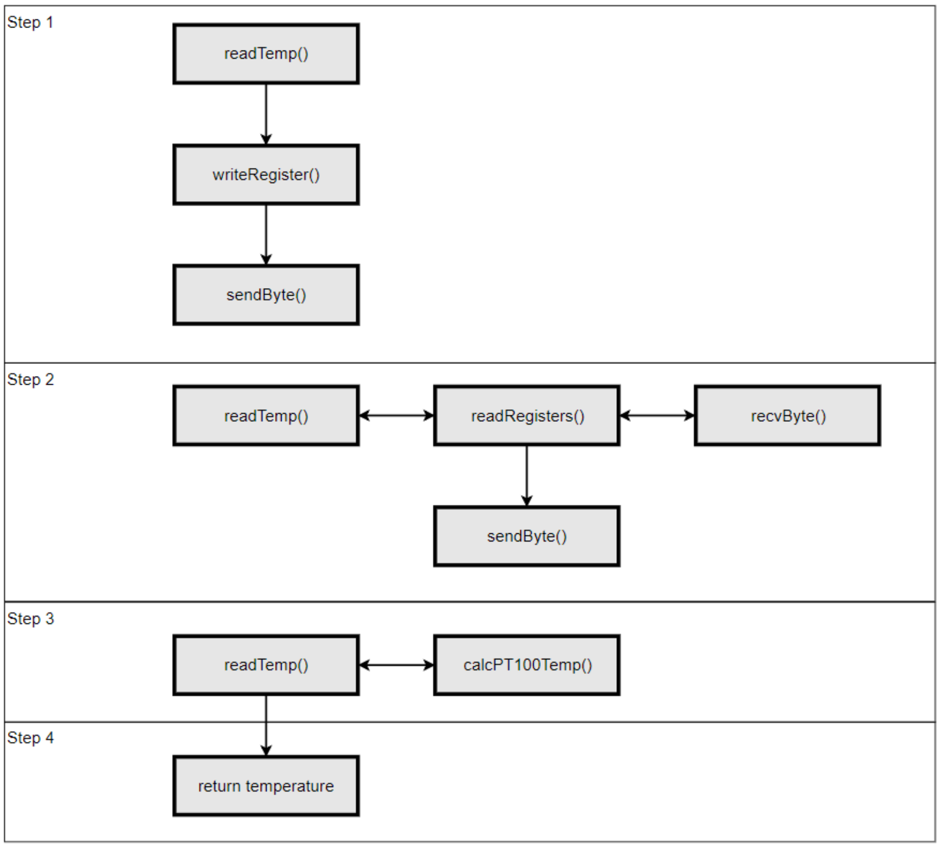


Figure 5-8: The steps for reading temperature.

5.3.3 Converting Resistance to Temperature

This subchapter is based on reference [16]. For converting the resistance to temperature, the Callendar-Van Dusen equation, shown in equation (5.1), can be used. The resistance vs temperature curve isn't completely linear, which the Callendar-Van Dusen equation describes. A straight-line approximation would give error 0 °C at 0 °C, but for 100 °C the error is -1.4 °C and error -1.75 °C for temperature -100 °C.

$$R(T) = R_0(1 + a\,T + b\,T^2 + c\,(T - 100)T^3 \tag{5.1}$$

Where T is temperature in °C, R(T) is resistance at T and R_0 is resistance at T equal to 0 °C. The Callendar-Van Dusen coefficients value are shown in equation (5.2).

$$\begin{gathered} a = 3.90830 \cdot 10^{-3} \\ b = -5.77500 \cdot 10^{-7} \\ c = 0\ for\ 0°\ C \le T \le 850°C\ and \\ c = -4.18301 \cdot 10^{-12}\ \ for - 200°\ C \le T \le 0°C \end{gathered} \tag{5.2}$$

Callendar-Van Dusen solved for temperature, in the range 0 to 850 °C, is shown in equation (5.3).

$$T = (-R_0 \cdot a + \frac{((R_0 \cdot a)^2 - 4 \cdot R_0 \cdot B \cdot (R_0 - R(T)))^{\frac{1}{2}}}{2} \cdot R_0 \cdot B \tag{5.3}$$

R(T) is found by using equation (5.4).

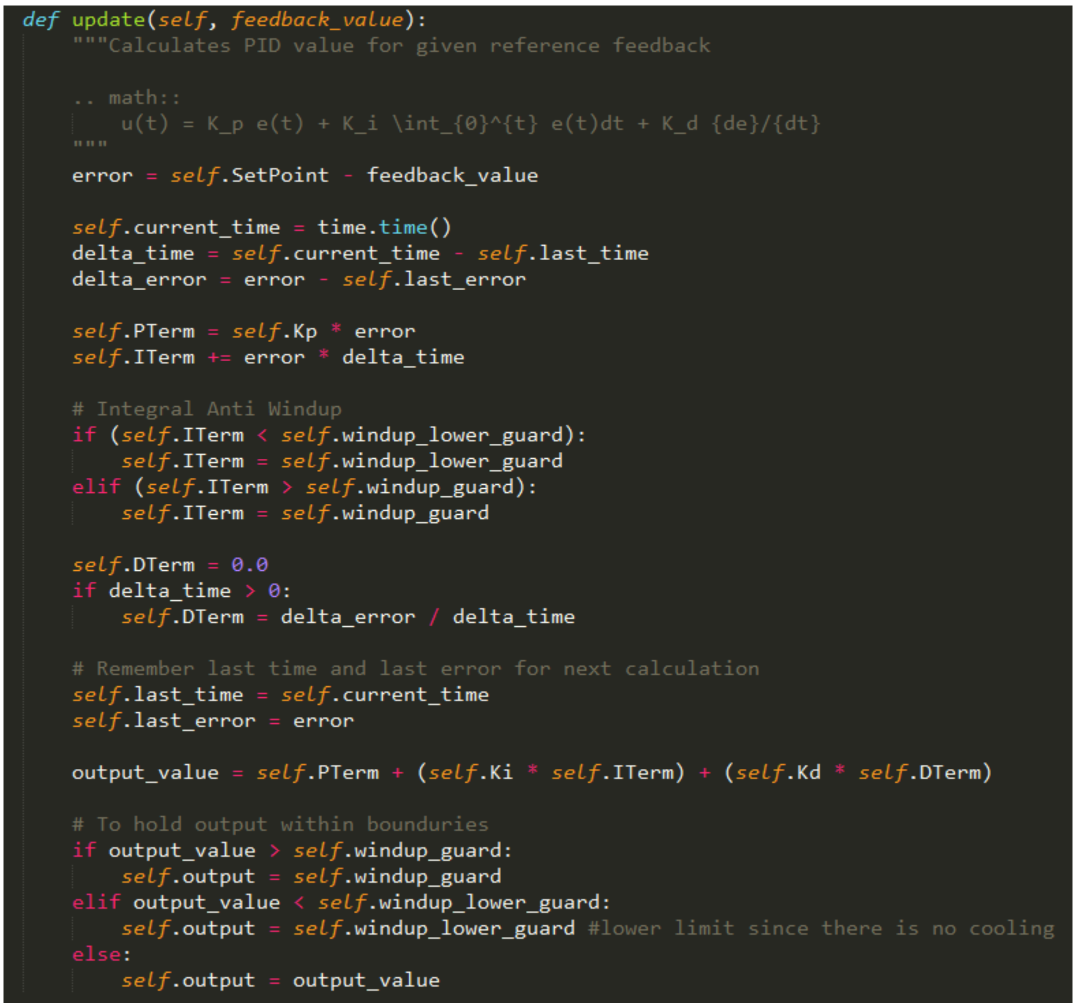
$$R(T) = \frac{ADC_{code} * R_{ref}}{FS - ADC_{code}} \tag{5.4}$$

Where FS is the ADC's full-scale code, ADC_{code} is the ADC's output code and R_{ref} is the reference resistor. For temperatures under 0 °C the straight-line approximation will be used, as shown in equation (5.5).

$$T = \left(\frac{ADC_{code}}{32}\right) - 256 \tag{5.5}$$

5.3.4 PID Class

The PID class is based on the PID code in reference [23]. Figure 5-9 shows the code for the method with the PID algorithm. For explanation of the algorithm see chapter 2.2. This method is called each Control loop with an updated temperature value. There is implemented Anti Windup by keeping the I-term within maximum and minimum values. By using if and and elif, else if, statements to check if I-term is crossing these limits. For example if the I-term exceeds the upper limit, it is sat equal to the upper limit. The *output_value* is as well checked and if it has outside the saturation limits it will be sat equal to the respective saturation limit it has breached.



```python
def update(self, feedback_value):
    """Calculates PID value for given reference feedback

    .. math::
        u(t) = K_p e(t) + K_i \int_{0}^{t} e(t)dt + K_d {de}/{dt}
    """
    error = self.SetPoint - feedback_value

    self.current_time = time.time()
    delta_time = self.current_time - self.last_time
    delta_error = error - self.last_error

    self.PTerm = self.Kp * error
    self.ITerm += error * delta_time

    # Integral Anti Windup
    if (self.ITerm < self.windup_lower_guard):
        self.ITerm = self.windup_lower_guard
    elif (self.ITerm > self.windup_guard):
        self.ITerm = self.windup_guard

    self.DTerm = 0.0
    if delta_time > 0:
        self.DTerm = delta_error / delta_time

    # Remember last time and last error for next calculation
    self.last_time = self.current_time
    self.last_error = error

    output_value = self.PTerm + (self.Ki * self.ITerm) + (self.Kd * self.DTerm)

    # To hold output within boundaries
    if output_value > self.windup_guard:
        self.output = self.windup_guard
    elif output_value < self.windup_lower_guard:
        self.output = self.windup_lower_guard #lower limit since there is no cooling
    else:
        self.output = output_value
```

Figure 5-9: Snippet of the PID algorithm code.

