



Warsaw University of Technology
Faculty of Electronics and Information Technology
Institute of Electronic Systems

Wojciech Obuchowicz

ID number: 252852

Bachelor's diploma thesis

**Firmware for Sinara's two-channel temperature
controller**

Thesis supervisor:

Maciej Linczuk, PhD

Warsaw 2019

Summary

Title: Firmware for Sinara's two-channel temperature controller

The goal of the thesis is to design firmware for 2-channel temperature controller of the Sinara hardware family. Consecutive project phases are formulating design assumptions, designing the firmware, testing the firmware using the prototype, testing the firmware with the device and presenting results. The firmware is designed for *TM4C1294NCPDT* microcontroller from Texas Instruments. The project implements various solutions: Ethernet communication, MQTT protocol, RTOS, PID control and others.

Keywords: *firmware, microcontroller, PID controller, ARM, Ethernet, MQTT*

Streszczenie

Tytuł: Oprogramowanie modułu dwukanałowego kontrolera temperatury projektu Sinara

Celem niniejszej pracy jest zaprojektowanie oprogramowania dla dwukanałowego kontrolera temperatury projektu Sinara. Na kolejne etapy pracy składa się sformułowanie założeń projektowych, opracowanie oprogramowania, przetestowanie oprogramowania z użyciem prototypu, przetestowanie działania urządzenia z załadowanym oprogramowaniem oraz przedstawienie wyników pracy. Oprogramowanie zaprojektowano dla mikrokontrolera *TM4C1294NCPDT* firmy Texas Instruments. W projekcie wykorzystano wiele różnych rozwiązań: komunikację Ethernet, protokół MQTT, RTOS, regulację PID i inne.

Słowa kluczowe: *firmware, mikrokontroler, regulator PID, ARM, Ethernet, MQTT*

Tę kartkę należy zastąpić oświadczeniem o autorstwie pracy.

Contents

Contents	1
1 Introduction	3
1.1 About temperature stabilization	3
1.2 About Sinara project	3
2 Review of existing solutions and the purpose of the thesis	5
2.1 Review of existing solutions	5
2.2 The purpose of the thesis	6
3 Design assumptions and design concept	7
3.1 About thermostat project	7
3.2 Design assumptions	12
3.3 Design concept	13
4 Design of the firmware	15
4.1 Prototyping	15
4.2 Description of the firmware for thermostat_thorlabs	18
4.3 Description of the firmware for thermostat_max	28
5 Testing of the firmware	35
5.1 Testing the thermostat_thorlabs version	35
5.2 Testing the thermostat_max version	37
6 Conclusions	45
List of Figures	47
List of Tables	49
A Appendix	51
A.1 Source code of the firmware	51
A.2 Schematics of the device	64
Bibliography	73

1 Introduction

1.1 About temperature stabilization

Throughout history, humanity has struggled with the problem of temperature stabilization. The problem arose from the need to provide comfortable living conditions. It is believed that early humans used caves as their living places [6]. They served as shelters from adverse weather conditions, wild animals, but also from excessive temperature fluctuations. The structure of the cave ensured that in winter the temperature was higher than outside and in summer it was lower than outside. If the cave itself did not provide the right temperature, humans wore fur clothing – another tool to control the temperature. The discovery of fire was a milestone for temperature stabilization – with controlled fire humans could easily keep their homes warm. Throughout the centuries the control of the temperature had to be conducted by people. They assessed subjectively whether the temperature was comfortable and, consequently, they increased or decreased the heat. The discovery of electricity has brought significant changes in the area of temperature control. The use of relays made it possible to perform objective and more precise control of the temperature. Electronics enabled further improvements in the field of temperature stabilization. Nowadays, advanced temperature control systems are used not only in homes but also in science. For instance, high precision temperature control is required in the field of quantum physics. Many research teams struggle with high requirements for temperature stabilization. One of them is the team of Sinara project.

1.2 About Sinara project

Sinara is an open source instrumentation family built for the purposes of the ARTIQ project [17]. ARTIQ (Advanced Real-Time Infrastructure for Quantum Physics) is an initiative created within partnerships between M-Labs and the National Institute of Standards and Technology (NIST), which currently has more than a dozen research institutions from around the world [4, 10]. As part of ARTIQ, a control system is being developed for applications in experiments in the field of quantum computing. The Sinara project was created in response to the need to improve the quality, functionality and scalability of ARTIQ systems. Hardware Sinara is designed to be reproducible, modular and easy to adapt. Because experiments in the field of quantum computing put high demands on fast data processing in real time, Sinara modules are produced in two forms – *MicroTCA* and *Eurocard Extension Module* (EEM). Currently, over 40 modules of the Sinara family are being developed [18]. One of the institutions involved in the Sinara project is the Institute of Electronic Systems at Warsaw University of Technology [16].

As mentioned before, the Sinara team faces high demands on temperature control. Experiments in the field of quantum computing require temperature stability at the level of millikelvins. That is why the idea of implementing a two-channel temperature control module in ARTIQ system was born. The module must provide the required temperature stability and be compatible with the ARTIQ system (Ethernet communication, proper PCB form etc.).

2 Review of existing solutions and the purpose of the thesis

2.1 Review of existing solutions

Owing to the fact that the desired device has very specific requirements for temperature stability and compatibility, there are no ready-made solutions on the market. It is also difficult to find modules with similar functionality. However, by inspecting the market in search of similar devices, one can find modules that provide temperature control via Ethernet.

2.1.1 AR654



Figure 2.1. Four-channel universal controller *AR654* from *Apar* [3]

The *Apar's* universal controller *AR654* enables temperature control in four channels via Ethernet [3]. Both control through a web application and a remote terminal are provided. However, this device is very advanced when it comes to functionality. Not only temperature control, but also control of other physical quantities (humidity, pressure, flow, velocity, etc.) is possible. The controller is not designed in any of the standards used by the Sinara project (EEM, MicroTCA) and requires a mains or low-voltage power supply from a separate socket. It also has parts unnecessary from the point of view of the Sinara project: a graphic LCD display, a keyboard and a USB port. The software of this device enables displaying, keyboard handling, writing to flash memory and many functions redundant in the ARTIQ system.

2.1.2 Uniplex III



Figure 2.2. *Uniplex III* heating controller from *Klöpper Therm* [3]

The *Uniplex III* temperature controller from *Klöpper Therm* is produced in the form of EEM [24]. However, also in this case, the module uses power from a separate socket. In addition, the device provides support for only one channel and has a display and a keyboard that are unnecessary when it comes to the Sinara project. Service via the web application is not provided. From the point of view of the firmware, the module may have similar solutions regarding the remote terminal to those that might be used in the desired module.

Due to the lack of ready-made solutions on the market, Sinara team decided to create their own temperature controller called *thermostat*. The PCB of the project was designed in the Institute of Electronic Systems at Warsaw University of Technology. Firmware of the device is the subject of this thesis.

2.2 The purpose of the thesis

The purpose of the thesis is to design firmware for the two-channel temperature controller of the Sinara project. This includes, in particular, the ability to control the settings of the device and read data from it using the web application or a remote terminal enabling easy automation with Python scripts. The firmware must also provide temperature control mechanism.

The purpose of the device is to stabilize the temperature at the level of 1 mK, however, reaching this level of stability is not the purpose of the thesis. For many reasons, mainly regarding laboratory capabilities, tuning the device was left in the hands of physicists of Sinara project.

3 Design assumptions and design concept

3.1 About thermostat project

The temperature controller module developed by the Sinara team is called *thermostat* [20]. Because temperature control for ARTIQ system applications does not set the highest requirements for signal phase control, the device has been designed in the form of EEM, which is cheaper than the form of MicroTCA also used by ARTIQ [30], nevertheless, it is sufficient for the project. Communication with the controller occurs via the RJ-45 connector, the temperature can be controlled in two channels.

At the Institute of Electronic Systems, a PCB was designed for two versions of the device [20, 21]. One of them uses a ready-made TEC driver from *Thorlabs*, with which the microcontroller communicates using UART. This version was called *thermostat_thorlabs* and was not finally produced due to the unsatisfactory performance of the TEC driver. However, the firmware of this version has been created and is described in this thesis.

The simplified schematic diagram of *thermostat_thorlabs* looks as follows:

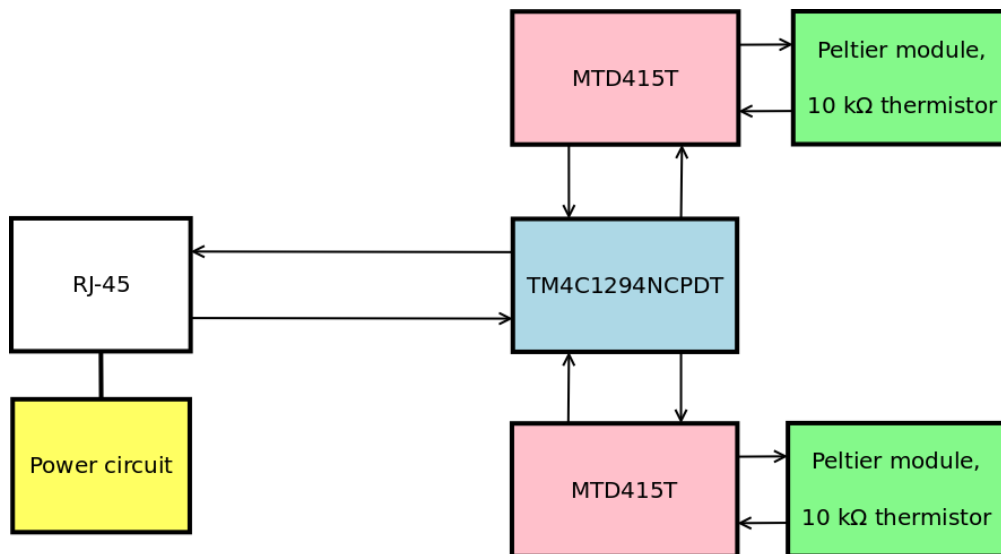


Figure 3.1. Simplified schematic diagram of *thermostat_thorlabs*

The *thermostat_thorlabs* is powered via an RJ-45 connector, using the *Ag5300* module from *Silvertel*. *Ag5300* makes it possible to obtain an output signal of 12V and power up to 30W, using PoE (Power over Ethernet) technology.

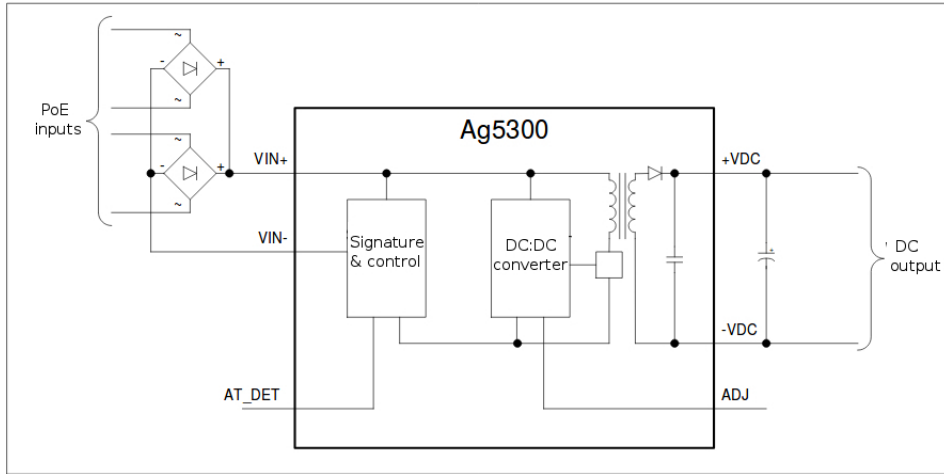


Figure 3.2. Block diagram of Ag5300 [2]

Two input pins – $VIN+$ and $VIN-$ – are connected to the positive and negative pins of the PoE. AT_DET pin of Ag5300 is set automatically when the power is supplied to the Ag5300 via Cat 5e cable. This pin is not used in the project. Thanks to the DC/DC converter of the module, 12V voltage is provided between pins $-VDC$ and $+VDC$. The converter has built-in output overload and short-circuit protection.

Input signals coming to the thermostat_thorlabs via the RJ-45 connector are processed by the Texas Instruments' *TM4C1294NCPDT* microcontroller equipped with a 32-bit ARM Cortex M4 core and an integrated Ethernet hardware support.

Table 3.1. Selected parameters of the *TM4C1294NCPDT* microcontroller [19].

Performance	
Core	ARM Cortex-M4F
Performance	120 MHz
Flash	1024 kB
SRAM	256 kB
EEPROM	6 kB
Communication Interfaces	
UART	8×
QSSI	4×
I ² C	10×
CAN	2×
Ethernet MAC	10/100
USB	USB 2.0
System Integration	
DMA	32-channel configurable DMA controller
General-Purpose Timers	8×
Watchdog Timers	2×
GPIO	15×

The microcontroller is based on the most widely used instruction set architecture – ARM [12]. 1024 kB flash and 120 MHz clock rate are more than sufficient for project’s applications. 6 kB EEPROM can be used to save modifiable parameters of the thermostat. The device provides multiple communication interfaces and Ethernet is the most important for the designed device. Many UART and QSSI interfaces ensure effective communication with units of the thermostat. General-purpose timers can be used in the project for time management and pulse shaping. Also, watchdog timers are useful to detect and recover from potential malfunctions.

In each of the two channels, there is *Thorlabs’ MTD415T* TEC driver, which controls the Peltier module using an external temperature sensor. The microcontroller communicates with the *MTD415T* module via the UART interface.



Figure 3.3. *MTD415T* [22]

Table 3.2. Selected parameters of the *MTD415T* [22]

Output Power	up to 6.0 W
Maximum Temperature Control Range	+5°C to +45°C
Temperature Setting Resolution	1 mK
Temperature Measurement Resolution	better than 10 mK, typically 2 mK
Absolute Temperature Accuracy	± 0.5°C
Temperature Stability over 8 h, typically	better than 20 mK
Interface	UART, 115 200 b/s, 8 data bits, 1 stop bit

The maximum output power of the module is 6.0 W – that is more than sufficient as the power circuit of the thermostat capable of providing up to 8W. Range of the controlled temperature is adequate – there is no need to provide control of temperatures lower than +5°C or higher than +45°C. Temperature setting resolution is good enough, however, measurement resolution, as well as temperature stability, was finally found not satisfactory. For this reason, the manufacturing of thermostat_thorlabs was abandoned.

As the schematic diagram shows, a Peltier module and a temperature sensor are connected to each *MTD415T*.

The second version of the device is a version using two *MAX1968* systems from *Maxim IC* – switch-mode drivers for Peltier modules and *AD7172-2* from *Analog Devices* – a sigma-delta analog-to-digital converter. The simplified schematic diagram of this version of the project looks as follows:

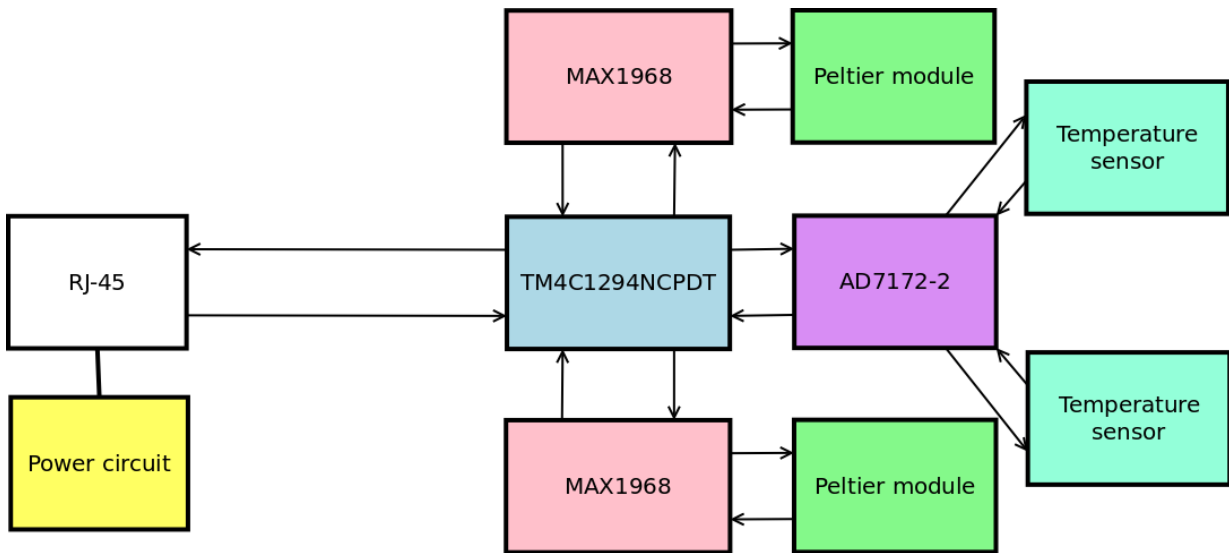


Figure 3.4. Simplified schematic diagram of thermostat_max

The difference with the previous version is the use of *MAX1968* chips. *MAX1968* allows to control the current and voltage of the Peltier module.

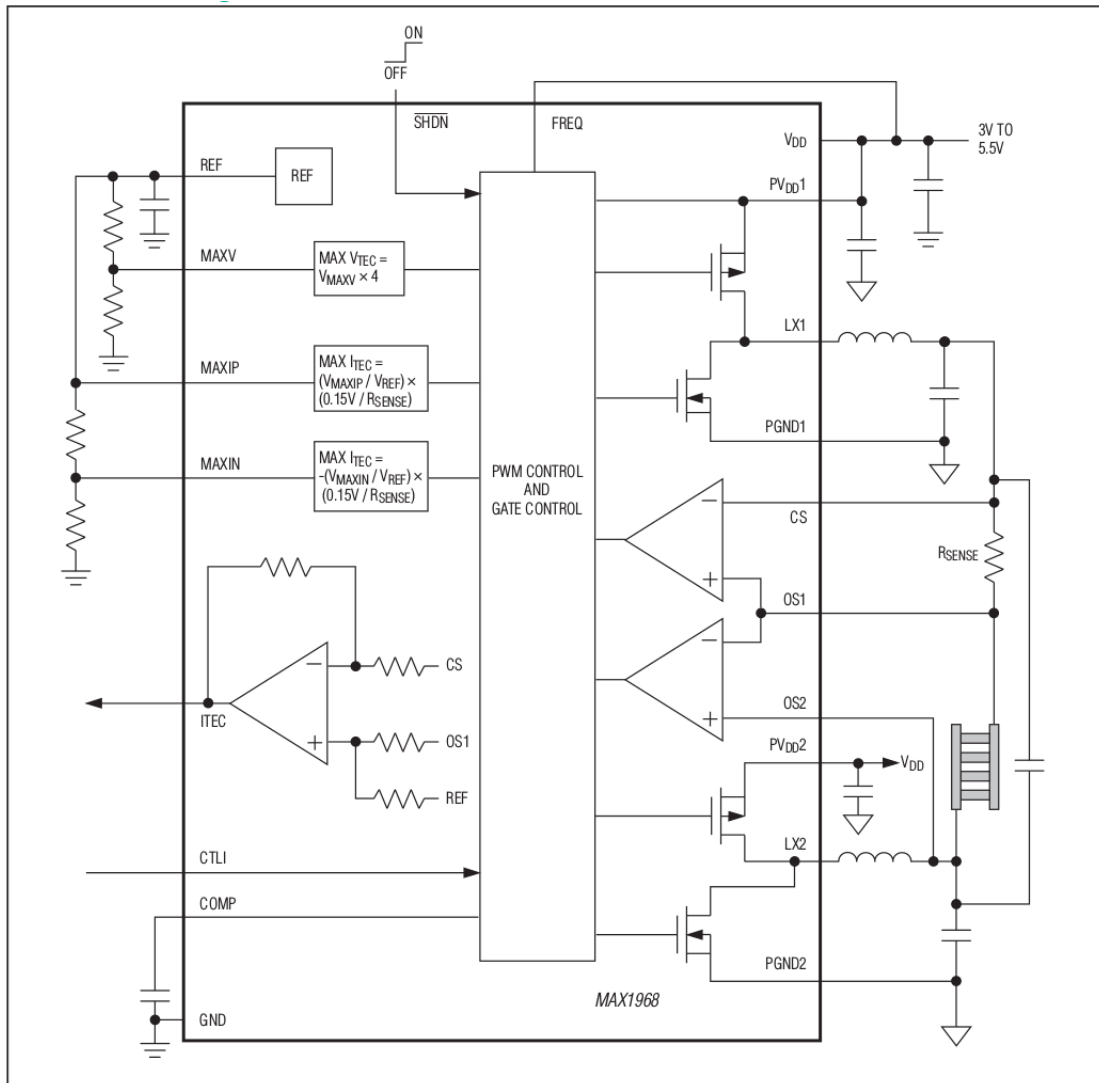


Figure 3.5. Block diagram of *MAX1968* [11]

As the manufacturer points, the *MAX1968* consists of two switching buck regulators operating together to directly control TEC current. *REF* is the output pin with 1.5V reference voltage. *MAXV*, *MAXIP* and *MAXIN* are input pins used to set maximum bipolar TEC voltage, maximum positive TEC current and maximum negative TEC current respectively. PWM signals from microcontroller are used in the project as signals to set these values. *ITEC* is an output signal corresponding to the value of the actual TEC current, while *CTLI* is an input signal to set TEC current. *FREQ* pin allows selection of switching frequency. *SHDN* is shutdown control input – it is used in the project to enable or disable TEC driver using the microcontroller GPIO signal. The other pins of the *MAX1968* device are significant from the design point of view – the choice of the connected resistors, capacitors and inductors determines the resonant frequency of output filter and other design features.

To read the voltage on thermistors, the *AD7172-2* analog-to-digital converter is used.

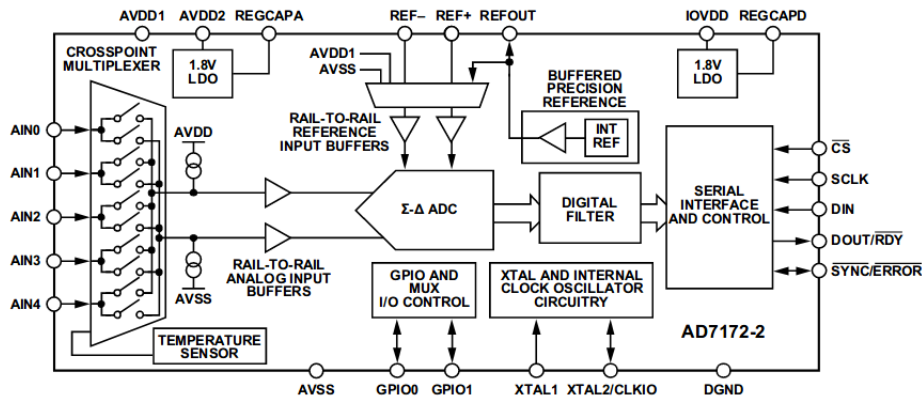


Figure 3.6. Block diagram of *AD7172-2* [1]

The AD7172-2 is a low noise, low power ADC with 2 fully differential channels or 4 single-ended channels. The maximum channel scan rate of the device is of 6.21 kSPS. The output data rates range from 1.25 SPS to 31.25 kSPS. The analog-to-digital converter has three essential blocks:

- $\Sigma - \Delta$ ADC block
- digital filter block
- serial interface and control block

ADC block handles fundamental functionality that is converting an analog signal to a digital signal. Digital filter block is responsible for filtering the output signal to reach the desired optimization of noise, settling time, and rejection. Serial interface and control block handles SPI communication.

Input pins for analog signals are from *AIN0* to *AIN4*. The first four are connected to temperature sensors used in the project. *CS*, *SCLK*, *DIN*, *DOUT/RDY* pins are connected to the microcontroller – they are used for SPI communication. *AVDD1*, *AVDD2*, *REGCAPA*, *REF-*, *REF+*, *IOVDD*, *REGCAPD*, *AVSS* and *DGND* are pins for power purposes. The other pins, including general purpose input/output pins and pins for external crystal oscillator, are not used in the designed device.

In the further part of the thesis the name *thermostat* refers to the whole project, the name *thermostat_thorlabs* – to the version of the project with *MTD415T* modules, and the name *thermostat_max* – to the version with the *MAX1968* chips. Schematics of both versions of the device are included as an attachment to the thesis.

3.2 Design assumptions

The firmware of the thermostat module for the *TM4C1294NCPDT* microcontroller must meet the following requirements:

- The firmware must be error-proof and easy to modify.
- Two *MTD415T/MAX1968* controllers must be serviced in two channels from the web application level.
- Temperature control should be possible via a remote terminal enabling automation using Python scripts.

- The configuration of the connection should be as automatic as possible, but it should be possible to change and save.
- In the case of using external solutions (e.g. libraries), they should be open source solutions.

3.3 Design concept

To ensure the reliability of the created firmware, it should be as simple as possible, should not use unnecessary functions in the code, should provide control of set values and error handling. The simplicity of the modification is assured if each functionality is served by a separate function, and the code is well described.

In order to implement the communication protocols used in project, it was necessary to use the TCP/IP stack. Lightweight IP (lwIP) is the widely used TCP/IP stack for embedded systems [25]. The popularity of this stack implies that it is well-tested on different microcontrollers. Using lwIP ensures low memory usage while still having a full-sized TCP/IP stack [9]. An additional advantage is that lwIP is open source.



Figure 3.7. *lwIP* logo [9]

The project uses a real-time operating system (RTOS) for embedded devices. The advantages of this solution are as follows:

- Simultaneous service of the web application, remote terminal, communication with the controller and other side tasks should be much simpler with the use of RTOS multithreading. Scheduling and synchronization of these tasks without an operating system would be a complex problems by themselves, implicating a potentially greater number of errors.
- The use of RTOS ensures higher processor performance, because there is no time wasted on completing various threads.
- Better management of resources. When a thread connected to a web server has to share resources with a thread related to a remote terminal, RTOS solves it by inversion of priorities.
- Using the operating system should provide a more simple and transparent code structure.
- Portability - using RTOS simplifies the implementation of the created firmware in case of need to change the microcontroller in the device.

The idea of implementing a dedicated real-time operating system was abandoned and one of the systems available on the market was implemented. Writing a dedicated RTOS would involve a relatively high cost of time and a potentially greater risk of unreliability in the first period of use of the device. The popular operating systems for embedded devices available on the market should no longer deal with many defects. What is more, popular RTOS has probably been tested many times in integration with the lwIP library.

FreeRTOS was chosen from the real-time operating systems available on the market. The advantage of this system is that it is one of the two most popular RTOS for embedded devices [26]. That brings with



Figure 3.8. *FreeRTOS* logo [8]

it higher reliability, a larger knowledge base about the system and a larger community that supports itself in solving design problems.

An additional advantage of *FreeRTOS* is that it is open source, which is consistent with the design assumptions of the Sinara hardware family. Another benefit is a set of already existing Ethernet projects implemented on the *TM4C1294NCPDT* microcontroller. These working projects are provided by the manufacturer with *Tivaware* package designed to help developers in their work. In fact, *Texas Instruments* attached the *FreeRTOS* system in *Tivaware* library.

The last significant choice is the issue of how to implement the remote text terminal. One solution would be to use the SCPI communication standard and the Telnet protocol. However, choosing this option would require defining the way of issuing commands (e.g. the commands would have to end with the "carriage return" sign). The way the command should be handled would have to be complex and, therefore, it would be non-standard (i.e. limited to this specific device). Using this solution would imply problems in case of need to change microcontroller. Moreover, Telnet by itself does not provide sufficient certainty of delivery. Additional functions implemented on the server side as well as on the client side would be necessary for this purpose. For all of these reasons, the idea of implementing the remote terminal has been abandoned.

Eventually, the MQ Telemetry Transport protocol (MQTT) based on the publication/subscription pattern, was implemented in the project. This protocol perfectly fits for applications in embedded systems – it is extremely simple and light (i.e. it needs little resources). It does not need a wide bandwidth, it provides high reliability and allows to manage the assurance of delivery [31]. Furthermore, many ARTIQ software tools use MQTT successfully [20], so it will be easier to adapt and use the device in the ARTIQ system.

4 Design of the firmware

The firmware of the thermostat project was written in C language in the C99 standard and is divided into many source files. The full source code of both versions of the project is on the CD attached to the thesis.

4.1 Prototyping

Before the final version of the thermostat firmware was achieved, the project had been gradually developed using the tools available at the university.

4.1.1 Hardware

Before working with the final version of the thermostat device, the firmware was developed using the *TM4C1294XL* evaluation board from *Texas Instruments*, which is based on the *TM4C1294NCPDT* microcontroller with a built-in Ethernet MAC and PHY chip [23]. The development board contains multiple peripherals useful while designing firmware with Ethernet connectivity. Among these peripherals there are:

- Ethernet port
- micro USB connector
- debugger, programmer and UART-USB converter
- 4 user LEDs
- 2 user buttons
- a set of pins enabling, for example, a connection of additional peripheral devices

As the project required the use of three UART interfaces, a USB-UART converter module based on the CP2102 chipset was used for prototyping.

4.1.2 Software

The thermostat firmware was created in the *Code Composer Studio* - an integrated development environment dedicated to projects for *Texas Instruments* microcontrollers. *CCS* includes a C/C++ optimizing compiler, a friendly source code editor and a debugger [7]. *TM4C1294NCPDT* microcontroller used in the project thermostat is supported by *CCS* with no limit on the size of the code.

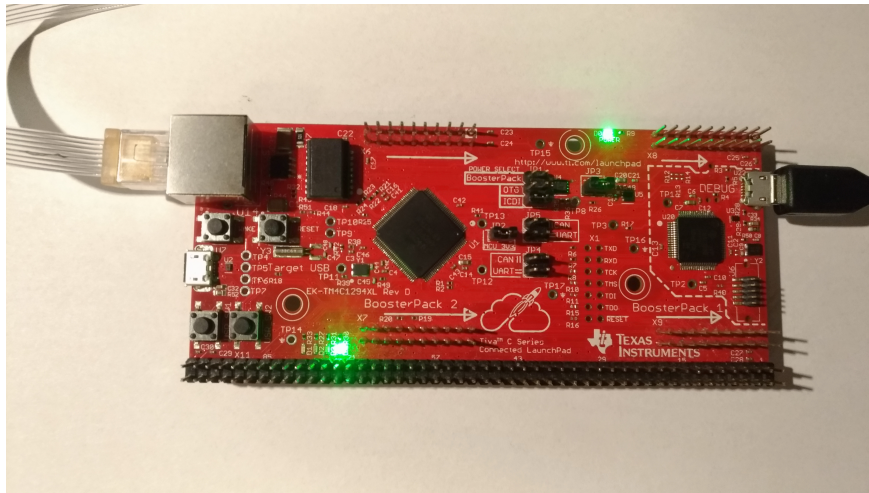


Figure 4.1. *TM4C1294XL* evaluation board

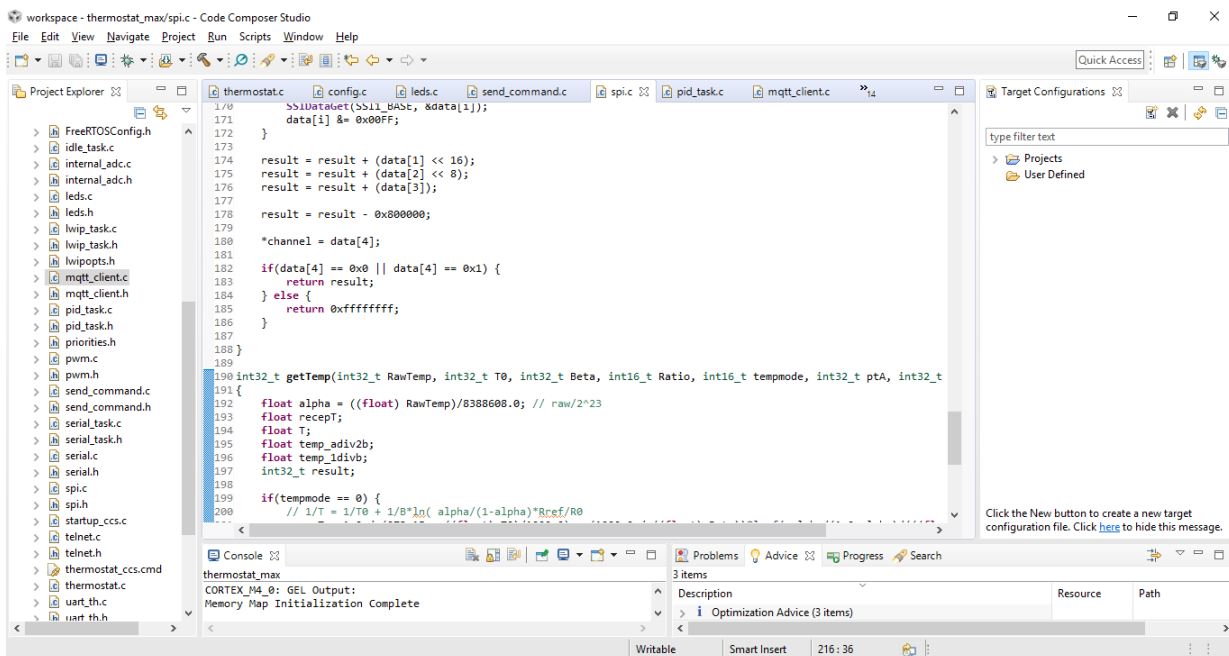


Figure 4.2. GUI of *Code Composer Studio*

Realterm software was used to work with UART. *Realterm* is a console emulator that allows capturing, controlling and debugging data streams from, for example, UART [15].

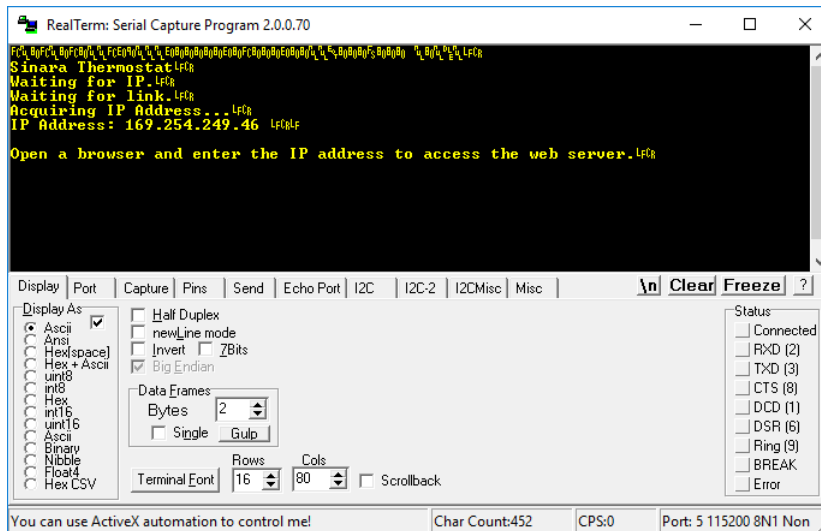


Figure 4.3. GUI of *Realterm*

The thermostat project uses Ethernet to communicate with a microcontroller. To successfully develop a part of the project responsible for network connectivity with the PCB, it was necessary to use a program that allows capturing and analyzing data flowing in the network. A popular, open source *Wireshark* program has been selected for these applications. A big advantage of *Wireshark* is relatively simple handling with a graphical user interface.

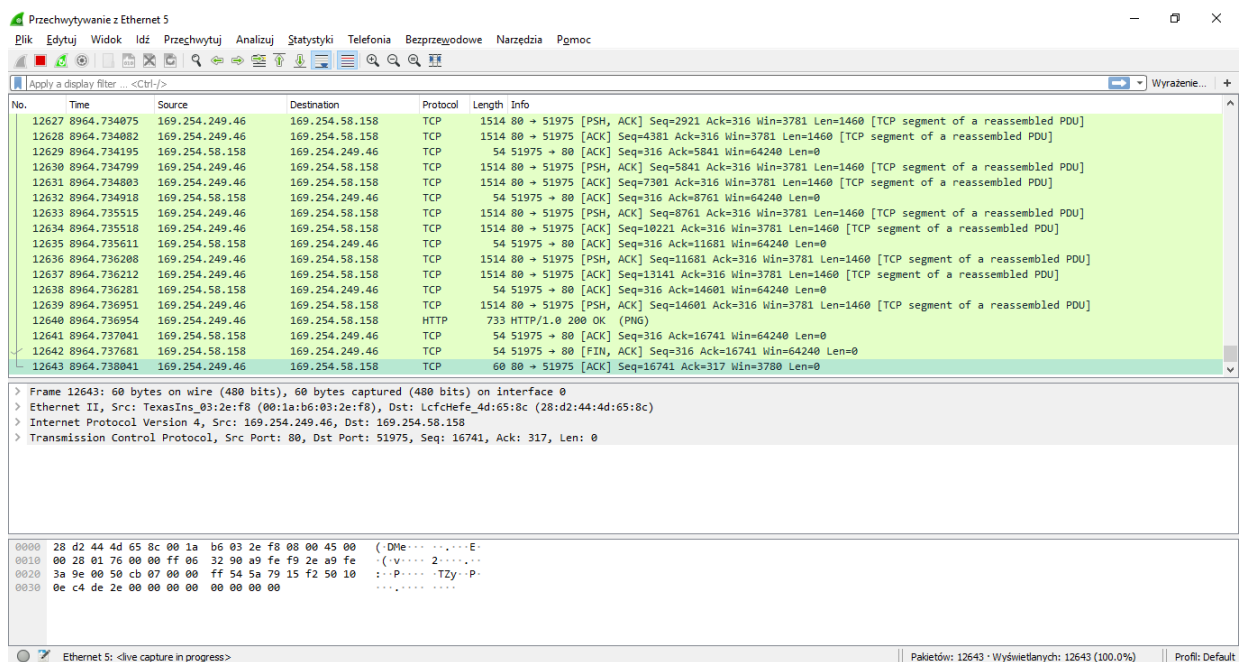


Figure 4.4. GUI of *Wireshark*

As mentioned earlier, the project uses the MQTT data transmission protocol. This protocol is based on the publication/subscription pattern, therefore, the presence of a server is required. This server is called MQTT broker and it manages all publications and subscriptions in the MQTT network. During the development and testing process of the MQTT part of the project, a popular open source *Mosquitto* broker was used. The advantage of *Mosquitto* is its versatility and low usage of memory resources [13].

The use of *MQTT Spy* software was very helpful in the development of the MQTT part. This is a universal application of MQTT client written in Java. It enables to monitor the activity of the MQTT network.

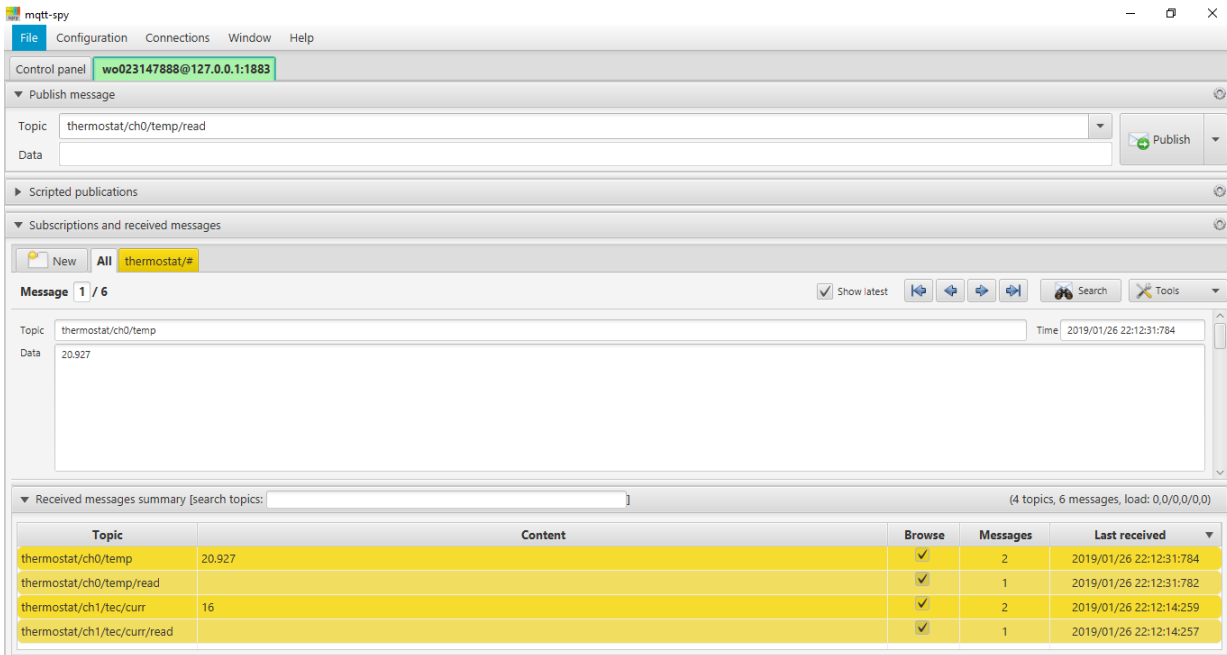


Figure 4.5. GUI of *MQTT Spy*

4.2 Description of the firmware for thermostat_thorlabs

4.2.1 FreeRTOS tasks

Processes in the *FreeRTOS* system are carried out using tasks. Each task has its own registers and its stack. *FreeRTOS* is a preemptive system. This means that the scheduler decides which task is executed at the moment. The developer's issue is to construct and initiate the tasks that are subsequently managed by the scheduler [29].

The following *FreeRTOS* tasks have been implemented in the thermostat firmware:

- a task related to TCP/IP stack handling – this task is initiated by lwIP library functions
- a task that supports Telnet-UART communication – implemented in the *SerialTask* function in the "serial_task.c" file
- *vApplicationIdleHook* – the lowest priority function, called by the system when no higher priority task requires execution

The task regarding the TCP/IP stack is initialized in the *main* function using the function *lwIPTaskInit* [A.1.1]. For this purpose, the MAC address is read from the microcontroller registers and it is passed to the function together with static IP address, subnet mask, default gateway for the connection and connection type. In this case it is used a connection with DHCP. As the documentation of the lwIP library says, if there is no possibility of connecting with the use of DHCP (e.g. there is no DHCP server in the network), AUTOIP mode is used instead. This mode is created specifically for the lwIP library [5]. Establishing a connection in the AUTOIP mode starts with attempting to take a randomly selected static IP address of form 169.254.xxx.xxx. Subsequently, there is an attempt to connect to the network. A

certain type of protection against competing addresses has been implemented here - after drawing an address of the above-mentioned form, *ARP request* for such address is sent several times. In the absence of a response, the drawn address is accepted as its own. However, it is not certain that there will be no duplication of the IP address in the network.

The *SerialTask* function is a task that handles communication over the UART and Telnet serial port [A.1.2]. The *SerialTask* uses system queues that store data coming from UART and outgoing to UART. Each character arriving via a given Telnet channel is sent directly to the corresponding UART port. In the same way, each character coming from UART is sent to the appropriate Telnet channel. This task has been implemented so that there is a simple way of direct communication (via Telnet) with two *MTD415T* modules (two UART ports correspond to two *MTD415T* devices). Thanks to that it is easier to validate the thermostat_thorlabs module. It also provides a simple text terminal in case there are difficulties with MQTT part of the project.

In the *vApplicationIdleHook* function [A.1.3], there is a code that enables changing the IP address of the thermostat module if necessary (e.g. changing the IP address by the user, disconnecting or connecting the module to the network). The new IP address is sent to the UART port available to the user (*PA0* and *PA1* pins of the microcontroller).

4.2.2 Ethernet communication

As mentioned above, the *FreeRTOS* task responsible for handling the TCP/IP stack is initiated using the lwIP library. Ethernet communication is an important part of the designed firmware, therefore the code responsible for this part of the project is the most extensive and plays a key role. The connection is established in the *lwIPTaskInit* function. In this function the Telnet session is opened (*TelnetInit*, *TelnetListen* and *TelnetOpen* functions of the "telnet.c" file) and the web server is launched (the *httpd_init* function of the *lwIP* library).

The web server in the designed firmware uses the SSI (Server Side Includes) scripting mechanism and the CGI (Common Gateway Interface) interface. SSI allows nesting specific values in the code of the document sent by the server to the client. These values can be, for example, variables or function results. In the thermostat project values of measured temperature, voltage, current is sent to the client with the help of SSI. As for CGI, it allows communication between the web server functions and other functions of the project. In the case of this firmware, CGI is primarily used to handle the requested values of the set temperature, maximum current, voltage, sent by the HTTP client.

The *http_set_ssi_handler* function specifies a function that handles all SSI queries from a web server. The *http_set_cgi_handlers* function assigns functions that support all of the CGI requests provided in the project. These functions (*http_set_ssi_handler* and *http_set_cgi_handlers*) are called during initialization, in the *ConfigWebInit* function implemented in the "config.c" file.