6 Testing and Tuning Prototype

This chapter will present PID tests on the simulator and on the prototype. The tests are conducted to find the PID parameters which will provide the best performance in temperature stability.

6.1 Testing Anti Windup and Sample Time Values

In the following subchapters different Anti Windup and Sample Time values will be tested and compared.

6.1.1 Testing different Anti Windup Boundaries

The tests were conducted in an air environment of 25 °C. During the tests of different anti windup boundaries all other parameters were kept constant, these can be seen in Table 2.1. The test results are shown in Figure 6-1, Figure 6-2, Figure 6-3 and Figure 6-4, where the anti windup upper limit was decreased between each test. In the controller output is shown as a blue line rising from the bottom. From the figures it can be seen that the first test is the only one standing out. It has longer time between peak values, as well as larger overshoot and a larger undershoot.

From the plots it can also be seen a trend, that the lower upper anti windup saturation limit gives higher frequency in the control signal oscillation. It’s decided to go forward with anti windup boundaries from 0 to 80, presented in Figure 6-3, since that was the one keeping closest to Set Point.

Table 6.1: Constant PID parameters during Anti Windup boundary test.

PID Parameter:	**Value**
K_p	0.2
K_i	1.7 [s]
K_d	0.000001 [s]
Set Point	31°C
Sample Time	0.5 [s]

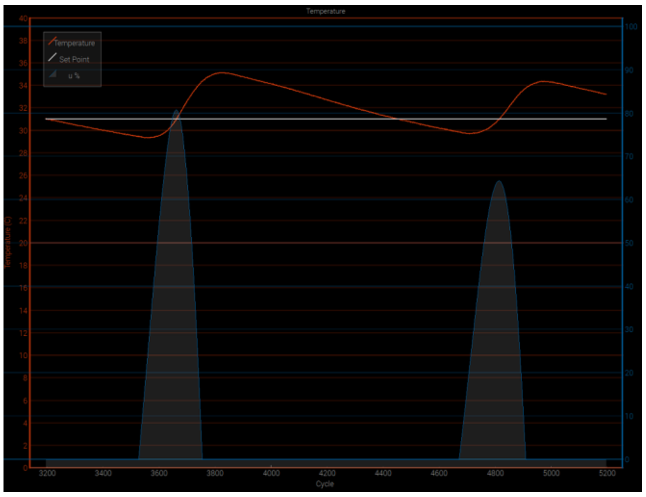


Figure 6-1: Windup boundary test 1, with saturation limit from 0 to 180.

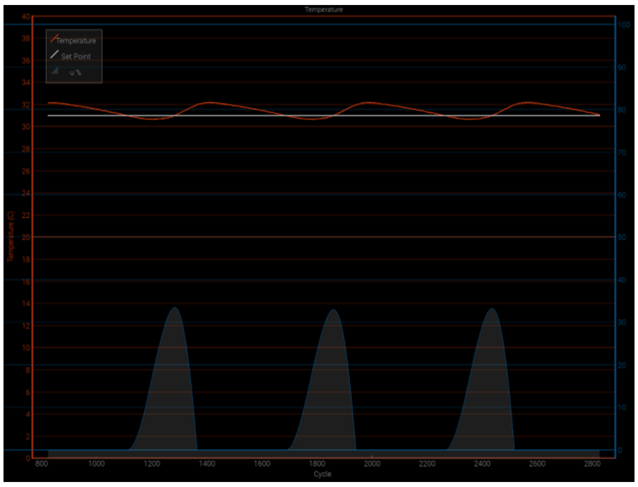


Figure 6-2: Windup boundary test 2, with saturation limit from 0 to 100.

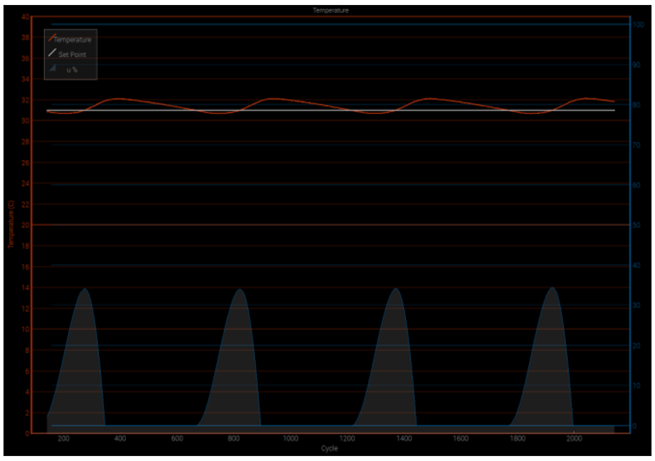


Figure 6-3: Windup boundary test 3, with saturation limit from 0 to 80.

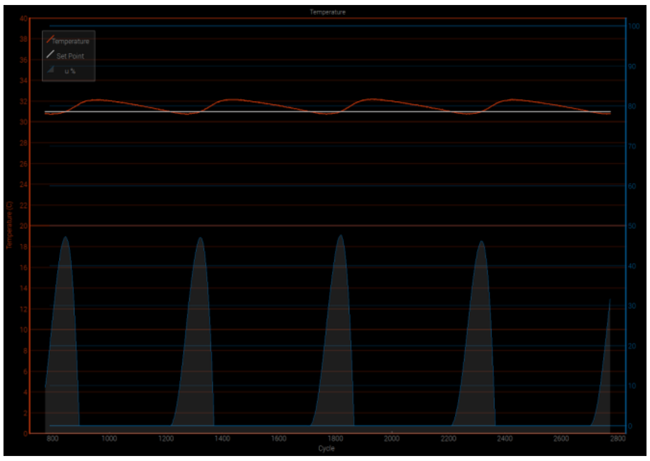


Figure 6-4: Windup boundary test 4, with saturation limit from 0 to 30.

6.1.2 Sample Time

The test was done in air environment at around 25 °C. During the tests of different sample times all other parameters were kept constant, these can be seen in Table 6.2: Constant PID parameters during Sample Time test.Table 6.2. The tests are presented in Figure 6-5, Figure 6-6 and Figure 6-7 with Sample Time values 0.25 [s], 0.5 [s] and 1 [s] respectively. The performance in temperature stability is equal for all the values tested. It was decided to use Sample Time equal to 0.5 [s].

Table 6.2: Constant PID parameters during Sample Time test.

PID Parameter:	**Value**
K_p	0.2
K_i	1.7 [s]
K_d	0.000001 [s]
Set Point	31°C
Anti Windup	0 to 80

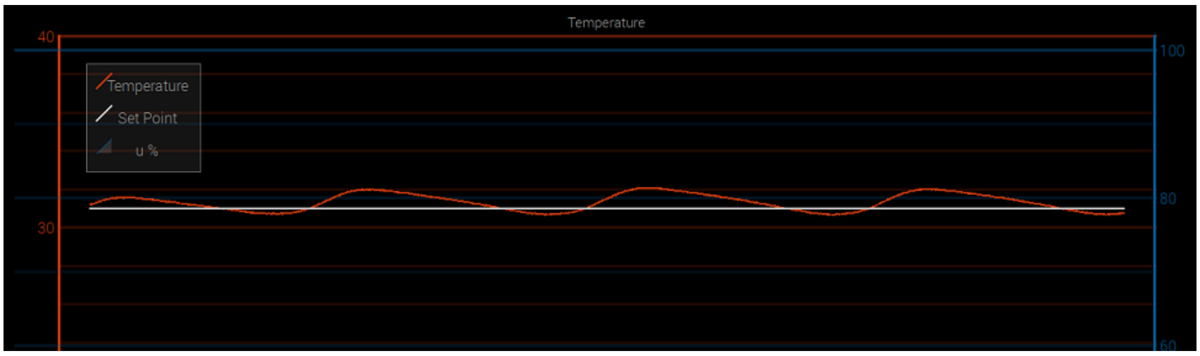


Figure 6-5: Sample Time test 1, with Sample Time equal to 0.25 [s].