The "config.c" file contains a significant part of the code responsible for handling the web server – this is the backend of the created web application. In the file there are definitions of all of the functions responsible for executing CGI requests and the function *ConfigSSIHandler* responding to SSI queries. The array below is passed to the *http_set_cgi_handlers* function and defines which functions are responsible for each CGI request:

```
1 static const tCGI g_psConfigCGIURIs[] =
  {
    { "/config.cgi", ConfigCGIHandler },           // CGI_INDEX_CONFIG
    { "/misc.cgi", ConfigMiscCGIHandler },       // CGI_INDEX_MISC
5    { "/defaults.cgi", ConfigDefaultsCGIHandler }, // CGI_INDEX_DEFAULTS
```

```

10  { "/ip.cgi", ConfigIPCGIHandler }, // CGI_INDEX_IP
    { "/read.cgi", ConfigERRCGIHandler }, // CGI_INDEX_ERR
    { "/reserr.cgi", ConfigRESERRCGIHandler }, // CGI_INDEX_RESERR
    { "/teclim.cgi", ConfigTecLimCGIHandler }, // CGI_INDEX_TECLIM
15  { "/tempset.cgi", ConfigTempSetCGIHandler }, // CGI_INDEX_TEMPSET
    { "/tempwin.cgi", ConfigTempWinCGIHandler }, // CGI_INDEX_TEMPWIN
    { "/tempdel.cgi", ConfigTempDelCGIHandler }, // CGI_INDEX_TEMPDEL
    { "/clgain.cgi", ConfigCLGainCGIHandler }, // CGI_INDEX_CLGAIN
    { "/clperiod.cgi", ConfigCLPeriodCGIHandler }, // CGI_INDEX_CLPERIOD
15  { "/cltime.cgi", ConfigCLTimeCGIHandler }, // CGI_INDEX_CLTIME
    { "/clp.cgi", ConfigCLPCGIHandler }, // CGI_INDEX_CLP
    { "/cli.cgi", ConfigCLICGIHandler }, // CGI_INDEX_CLI
    { "/cld.cgi", ConfigCLDCGIHandler }, // CGI_INDEX_CLD
20  { "/soft_reset.cgi", ConfigSoftResetCGIHandler }, // CGI_INDEX_SOFT_RESET
    { "/setmqtt.cgi", ConfigSetMQTTTCGIHandler } // CGI_INDEX_SETMQTT
};

```

For example: if the web client sends a request "tempset.cgi", the function *ConfigTempSetCGIHandler* will be launched and the target temperature will be set.

Table 4.1. Functions that handle CGI requests and actions performed by these functions

<i>ConfigCGIHandler</i>	changes Telnet protocol settings (port number, mode, IP, timeout)
<i>ConfigMiscCGIHandler</i>	changes the module name
<i>ConfigDefaultsCGIHandler</i>	restores the default connection parameters
<i>ConfigIPCGIHandler</i>	changes connection parameters (IP address, subnet mask, default gateway)
<i>ConfigERRCGIHandler</i>	reads error register of MTD415T module
<i>ConfigRESERRCGIHandler</i>	resets error register of MTD415T module
<i>ConfigTecLimCGIHandler</i>	sets the TEC current limit
<i>ConfigTempSetCGIHandler</i>	sets the target temperature
<i>ConfigTempWinCGIHandler</i>	sets the temperature window
<i>ConfigTempDelCGIHandler</i>	sets the diode delay time
<i>ConfigCLGainCGIHandler</i>	sets the critical gain of PID controller
<i>ConfigCLPeriodCGIHandler</i>	sets the critical period of PID controller
<i>ConfigCLTimeCGIHandler</i>	sets the cycling time of PID controller
<i>ConfigCLPCGIHandler</i>	sets the P share of PID controller
<i>ConfigCLICGIHandler</i>	sets the I share of PID controller
<i>ConfigCLDCGIHandler</i>	sets the D share of PID controller
<i>ConfigSoftResetCGIHandler</i>	resets the device
<i>ConfigSetMQTTTCGIHandler</i>	changes address of the MQTT broker

All functions supporting CGI requests have a similar structure, therefore a description of one of them will give a picture of all others. For example, the function that handles the request to change the current limit looks as below:

```

1  static const char *
  ConfigTecLimCGIHandler(int iIndex, int iNumParams, char *pcParam[], char *pcValue[])
  {
    int32_t i32Port;
5   int32_t i32Value;
    bool bParamError = false;
    bool bCommandError;

    //
10  // Get the port number.

```

```

//
i32Port = ConfigGetCGIParam("port", pcParam, pcValue, iNumParams, &bParamError);
i32Value = (uint32_t)ConfigGetCGIParam("teclim", pcParam, pcValue, iNumParams, &bParamError);
15     if(bParamError || ((i32Port != 0) && (i32Port != 1)))
        {
            return (PARAM_ERROR_RESPONSE);
        }
20     bCommandError = Set_TEC_limit(i32Port, i32Value);

    if(bCommandError == 0)
    {
25         return (PARAM_ERROR_RESPONSE);
    } else
    {
        if(i32Port == 0)
30         {
            return (TEC0_PAGE_URI);
        } else
        {
            return (TEC1_PAGE_URI);
        }
35     }
}

```

The *iNumParams* variable specifies the number of parameters passed by the CGI request, the *pcParam* table contains pointers to the names of the passed parameters, and the *pcValue* table – pointers to the values of the passed parameters (in the form of char). The *ConfigGetCGIParam* function is used to read the values of the passed parameters. It finds the name of the parameter and converts its value from char to int. If the passed parameter has an incorrect value, the error page is returned to the client. If the parameter value is correct, the function handling the request is called. In this case, the *Set_TEC_limit* function sets the appropriate current limit value. After the value is set correctly, the proper web page is sent to the client. In this case, the page corresponding to the definition *TEC0_PAGE_URI* is *tec0.shtm*.

When handling a request to change the target temperature, the corresponding function looks slightly different. That is because the value of temperature is stored in memory in units of $m^{\circ}C$ whereas the user sends the temperature value in $^{\circ}C$ with the separator (".", " or ",").

The *ConfigSSIHandler* function handles SSI queries. When called, a string of characters is sent to functions that handle the web server. These characters are then placed in HTML code and the appropriate value is visible by the web client.

```

1  static uint16_t ConfigSSIHandler(int iIndex, char *pcInsert, int iInsertLen)
    {
        uint32_t ui32Port;
        int iCount;
2     const char *pcString;
        char answer[32];
        uint32_t ans_len = 0;
        ip_addr_t *addr_temp;

10     switch(iIndex)
        {
            //
            // IP address
            //
15         case SSI_INDEX_IPADDR:
            {
                uint32_t ui32IPAddr;

                ui32IPAddr = lwIPLocalIPAddrGet();
20         return(usnprintf(pcInsert, iInsertLen, "%d.%d.%d.%d",
                            ((ui32IPAddr >> 0) & 0xFF),

```

```

25         ((ui32IPAddr >> 8) & 0xFF),
           ((ui32IPAddr >> 16) & 0xFF),
           ((ui32IPAddr >> 24) & 0xFF));
    }
    //...

```

The *iIndex* variable contains the SSI tag number, according to the *g_pcConfigSSITags* pointers table passed to the *http_set_ssi_handler* function at the web server initialization.

```

1  static const char *g_pcConfigSSITags [] =
    {
        "ipaddr",    // SSI_INDEX_IPADDR
        "macaddr",  // SSI_INDEX_MACADDR
5     "p0br",       // SSI_INDEX_POBR
        "p0sb",     // SSI_INDEX_POsb
        "p0p",      // SSI_INDEX_POp
    //...

```

The answer to the SSI query is saved in the memory location pointed by *pcInsert*. The variable *iInsertLen* specifies the maximum length of this answer.

SSI queries are handled in switch ... case statement with a vast number of cases. For different SSI tags different actions are performed. Below is the case of TEC current limit:

```

1  //...
    case SSI_INDEX_TECLIM0:
    case SSI_INDEX_TECLIM1:
    {
5     ui32Port = (iIndex == SSI_INDEX_TECLIM0) ? 0 : 1;
        ans_len = Read_TEC(ui32Port, 0, answer);
        return(usnprintf(pcInsert, iInsertLen, "%s", answer));
    }
    //...

```

When the web server receives a request to load a web page, it is handled by a file system that is implemented in the file "enet_fs.c".

4.2.3 File system

Each page on the web server is saved in the form of a char table in the file "enet_fsdata.h". A simplified tree is implemented to search for the desired page. It consists of structures containing pointers to tables that store the content of web pages. The function *fs_open* [A.1.4] from the file "enet_fs.c" searches the tree and if it succeeds, the proper web page can be loaded into the output buffer.

4.2.4 EEPROM memory

The firmware of thermostat uses the available EEPROM memory to save and read parameters important for the project. The *Tiware* library provided by the manufacturer contains functions allowing simple EEPROM handling. All actions regarding EEPROM are implemented in the functions of the "config.c" file. While bootup, the microcontroller tries to read the parameters from EEPROM.

```

1  void
    ConfigLoad(void)
    {
        uint8_t *pui8Buffer;
5     pui8Buffer = EEPROMPBGet();

        if(pui8Buffer)
        {

```



```

10     g_sParameters = *(tConfigParameters *)pui8Buffer;
        g_sWorkingDefaultParameters = g_sParameters;
    }
}

```

The function *ConfigSave* is used to save parameters [A.1.5]. The list of parameters that are saved in the EEPROM memory is as follows:

- UART parameters (baud rate, data size, parity, stop bits, flow control)
- Telnet parameters (timeout, port number, IP address)
- Module name
- IP address of module
- Subnet mask
- Default gateway
- IP address of MQTT broker

4.2.5 Front-end of the web application

In the part responsible for the front-end of the web application, the HTML markup language, the CSS style sheet language and some JavaScript are used. All *.html and *.shtm documents as well as *.js and *.css files are located in the *fs* subdirectory of the main project directory. The documents use the SSI scripting language to retrieve values from the web server and the CGI interface that allows to send commands to the web server. An example of a fragment of a document that uses SSI and CGI:

```

1  <table width="100%" border="0" cellpadding="2" cellspacing="2">
    <tbody>
        <tr>
            <td width="30%" class="gr"></td>
5         <td width="30%" class="gl">Current </td>
            <td width="30%" class="gl">Updated </td>
            <td width="10%" class="gl"></td>
        </tr>
        <tr>
10         <td class="gr">TEC current limit: </td>
            <td><!--#teclim0 --> mA</td>
            <form name="teclim" action="teclim.cgi" method="get">
                <input name="port" value="0" type="hidden">
                <td><input maxLength="4" size="5" name="teclim">mA</td>
15             <td>
                <input name="mysubmit" value="Set" type="submit">
            </td>
            </form>
        </tr>
20         <tr>
            <td class="gr">Actual TEC current: </td>
            <td><!--#teccur0 --> mA</td>
            <td></td>
        </tr>
25         <tr>
            <td class="gr">Actual TEC voltage: </td>
            <td><!--#tecvol0 --> mV</td>
            <td></td>
        </tr>
30     </tbody>
</table>

```

When the page is loaded, the value of the maximum TEC current for the channel 0 will be inserted in the `<!--#teclim0-->` location. When sending the form *teclim*, a request *teclim.cgi* with the parameter *port*

= 0 and the parameter *teclim* with the value entered by the user in the input field will be sent to the server. After receiving such request, the web server will run the function setting the current limit in channel 0 to the received value.

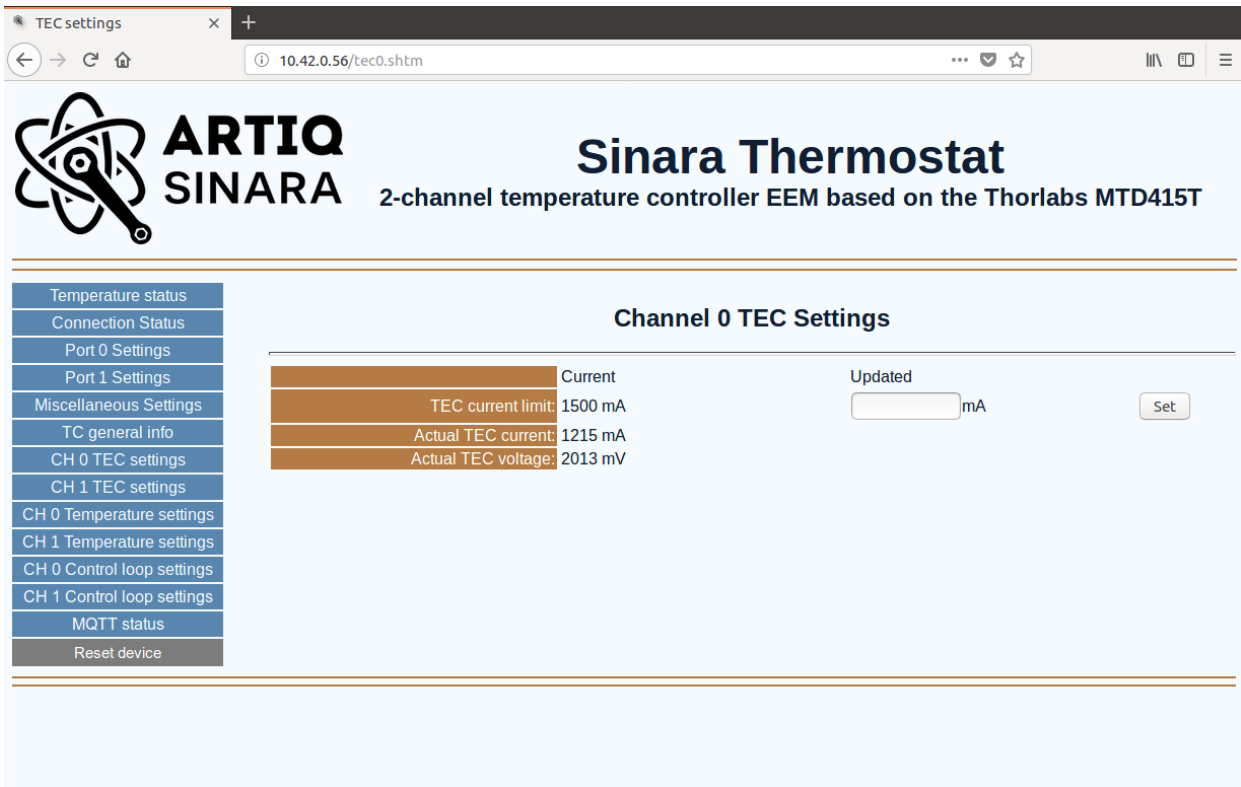


Figure 4.6. Layout of the sample webpage

4.2.6 Communication with MTD415T modules

The microcontroller *TM4C1294NCPDT* communicates with two *MTD415T* modules using UART. The microcontroller’s RX line for the first channel is, according to the device schematics, on the *PA4* pin, and the TX line is on the *PA5* pin. For the second channel, the RX line corresponds to the *PK0* pin and the TX line to the *PK1* pin.

The functions of *Tivaware* library used in the firmware allow simple sending (function *SerialSend* in the file "serial.c") and receiving (function *SerialReceive* in the "serial.c" file) characters with UART.

The "send_command.c" file of the project contains all the functions that support UART communication with the *Thorlabs* modules. In particular, the *Send_Command* and *Receive_Answer* functions play an important role of sending properly formatted commands to the *MTD415T* modules and receiving answers from them [A.1.6, A.1.7].

In the function that receives the response from the *MTD415T* chip, a simple timeout mechanism was implemented to prevent the device from hanging if there is any difficulty in communicating with the *Thorlabs* device. All UART receiving and transmitting actions are non-blocking in the firmware. That also prevents the program from hanging. If the timeout is reached, the *Receive_Answer* function sets the response "timeout" and this response is sent to the web client.

The functions *Send_Command* and *Receive_Answer* are the internal functions used by the other functions from the file "send_command.c". The other functions are responsible for the construction of commands sent to the devices of *Thorlabs*, consistent with the documentation provided by the manufacturer. To follow how these functions work, one can inspect some commands of *MTD415T*:

Table 4.2. Extract of the datasheet of *MTD415T*

Command	<i>MTD415T</i> action
Lx (x - current in range 200-2000 [mA])	Sets the TEC current limit to x
L?	Reads the TEC current limit [mA]
A?	Reads the actual TEC current [mA]
U?	Reads the actual TEC voltage [mV]

The *Set_TEC_limit* and *Read_TEC* functions are used to handle the commands presented in the above table.

```

1  bool Set_TEC_limit(uint32_t ui32Port, uint32_t limit)
   {
       char command[5];
5   command[0] = 'L';

       if(limit >= 200 && limit < 1000)
           {
               command[1] = ((limit/100) % 10) + '0';
10  command[2] = ((limit/10) % 10) + '0';
               command[3] = ((limit) % 10) + '0';
               Send_Command(ui32Port,command,4);
               return true;
           } else if(limit >= 1000 && limit <= 2000)
15  {
               command[1] = ((limit/1000) % 10) + '0';
               command[2] = ((limit/100) % 10) + '0';
               command[3] = ((limit/10) % 10) + '0';
               command[4] = ((limit) % 10) + '0';
20  Send_Command(ui32Port,command,5);
               return true;
           } else
           {
25  return false;
           }
   }

```

The function *Set_TEC_limit* constructs the command, but before sending the command to the TEC controller, the current limit value is validated. In case of an incorrect command (e.g. with a value out of range), the *MTD415T* reaction would be unpredictable from the designer's point of view, as the manufacturer did not describe how the TEC driver behaves in such circumstances. If the value of the variable *limit* does not meet the requirements, the function returns the value *false*, which results in sending the web page informing about the error in the parameter value ("error.shtm") to the web client. If the value of this variable is correct, the function constructs a command and uses the *Send_Command* function to send the command to the *Thorlabs* device. Validating the values is different for each parameter, that is why each setting command has a separate handling function.

```

1  uint32_t Read_TEC(uint32_t ui32Port, uint32_t mode, char *answerBuf)
   {
       ASSERT((mode == 0) || (mode == 1) || (mode == 2));
5   char command[2];

```

```

uint32_t ans_len;

switch(mode)
{
10  case 0:
        command[0] = 'L'; //reads the TEC current limit
        break;
    case 1:
15  case 1:
        command[0] = 'A'; //reads the actual TEC current
        break;
    case 2:
        command[0] = 'U'; //reads the actual TEC voltage
        break;
    }
20
    command[1] = '?';

    Send_Command(ui32Port ,command,2);
    ans_len = Receive_Answer(ui32Port , answerBuf);
25
    return ans_len;
}

```

The function responsible for the read commands has a simpler form, because there is no need to validate any data. After constructing and transmitting the command, the function calls the *Receive_Answer* function to receive data. There is no need to delay before calling this function because of the non-blocking nature of designed UART and the timeout mechanism.

Functions created in the "send_command.c" file are used while handling CGI requests and responding to SSI queries.

4.2.7 MQTT client

The MQTT protocol does not use the traditional client-server model, but it is based on the publication/subscription pattern. Therefore, there is no direct connection between the sender of the message (publisher) and its recipient (subscriber). The connection between the publisher and the subscriber is handled by a broker. Broker analyzes the messages that it receives and distributes them to subscribers.

MQTT is based on the TCP/IP model, therefore the broker, as well as each client must have a TCP/IP stack. In the thermostat project, the TCP/IP stack is implemented using the *lwIP* library. In the latest version of the *lwIP 2.1.2* library, the MQTT functionality has been added. However, *lwIP 2.1.2* is a new version and it has not yet been included in the libraries supplied by the microcontroller manufacturer. It was necessary to adapt the rest of the manufacturer's libraries to the latest version of lwIP (it was a need to define some functions, change the names of the lwIP functions).

Each MQTT message contains a topic – a string of characters that the broker uses to determine which clients should obtain a message. Topic has a hierarchical structure with a slash as a level separator. The form of topic should be consistent in the designed MQTT system. The good practices of designing the form of topic [14] are as follows:

- lack of leading forward slash
- not using spaces in a topic
- using only ASCII characters
- keeping the topic short and concise (save of resources)
- enabling extensibility

In the designed MQTT client, each action is a response to the publication of another client, i.e. it corresponds to the command-reaction model. Therefore, the client does not publish any messages by itself, e.g. it does not send information about temperature periodically. The designed form of the topic can be read from the table:

Topic				Action	
thermostat	/reset			Resets device	
	/ch0 or /ch1	/gen	/ver	/read	Reads the version of hardware and software of MTD415T
			/id	/read	Reads the ID of MTD415T
			/err	/read	Reads the error register of MTD415T
	/reset	Resets the error register			
	/tec	/curr	/read	Reads the actual current of TEC driver	
			/lim	/set	Sets the current limit of TEC driver
				/read	Reads the current limit of TEC driver
		/volt	/read	Reads the TEC voltage	
	/temp	/read	Reads the actual temperature		
			/set	/set	Sets the target temperature
		/win	/read	Reads the set temperature	
			/set	Sets the temperature window	
		/del	/read	Reads the temperature window	
			/set	Sets the delay time of status pin	
		/cloop	/gain	/read	Reads the delay time of status pin
				/set	Sets the critical gain
	/period		/read	Reads the critical gain	
			/set	Sets the critical period	
	/cyctime		/read	Reads the critical period	
			/set	Sets the cycling time	
	/P		/read	Reads the cycling time	
			/set	Sets the P share	
	/I		/read	Reads the P share	
/set			Sets the I share		
/D	/read	Reads the I share			
	/set	Sets the D share			
		/read	Reads the D share		

In the above table, the topic has been divided into hierarchical parts, therefore, to read the final form of the topic, it should be constructed from the appropriate cells of the table. For example, if reading the current limit in the channel 0, the appropriate topic is "thermostat/ch0/tec/curr/lim/read".

As for functions that require values to set, e.g. setting the current limit, the set value should be given in payload of MQTT messages in a format compatible with the web application.

All functions related to MQTT handling are implemented in the "mqtt_client.c" file.

Establishing a connection with the broker takes place in the function *do_connect* [A.1.8]. After successfully connecting to the broker or failing to connect several times, the *mqtt_connection_cb* function is called [A.1.9].

If the client and the broker are connected, the client starts the subscription of "thermostat/#", i.e. it starts to track all the topics that begin with the string "thermostat". The function *mqtt_incoming_publish_cb* is a callback function called when any message from the subscribed topic reaches the MQTT client [A.1.10].

One of the input arguments of this function is a pointer to a string containing the topic of the MQTT message. Depending on the content of this string, the different action is taken. The function *parse_topic*

is used to divide the topic into hierarchical parts. The extended *if* statement is a place, where the appropriate action is chosen. In case of setting a parameter, the appropriate number is assigned to the variable *inpub_id*. This variable is used by the function that analyzes the payloads of MQTT messages. In the case of reading action, the proper function that deals with the publication of data in the MQTT network is called.

The function *mqtt_incoming_data_cb* is a callback function called in response to a payload coming with a MQTT message [A.1.11]. It is responsible for ensuring that the parameter defined in the topic of the MQTT message takes the value sent in payload.

In the *if* statement of the function, appropriate functions are called depending on the value of the variable *inpub_id*. For example, for the topic "thermostat/ch0/tec/curr/lim/set", the *mqtt_incoming_publish_cb* function assigns 6 to the *inpub_id* variable. Therefore, in the function *mqtt_incoming_data_cb* the function *Set_TEC_limit* is called. This function is responsible for setting the current limit.

The "mqtt_client.c" file also contains a set of functions that are responsible for publishing the data, when the read command is received from the MQTT broker. For example - assuming that MQTT client received a message with the topic "thermostat/ch0/tec/curr/lim/read", the variable *inpub_id* takes the value 0 and the function *publish_tec* is called.

```

1 void publish_tec(mqtt_client_t *client, void *arg, const char *topic)
  {
    char answer[32];
    err_t err;
5   u8_t qos = 1; /* quality of service */
    u8_t retain = 1; /* do retain */

    if(inpub_id >= 0 && inpub_id <= 2) {
        ans_len = Read_TEC(ui32Port, inpub_id, answer);
10        err = mqtt_publish(client, topic, answer, ans_len, qos, retain, mqtt_pub_request_cb, arg);
        if(err != ERR_OK) {
            UARTprintf("Publish error: %d\n", err);
        }
    }
15 }

```

The function *Read_TEC* and the library function *mqtt_publish* that publishes the message in the MQTT network are used here.

In the entire part of the project responsible for MQTT communication, the UART interface available to the user (*PA0* and *PA1* pins of the microcontroller) was used to send messages about MQTT status and any errors.

4.3 Description of the firmware for thermostat_max

The temperature controller in the version with *MAXI968* modules differs significantly from the thermostat_thorlabs version in the part responsible for temperature control. In the device with *Thorlabs* modules temperature reading, Peltier current setting and PID control were handled by the firmware of *MTD415T*. In the thermostat_max version, all these functionalities had to be implemented in the designed firmware. To perform the above-mentioned tasks, it is necessary to make microcontroller work with two *MAXI968* drivers and *AD7172-2* analog-to-digital converter.

The part of the project responsible for Ethernet communication, file system, EEPROM memory remained the same as in the thermostat_thorlabs version. As for the web application and MQTT communication, it was only necessary to adapt the relevant parts of the project to the version with

MAX1968 drivers.

4.3.1 The concept of temperature control in the thermostat_max version

In the firmware of this version of the device, a proportional-integral-derivative controller has been implemented, consisting of three parts: proportional, integral and differential. The input variable of the controller is the temperature, and the output variable – the current of the Peltier module. The measured temperature values are obtained by reading the voltage on temperature sensors using AD7172-2 ADC. Communication of the microcontroller with the ADC is via the SPI interface. The current value for the Peltier module is set by MAX1968 basing on PWM signals on specific MAX1968 pins. Providing appropriate PWM signals on some pins of the MAX1968 allows also to set the positive and negative current limit and the voltage limit of the Peltier module. The MAX1968 driver, apart from the output pins to be connected to the Peltier module, has two outputs. The voltage on these outputs is proportional to the actual Peltier voltage and current. These values can be calculated using the formula provided by the manufacturer. The firmware uses an internal analog-to-digital converter of TM4C1294NCPDT to read the voltage on both these MAX1968 outputs.

It was assumed in the project that the temperature would be measured either by Pt100/Pt1000 resistance temperature detectors (RTDs) or by NTC/PTC thermistors.

4.3.2 SPI communication

SPI communication with the Analog Devices' ADC has been implemented in the project's "spi.c" file. The *spi_config* function is responsible for the initialization of the SPI and the proper configuration of the converter. It activates and changes settings of both input channels and selects wanted operating mode (continuous conversion mode).

The *spi_getvalue* function is responsible for getting the value from the analog-to-digital converter [A.1.12]. As a result, the value taken from the ADC is returned, while the status of ADC is saved in the memory location pointed by *channel*. Status of ADC determines the channel for which the conversion was done.

The temperature of the thermistor or resistance temperature detector is calculated in the *getTemp* function [A.1.13]. For the NTC/PTC thermistor, a simple equation describing the dependence of thermistor resistance on its temperature was used [27].

$$R_T = R_{T_0} e^{\beta \left(\frac{1}{T} - \frac{1}{T_0} \right)} \quad (4.1)$$

, where R_T is thermistor resistance at temperature T , R_{T_0} is thermistor resistance at reference temperature T_0 , β is parameter specific for a thermistor.

For a Pt100/Pt1000 resistance thermometer, the value of temperature is calculated using the Callendar - Van Dusen equation for temperatures above $0^\circ C$ [32].

$$R_T = R_{T_0} \cdot \left(1 + A \cdot (T - T_0) + B \cdot (T - T_0)^2 \right) \quad (4.2)$$

R_T - resistance of RTD at temperature T , R_{T_0} - resistance of thermistor at reference temperature T_0 , A and B - parameters specific for the material of RTD (platinum).

All material parameters needed for temperature calculations are entered by the user via the web application or using the MQTT protocol. The ratio $\frac{R_T}{R_{T_0}}$ is calculated from the value of the voltage read from the ADC.

4.3.3 PWM

The functioning of *MAX1968* depends on PWM signals on some pins of the driver. The average voltage applied to the input pins of *MAX1968* is regulated by using the appropriate duty cycle. Average voltage on these particular pins is used by *MAX1968* to determine the Peltier current and the current and voltage limits. The functions responsible for control of PWM signals are implemented in the project's "pwm.c" file. In the function *pwm_config*, all needed microcontroller resources are initialized. The other functions of the file are responsible for setting individual signals, e.g. when setting the PWM signal responsible for the Peltier current of channel 0, the *setCurr0* function is called.

```

1 void setCurr0(int32_t Curr) {
    if( Curr > IMAX ) {
        Curr = IMAX;
    }
5   if( Curr < -IMAX ) {
        Curr = -IMAX;
    }

    Curr = (1500+Curr*1500/3000);
10  if(Curr != 0) {
        TimerMatchSet(TIMER3_BASE, TIMER_A, TimerLoadGet(TIMER3_BASE, TIMER_A) * \
            (3300 - Curr) /3300);
    } else {
        TimerMatchSet(TIMER3_BASE, TIMER_A, TimerLoadGet(TIMER3_BASE, TIMER_A) - 1);
15  }
}

```

The conditional statements are used here to prevent exceeding the current limit specified by the *IMAX* definition. Duty cycle calculation is based on the datasheet of *MAX1968* driver, where it is specified how the voltage level at the PWM pin translates into (in this case) the value of current.

4.3.4 Internal ADC

The voltages on the two output pins of the *MAX1968* driver are proportional to the current and voltage of the Peltier module. In the thermostat project, to read these voltages, microcontroller's built-in ADC is used. It is a 12-bit precision analog-to-digital converter with a maximum sample rate of 2 million samples per second and hardware averaging of up to 64 samples.

The code responsible for handling the internal ADC can be found in the file "internal_adc.c" of the project. The function *internal_adc_config* is responsible for initializing the converter, while the functions *internal_adc_getvalues0* and *internal_adc_getvalues1* - for getting the value from it.

```

1 void internal_adc_getvalues0(uint32_t* Value) {
    uint32_t ADC_SEQ = 1;
5   ADCProcessorTrigger(ADC0_BASE, ADC_SEQ);

    while(!ADCIntStatus(ADC0_BASE, ADC_SEQ, false))
    {
10  }

    ADCIntClear(ADC0_BASE, ADC_SEQ);

    ADCSequenceDataGet(ADC0_BASE, ADC_SEQ, Value);

```


Calling the *ADCProcessorTrigger* function starts the sampling sequence defined in the initializing function. The value from the ADC is taken in the *ADCSequenceDataGet* function and is written to the memory location pointed by *Value*.

4.3.5 PID controller

The purpose of the PID controller is to maintain the temperature at the level of the value set by the user via the web application or via the MQTT protocol. The PID controller in the project determines the value of the Peltier current based on the difference between the measured and set temperature. The value of current is calculated by adding three parts:

- proportional P – the greater the difference between measured and set temperature is, the greater its share is
- integral I – it is responsible for compensating for past errors and its contribution is proportional to the sum of the specified number of recently recorded errors
- differential D – it is to compensate for the expected error and its contribution is greater when changes in the measured temperature are greater

The basic formula describing the operation of the PID controller is as follows [28]:

$$u(t) = P \cdot e(t) + I \cdot \int_0^t e(\tau) d\tau + D \cdot \frac{de}{dt} \quad (4.3)$$

e - error value, in this case the difference between the measured and set temperature

u - control output, for the thermostat it is a value of Peltier current

P, I, D - parameters determined by the user, by changing the values of these parameters the user changes the share of PID parts in control output

The thermostat project uses a discrete version of the 4.3 equation. Instead of the integral presented, the sum of the errors multiplied by the relevant time was used.

$$\sum_{k=0}^n e(t_k) \cdot (t_k - t_{k-1})$$

Instead of a derivative, the appropriate quotient was used.

$$\frac{e(t_n) - e(t_{n-1})}{t_n - t_{n-1}}$$

The part of the code responsible for PID control can be found in the project's "pid_task.c" file. The function that handles PID control has been defined as the task of the FreeRTOS system and is initialized in the *PIDTaskInit* function. The task's name is *PIDTask* [A.1.14].

Each time the task is launched by the system, the temperature value is taken from the *AD7172-2* converter. Then the corresponding Peltier current is calculated and set using *setCurr0* or *setCurr1* function. The function *SetStat* is used to switch the diode on or off depending on whether the value of the measured temperature is within or out of the user-defined range around the set temperature. The while loop of the *PIDTask* function is run by the *FreeRTOS* every 75 ms.

4.3.6 Watchdog

In the thermostat project Watchdog was implemented to protect the system from hanging and the resulting need to manually reset the device. In the *TM4C1294NCPDT* microcontroller, the Watchdog functioning looks as below [19]:

- After enabling, the Watchdog counter starts working. When it reaches 0, the Watchdog interrupt is called and the value from the WDTLOAD register is loaded into the Watchdog counter. The counter starts counting down again.
- If the counter reaches 0 again and the Watchdog interrupt flag is not cleared, the microcontroller resets.
- In case the Watchdog interrupt flag is cleared before the second interrupt, the value from the WDTLOAD register is loaded into the Watchdog counter and the countdown starts again.

In the thermostat_max project, the value of WDTLOAD was chosen so that the Watchdog interrupt occurs every 1 second. Watchdog initialization takes place inside the *Watchdog_config* function defined in the project's "watchdog.c" file. Clearing the Watchdog interrupt flag takes place inside a while loop of the *PIDTask*, which should be called every 75 ms if the device works properly.

4.3.7 Changes in other parts of the project

Thanks to the modularity of the designed firmware, adapting it to the version with *MAX1968* drivers was relatively easy. The parts responsible for the web application and MQTT communication needed to be changed. Also, functions used by the web server and the MQTT client from the "send_command.c" file required some changes.

SSI tags and related functions were added to the "config.c" file. Functions that were not implemented in thermostat_thorlabs version include in particular those related to the calibration of Pt100/Pt1000 RTDs and NTC/PTC thermistors. Handling new CGI requests has been implemented in this version of the web application.

Parameters that previously were sent to *MTD415T* drivers, in this version are saved to the EEPROM memory. Therefore, the list of parameters that are saved to the EEPROM memory looks as below:

- UART parameters (baud rate, data size, parity, stop bits, flow control)
- Telnet parameters (timeout, port number, IP address)
- Module name
- IP address of module
- Subnet mask
- Default gateway
- IP address of MQTT broker
- PID parameters for each channel
- Target temperature for each channel
- Calibration parameters for Pt100/Pt1000 and NTC/PTC
- Current limit, voltage limit, temperature window, *MAX1968* switching frequency for each channel

Significant changes were made in the file "send_command.c". The *Send_Command* and *Receive_Avwer* functions responsible for UART communication with the *MTD415T* modules were removed. In the version with the Thorlabs chips, most of the functions in this file were to send commands and receive responses from the *MTD415T* drivers. In thermostat_max most functions reduces to writing or reading from the EEPROM memory. An example is *Read_MaxVolt* function responsible for reading the voltage limit:

```
1  uint32_t Read_MaxVolt(uint32_t ui32Port, char *answerBuf)
   {
   5      uint32_t ans_len;
      if(ui32Port == 0) {
          ans_len = usnprintf(answerBuf, 32, "%d", g_sParameters.ui16MaxVolt0);
      } else {
10         ans_len = usnprintf(answerBuf, 32, "%d", g_sParameters.ui16MaxVolt1);
      }
      return ans_len;
   }
```

In turn, one of the exceptions is *Set_MaxVolt*, which is called when setting the voltage limit:

```
1  bool Set_MaxVolt(uint32_t ui32Port, int32_t voltage)
   {
   5      if(voltage < 0 || voltage > 5000) {
          return false;
      }
      if(ui32Port == 0) {
10         g_sParameters.ui16MaxVolt0 = voltage;
          setMaxVolt0((uint16_t) voltage);
      } else {
          g_sParameters.ui16MaxVolt1 = voltage;
          setMaxVolt1((uint16_t) voltage);
      }
15      return true;
   }
```

Here, besides the write to the memory, the appropriate PWM signal is set using *setMaxVolt0* or *setMaxVolt1* function.

The way the MQTT client works remained the same. However, the form of the MQTT topic was changed to adapt it to new functionalities:

		Topic		Action			
thermostat	/reset			Resets device			
		/saveall		Saves all parameters to EEPROM			
	/ch0 or /ch1	/on		Turns TEC driver on			
		/off		Turns TEC driver off			
	/onoff?				Sends 'ON' if TEC driver is on or 'OFF' if it is off		
	/cal	/t0	/set		Sets the reference temperature for calibration		
			/read		Reads the reference temperature for calibration		
		/beta	/set		Sets the β parameter for calibration		
			/read		Reads the β parameter for calibration		
		/ratio	/set		Sets the ratio parameter for calibration		
			/read		Reads the ratio parameter for calibration		
		/tempmode	/set		Sets the type of temperature sensor; '0' - NTC/PTC, '1' - Pt100/Pt1000		
			/read		Reads the type of temperature sensor		
		/pta	/set		Sets the A parameter for Pt100/Pt1000		
			/read		Reads the A parameter for Pt100/Pt1000		
		/ptb	/set		Sets the B parameter for Pt100/Pt1000		
			/read		Reads the B parameter for Pt100/Pt1000		
		/freq	/set		Sets the switching frequency of TEC driver		
			/read		Reads the switching frequency of TEC driver		
	/rawtemp		/read		Reads the raw value received from AD7172-2		
	/tec	/curr	/read		Reads the actual current of TEC driver		
			/lim	/pos	/set	Sets the positive current limit of TEC driver	
				/read		Reads the current limit of TEC driver	
			/neg	/set	Sets the negative current limit of TEC driver		
		/read		Reads the negative current limit of TEC driver			
		/volt	/read		Reads the TEC voltage		
			/lim	/set	Sets the voltage limit of TEC driver		
	/read			Reads the voltage limit of TEC driver			
	/temp	/read		Reads the actual temperature			
		/set	/set		Sets the target temperature		
			/read		Reads the set temperature		
		/win	/set		Sets the temperature window		
			/read		Reads the temperature window		
	/loop	/P	/set		Sets the P share		
			/read		Reads the P share		
		/I	/set		Sets the I share		
			/read		Reads the I share		
		/D	/set		Sets the D share		
	/read		Reads the D share				

The above table should be read as in the thermostat_thorlabs version, i.e. the final form of the topic should be constructed from the table cells. When reading the reference temperature for calibration in channel 1, the topic is "thermostat/ch1/cal/t0/read".

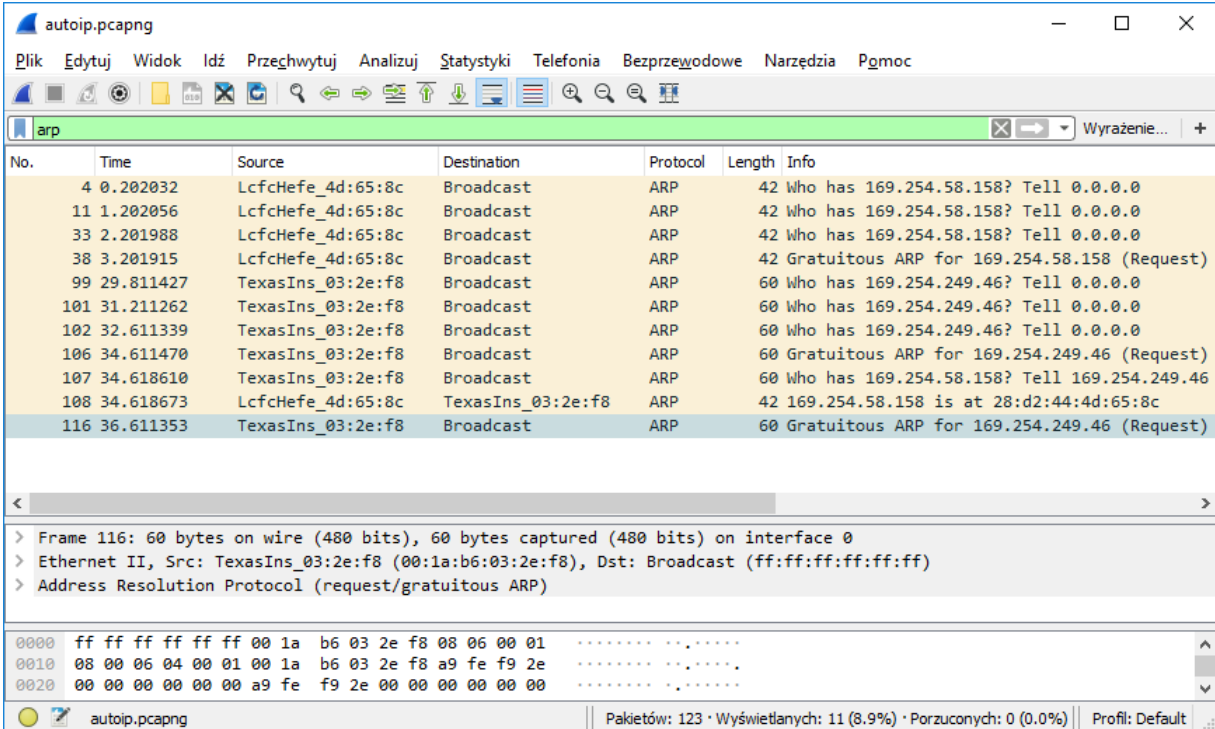
If required, a set value should be given in the payload of MQTT message in a format compatible with the web application.

5 Testing of the firmware

5.1 Testing the thermostat_thorlabs version

The PCB project of the thermostat with *MTD415T* drivers was not finally sent for production, therefore all tests were performed using the *TM4C1294XL* evaluation board from *Texas Instruments* and the USB-UART converter module based on CP2102.

The module correctly establishes communication via Ethernet in all modes - AUTOIP, DHCP and Static IP. DHCP mode was tested using the *ZTE ZXV10 H202N* router. Below is the window of the *Wireshark* program, where one can see the way AUTOIP mode works.



No.	Time	Source	Destination	Protocol	Length	Info
4	0.202032	LcfcHefe_4d:65:8c	Broadcast	ARP	42	Who has 169.254.58.158? Tell 0.0.0.0
11	1.202056	LcfcHefe_4d:65:8c	Broadcast	ARP	42	Who has 169.254.58.158? Tell 0.0.0.0
33	2.201988	LcfcHefe_4d:65:8c	Broadcast	ARP	42	Who has 169.254.58.158? Tell 0.0.0.0
38	3.201915	LcfcHefe_4d:65:8c	Broadcast	ARP	42	Gratuitous ARP for 169.254.58.158 (Request)
99	29.811427	TexasIns_03:2e:f8	Broadcast	ARP	60	Who has 169.254.249.46? Tell 0.0.0.0
101	31.211262	TexasIns_03:2e:f8	Broadcast	ARP	60	Who has 169.254.249.46? Tell 0.0.0.0
102	32.611339	TexasIns_03:2e:f8	Broadcast	ARP	60	Who has 169.254.249.46? Tell 0.0.0.0
106	34.611470	TexasIns_03:2e:f8	Broadcast	ARP	60	Gratuitous ARP for 169.254.249.46 (Request)
107	34.618610	TexasIns_03:2e:f8	Broadcast	ARP	60	Who has 169.254.58.158? Tell 169.254.249.46
108	34.618673	LcfcHefe_4d:65:8c	TexasIns_03:2e:f8	ARP	42	169.254.58.158 is at 28:d2:44:4d:65:8c
116	36.611353	TexasIns_03:2e:f8	Broadcast	ARP	60	Gratuitous ARP for 169.254.249.46 (Request)

> Frame 116: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0
> Ethernet II, Src: TexasIns_03:2e:f8 (00:1a:b6:03:2e:f8), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
> Address Resolution Protocol (request/gratuitous ARP)

```
0000 ff ff ff ff ff ff 00 1a b6 03 2e f8 08 06 00 01 .....  
0010 08 00 06 04 00 01 00 1a b6 03 2e f8 a9 fe f9 2e .....  
0020 00 00 00 00 00 00 a9 fe f9 2e 00 00 00 00 00 00 .....
```

Figure 5.1. AUTOIP mode captured by *Wireshark*

In this mode, a static IP address of 169.254.xxx.xxx is randomly generated, and then ARP request for this address is sent several times. In the above case, the answer did not happen, consequently, the drawn address was accepted as its own.

Since the thermostat_max version was not finally produced, testing of the web application and MQTT communication was troublesome, because all interactions of the microcontroller with the *MTD415T* modules had to be simulated using the USB-UART converter and the *Realterm* software. With the

assistance of these tools, it was possible to observe the commands reaching *MTD415T* and the thermostat responses to UART messages coming from the (simulated) Thorlabs devices. With this *MTD415T* simulation, it was found that the firmware is working properly. Particular actions taken by the user of the web application invoke the transmission of the expected commands to the USB-UART converter. In turn, when sending UART messages to the microcontroller, the microcontroller correctly interprets them. For example, when the microcontroller sent "Te?" command to the USB-UART converter (query for the actual temperature) and *Realterm* responded "23050", the temperature value was correctly displayed in the web application, and also correctly sent in the MQTT publication.