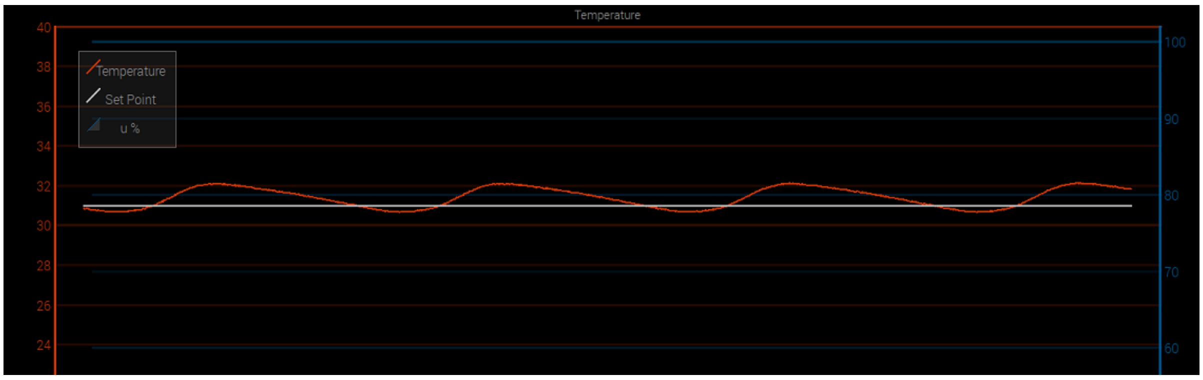


Figure 6-6: Sample Time test 2, with Sample Time equal to 0.5 [s].

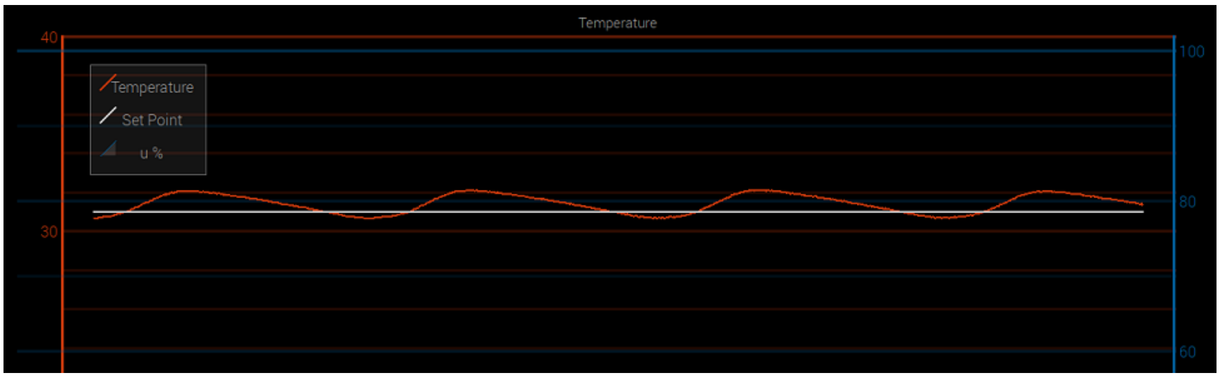


Figure 6-7: Sample Time test 3, with Sample Time equal to 1 [s].

6.2 Tuning PID on Simulation

Simulation was tuned using Try and Error method. Started with an initial guess of K_p equal to 0.1 and K_d equal to 0.000001. Also using the values for Anti Windup and Sample Time found previously. There were systematically tried different values for K_i in a Set Point step response from 26 °C to 31 °C. Environment temperature was set to 25 °C and self heating variable H_c was sat to 0. Since the prototype doesn't have any self heating from electrical components.

After settling on a K_i equal to 0.2, K_p was systemically tested and so on K_d. PID parameters found was K_p equal to 0.1, K_i equal to 0.2 and K_d equal to 0.001. The simulated step response with these parameters is shown in Figure 6-8.

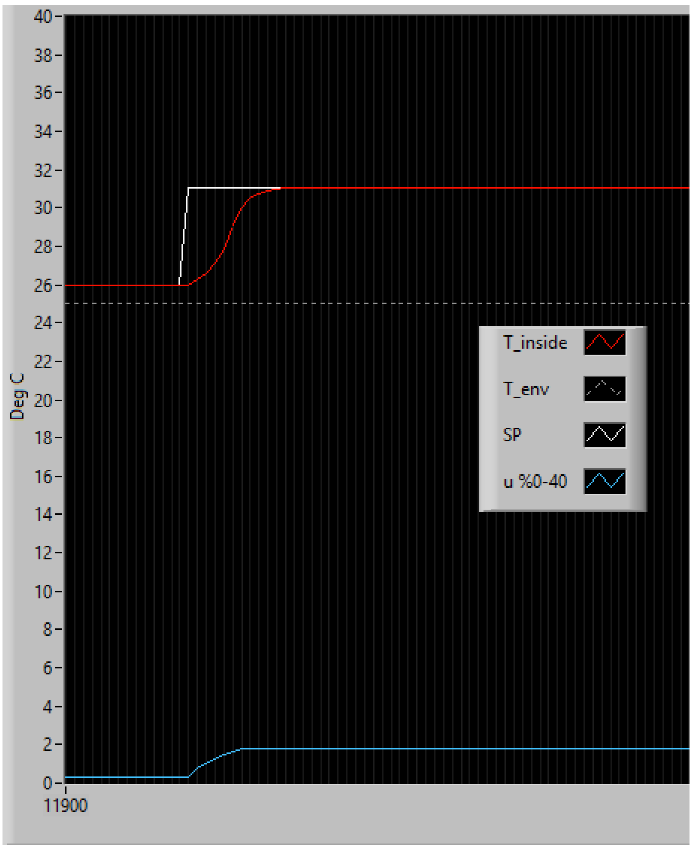


Figure 6-8: Step response in Set Point from 26 to 31 on Simulation.

6.3 Prototype Tuning

The PID parameters found for PID on the simulation is tested on the prototype under similar conditions. Set Point step response from 26 °C to 31 °C in air environment at 25 °C. The results are presented in Figure 6-9 and is not providing satisfactory temperature stability.