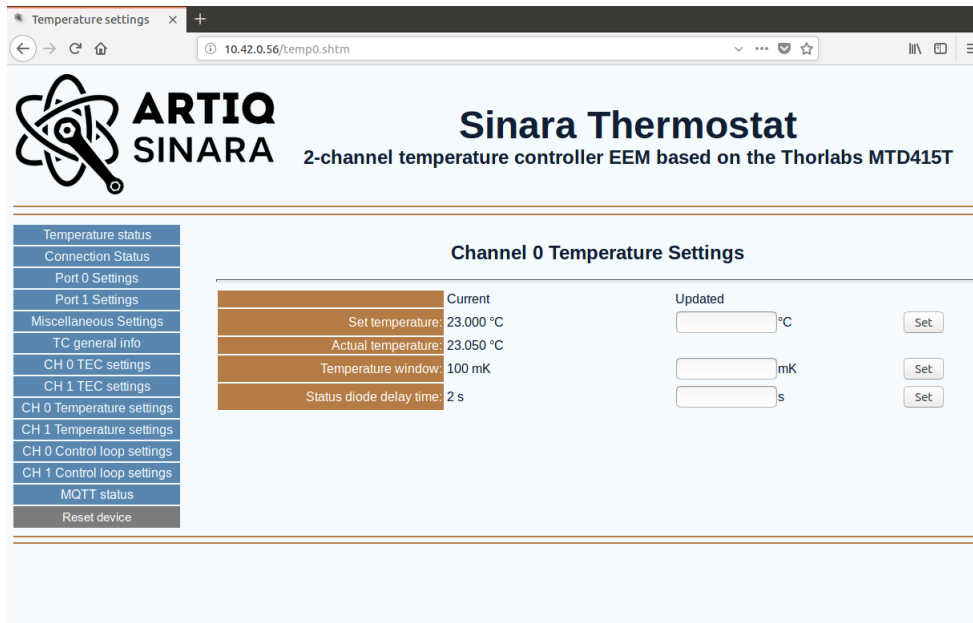


Figure 5.2. Testing of the web application

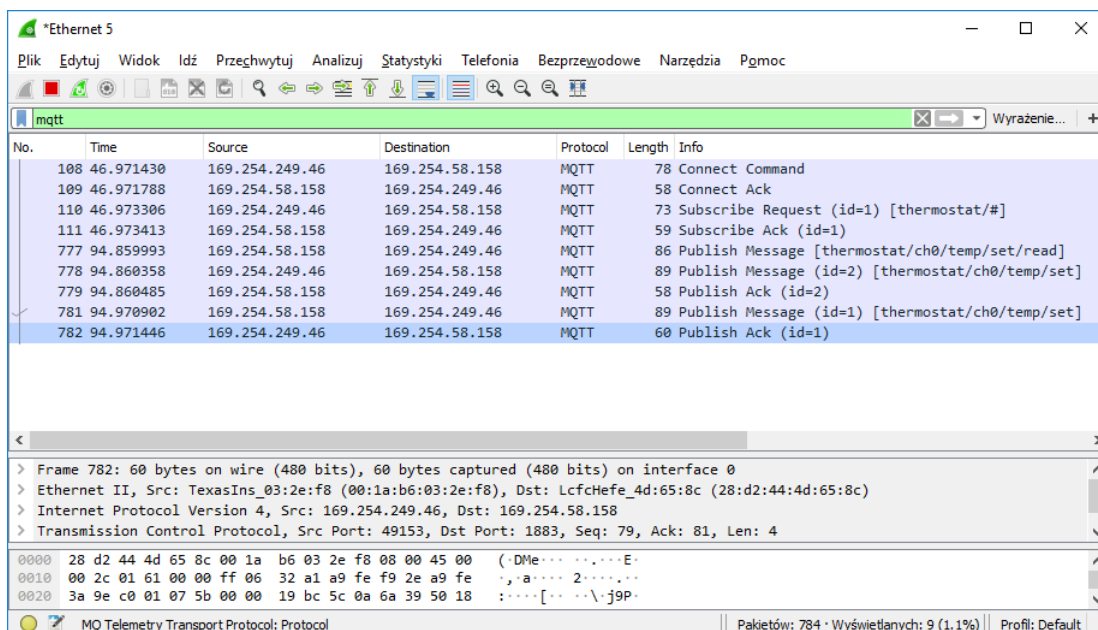


Figure 5.3. MQTT communication captured by *Wireshark*

As one can see in the example above, MQTT communication starts with *connect command* sent by the thermostat. To initiate a connection, the implemented client sends a command message with a proper content: client id, clean session flag, last will topic, keep alive interval. The broker accepts the connection and it responds with a *connect ack* message. This message contains two data entries: session present flag and connect acknowledge flag. Subsequently, the thermostat sends *subscribe request*, that consists of three components: topic length, topic name and requested quality of service. The *subscribe ack* message with a return code is then sent back by the broker. The user of the thermostat sends the *publish message* with the topic "thermostat/ch0/temp/set/read". The broker forwards the message to the device and, after processing, the thermostat sends the *publish message* with a value of the actual temperature in the payload. Each time the *publish message* has QOS level 1, the receiver answers with *publish ack*.

All tests possible to perform using the evaluation board showed the proper functioning of the designed firmware.

5.2 Testing the thermostat_max version

Tests of the thermostat_max version were made using a manufactured PCB of thermostat, two Peltier modules with a maximum operating current of 10A, two PTC thermistors (*KTY81-110* from *NXP*) and two Pt1000 RTDs from *Jumo*.

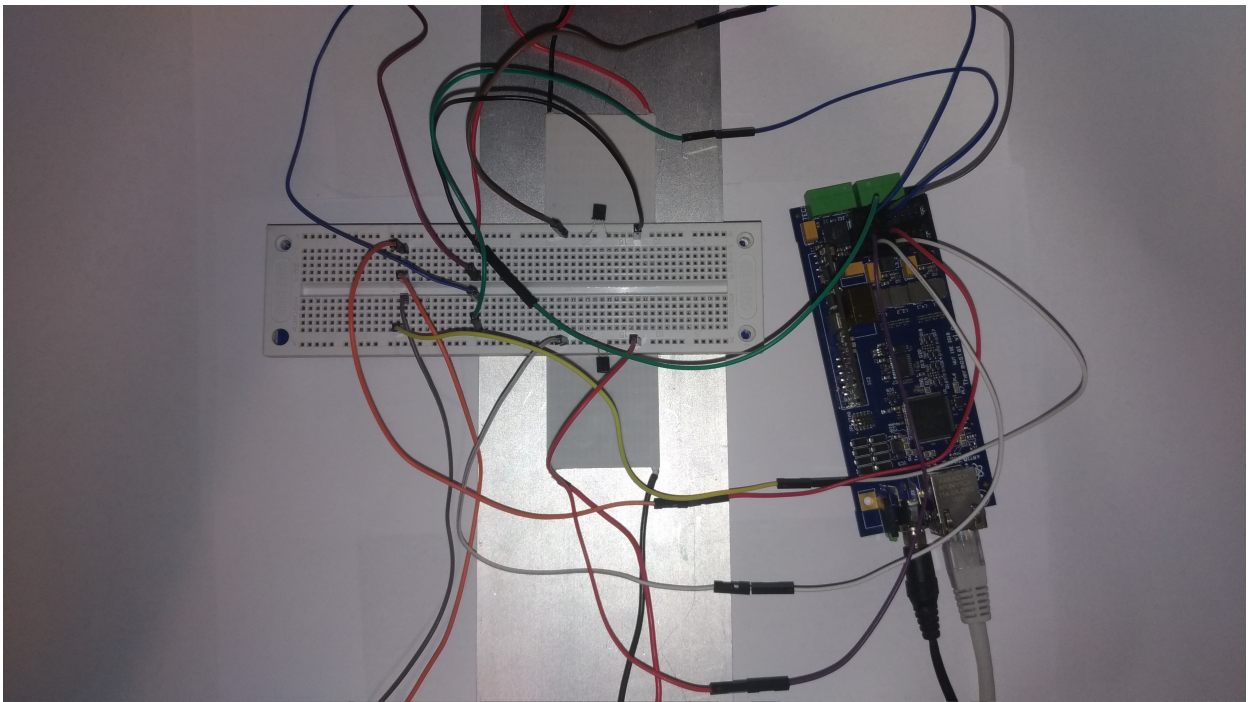


Figure 5.4. Photograph of a system for testing with PTC thermistors

5.2.1 Functional tests of individual parts of the project

UART communication in the module works correctly, messages about the status of the Ethernet and MQTT connection appear on the *PA1* pin. Figure 4.3 shows the window of *Realterm* while validating UART communication.

The module successfully establishes an Ethernet connection and MQTT connection. It was tested using the Wireshark program.

The web application was tested using two browsers - *Firefox* v. 65.0.2 and *Opera* v. 58.0.3135.47. All application hyperlinks work correctly and lead to correct subpages. If one tries to open a non-existent subpage, page 404 is displayed. All SSI queries sent by the web client are correctly interpreted by the microcontroller - correct values returned by the web server appear in the right places. Also, all CGI requests sent using forms on web pages are well interpreted by the microcontroller.

The part of the firmware responsible for MQTT communication works properly. Using the *Mosquitto* broker and the *MQTT Spy* application, all designed MQTT topics were tested. For each MQTT publication from the *MQTT Spy*, the microcontroller reacts as expected.

The microcontroller correctly handles EEPROM memory, after power cycling values of parameters are loaded from memory.

The watchdog was tested by adding while loop with condition that is always true so that the watchdog counter is not loaded. After the relevant time, the module resets and the factory parameters are loaded when restarting.

The microcontroller successfully communicates through the SPI with the ADC and correctly converts the values returned by it to the temperature. Setting the PWM signals also works properly – it was tested using the *YF-3503* digital multimeter from *YU FUNG*.

5.2.2 Functional tests of the entire project

After successfully testing all parts of the thermostat firmware, functional tests of the entire project were also performed. The purpose of these tests was to verify whether the module works properly after setting temperature controller parameters. It is expected that after turning on the temperature control in a given channel, the temperature measured by the sensor will approach the set temperature, and when it is reached, it will be maintained in the close neighborhood of the set temperature.

The measurement system shown in the picture 5.4 was set up. PTC thermistors were used as temperature sensors. This system was used to examine whether the thermostat module works properly in both channels with PTC/NTC thermistors. It has been noted that the functioning of the module is correct. When the measured temperature is greater than the set temperature, the device forces the Peltier current that causes cooling the PTC side of the Peltier module. In turn, when the measured temperature is lower than the set temperature, the PTC side of the Peltier module heats up.

In order to illustrate the performance of the temperature controller, a script for the *MQTT Spy* was written. The script enables temperature control in a given channel, and then every half a second reads the value of the measured temperature in this channel. In one of the channels, the target temperature was set to be lower than the ambient temperature and in the second - higher than the ambient temperature. The test results are presented in the charts below.

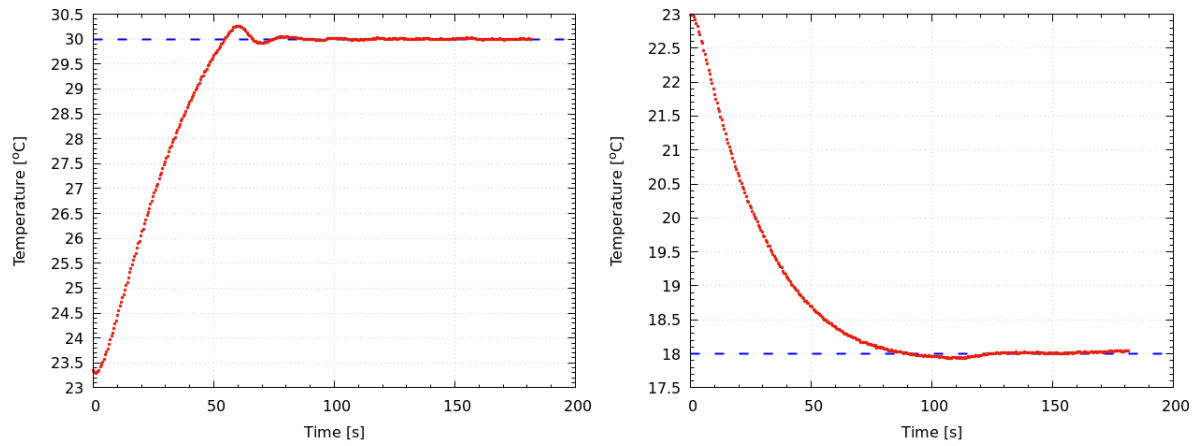


Figure 5.5. Temperature measured by the PTC thermistor as a function of time in the channel 0 and 1 respectively. $P = -2000 \frac{mA}{K}$, $I = -300 \frac{mA}{K \cdot s}$, $D = -500 \frac{mA \cdot s}{K}$

The way the controller works depends on the values of parameters P, I, D. The conclusion from the above measurements is that the thermostat module works as expected with PTC thermistors. If PID parameters are tuned, the actual temperature reaches the target temperature and performs smaller and smaller wobbles around it.

For further tests, PTC thermistors were replaced with Pt1000 sensors. The MQTT Spy script was used again and measurements were taken. The results are presented in the graphs below.

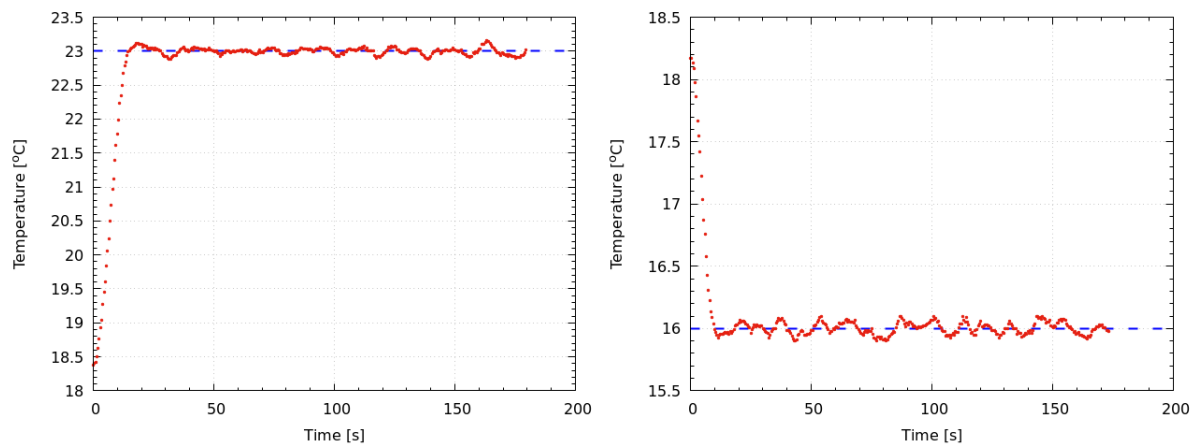


Figure 5.6. The left graph shows the temperature measured by Pt1000 on channel 0 as a function of time ($P = -1500 \frac{mA}{K}$, $I = -145 \frac{mA}{K \cdot s}$, $D = -2500 \frac{mA \cdot s}{K}$). The right graph shows the same for channel 1. ($P = -1000 \frac{mA}{K}$, $I = -100 \frac{mA}{K \cdot s}$, $D = -1500 \frac{mA \cdot s}{K}$).

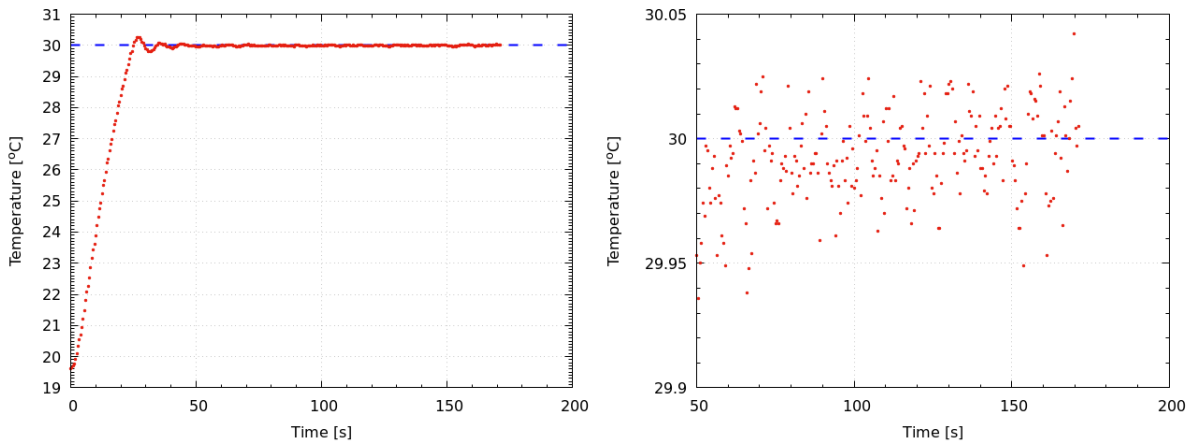


Figure 5.7. The temperature measured by Pt1000 on channel 0 as a function of time. The left graph shows all measurements, the right graph shows the situation after 50 seconds. $P = -2000 \frac{mA}{K}$, $I = -200 \frac{mA}{K \cdot s}$, $D = -500 \frac{mA \cdot s}{K}$

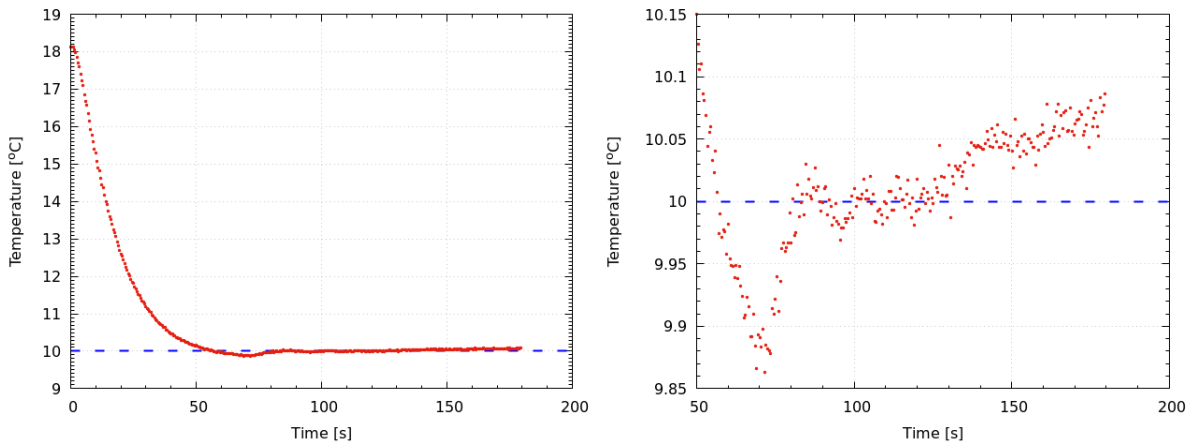


Figure 5.8. The temperature measured by Pt1000 on channel 1 as a function of time. The left graph shows all measurements, the right graph shows the situation after 50 seconds. $P = -2000 \frac{mA}{K}$, $I = -300 \frac{mA}{K \cdot s}$, $D = -300 \frac{mA \cdot s}{K}$

As one can see, selected parameters of PID controller ensure the maintenance of temperature at the level of even $\pm 15mK$ around the target temperature. It may be supposed that the use of software PID tuners of the Sinara team can increase the stability to the target level of $\sim 1mK$.

All the above-presented measurements used data from an analog-to-digital converter as a source of knowledge about the state of the system. Therefore, they are subject to possible errors related to ADC, e.g. its non-linearity. In order to make a decisive test of the thermostat, the measuring system shown in the diagram below was set up.

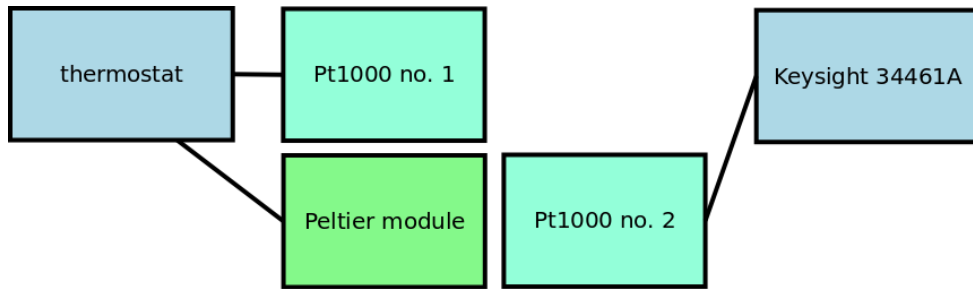


Figure 5.9. Block diagram of a measuring system with a *Keysight 34461A* digital multimeter

One of the Pt1000 sensors was connected to channel 0 of the thermostat device. The second one was connected to the inputs of *Keysight 34461A* digital multimeter available at the Institute of Electronic Systems. Both sensors were placed on one Peltier module connected to channel 0 of the board. The picture 5.10 shows the Peltier module together with the Pt1000 sensors in the measurement system.

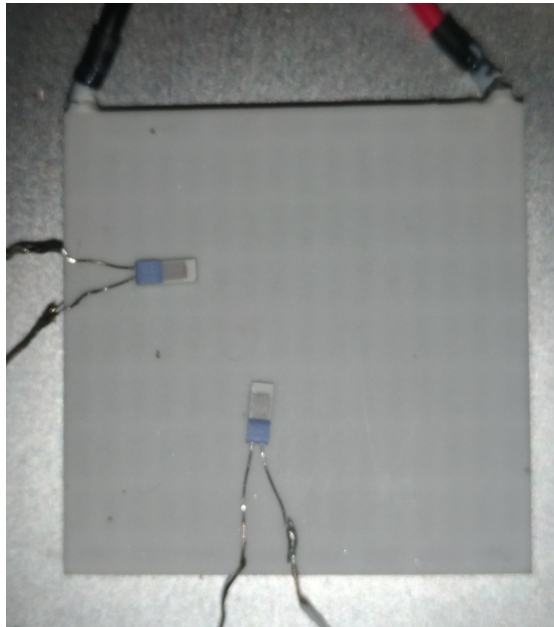


Figure 5.10. Photograph of the Peltier module and Pt1000 sensors

The measurements consisted of the simultaneous use of a digital multimeter and the script for *MQTT Spy*. The script was getting temperature values from the thermostat module using the MQTT protocol. At the same time, the resistance of the second Pt1000 sensor was being measured using the *Keysight 34461A* multimeter. The *Benchvue* software made available by the manufacturer under the trial license was used to operate the multimeter. With the assistance of the *Benchvue* program, it was possible to save data from the multimeter to the PC.

Measurements were taken for different set temperatures. The appropriate parameters of the PID controller have been determined for all measurements. The graphs showing the results of the tests are listed below.