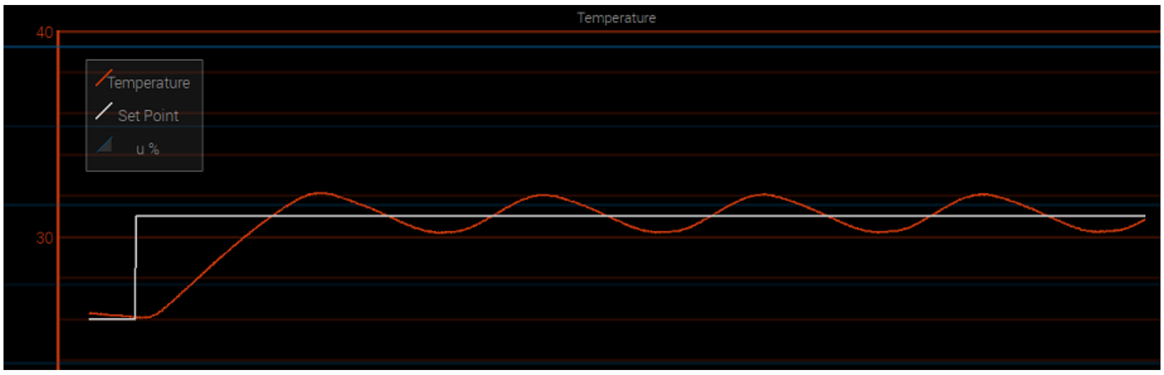
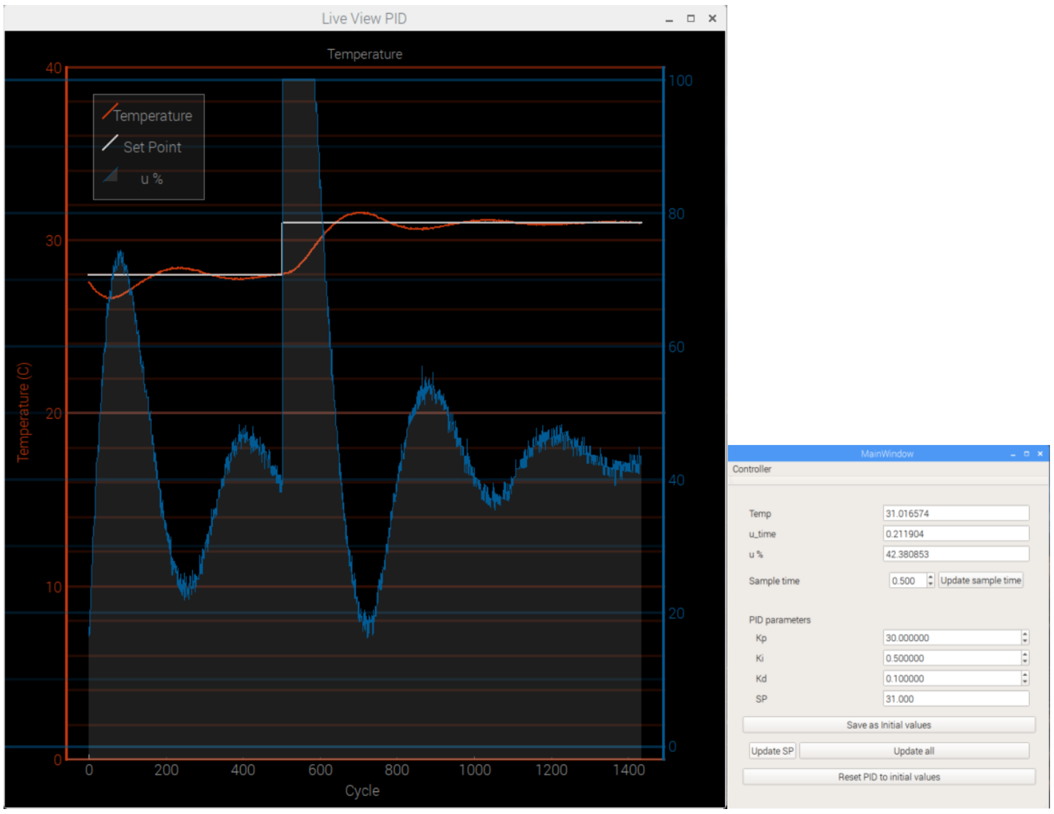


Figure 6-9: Simulation tuned PID parameters tested on prototype by step response in Set Point from 26 °C to 31 °C. The test is conducted in air environment at 25 °C.

Further testing was done by doing step response Set Point in 25 °C air, 3 °C water and air 0 °C. Temperature stability during a change in environment from air 25 °C to water 3 °C and from air 25 °C to air 0 °C. PID parameters found was K_p equal to 30, K_i equal to 0.5 and K_d equal to 0.1. A set point step response with these parameters in environment 0°C is shown in Figure 6-10.



Temp	31.016574
u_time	0.211904
u %	42.380853
Sample time	0.500
PID parameters	
Kp	30.000000
Ki	0.500000
Kd	0.100000
SP	31.000

Figure 6-10: Step response 0°C environment.

Before reaching the parameters, shown in the figure above, it was conducted an environment disturbance test from air to water 3 °C shown in Figure 6-11. The parameters in this test was Kp = 30, Ki = 0.29 and Kd = 0.01. It's pretty obvious after making the plot in excel that it converges into a steady-state error but it wasn't noticed from the *LivePlot*. Therefor these parameters were further tested in air environment of 0 °C, were the steady-state was clearer as shown in Figure 6-12.

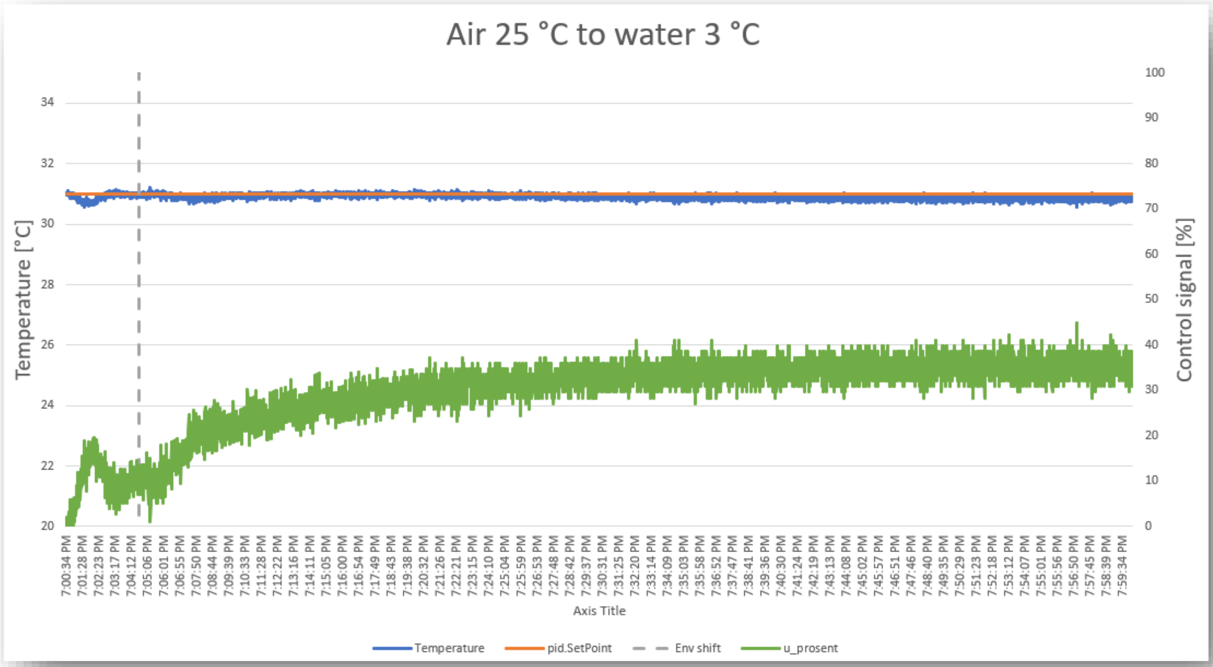


Figure 6-11: Disturbance test, from air 25 °C to water 3 °C. Changes at grey line.

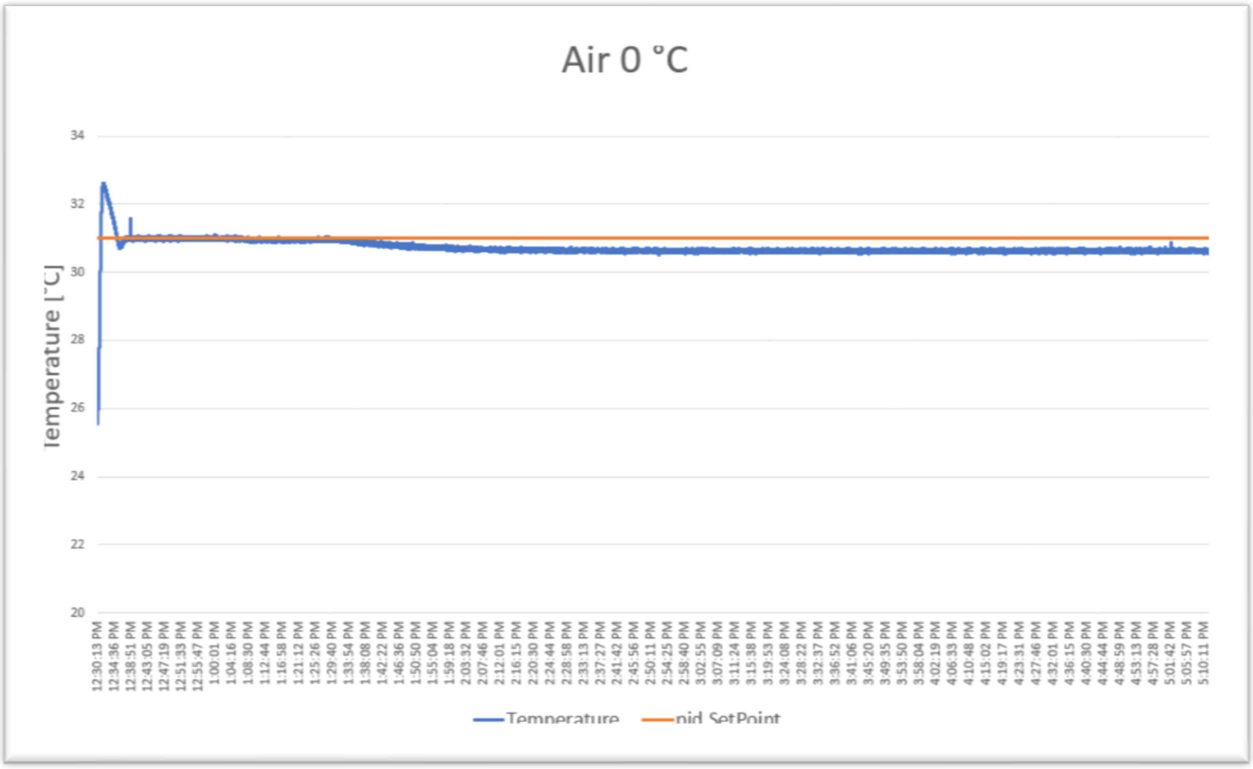


Figure 6-12: PID parameter test in 0 °C air environment.

6.3.1 Disturbance in Environment test.

A last test with environment change every 12 minutes over 3 hours timespan was conducted, as shown in Figure 6-14. The test is based on heat loss development from a test shown in Figure 6-13. The environment was changed between air 22 °C to air 0 °C. Initial system temperature was 22.6 °C. The test was started by putting the prototype into air 0 °C from air 22 °C. The performance of temperature stability was +/- 0.1 °C. The first 15 minutes of the test is shown in Figure 6-15. The error compared with environment change is shown in Figure 6-16, worth mentioning that the largest error peak is confirmed as noise. The other 2 peaks are probably also noise but needs more data to confirm, or implement a filter and see if it still occurs. Figure 6-17 shows the correlation between environment change and control signal.

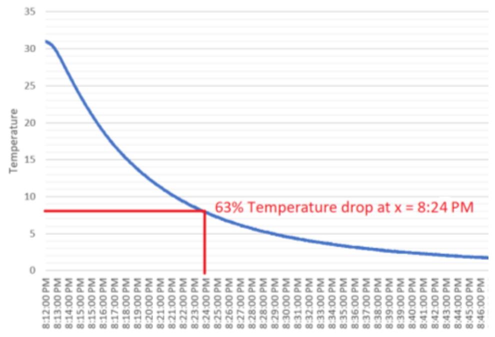


Figure 6-13: Heat loss development from 31 °C internal temperature, in 0 °C air environment. Conducted by turning off heating after being stable at 31 °C.

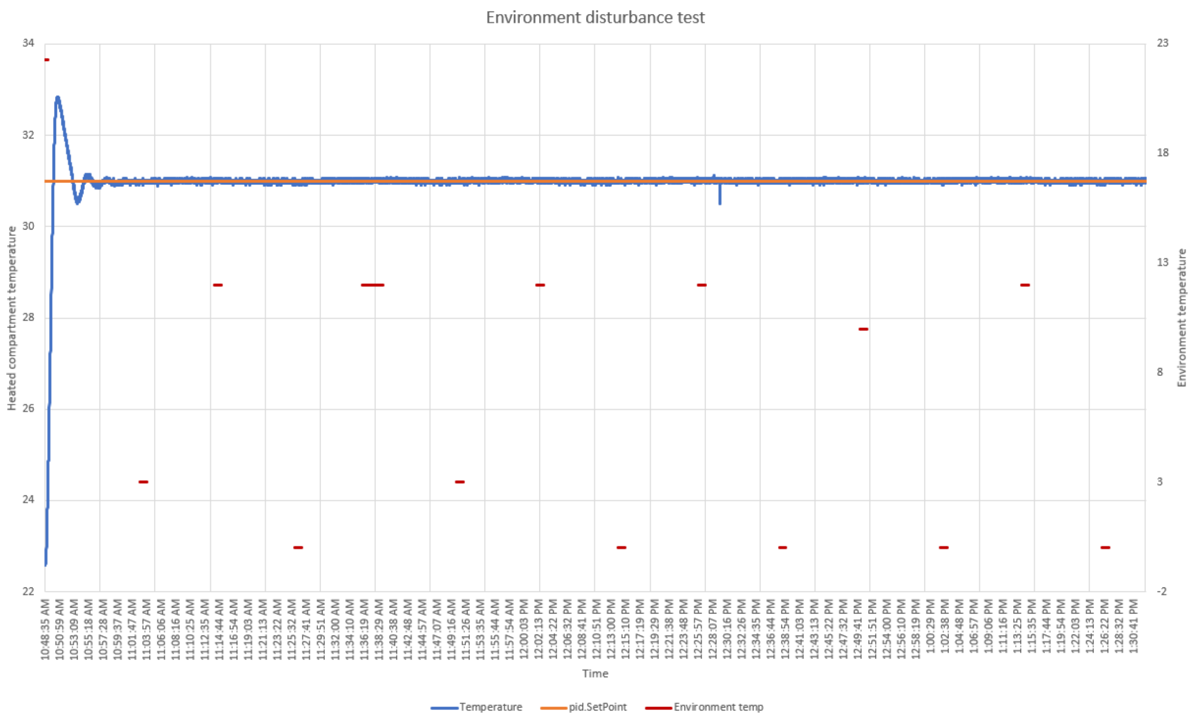


Figure 6-14: Prototype PID parameter test, by every 12 minutes changing between air 22 °C to air 0 °C. The test was conducted over 3 hours. The environment temperature is showed in red and is collected at time the change occurs.

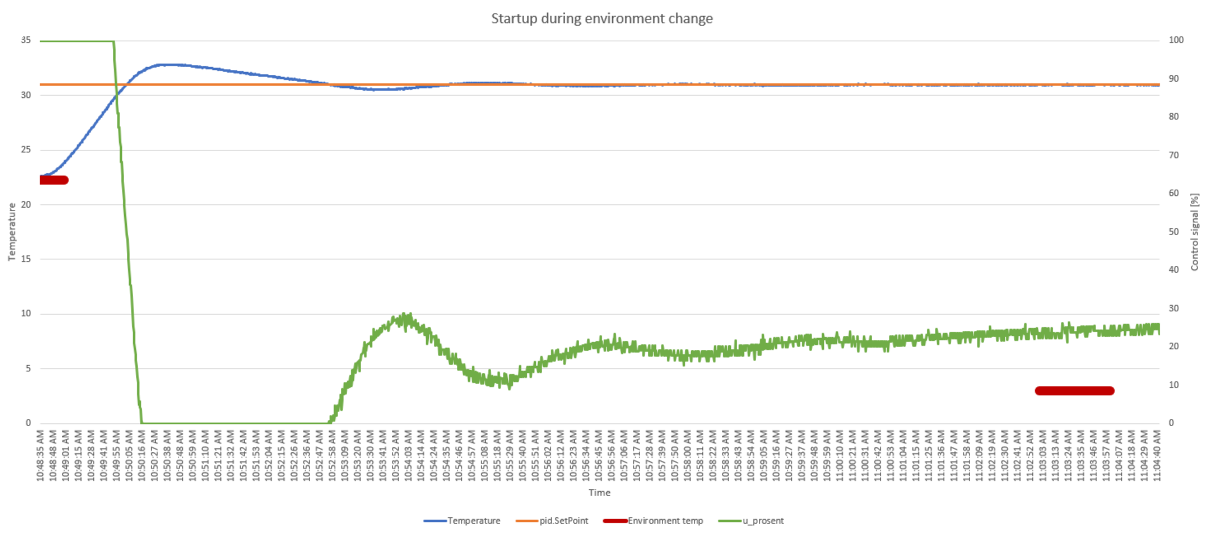


Figure 6-15: The first ~15 minutes of the test.

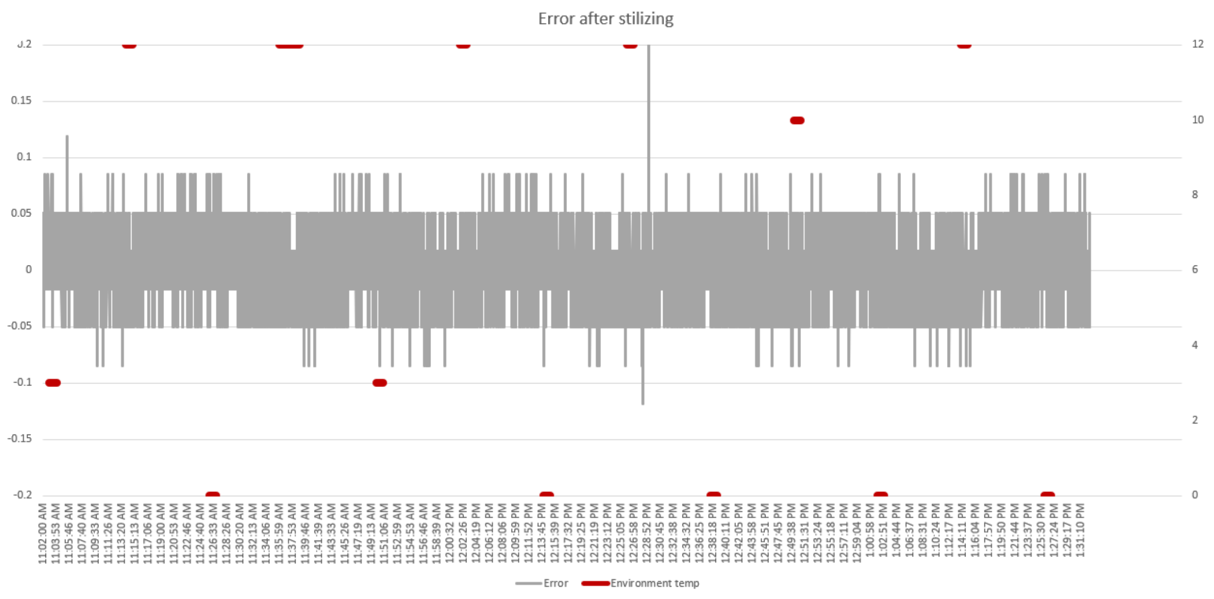


Figure 6-16: Error from setpoint compared with sampled environment temperature.