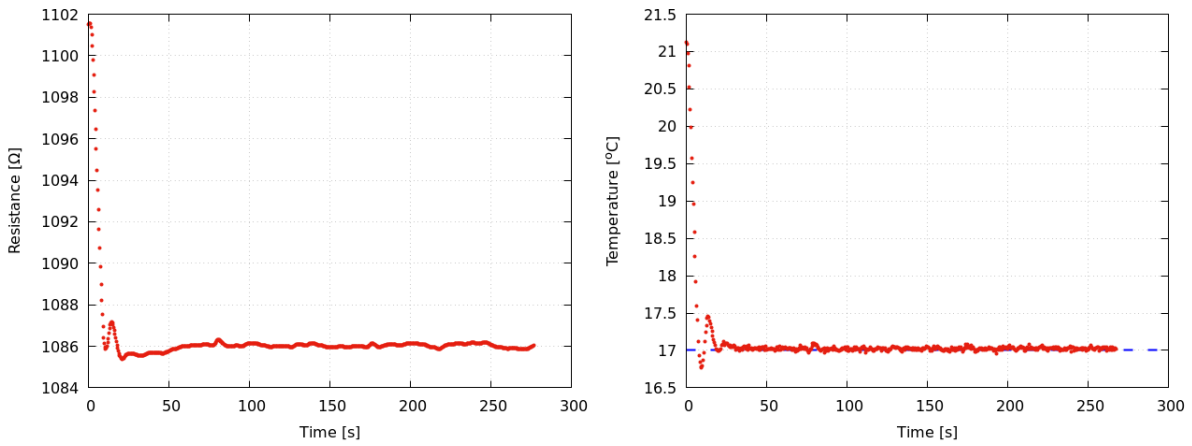


Figure 5.11. 150/5000 Left: Pt1000 resistance measured by *Keysight 34461A* as a function of time. Right: Temperature measured by thermostat as a function of time. $P = -1000 \frac{mA}{K}$, $I = -400 \frac{mA}{K \cdot s}$, $D = -100 \frac{mA \cdot s}{K}$

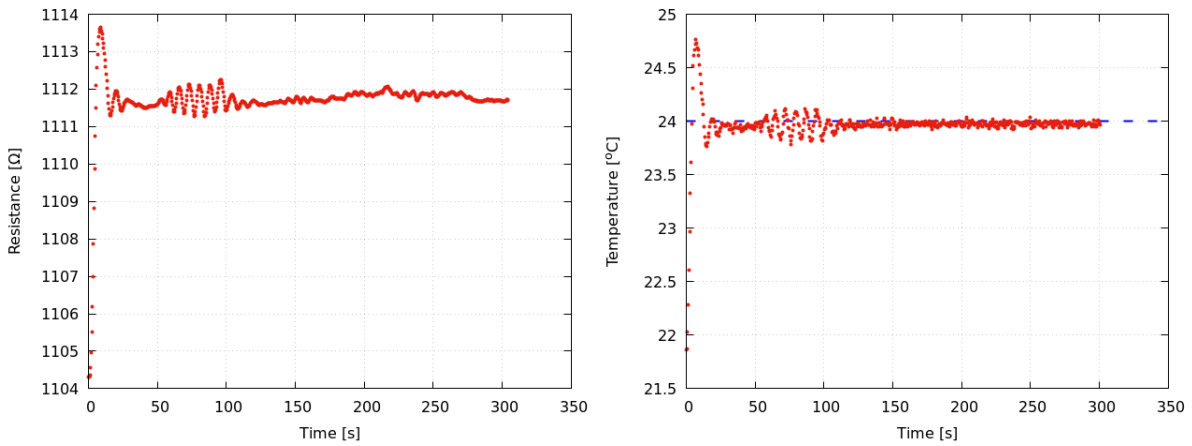


Figure 5.12. Left: Pt1000 resistance measured by *Keysight 34461A* as a function of time. Right: Temperature measured by thermostat as a function of time. $P = -2000 \frac{mA}{K}$, $I = -300 \frac{mA}{K \cdot s}$, $D = -200 \frac{mA \cdot s}{K}$

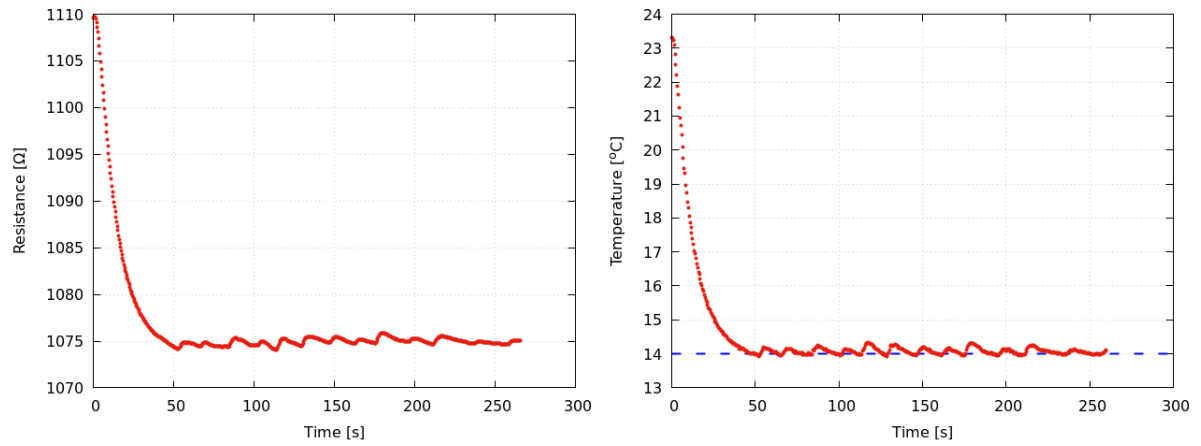


Figure 5.13. Left: Pt1000 resistance measured by *Keysight 34461A* as a function of time. Right: Temperature measured by thermostat as a function of time. $P = -2000 \frac{m\Omega}{K}$, $I = -1000 \frac{m\Omega}{K \cdot s}$, $D = -2000 \frac{m\Omega \cdot s}{K}$

It can be noticed that changes in resistance measured by the *Keysight 34461A* multimeter are very similar to the changes in temperature read from the thermostat. This means that thermostat measurements are not biased by ADC errors more than a high-class multimeter is. The tests can therefore be considered successful as they showed the correct functioning of the designed firmware.

6 Conclusions

The main result of the thesis is the designed firmware for two-channel temperature controller of the Sinara hardware family. The firmware was created for two versions of the device: the one with *MTD415T* chips and the one with *MAX1968* chips. The project meets all the requirements presented in the design assumptions. Temperature can be controlled in two channels independently using web application or MQTT messaging protocol. All the external libraries used in the project are open-source. This includes FreeRTOS and lwIP libraries. The configuration of the connection can be modified via web application if need. The modularity of the firmware as well as using RTOS allows easy modification in case of any changes of the PCB or need to add functionalities.

Although complex testing of the firmware for the version with *Thorlabs* chips was not possible due to abandoning the production of this version, the tests were performed with the use of evaluation board and USB-UART converter. It turned out, that the firmware works correctly, however, the ultimate validation of the functioning should be made in case of manufacturing the *thermostat_thorlabs* PCB. The firmware for the thermostat with *MAX1968* chips was thoroughly tested. Besides testing all functionalities of the firmware, the tests of the whole device were made. It turned out, that, if PID parameters were tuned, the device managed to quickly reach and hold the target temperature. The maximum stability of the temperature obtained in tests was at the level of $\sim 15mK$. However, as the reaching the target stability level of $\sim 1mK$ was not the purpose of the thesis, tuning the PID controller was left in the hands of physicists of Sinara project.

Before adopting the device in the *ARTIQ* system, other tests should be performed, including measuring errors induced by TEC current and measuring the influence of the noise on the device's measurements.

The thermostat with the designed firmware is meant to be used in the working *ARTIQ* systems after the next necessary tests. The further development of the firmware is relatively simple and it should not be a problem to introduce new solutions. The suggested modifications are:

- simplification of the code, i.e. review of all functions and seek for possible redundancies
- usage of a template system for a web application to ensure easier changes of web pages
- adding a possibility to change the form of the MQTT topic from the web application – it would be very useful to add a unique identifier of the device to the topic in case of more than one thermostat in the MQTT system

As for the possible modifications of the device, it is worth considering to replace *TM4C1294NCPDT* with *STM32* microcontroller and Ethernet PHY chip. It would reduce the cost of the device, while the cost of the changes in the firmware would be low.

List of Figures

2.1	Four-channel universal controller <i>AR654</i> from <i>Apar</i> [3]	5
2.2	<i>Uniplex III</i> heating controller from <i>Klöpper Therm</i> [3]	6
3.1	Simplified schematic diagram of thermostat_ <i>thorlabs</i>	7
3.2	Block diagram of <i>Ag5300</i> [2]	8
3.3	<i>MTD415T</i> [22]	9
3.4	Simplified schematic diagram of thermostat_ <i>max</i>	10
3.5	Block diagram of <i>MAX1968</i> [11]	11
3.6	Block diagram of <i>AD7172-2</i> [1]	12
3.7	<i>lwIP</i> logo [9]	13
3.8	<i>FreeRTOS</i> logo [8]	14
4.1	<i>TM4C1294XL</i> evaluation board	16
4.2	GUI of <i>Code Composer Studio</i>	16
4.3	GUI of <i>Realterm</i>	17
4.4	GUI of <i>Wireshark</i>	17
4.5	GUI of <i>MQTT Spy</i>	18
4.6	Layout of the sample webpage	24
5.1	AUTOIP mode captured by <i>Wireshark</i>	35
5.2	Testing of the web application	36
5.3	MQTT communication captured by <i>Wireshark</i>	36
5.4	Photograph of a system for testing with PTC thermistors	37
5.5	Temperature control graph for PTC	39
5.6	Temperature control for Pt1000	39
5.7	Temperature control chart in channel <i>0</i> for Pt1000	40
5.8	Temperature control chart in channel <i>1</i> for Pt1000	40
5.9	Block diagram of a measuring system with a <i>Keysight 34461A</i> digital multimeter	41
5.10	Photograph of the Peltier module and Pt1000 sensors	41
5.11	Measurements thermostat + <i>Keysight 34461A</i> for a set temperature of $17^{\circ}C$	42
5.12	Measurements thermostat + <i>Keysight 34461A</i> for a set temperature of $24^{\circ}C$	42
5.13	Measurements thermostat + <i>Keysight 34461A</i> for a set temperature of $14^{\circ}C$	43

List of Tables

3.1	Selected parameters of the <i>TM4C12294NCPDT</i> microcontroller [19].	8
3.2	Selected parameters of the <i>MTD415T</i> [22]	9
4.1	Functions that handle CGI requests and actions performed by these functions	20
4.2	Extract of the datasheet of <i>MTD415T</i>	25

A Appendix

A.1 Source code of the firmware

A.1.1 lwIPTaskInit

```
1  uint32_t
   lwIPTaskInit(void)
   {
       uint32_t ui32User0, ui32User1;
5     uint8_t pui8MAC[6];

       //
       // Get the MAC address from the user registers.
       //
10    MAP_FlashUserGet(&ui32User0, &ui32User1);
       if((ui32User0 == 0xffffffff) || (ui32User1 == 0xffffffff))
       {
           return(1);
       }
15

       //
       // Convert the 24/24 split MAC address from NV ram into a 32/16 split MAC
       // address needed to program the hardware registers, then program the MAC
       // address into the Ethernet Controller registers.
20    //
       pui8MAC[0] = ((ui32User0 >> 0) & 0xff);
       pui8MAC[1] = ((ui32User0 >> 8) & 0xff);
       pui8MAC[2] = ((ui32User0 >> 16) & 0xff);
       pui8MAC[3] = ((ui32User1 >> 0) & 0xff);
25    pui8MAC[4] = ((ui32User1 >> 8) & 0xff);
       pui8MAC[5] = ((ui32User1 >> 16) & 0xff);

       //
       // Lower the priority of Ethernet interrupt
30    //
       MAP_IntPrioritySet(INT_EMAC0, ETHERNET_INT_PRIORITY);

       //
       // Set the link status
35    //
       g_bLinkStatusUp = GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_3) ? false : true;

       //
       // Initialize lwIP.
40    //
       lwIPInit(g_ui32SysClock, pui8MAC, 0, 0, 0, IPADDR_USE_DHCP);

       //
       // Setup remaining services of TCP/IP
45    //
       tcpip_callback(SetupServices, 0);

       return(0);
   }
```

A.1.2 SerialTask

```

1  static void
SerialTask(void *pvParameters)
{
    tSerialEvent sEvent;
5
    while(1)
    {
        //
        // Block until a message is put on g_QueueSerial by interrupt handler.
10        //
        xQueueReceive(g_QueueSerial, (void*) &sEvent, portMAX_DELAY);

        //
        // The first part of the message is type of event.
15        //

        if(sEvent.eEventType == RX)
        {
            //
            // If Telnet protocol is enabled, check for incoming IAC character and escape it.
20            //
            if((g_sParameters.sPort[sEvent.ui8Port].ui8Flags &
                PORT_FLAG_PROTOCOL) == PORT_PROTOCOL_TELNET)
            {
                //
                // If this is a IAC character, write it twice.
                //
                if((sEvent.ui8Char == TELNET_IAC) &&
                    (RingBufFree(&g_sRxBuf[sEvent.ui8Port]) >= 2))
30                {
                    RingBufWriteOne(&g_sRxBuf[sEvent.ui8Port], sEvent.ui8Char);
                    RingBufWriteOne(&g_sRxBuf[sEvent.ui8Port], sEvent.ui8Char);
                }

                //
                // If not IAC character, write it just once.
                //
                else if((sEvent.ui8Char != TELNET_IAC) &&
40                    (RingBufFree(&g_sRxBuf[sEvent.ui8Port]) >= 1))
                {
                    RingBufWriteOne(&g_sRxBuf[sEvent.ui8Port], sEvent.ui8Char);
                }
            }

            //
            // If not Telnet, then write the data once.
            //
            else
50            {
                RingBufWriteOne(&g_sRxBuf[sEvent.ui8Port], sEvent.ui8Char);
            }
        }
        //
        // Check if it is a TX interrupt
        //
55        else
        {
            //
            // Loop while there is space in the transmit FIFO and characters to be sent.
            //
60            while(!RingBufEmpty(&g_sTxBuf[sEvent.ui8Port]) &&
                UARTSpaceAvail(g_ui32UARTBase[sEvent.ui8Port]))
            {
                //
                // Write the next character into the transmit FIFO.
                //
65                UARTCharPut(g_ui32UARTBase[sEvent.ui8Port],
                    RingBufReadOne(&g_sTxBuf[sEvent.ui8Port]));
            }
70        }
    }
}

```

A.1.3 vApplicationIdleHook

```
1 void
  vApplicationIdleHook(void)
  {
5     uint32_t ui32Temp;
      portTickType ui32CurrentTick, ui32InitialTick;

      //
      // Get current IP address.
      //
10     ui32Temp = lwIPLocalIPAddrGet();

      //
      // See if IP address has changed.
      //
15     if(ui32Temp != g_ui32IPAddress)
        {
            //
            // Save current IP address.
            //
20             g_ui32IPAddress = ui32Temp;

            //
            // Display IP address.
            //
25             DisplayIP(ui32Temp);
        }

      //
      // Check for IP update request.
      //
30     if(g_ui8UpdateRequired)
        {

35         //
        // Check if "ui32InitialTick" is to be initialized.
        //
        if(bFirst)
40         {
            //
            // Get the initial tick count (to calculate two seconds delay)
            //
            ui32InitialTick = xTaskGetTickCount();

45             //
            // Reset 'bFirst'
            //
            bFirst = false;
50         }

            //
            // Get current Tick count.
            //
55             ui32CurrentTick = xTaskGetTickCount();

            //
            // Check if 2 seconds have lapsed.
            //
60             if((ui32CurrentTick - ui32InitialTick) > 2000 / portTICK_RATE_MS)
                {

65                 //
                // Updating only IP address?
                //
                if(g_ui8UpdateRequired & UPDATE_IP_ADDR)
                    {
70                     //
                        // Update IP address.
```

```

//
g_ui8UpdateRequired &= ~UPDATE_IP_ADDR;
75 ConfigUpdateIPAddress ();
}

//
// Updating all parameters?
80 //
//
if(g_ui8UpdateRequired & UPDATE_ALL)
{
//
// Update all.
85 //
//

g_ui8UpdateRequired &= ~UPDATE_ALL;
ConfigUpdateAllParameters(true);
90 }

//
// Set 'bFirst'
//
95 bFirst = true;
}
}
}

```

A.1.4 fs_open

```

1 err_t fs_open(struct fs_file *psFile, const char *pcName)
{
const struct fsdata_file *psTree;
5 err_t err;

//
// Allocate memory
//
10 if(psFile == NULL)
{
err = ERR_MEM;
return(err);
}

15 //
// Initialize the file system tree pointer
//
psTree = FS_ROOT;

20 //
// Search for requested file name.
//
while(NULL != psTree)
25 {
//
// Compare the requested name to name in current node.
//
30 if(ustrncmp(pcName, (char *)psTree->name, psTree->len) == 0)
{

psFile->data = (char *)psTree->data;
psFile->len = psTree->len;

35 //
// Setup read index to end of file
//
psFile->index = psTree->len;

40 //
// Not using any file system extensions
//

```



```

        psFile->pextension = NULL;
45
        err = ERR_OK;
        break;
    }
50
    //
    // Did not find the file at this node. Get the next element in the list.
    //
    psTree = psTree->next;
55
    err = ERR_ARG;
}

//
60 // Didn't find the file -> psTree will be NULL
//
if(psTree == NULL)
{
    mem_free(psFile);
65     psFile = NULL;
}

return(err);
}

```

A.1.5 ConfigSave

```

1 void
  ConfigSave(void)
  {
    uint8_t *pui8Buffer;
5
    //
    // Save working defaults parameter block
    //
    EEPROMBsave((uint8_t *)&g_sWorkingDefaultParameters);
10
    //
    // Get pointer to recently saved buffer
    //
    pui8Buffer = EEPROMBGet();
15
    //
    // Update default parameter pointer
    //
    if(pui8Buffer)
20     {
        g_psDefaultParameters = (tConfigParameters *)pui8Buffer;
    }
    else
    {
25     g_psDefaultParameters = (tConfigParameters *)g_psFactoryParameters;
    }
}

```

A.1.6 Send_Command

```

1 void Send_Command(uint32_t ui32Port, const char *commandBuf, uint32_t ui32Len)
  {
    ASSERT((ui32Port == 0) || (ui32Port == 1));
5
    unsigned int charNum = 0;
    while(charNum < ui32Len)
    {

```

```

10     if(! SerialSendFull(ui32Port))
        {
            SerialSend(ui32Port , commandBuf[charNum]);
            charNum = charNum + 1;
        }
    }
15     if(! SerialSendFull(ui32Port)) SerialSend(ui32Port , '\r');
    if(! SerialSendFull(ui32Port)) SerialSend(ui32Port , '\n');
}

```

A.1.7 Receive_Answer

```

1  uint32_t Receive_Answer(uint32_t ui32Port , char *answerBuf)
    {
        ASSERT((ui32Port == 0) || (ui32Port == 1));
5
        unsigned int charNum = 0;
        int32_t buf;
        int8_t cbuf;
        int i = 0;
10       int imax = 3000000; // timeout about 1 second
        bool timein = true;

        while(timein)
        {
15           cbuf = -1;

            if(ui32Port == 0)
            {
                while(!UARTCharsAvail(S2E_PORT0_UART_PORT) && timein)
20                 {
                    if(i > imax)
                    {
                        timein = false;
                        charNum = 0xFF;
25                     }
                    i = i + 1;
                }
                if(UARTCharsAvail(S2E_PORT0_UART_PORT))
                {
30                     buf = MAP_UARTCharGetNonBlocking(S2E_PORT0_UART_PORT);
                    cbuf = (unsigned char)(buf & 0xFF);
                }
            } else
            {
35                 while(!UARTCharsAvail(S2E_PORT1_UART_PORT) && timein)
                {
                    if(i > imax)
                    {
                        timein = false;
                        charNum = 0xFF;
40                     }
                    i = i + 1;
                }
                if(UARTCharsAvail(S2E_PORT1_UART_PORT))
                {
45                     buf = MAP_UARTCharGetNonBlocking(S2E_PORT1_UART_PORT);
                    cbuf = (unsigned char)(buf & 0xFF);
                }
            }
50           if(cbuf != -1)
            {
                if((cbuf == '\r') || (cbuf == '\n') || (cbuf == 0x1b))
                {
55                     answerBuf[charNum] = cbuf;
                    break;
                }
            } else
            {

```

```

60         answerBuf[charNum] = cbuf;
           charNum = charNum + 1;
           }
       }
65     }
       if(charNum == 0xFF)
       {
           answerBuf[0] = 't'; answerBuf[1] = 'i'; answerBuf[2] = 'm'; answerBuf[3] = 'e';
           answerBuf[4] = 'o'; answerBuf[5] = 'u'; answerBuf[6] = 't'; answerBuf[7] = '\n';
70         charNum = 8;
       }

       return charNum;
   }

```

A.1.8 do_connect

```

1 void do_connect()
  {
    struct mqtt_connect_client_info_t ci;
    err_t err;
5
    /* Setup an empty client info structure */
    memset(&ci, 0, sizeof(ci));

    ci.client_id = "thermostat";
10    ci.will_topic = "unexpected exit";

    /* Initiate client and connect to server*/

    ip_addr_t* ip_addr;
15    ip_addr = (ip_addr_t*) mem_malloc(sizeof(ip_addr_t));
    ip_addr->addr = g_sParameters.ui32mqttip;

    err = mqtt_client_connect(client, ip_addr, MQTT_PORT, mqtt_connection_cb, 0, &ci);

20    /* Print the result code if something goes wrong*/
    if(err != ERR_OK) {
        UARTprintf("mqtt_connect return %d\n", err);
    }

25    mem_free((void *) ip_addr);
  }

```

A.1.9 mqtt_connection_cb

```

1 static void mqtt_connection_cb(mqtt_client_t *client, void *arg, mqtt_connection_status_t status)
  {
    err_t err;
    if(status == MQTT_CONNECT_ACCEPTED) {
5        UARTprintf("MQIT: Successfully connected\n");

        mqtt_conn = true;

        /* Setup callback for incoming publish requests */
10        mqtt_set_inpub_callback(client, mqtt_incoming_publish_cb, mqtt_incoming_data_cb, arg);

        /* Subscribe to a topic named "thermostat/#" with QoS level 1,
           call mqtt_sub_request_cb with result */
        err = mqtt_subscribe(client, "thermostat/#", 1, mqtt_sub_request_cb, arg);
15
        if(err != ERR_OK) {
            UARTprintf("MQIT Error: mqtt_subscribe return: %d\n", err);
        }
    } else
20    {
        UARTprintf("MQIT: Disconnected, reason: %d\n", status);
    }
  }

```

```

    mqtt_conn = false;
}
}

```

A.1.10 mqtt_incoming_publish_cb

```

1 static void mqtt_incoming_publish_cb(void *arg, const char *topic, u32_t tot_len)
  {

    char *topic_p[6];
5   char *topic_pub = (char *) mem_malloc(64, sizeof(char));
    int i, topics_number;
    for(i = 0; i < 6; i = i+1) {
        topic_p[i] = (char *) mem_malloc(32, sizeof(char));
    }
10
    if(strcmp(topic, "thermostat/reset") == 0) {
        SysCtlReset();
    }

15   topics_number = parse_topic(topic, topic_p, strlen(topic));

    inpub_id = -1;

    /* Decode topic string into a user defined reference */
20   if(topics_number >= 4 && topics_number <= 6) {
        if(strcmp(topic_p[1], "ch0") == 0 || strcmp(topic_p[1], "ch1") == 0) {
            strncpy(topic_pub, topic, strlen(topic) - strlen(topic_p[topics_number - 1]) - 1);
            ui32Port = atoi(topic_p[1]+2);
            if(strcmp(topic_p[2], "gen") == 0) {
25                if(strcmp(topic_p[3], "ver") == 0 && strcmp(topic_p[4], "read") == 0) {
                    inpub_id = 0;
                    publish_general(client, arg, topic_pub);
                } else if(strcmp(topic_p[3], "id") == 0 && strcmp(topic_p[4], "read") == 0) {
                    inpub_id = 1;
                    publish_general(client, arg, topic_pub);
30                } else if(strcmp(topic_p[3], "err") == 0 && strcmp(topic_p[4], "read") == 0) {
                    inpub_id = 2;
                    publish_general(client, arg, topic_pub);
                } else if(strcmp(topic_p[3], "err") == 0 && strcmp(topic_p[4], "reset") == 0) {
35                    inpub_id = 3;
                    publish_general(client, arg, topic_pub);
                }
            } else if(strcmp(topic_p[2], "tec") == 0) {
                if(strcmp(topic_p[3], "curr") == 0 && strcmp(topic_p[4], "read") == 0) {
40                    inpub_id = 1;
                    publish_tec(client, arg, topic_pub);
                } else if(strcmp(topic_p[3], "curr") == 0 && strcmp(topic_p[4], "lim") == 0 \
&& strcmp(topic_p[5], "set") == 0) {
                    inpub_id = 6;
45                } else if(strcmp(topic_p[3], "curr") == 0 && strcmp(topic_p[4], "lim") == 0 \
&& strcmp(topic_p[5], "read") == 0) {
                    inpub_id = 0;
                    publish_tec(client, arg, topic_pub);
                } else if(strcmp(topic_p[3], "volt") == 0 && strcmp(topic_p[4], "read") == 0) {
50                    inpub_id = 2;
                    publish_tec(client, arg, topic_pub);
                }
            } else if(strcmp(topic_p[2], "temp") == 0) {
                if(strcmp(topic_p[3], "read") == 0) {
55                    inpub_id = 1;
                    publish_temp(client, arg, topic_pub);
                } else if(strcmp(topic_p[3], "set") == 0 && strcmp(topic_p[4], "set") == 0) {
                    inpub_id = 7;
                } else if(strcmp(topic_p[3], "set") == 0 && strcmp(topic_p[4], "read") == 0) {
60                    inpub_id = 0;
                    publish_temp(client, arg, topic_pub);
                } else if(strcmp(topic_p[3], "win") == 0 && strcmp(topic_p[4], "set") == 0) {
                    inpub_id = 8;
                } else if(strcmp(topic_p[3], "win") == 0 && strcmp(topic_p[4], "read") == 0) {
65                    inpub_id = 2;
                }
            }
        }
    }
}

```

```

        publish_temp(client, arg, topic_pub);
    } else if(strcmp(topic_p[3], "del") == 0 && strcmp(topic_p[4], "set") == 0) {
        inpub_id = 9;
    } else if(strcmp(topic_p[3], "del") == 0 && strcmp(topic_p[4], "read") == 0) {
70         inpub_id = 3;
        publish_temp(client, arg, topic_pub);
    }
} else if(strcmp(topic_p[2], "cloop") == 0) {
    if(strcmp(topic_p[3], "gain") == 0 && strcmp(topic_p[4], "set") == 0) {
75         inpub_id = 10;
    } else if(strcmp(topic_p[3], "gain") == 0 && strcmp(topic_p[4], "read") == 0) {
        inpub_id = 0;
        publish_cloop(client, arg, topic_pub);
    } else if(strcmp(topic_p[3], "period") == 0 && strcmp(topic_p[4], "set") == 0) {
80         inpub_id = 11;
    } else if(strcmp(topic_p[3], "period") == 0 && strcmp(topic_p[4], "read") == 0) {
        inpub_id = 1;
        publish_cloop(client, arg, topic_pub);
    } else if(strcmp(topic_p[3], "cyctime") == 0 && strcmp(topic_p[4], "set") == 0) {
85         inpub_id = 12;
    } else if(strcmp(topic_p[3], "cyctime") == 0 && strcmp(topic_p[4], "read") == 0) {
        inpub_id = 2;
        publish_cloop(client, arg, topic_pub);
    } else if(strcmp(topic_p[3], "P") == 0 && strcmp(topic_p[4], "set") == 0) {
90         inpub_id = 13;
    } else if(strcmp(topic_p[3], "P") == 0 && strcmp(topic_p[4], "read") == 0) {
        inpub_id = 3;
        publish_cloop(client, arg, topic_pub);
    } else if(strcmp(topic_p[3], "I") == 0 && strcmp(topic_p[4], "set") == 0) {
95         inpub_id = 14;
    } else if(strcmp(topic_p[3], "I") == 0 && strcmp(topic_p[4], "read") == 0) {
        inpub_id = 4;
        publish_cloop(client, arg, topic_pub);
    } else if(strcmp(topic_p[3], "D") == 0 && strcmp(topic_p[4], "set") == 0) {
100         inpub_id = 15;
    } else if(strcmp(topic_p[3], "D") == 0 && strcmp(topic_p[4], "read") == 0) {
        inpub_id = 5;
        publish_cloop(client, arg, topic_pub);
    }
}
}
}
}

mem_free(topic_pub);
110 for(i = 0; i < 6; i = i + 1) {
    mem_free(topic_p[i]);
}
}
}

```

A.1.11 mqtt_incoming_data_cb

```

1 static void mqtt_incoming_data_cb(void *arg, const u8_t *data, u16_t len, u8_t flags)
{
    bool bRetcode;
    int32_t i32Value = 0;
5     int i = 0;
    int j = 0;
    char *value;

    if(flags & MQTT_DATA_FLAG_LAST & (len > 0)) {
10         value = (char *) mem_malloc(32, sizeof(char));
        strncpy(value, (char *) data, len);
        /* Last fragment of payload received */

        /* Call function or do action depending on inpub_id */
15         if(inpub_id == 6) { //set current limit
            bRetcode = ConfigCheckDecimalParam(value, &i32Value);
            if(bRetcode) {
                if(!Set_TEC_limit(ui32Port, i32Value)) {
                    UARTprintf("MQTT: incorrect value\n");
                }
            }
        }
    }
}

```

```

20     }
    } else {
        UARTprintf("MQIT: incorrect value\n");
    }
} else if(inpub_id == 7) { //set temperature
25     while(data[i] != '.' && data[i] != ',' && i < len) {
        value[i] = data[i];
        i = i + 1;
    }
    while(i < len - 1) {
30         value[i] = data[i+1];
        if(isdigit(value[i])) {
            j = j + 1;
        }
        i = i+1;
35     }
    if(j > 3) {
        value[i-j+3] = NULL;
    }
    while(j < 3) {
40         value[i] = '0';
        i = i + 1;
        j = j + 1;
    }
    value[i] = NULL;
45     bRetcode = ConfigCheckDecimalParam(value, &i32Value);
    if(bRetcode) {
        if(!Set_Temp(ui32Port, i32Value)) {
            UARTprintf("MQIT: incorrect value\n");
        }
50     } else {
        UARTprintf("MQIT: incorrect value\n");
    }
} else if(inpub_id == 8) { //set temperature window
    bRetcode = ConfigCheckDecimalParam(value, &i32Value);
55     if(bRetcode) {
        if(!Set_Temp_Window(ui32Port, i32Value)) {
            UARTprintf("MQIT: incorrect value\n");
        }
    } else {
60         UARTprintf("MQIT: incorrect value\n");
    }
} else if(inpub_id == 9) { //set temperature diode delay time
    bRetcode = ConfigCheckDecimalParam(value, &i32Value);
    if(bRetcode) {
65         if(!Set_Temp_Delay(ui32Port, i32Value)) {
            UARTprintf("MQIT: incorrect value\n");
        }
    } else {
70         UARTprintf("MQIT: incorrect value\n");
    }
} else if(inpub_id == 10) { //set gain
    bRetcode = ConfigCheckDecimalParam(value, &i32Value);
    if(bRetcode) {
75         if(!Set_Gain(ui32Port, i32Value)) {
            UARTprintf("MQIT: incorrect value\n");
        }
    } else {
        UARTprintf("MQIT: incorrect value\n");
    }
80 } else if(inpub_id == 11) { //set period
    bRetcode = ConfigCheckDecimalParam(value, &i32Value);
    if(bRetcode) {
        if(!Set_Period(ui32Port, i32Value)) {
85             UARTprintf("MQIT: incorrect value\n");
        }
    } else {
        UARTprintf("MQIT: incorrect value\n");
    }
} else if(inpub_id == 12) { //set cycling time
90     bRetcode = ConfigCheckDecimalParam(value, &i32Value);
    if(bRetcode) {
        if(!Set_CT(ui32Port, i32Value)) {

```

```

        UARTprintf("MQTT: incorrect value\n");
    }
95     } else {
        UARTprintf("MQTT: incorrect value\n");
    }
    } else if(inpud_id >= 13 && inpud_id <= 15) { //set PID
        bRetcode = ConfigCheckDecimalParam(value, &i32Value);
100     if(bRetcode) {
        if(!Set_PID(ui32Port, inpud_id - 13, i32Value)) {
            UARTprintf("MQTT: incorrect value\n");
        }
    } else {
105     UARTprintf("MQTT: incorrect value\n");
    }
}
    mem_free(value);
110 }
}

```

A.1.12 spi_getvalue

```

1  int32_t spi_getvalue(uint16_t *channel) {
    uint32_t command[5];
    uint32_t data[5];
5   uint32_t i;
    int32_t result = 0;
    command[0] = READ_DATA_REG;
    command[1] = 0x0;
10   command[2] = 0x0;
    command[3] = 0x0;
    command[4] = 0x0;
    for(i = 0; i < 5; i = i + 1) {
15     SSIDataPut(SSII_BASE, command[i]);
    }
    while(SSIBusy(SSII_BASE))
    {
20   }
    for(i = 0; i < 5; i = i + 1) {
        SSIDataGet(SSII_BASE, &data[i]);
        data[i] &= 0x00FF;
25   }
    data[4] &= 0x000F;
    result = result + (data[1] << 16);
30   result = result + (data[2] << 8);
    result = result + (data[3]);
    result = result - 0x800000;
35   *channel = data[4];
    if(data[4] == 0x0 || data[4] == 0x1) {
        return result;
    } else {
40     return 0xffffffff;
    }
}

```

A.1.13 getTemp

```

1  int32_t getTemp(int32_t RawTemp, int32_t T0, int32_t Beta, int16_t Ratio, \
    int16_t tempmode, int32_t ptA, int32_t ptB)
    {
5     float alpha = ((float) RawTemp)/8388608.0; // raw/2^23
        float recepT;
        float T;
        float temp_adi2b;
        float temp_ldivb;
        int32_t result;
10
        if(tempmode == 0) {
            // 1/T = 1/T0 + 1/B*ln( alpha/(1-alpha)*Rref/R0
            recepT = 1.0 / (273.15 + ((float) T0)/1000.0) + \
                (1000.0 / ((float) Beta))*logf( alpha/(1.0-alpha)/(((float) Ratio)/1000.0) );
15            T = 1.0/recepT - 273.15;

        } else if(tempmode == 1 && ptB != 0) {
            temp_adi2b = 1000.0*(((float) ptA)/((float) ptB) ) / 2.0 ;
            temp_ldivb = 100000000000.0 / ((float) ptB);
20            if(ptB > 0) {
                T = - temp_adi2b + sqrt(temp_adi2b*temp_adi2b - \
                    temp_ldivb + temp_ldivb * alpha/(1.0-alpha)/(((float) Ratio)/1000.0));
            } else {
                T = - temp_adi2b - sqrt(temp_adi2b*temp_adi2b - \
25                    temp_ldivb + temp_ldivb * alpha/(1.0-alpha)/(((float) Ratio)/1000.0));
            }
            T = T + (float) T0 / 1000.0;

        } else if(tempmode == 1 && ptA != 0) {
            T = ( 1000000000.0 / ((float) ptA) ) * \
                ( alpha/(1.0-alpha)/(((float) Ratio)/1000.0) - 1.0);
            T = T + (float) T0 / 1000.0;
30        } else {
            T = 666.666;
35        }

        result = (int32_t) (T*1000.0); //result in mC
        return result;
40 }

```

A.1.14 PIDTask

```

1  static void
    PIDTask(void *pvParameters)
    {
5     uint32_t RawTemp;
        uint16_t channel;
        channel = 2;
        float integrator_max;
        float result = 0.0;
10        float err, derivative;

        //
        // Loop forever.
        //
15        while(1) {

            WatchdogClear();

20            RawTemp = spi_getvalue(&channel);

            if(channel == 1) {
                ui32RawTemp0 = RawTemp;
                i32ActTemp0 = getTemp(RawTemp, g_sParameters.i32T00, g_sParameters.i32Beta0, \ \
25                    g_sParameters.i32Ratio0, g_sParameters.ui16TempMode0, g_sParameters.i32ptA0, \ \
                    g_sParameters.i32ptB0);

```



```

pid_dt0 = xTaskGetTickCount() * portTICK_PERIOD_MS - pid_t0;

30 pid_t0 = xTaskGetTickCount() * portTICK_PERIOD_MS;
err = (float) g_sParameters.i32SetTemp0 - (float) i32ActTemp0;
derivative = (err - (float) pid_error0) / ((float) pid_dt0);
pid_error0 = g_sParameters.i32SetTemp0 - i32ActTemp0;
pid_integral0 = pid_integral0 + pid_error0 * pid_dt0;
35 if(g_sParameters.i16cli0 != 0) {
    integrator_max = fabs(pid_integrallimit0*g_sParameters.i16cli0);
    if(pid_integral0 > (int32_t) integrator_max) {
        pid_integral0 = (int32_t) integrator_max;
    }
40 if(pid_integral0 < - (int32_t) (integrator_max)) {
    pid_integral0 = - (int32_t) integrator_max;
    }
}

45 result = ((float) g_sParameters.i16clp0)*pid_error0/1000.0 + \
((float) g_sParameters.i16cli0)*pid_integral0/1000000.0 + \
((float) g_sParameters.i16cld0)*derivative;
setCurr0((int32_t) result);

50 if(pid_error0 < g_sParameters.ui16TempWindow0 && \
    pid_error0 > -g_sParameters.ui16TempWindow0) {
    SetStat(0, 1);
} else {
    SetStat(0, 0);
55 }

} else if(channel == 0) {
    ui32RawTemp1 = RawTemp;
    i32ActTemp1 = getTemp(RawTemp, g_sParameters.i32T01, g_sParameters.i32Beta1, \
60 g_sParameters.i32Ratio1, g_sParameters.ui16TempModel, g_sParameters.i32ptA1, \
    g_sParameters.i32ptB1);
    pid_dt1 = xTaskGetTickCount() * portTICK_PERIOD_MS - pid_t1;

    pid_t1 = xTaskGetTickCount() * portTICK_PERIOD_MS;
65 err = (float) g_sParameters.i32SetTemp1 - (float) i32ActTemp1;
derivative = (err - (float) pid_error1) / ((float) pid_dt1);
pid_error1 = g_sParameters.i32SetTemp1 - i32ActTemp1;
pid_integrall1 = pid_integrall1 + pid_error1 * pid_dt1;
70 if(g_sParameters.i16cli1 != 0) {
    integrator_max = fabs(pid_integrallimit1*g_sParameters.i16cli1);
    if(pid_integrall1 > (int32_t) integrator_max) {
        pid_integrall1 = (int32_t) integrator_max;
    }
75 if(pid_integrall1 < - (int32_t) (integrator_max)) {
    pid_integrall1 = - (int32_t) integrator_max;
    }
}

80 result = ((float) g_sParameters.i16clp1)*pid_error1/1000.0 + \
((float) g_sParameters.i16cli1)*pid_integrall1/1000000.0 + \
((float) g_sParameters.i16cld1)*derivative;

setCurr1((int32_t) result);

85 if(pid_error1 < g_sParameters.ui16TempWindow1 && \
    pid_error1 > -g_sParameters.ui16TempWindow1) {
    SetStat(1, 1);
} else {
    SetStat(1, 0);
90 }
}

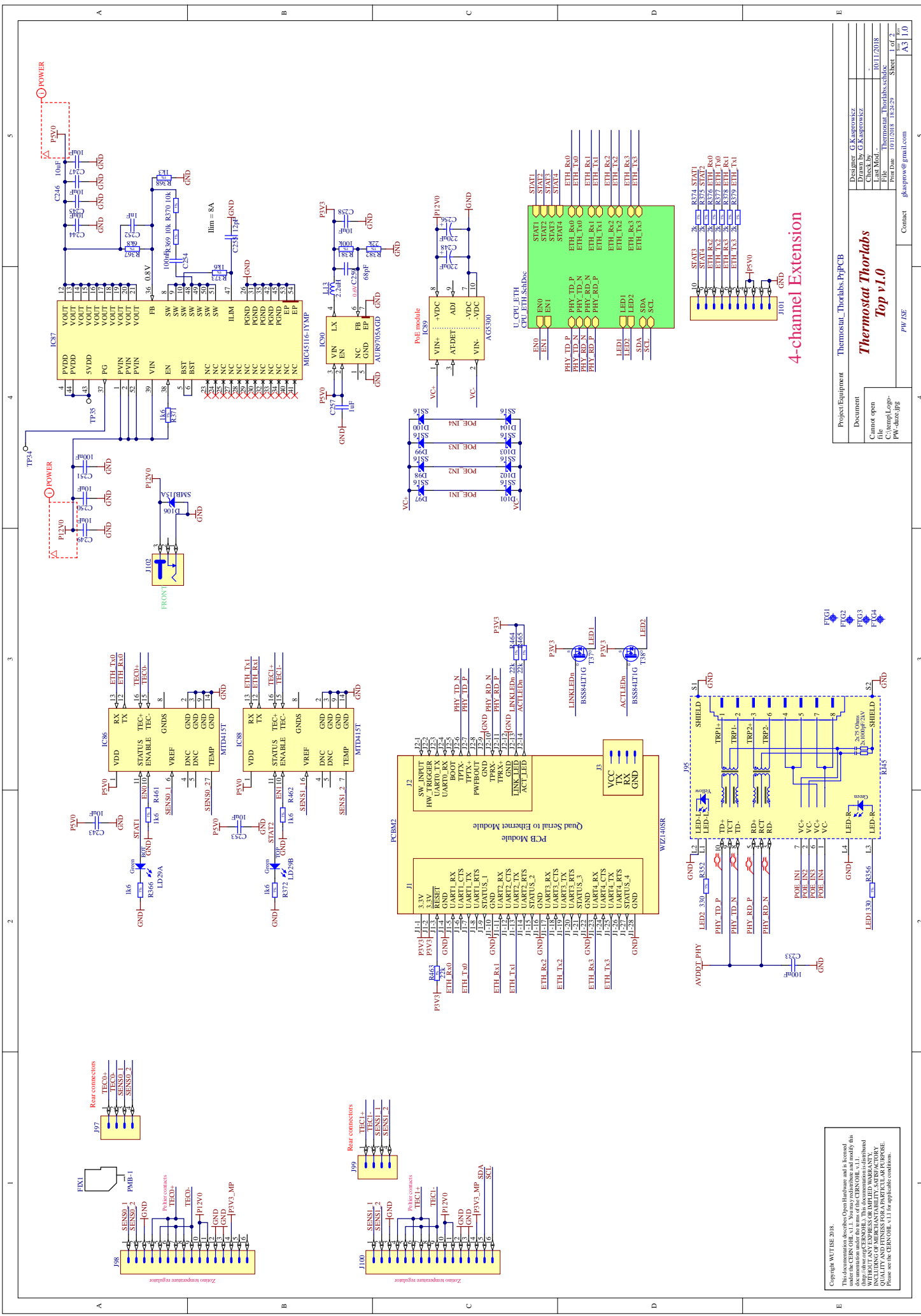
95 vTaskDelay(UPDATE_TIME/portTICK_PERIOD_MS);

}
}

```

A.2 Schematics of the device

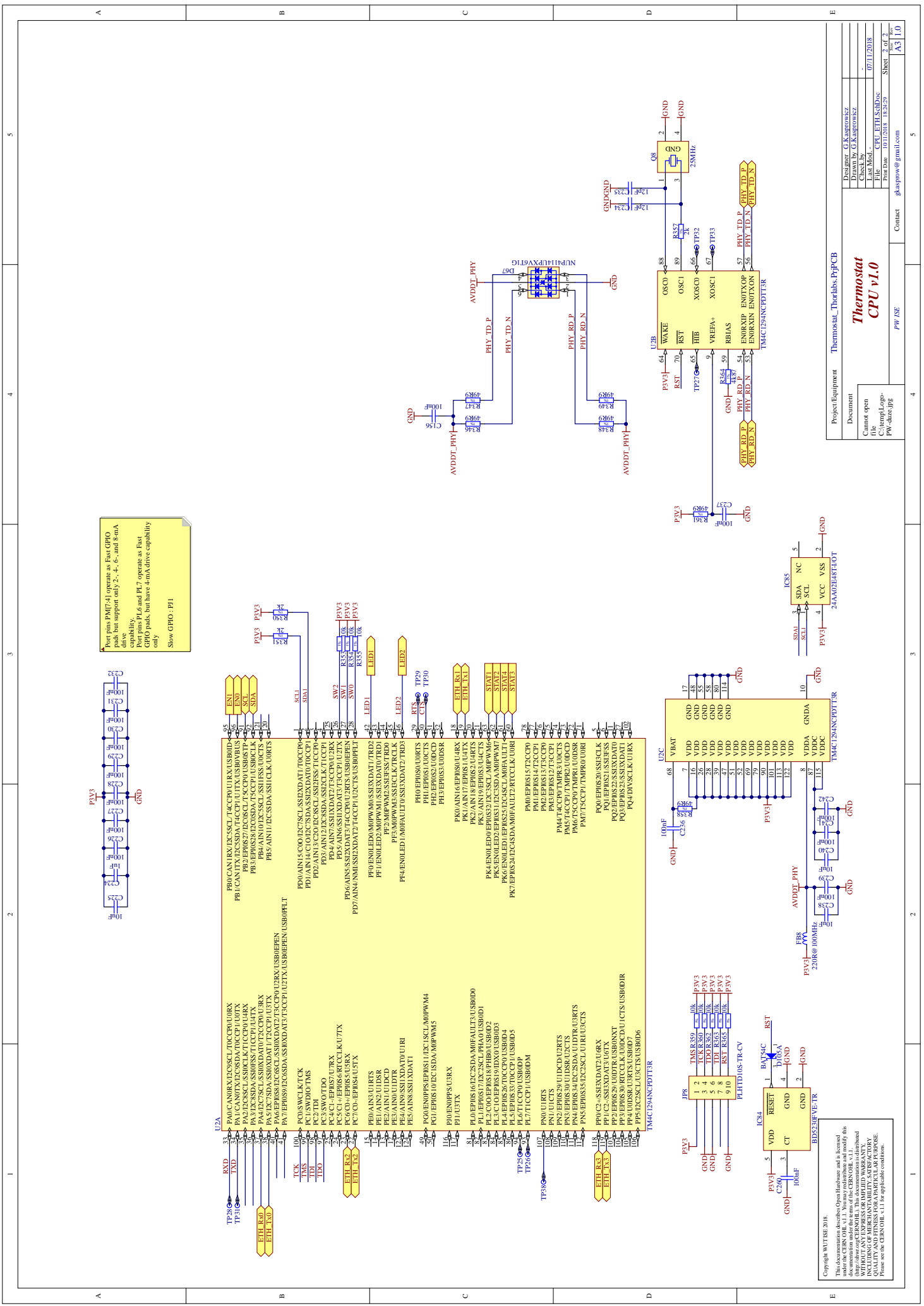
A.2.1 thermostat_thorlabs [21]



4-channel Extension

Copyright WITSIET 2018.
 This documentation describes Open Hardware and is licensed under the terms of the CERN OHL v1.1 (http://ohw.org/cernohl). This documentation is distributed IN CLIPPING OF MERCHANTABILITY AND FACTORY QUALITY AND BASIS FOR A PARTICULAR PURPOSE. Please see the LICENSE file for appropriate conditions.