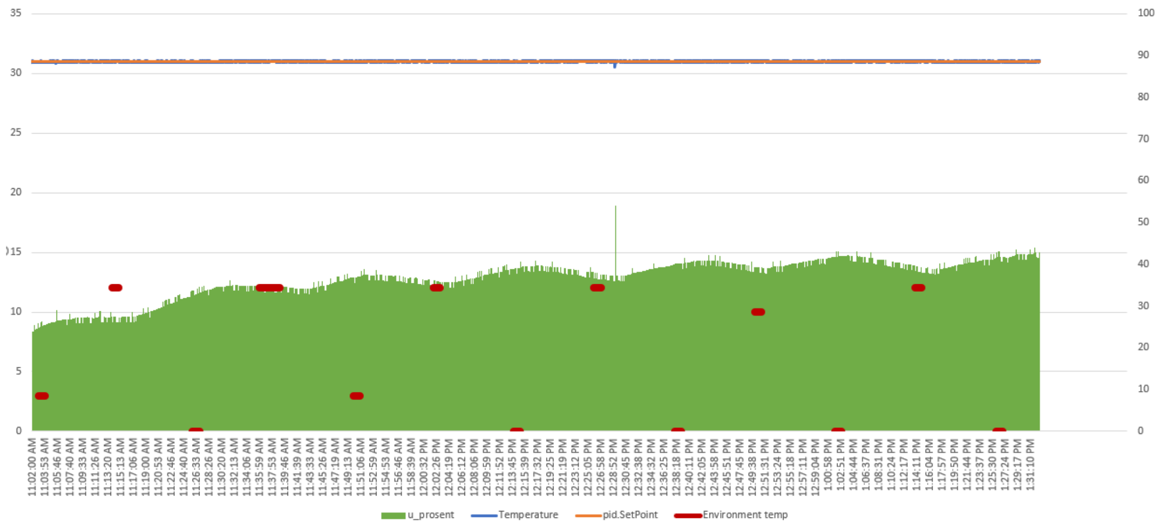


Figure 6-17: Control signal versus environment temperature.

7 Conclusion and Future Works

Developed software PID system worked as intended and proved good performance in temperature stability compared for subsea application. But before actual implementation more test would be needed.

The Raspberry Pi 3 as a SOC device for software PID implementation worked as intended. The ADC worked as intended.

The building of a Physical prototype subsea sensor system was as intended.

The simulation wasn’t good enough to be used for tuning, but there was correlation to the real application, so it might be a good starting point.

7.1 Improving Simulation

Heat response is very fast. Adding some sort of delay to the heat added (P_in) might solve this. Another possible solution is to use known data from real application as parameters A, B and C, since it simulation of a differential equation. E.g. : A x + B u + C.

7.2 Extending Prototype Software PID

- More water tests to confirm performance in temperature stability.
- Test software PID on real subsea sensor system.
- Extend to Cascade control system.
- Activate plot from GUI.
- Improve error handling. Only handling the most probable and important, errors at this moment.
- Add a measurement filter.
- Clean up and improve code. To increase computational speed and readability.

7.3 Developing for the Real System

Since C++ is needed for the real system, the options are:

- Manually convert Python to C++. E.g. find similar libraries in C++ as to the python libraries used.
- Treat Python code as prototype and start from scratch. Use the FURPS+ and USE CASE to develop the software PID in C++.

References/literature

1. Wikipedia. System on a chip [Available from: https://en.wikipedia.org/wiki/System_on_a_chip.

2. Python. What is Python? Executive Summary [Available from: https://www.python.org/doc/essays/blurb/.

3. EDGEFX. What is LabVIEW Programming and Why You Should Use it [Available from: http://www.edgefxkits.com/blog/labview-programming-language-uses/.

4. Wikipedia. Serial Peripepheral Interface Bus [Available from: https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus.

5. Wikipedia. Thread (computing) [Available from: https://en.wikipedia.org/wiki/Thread_(computing).

6. Haugen F. Reguleringsteknikk. Høgskolen i Telemark: Høgskolen i Telemark; 2012 02.02.2012.

7. Espedal TA. Temperature behaviour in subsea sensor system without effect from head pads. In: Mogård S, editor. 2018.

8. Lie B. Modeling of Dynamic Systems: Telemark University College; 2013 August 2013.

9. Wikipedia. Raspberry Pi [Available from: https://en.wikipedia.org/wiki/Raspberry_Pi.

10. RS-Online. Raspberry Pi 3 Model B SBC [Available from: https://no.rs-online.com/web/p/processor-microcontroller-development-kits/8968660/.

11. Wikipedia. Resistance thermometer [Available from: https://en.wikipedia.org/wiki/Resistance_thermometer#Four-wire_configuration.

12. RTD accuracy – Class A, Class B, 1/3 DIN, 1/10 DIN: Temperature Controls Pty Ltd; [Available from: http://www.temperature.com.au/Support/RTDSensors/RTDaccuracyClassAClassB13DIN110DIN.aspx.

13. RS-Online. RS Pro PT100 Sensor -50°C min +250°C max 10mm length x 2mm diameter [Available from: https://no.rs-online.com/web/p/platinum-resistance-temperature-sensors/8919141/.

14. adafruit. Adafruit PT100 RTD Temperature Sensor Amplifier - MAX31865 [Available from: https://www.adafruit.com/product/3328.

15. Digi-Key. Adafruit Industries LLC 3328 [Available from: https://www.digikey.com/product-detail/en/adafruit-industries-llc/3328/1528-1804-ND/6562952.

16. MAX31865 Data sheet: Maxim integrated; [Available from: https://datasheets.maximintegrated.com/en/ds/MAX31865.pdf.

17. Wikipedia. Solid-state relay [Available from: https://en.wikipedia.org/wiki/Solid-state_relay.

18. RS-Online. Sensata / Crydom 7 A SPNO Solid State Relay, DC, Surface Mount Transistor, 60 V dc Maximum Load [Available from: https://no.rs-online.com/web/p/solid-state-relays/7360873/.

19. crydom. DC60 Series [Available from: https://docs-emea.rs-online.com/webdocs/1380/0900766b813806a1.pdf.

20. element14.com. Raspberry Pi 3 GPIO Header [Available from: https://i.stack.imgur.com/RILry.png.

21. Wikipedia. FURPS [Available from: https://en.wikipedia.org/wiki/FURPS.

22. Smith SP. max31865.py [Available from: https://github.com/steve71/MAX31865/blob/master/max31865.py.

23. PID.py: Ivmech Mechatronics Ltd.; [Available from: https://github.com/ivmech/ivPID/blob/master/PID.py.

Appendices

Appendix A – Task Description

FMH606 Master's Thesis

Title: Compact temperature regulator for subsea sensors

Supervisor: Hans-Petter Halvorsen

External Partner: Octio

Task Description:

Development and implementation of a versatile and compact PID temperature controller in a SmartFusion2 SoC FPGA device, or similar ARM microcontroller, and verifying performance, temperature stability for use with a subsea pressure and gravity monitoring system.

Possible sub tasks (ranked):

1. Study and gain experience of a commercial PID controller in a "sample temperature regulation system"
2. Simulate a system model of a sample temperature regulation system including PID controller.
3. Develop and implement a SW PID controller architecture with comparable performance as the reference temperature controller.
4. Develop and implement a simple register based system for parameter setting, control and monitoring
5. Extend the system to an RTOS task (for integration into an existing SW system)
6. Extend the system to comprise a multistage (cascade) PID controller

The actual implementation can be done on a variety of ARM microcontroller platforms such as:

- Raspberry PI 3
- Arduino
- SmartFusion2 SoC FPGA
- others

The SW language should preferably be based on C/C++ or Python.

The simulation platform could be based on Matlab, Simulink or LabView.

The verification of the performance could be by comparison of the reference system with an industrial controller and the SW PID controller.

Task Background:

Commercial PID temperature controller often comes in enclosures including display or with rack mounts which are impractical for volume constrained subsea applications. Control and monitoring require a dedicated communication channel, normally RS-232 or ethernet, which adds on the complexity.

As few commercial or open core software temperature controller are available, developing a flexible SW architecture allowing integration into a general micro-controller system, is of interest.

Student Category: IIA students.

Practical Arrangements: The assignment is reserved for a student that already has a collaboration with the company in question.

OCTIO may provide a development platform (ARM). An example PID controller core (for Arduino) may be provided. Lab space and physical prototype is available for testing and/ or comparing systems.

The student needs to be located in Bergen, Norway.

Signatures:

Supervisor (date and signature):

Students (date and signature):

Appendix B – Software PID code

```
import time, math
#for filename
from time import gmtime, strftime
import RPi.GPIO as GPIO
import threading
#for GUI
import sys
#QT stuff
import pyqtgraph as pg
import PyQt5
from PyQt5 import QtCore
from PyQt5.QtWidgets import *

import numpy as np

# The window from QtCreator aka GUI
import mainwindow_auto
import csv

# create class for our Raspberry Pi GUI
class MainWindow(QMainWindow, mainwindow_auto.Ui_MainWindow):
    # access variables inside of the UI's file

    ### functions for the buttons to call
    # def pressedQuit(self):

    def pressedSaveInit(self):
        global SampleTime
        SP = gui.txtSetSP.value()
        P = gui.txtSetKp.value()
        I = gui.txtSetKi.value()
        D = gui.txtSetKd.value()
        fileName = 'pid_config.csv'
        mode = "w"
        data = []
        data = [SP, P, I, D, SampleTime]
        csv_writer(data, fileName, mode)

    def pressedUpdateSP(self):
        SP = gui.txtSetSP.value()
        newSetpoint(SP)

    def pressedUpdateAll(self):
          print("start program")
          SP = gui.txtSetSP.value()
          P = gui.txtSetKp.value()
          I = gui.txtSetKi.value()
          D = gui.txtSetKd.value()

          newAllParameters(SP,P,I,D)

    def pressedNewSampleTime(self):
        global SampleTime
        SampleTime = gui.txtSampleTime.value()

    def pressedResetPID(self):
        csv_reader('pid_config.csv')

        #event handler for when you X-out the window
    def closeEvent(self, event):
        #make sure control loop ends properly.
        global RunProgram
        RunProgram = False #This will end the while loop.

        event.accept() #Accepting closeing

    def __init__(self):
            super(self.__class__, self).__init__()
            self.setupUi(self) # gets defined in the UI file

            ### Hooks to for buttons
            self.btnSaveInitValues.clicked.connect(lambda: self.pressedSaveInit())
            self.btnUpdateSP.clicked.connect(lambda: self.pressedUpdateSP())
            self.btnUpdateAll.clicked.connect(lambda: self.pressedUpdateAll())
            self.btnResetPID.clicked.connect(lambda: self.pressedResetPID())
            self.btnNewSampleTime.clicked.connect(lambda: self.pressedNewSampleTime())

    class PID:
        """PID Controller
        """

        def __init__(self, P=0.2, I=0.0, D=0.0):

            self.Kp = P
            self.Ki = I
            self.Kd = D

            self.sample_time = 0.00
            self.current_time = time.time()
            self.last_time = self.current_time

            self.clear()

        def clear(self):
            """Clears PID computations and coefficients"""
            self.SetPoint = 0.0

            self.PTerm = 0.0
            self.ITerm = 0.0
            self.DTerm = 0.0
            self.last_error = 0.0
            # Windup Guard
            self.int_error = 0.0
            self.windup_lower_guard = 0.0
            self.windup_guard = 80.0

            self.output = 0.0

        def update(self, feedback_value):
            """Calculates PID value for given reference feedback

            .. math::
                u(t) = K_p e(t) + K_i \int_{0}^{t} e(t)dt + K_d {de}/{dt}
            """
            error = self.SetPoint - feedback_value

            self.current_time = time.time()
            delta_time = self.current_time - self.last_time
            delta_error = error - self.last_error

            self.PTerm = self.Kp * error
            self.ITerm += error * delta_time

            # Integral Anti Windup
            if (self.ITerm < self.windup_lower_guard):
                self.ITerm = self.windup_lower_guard
            elif (self.ITerm > self.windup_guard):
                self.ITerm = self.windup_guard

            self.DTerm = 0.0
            if delta_time > 0:
                self.DTerm = delta_error / delta_time

            # Remember last time and last error for next calculation
            self.last_time = self.current_time
            self.last_error = error
            output_value = self.PTerm + (self.Ki * self.ITerm) + (self.Kd * self.DTerm)

            # To hold output within boundturies
            if output_value > self.windup_guard:
                self.output = self.windup_guard
            elif output_value < self.windup_lower_guard:
                self.output = self.windup_lower_guard #lower limit since there is no cooling
            else:
                self.output = output_value



        def setKp(self, proportional_gain):
            self.Kp = proportional_gain

        def setKi(self, integral_gain):
            self.Ki = integral_gain

        def setKd(self, derivative_gain):
            self.Kd = derivative_gain

        def setWindup(self, upper_windup, lower_windup):
            self.windup_guard = upper_windup
            self.windup_lower_guard = lower_windup

        def setSP(self, set_point):
            self.SetPoint = set_point

    class ADC(object):
        """Reading Temperature from the MAX31865 with GPIO using
           the Raspberry Pi. """

            #The MIT License (MIT)
            #
            #Copyright (c) 2015 Stephen P. Smith
        #
        #Permission is hereby granted, free of charge, to any person obtaining a copy
        #of this software and associated documentation files (the "Software"), to deal
        #in the Software without restriction, including without limitation the rights
        #to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
        #copies of the Software, and to permit persons to whom the Software is
        #furnished to do so, subject to the following conditions:
        #
        #The above copyright notice and this permission notice shall be included in all
        #copies or substantial portions of the Software.

    def __init__(self, csPin = 8, misoPin = 9, mosiPin = 10, clkPin = 11):
        self.csPin = csPin
        self.misoPin = misoPin
        self.mosiPin = mosiPin
        self.clkPin = clkPin
        self.setupGPIO()

    def setupGPIO(self):
        GPIO.setwarnings(False)
        GPIO.setmode(GPIO.BCM)
        GPIO.setup(self.csPin, GPIO.OUT)
        GPIO.setup(self.misoPin, GPIO.IN)
        GPIO.setup(self.mosiPin, GPIO.OUT)
        GPIO.setup(self.clkPin, GPIO.OUT)

        GPIO.output(self.csPin, GPIO.HIGH)
        GPIO.output(self.clkPin, GPIO.LOW)
        GPIO.output(self.mosiPin, GPIO.LOW)

    def readTemp(self):

        self.writeRegister(0, 0xA2)
        # conversion time is less than 100ms
        time.sleep(.01) #giving it 10ms for conversion
        # read all registers
        out = self.readRegisters(0,8)

        # define least and most significant bit
        [rtd_msb, rtd_lsb] = [out[1], out[2]]
        # picking out the ADC code for the temperature reading
        rtd_ADC_Code = (( rtd_msb << 8 ) | rtd_lsb ) >> 1
        # converting ADC code to degrees Celsius
        temp_C = self.calcPT100Temp(rtd_ADC_Code)

        status = out[7]

        # 10 Mohm resistor is on breakout board to help
        # detect cable faults
        # bit 7: RTD High Threshold / cable fault open
        # bit 6: RTD Low Threshold / cable fault short
        # bit 5: REFIN- > 0.85 x VBias -> must be requested
        # bit 4: REFIN- < 0.85 x VBias (FORCE- open) -> must be requested
        # bit 3: RTDIN- < 0.85 x VBias (FORCE- open) -> must be requested
        # bit 2: Overvoltage / undervoltage fault
        # bits 1,0 don't care
        #print "Status byte: %x" % status

        if ((status & 0x80) == 1):
            raise FaultError("High threshold limit (Cable fault/open)")
        if ((status & 0x40) == 1):
            raise FaultError("Low threshold limit (Cable fault/short)")
        if ((status & 0x04) == 1):
            raise FaultError("Overvoltage or Undervoltage Error")

        return temp_C

    def writeRegister(self, regNum, dataByte):
        GPIO.output(self.csPin, GPIO.LOW)
```

```
        # 0x8x to specify 'write register value'
        addressByte = 0x80 | regNum;

        # first byte is address byte
        self.sendByte(addressByte)
        # the rest are data bytes
        self.sendByte(dataByte)

        GPIO.output(self.csPin, GPIO.HIGH)

    def readRegisters(self, regNumStart, numRegisters):
        out = []
        GPIO.output(self.csPin, GPIO.LOW)

        # 0x to specify 'read register value'
        self.sendByte(regNumStart)

        for byte in range(numRegisters):
            data = self.recvByte()
            out.append(data)

        GPIO.output(self.csPin, GPIO.HIGH)
        return out

    def sendByte(self,byte):
        for bit in range(8):
            GPIO.output(self.clkPin, GPIO.HIGH)
            if (byte & 0x80):
                GPIO.output(self.mosiPin, GPIO.HIGH)
            else:
                GPIO.output(self.mosiPin, GPIO.LOW)
            byte <<= 1
            GPIO.output(self.clkPin, GPIO.LOW)
    def recvByte(self):
        byte = 0x00
        for bit in range(8):
            GPIO.output(self.clkPin, GPIO.HIGH)
            byte <<= 1
            if GPIO.input(self.misoPin):
                byte |= 0x1
            GPIO.output(self.clkPin, GPIO.LOW)
        return byte

    def calcPT100Temp(self, RTD_ADC_Code):
        R_REF = 430.0 # Reference Resistor
        Res0 = 100.0; # Resistance at 0 degC for 400ohm R_Ref
        a = .00390830
        b = -.000000577500

        R_T = (RTD_ADC_Code * R_REF) / 32768.0 # PT100 Resistance

        temp_C = -(a*Res0) + math.sqrt(a*a*Res0*Res0 - 4*(b*Res0)*(Res0 - R_T))
        temp_C = temp_C / (2*(b*Res0))

        if (temp_C < 0): #use straight line approximation if less than 0
            temp_C = (RTD_ADC_Code/32) - 256
        return temp_C

class FaultError(Exception):
        #make sure control loop ends properly.
        global RunProgram
        RunProgram = False #This will end the while loop.
    pass

def get_temp():
    #import max31865
    csPin = 8
    misoPin = 9
    mosiPin = 10
    clkPin = 11
    adc = ADC(csPin,misoPin,mosiPin,clkPin)
    tempC = adc.readTemp()

    return tempC

def scale_u(x): # y = ax + b
    global SampleTime
    y1 = 0
    y2 = SampleTime
    x1 = pid.windup_lower_guard
    x2 = pid.windup_guard

    a = (y2 - y1)/(x2-x1)
    b = y1 - (a * x1)
    y = (a * x + b)
    return y

def scale_u_prosent(u): #Scale u PWM til prosent
    global SampleTime
    scaled = (u / SampleTime) * 100
    return scaled

def scale_u_for_plot(x): # y = ax + b
    y1 = 0.7
    y2 = 39.30
    x1 = 0
    x2 = 100

    a = (y2 - y1)/(x2-x1)
    b = y1 - (a * x1)
    y = (a * x + b)
    return y
```

```
def start_heating(duration):
    GPIO.setmode(GPIO.BCM)
    GPIO.setup(17, GPIO.OUT)
    GPIO.output(17, GPIO.HIGH)
    time.sleep(duration) #Signal width
    GPIO.output(17, GPIO.LOW)
    GPIO.cleanup()
    pass

def control_loop():
    start_time = time.time()
    prev_time = start_time

    lastTemp = 0.0

    csv_reader('pid_config.csv')

    #Creating measurement file with explanatory names in the first row
    #filename is unique. It's used date and time in filename to secure no overwriting
    fileName = strftime("%Y-%m-%d %H:%M:%S", gmtime()) + '.csv'
    data = []
    data = ['Date Time', 'Temperature', 'u_prosent','SampleTime', 'pid.SetPoint', 'pid.Kp',
'pid.Ki', 'pid.Kd','windup']
    mode = "a"
    csv_writer(data, fileName, mode)

    global SampleTime

    # Creating global arrays for plotting purposes
    global arrayTempC,arraySP,arrayCnt, arrayU
    arrayTempC = []
    arraySP = []
    arrayCnt = []
    arrayU = []
```

```
# counter for plotting
cnt=0

try:
    global RunProgram

    while RunProgram:

        temperatureC = get_temp()

        deltaTemp1 = temperatureC - lastTemp
        deltaTemp2 = lastTemp - temperatureC

        #filter to remove bad readings
        if cnt == 0:
            pass
        elif deltaTemp1 > 0.5:
            temperatureC = lastTemp
        elif deltaTemp2 > 0.5:
            temperatureC = lastTemp
        else:
            pass

        lastTemp = temperatureC
        # Run PID
        pid.update(temperatureC)
        # Scale pid PWM
        u_scaled = scale_u(pid.output)
        u_prosent = scale_u_prosent(u_scaled)
        # Update GUI
        gui.txtShowTemp.setValue(temperatureC)
        gui.txtShowUTime.setValue(u_scaled)
        gui.txtShowUPorsent.setValue(u_prosent)
        # Run heating
        start_heating(u_scaled) #PWM signal
        # Log measurement and parameters to .csv file
        data = []
        mode = "a"
        data = [strftime("%Y-%m-%d %H:%M:%S", gmtime()), temperatureC, u_prosent,
    SampleTime, pid.SetPoint, pid.Kp, pid.Ki, pid.Kd, pid.windup_guard]
        csv_writer(data, fileName, mode) #Call the writer
        # Log data for graph .txt file
        arrayTempC.append(temperatureC)
        arraySP.append(pid.SetPoint)
        arrayCnt.append(cnt)
        u_for_plot = scale_u_for_plot(u_prosent)
        arrayU.append(u_for_plot)
        txt_writer()

        cnt=cnt+1
        if(cnt>4000):           #If you have 4000 or more points, delete the first one from the
    array
            arrayTempC.pop(0)    #This allows us to just see the last 4000 data points
            arraySP.pop(0)
            arrayCnt.pop(0)
            arrayU.pop(0)

        # setting loop time
        # SampleTime is controlling looptime
        sleep = SampleTime - (time.time() - prev_time)

        if sleep >= 0.01:
            time.sleep(sleep-0.01) # Compensate time this calculation takes
        else:
            time.sleep(0.0001)

        prev_time = time.time()

        pass
        finally:
            GPIO.cleanup() #To secure that heating is shutdown.
            print("### PROPER SHUTDOWN ###")

    def txt_writer():
        global arrayTempC, arraySP, arrayCnt, arrayU

        file = open("temperature.txt","w")
        for index in range(len(arrayCnt)):
            file.write("\n" + str(arrayCnt[index])+
    ","+str(arrayTempC[index])+","+str(arraySP[index])+","+str(arrayU[index]))
        file.close()


    def csv_writer(data, path, mode):
        """
        Write data to a CSV file path
        """
        with open(path, mode, newline='') as csv_file:
            writer = csv.writer(csv_file, delimiter=',')
            #for line in data:
            writer.writerow(data)

    def csv_reader(path):
        parameter = []
        with open(path, newline='') as csv_file:
            reader = csv.reader(csv_file, delimiter=',')
            for row in reader:
                parameter = row

        #Also set new sampletime
        global SampleTime
        SampleTime = float(parameter[4])

    newAllParameters(float(parameter[0]),float(parameter[1]),float(parameter[2]),float(paramete
    r[3]))
```

```
def newSetpoint(SP):
    pid.SetPoint = SP

def newAllParameters(SP, P, I, D):
    pid.SetPoint = SP
    pid.setKp(P)
    pid.setKi(I)
    pid.setKd(D)
    newParametersToGui()

def newParametersToGui():
    global SampleTime
    gui.txtSetSP.setValue(pid.SetPoint)
    gui.txtSetKd.setValue(pid.Kd)
    gui.txtSetKi.setValue(pid.Ki)
    gui.txtSetKp.setValue(pid.Kp)
    gui.txtSampleTime.setValue(SampleTime)

def runGui():
    gui.show()
    newParametersToGui()
    # without this, the script exits immediately.
    sys.exit(app.exec_())

if __name__ == "__main__":

    global RunProgram, SampleTime
    SampleTime = 2.0
    RunProgram = True

    pid = PID(0, 0, 0) # object is accesible to all

    app = QApplication(sys.argv)
    gui = MainWindow()
```

```
#daemon not = True, so that control_loop will end properly
t = threading.Thread(target=control_loop)
t.start()

runGui()
```