Project/Equipment		Thermostat_Therlabs_PoEPCB	
Document		Thermostat_Therlabs_PoEPCB	
Cannot open C:\temp\Logo-PW-dtuzs.jpg		Thermostat_Therlabs_schdoc	
Designer	G. Kasprapowicz	Rev	1.0
Checked by	G. Kasprapowicz	Date	10/12/2018
Last Mod.		File	Thermostat_Therlabs_schdoc
Print Date	10/12/2018 18:32:29	Sheet	1 of 2
Part Name	gkaspr@protonmail.com	Comment	A3 1.0



Port pins PM1751 operate as Fast GPIO pins but support only 2-, 4-, 6-, and 8-mA drive capability.
 Port pins PL6 and PL7 operate as Fast GPIO pins, but have 4-mA drive capability only.
 Slow GPIO: P11

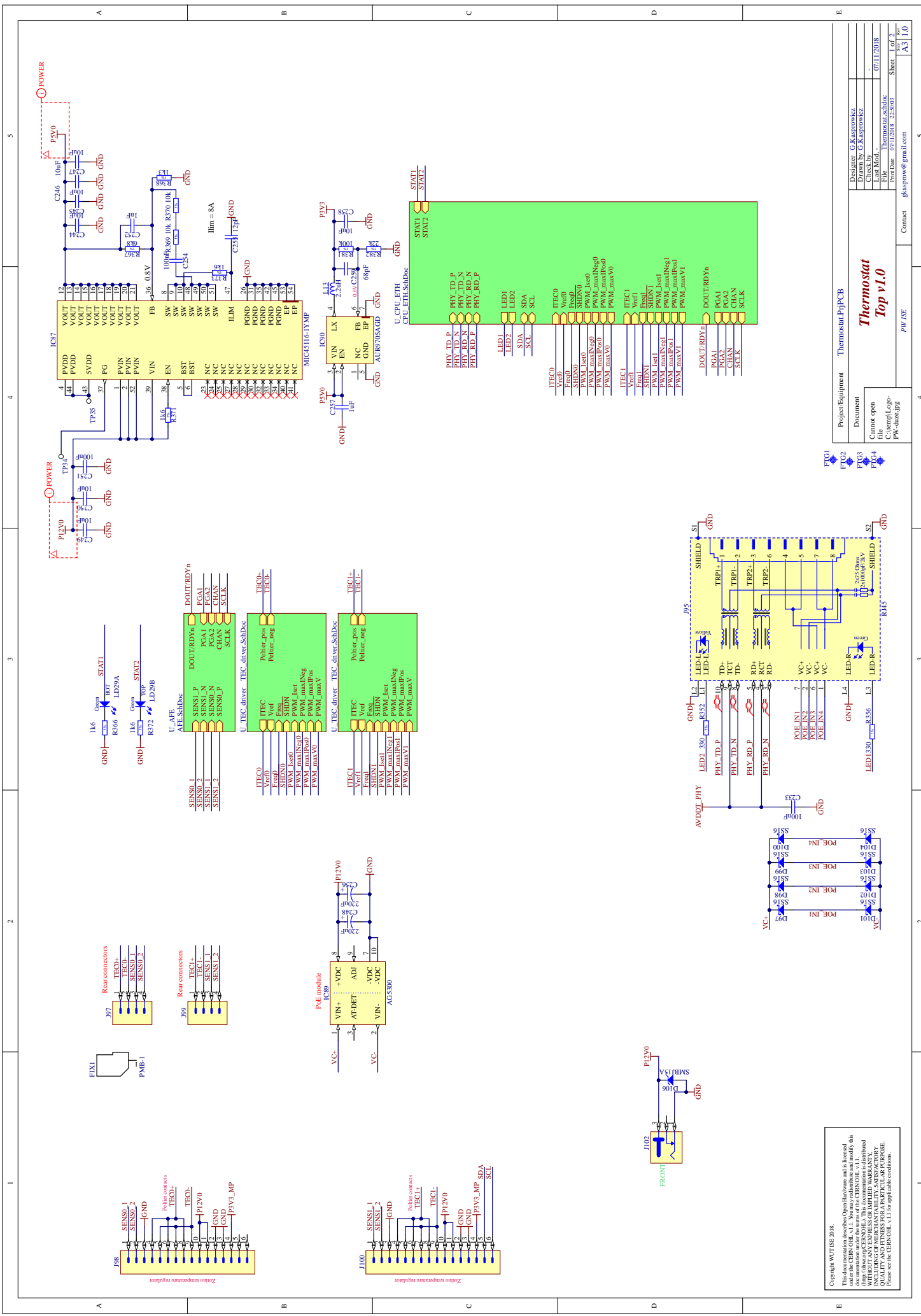
Copyright WITSE 2018.
 This documentation describes Open Hardware and is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/).
 The documentation is provided for informational purposes only. It is not intended to be used as a substitute for the manufacturer's documentation. The manufacturer's documentation is the only source for the most up-to-date information.
 INCLUDING OR MERCHANDISING ANY FACTORY QUALITY ASSURANCE TAGS OR PARTS IS PROHIBITED.
 Please see the README file for applicable conditions.

Project/Equipment	Thermoskat_Theralkts_P1PCB
Document	
Designer	C. Kasprawicz
Checked by	C. Kasprawicz
Last Mod.	07/11/2018
File	CPU_ETH_SchDoc
Print Date	01/12/2018 13:32:59
Sheet	3 of 3
Sheet No.	A3.1.0
Company	gkaspraw@gmail.com

**Thermoskat
CPU v1.0**

PW-015E

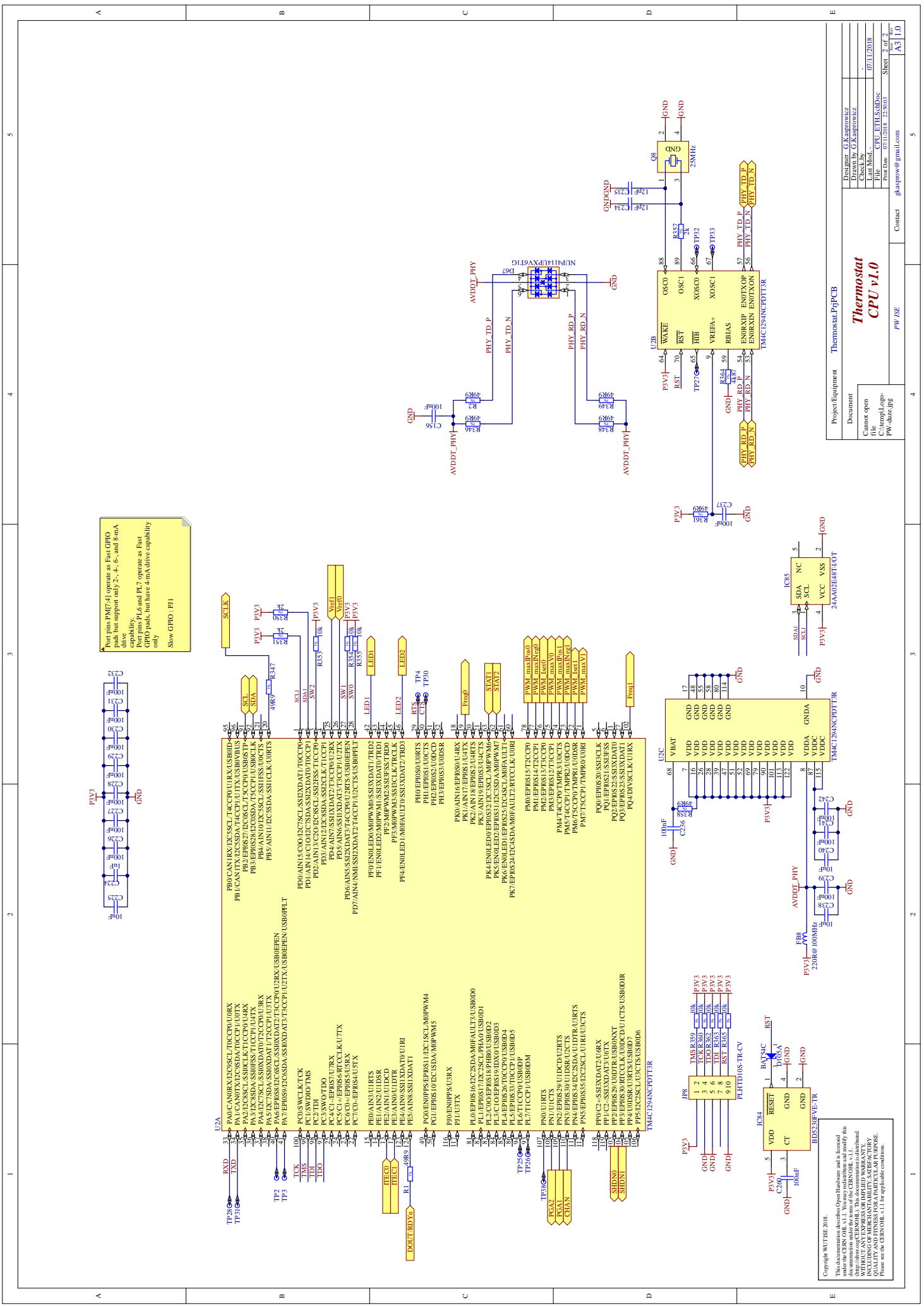
A.2.2 thermostat_max [20]



Copyright WUT/ISE 2018.
 This documentation describes Open Hardware and is licensed under the terms of the CERN OHL v1.1. It is provided for informational purposes only. It is not intended for use in any other project without the explicit permission of WUT/ISE. The documentation is distributed IN FULLY AS IS WITHOUT ANY WARRANTY, EXPRESS OR IMPLIED. ANY USE OF THIS DOCUMENTATION IS AT YOUR OWN RISK. WUT/ISE SHALL NOT BE LIABLE FOR ANY DAMAGES, INCLUDING CONSEQUENTIAL DAMAGES, ARISING FROM OR OUT OF THE USE OF THIS DOCUMENTATION, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

Project/Equipment		Thermostat_Top1PCB	
Document		Thermostat_Top v1.0	
Cannot open C:\temp\Log-PCW-dtuzs.jpg			
Designer	G. Kasprawicz	Sheet	1 of 3
Checked by	G. Kasprawicz	Sheet	1 of 3
Last Mod.	07/11/2018	Sheet	1 of 3
File	Thermostat_top.docx	Sheet	1 of 3
Print Date	07/11/2018 22:50:37	Sheet	1 of 3
Printed by	gkaspraw@pau.edu.pl	Sheet	1 of 3
Comment		Sheet	1 of 3
PW/ISE		A3 1.0	

- FC1
- FC2
- FC3
- FC4



Port pins PM7-14 operate as Fast GPIO pins but support only 2-, 4-, 6-, and 8-mA drive capability.
 Port pins PL6 and PL7 operate as Fast GPIO pins, but have 4-mA drive capability only.
 Slow GPIO: P11

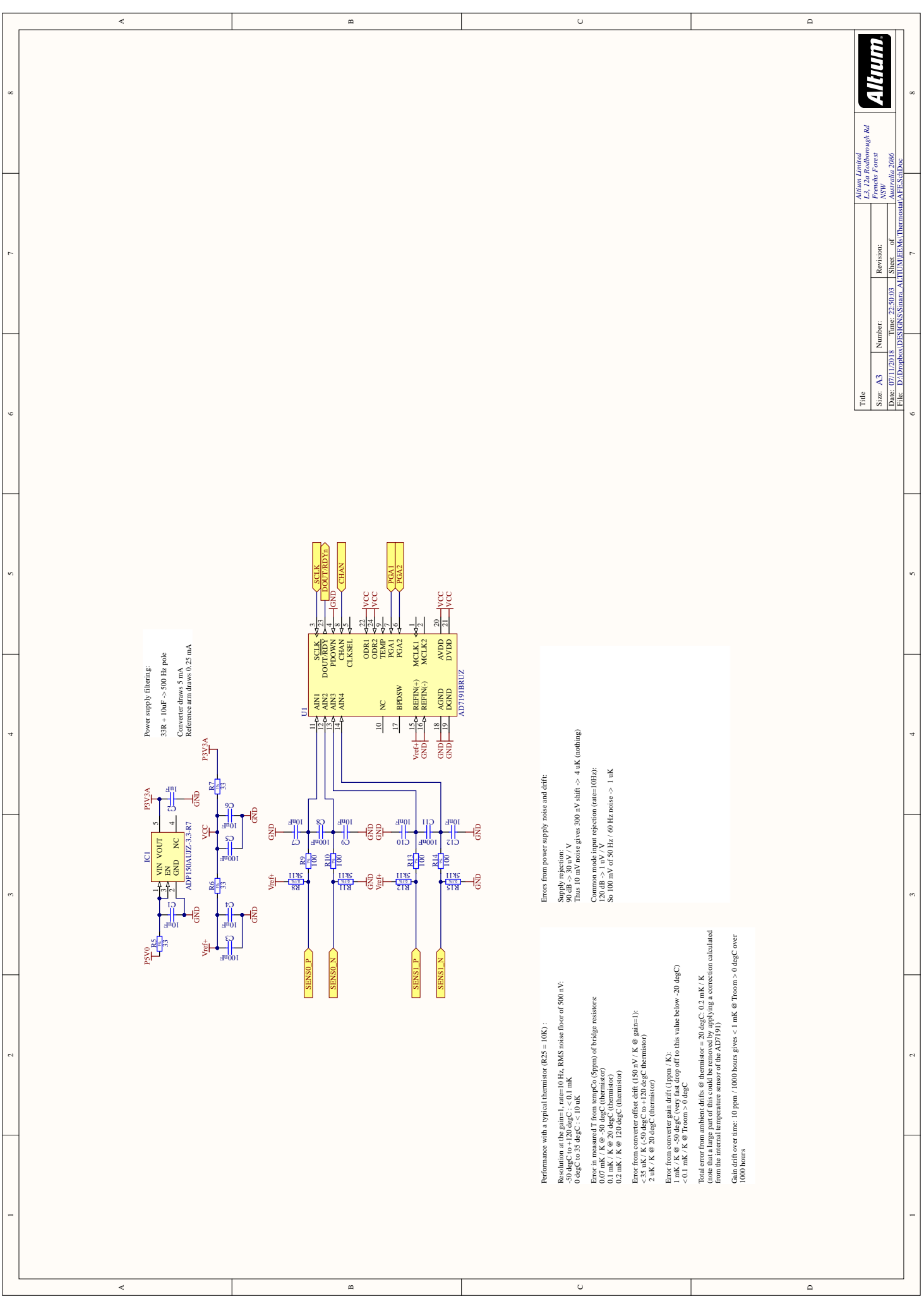
Copyright WITSEI 2018.
 This documentation describes Open Hardware and is licensed under the terms of the Creative Commons Attribution 4.0 International License (CC BY 4.0).
 The documentation is provided for informational purposes only. It is not intended to be used as a substitute for the manufacturer's documentation. The manufacturer's documentation is the only source of information for the product.
 INCLUDING OR MERCHANDISING ANY PRODUCT OF TRADE OR SERVICE OF ANY OTHER MANUFACTURER IS NOT THE INTENT OF THIS DOCUMENTATION.
 Please see the README file for appropriate conditions.

Project/Equipment	Thermostat_PjPjPCB
Document	
Designer	C. Kasprowicz
Checked by	C. Kasprowicz
Last Mod.	07/11/2018
File	CPU_ETH_SchDoc
Print Date	07/11/2018 22:03:03
Sheet	3 of 3
Sheet No.	A3.1.0

Thermostat CPU v1.0

PW-ISE

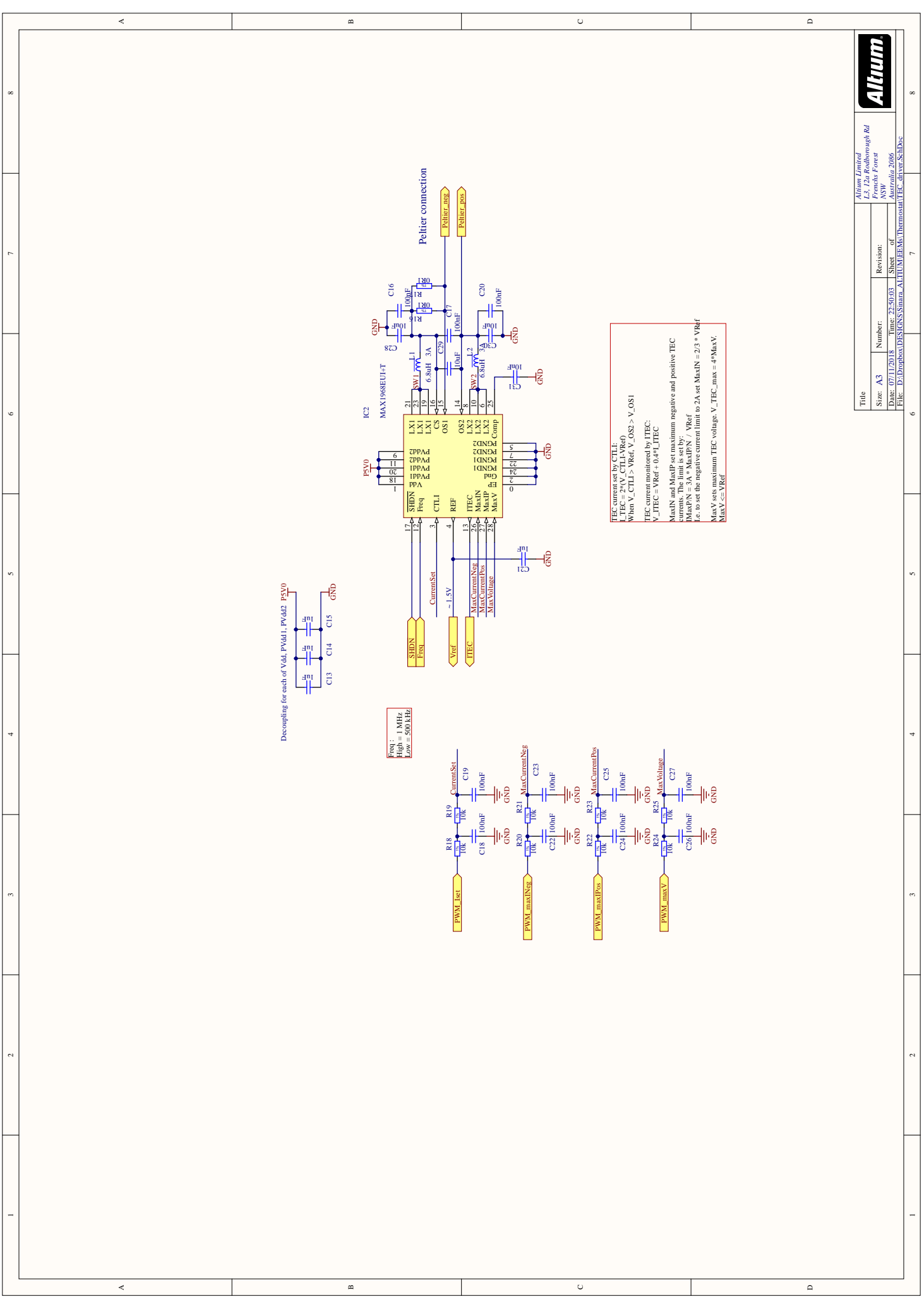
Comter gkaspro@pjal.com



Power supply filtering:
 $33R + 10\mu F \rightarrow 500 \text{ Hz pole}$
 Converter draws 5 mA
 Reference arm draws 0.25 mA

Errors from power supply noise and drift:
 Supply rejection:
 $90 \text{ dB} \rightarrow 30 \text{ nV/V}$
 Thus 10 mV noise gives 300 nV shift $\rightarrow 4 \mu\text{K}$ (nothing)
 Common mode input rejection (rate: 10Hz):
 120 dB
 So 100 mV of 50 Hz / 60 Hz noise $\rightarrow 1 \mu\text{K}$

Performance with a typical thermistor (R25 = 10K):
 Resolution at the gain=1, rate=10 Hz, RMS noise floor of 500 nV:
 -50 degC to $+120 \text{ degC}$; $< 0.1 \text{ mK}$
 0 degC to 35 degC ; $< 10 \mu\text{K}$
 Error in measured ΔT from tempCo (5ppm) of bridge resistors:
 0.07 mK/K @ 20 degC (thermistor)
 0.1 mK/K @ 20 degC (thermistor)
 0.2 mK/K @ 120 degC (thermistor)
 Error from converter offset drift (150 nV/K @ gain=1):
 2.35 mK/K @ 20 degC to 120 degC (thermistor)
 2 mK/K @ 20 degC (thermistor)
 Error from converter gain drift (1ppm/K):
 1 mK/K @ -50 degC (very flat drop off to this value below -20 degC)
 $< 0.1 \text{ mK/K}$ @ Troom $> 0 \text{ degC}$
 Total error from ambient drifts @ thermistor = 20 degC : 0.2 mK/K
 (note that a large part of this could be removed by applying a correction calculated from the internal temperature sensor of the AD7191)
 Gain drift over time: $10 \text{ ppm} / 1000 \text{ hours}$ gives $< 1 \text{ mK}$ @ Troom $> 0 \text{ degC}$ over 1000 hours



Decoupling for each of Vdd, PVdd1, PVdd2, PSV0

Freq: High = 1 MHz Low = 800 kHz

ITEC current set by CTLL:
 $V_{ITEC} = V_{REF} \cdot \frac{CTLL}{CTLL + 1}$
 When $V_{CTLL} > V_{REF}$, $V_{ITEC} > V_{OS1}$
 TEC current monitored by ITEC:
 $V_{ITEC} = V_{REF} + 0.4 \cdot I_{ITEC}$
 MaxIN and MaxIP set maximum negative and positive TEC currents. The limit is set by:
 $I_{MaxPN} = 3A \cdot \frac{MaxIPN}{V_{REF}}$
 I.e. to set the negative current limit to 2A set $MaxIN = 2/3 \cdot V_{REF}$
 MaxV sets maximum TEC voltage. $V_{TEC_max} = 4 \cdot MaxV$
 $MaxV < V_{REF}$

Bibliography

- [1] AD7172-2 Data Sheet (Rev. A). URL: <https://www.analog.com/media/en/technical-documentation/data-sheets/AD7172-2.pdf>.
- [2] Ag5300 Data Sheet. URL: <https://www.silvertel.com/images/datasheets/Ag5300-datasheet-smallest-30W-Power-Over-Ethernet-Plus-Module-PoEplusPD.pdf>.
- [3] AR654 Data Sheet. URL: https://www.apar.pl/v/product/cat_ar654_eng.pdf.
- [4] ARTIQ overview. URL: <https://m-labs.hk/artiq/index.html>.
- [5] Autoip - lwip wiki. URL: <https://lwip.fandom.com/wiki/AUTOIP>.
- [6] Cave - Wikipedia. URL: <https://en.wikipedia.org/wiki/Cave>.
- [7] Code Composer Studio - Wikipedia. URL: https://en.wikipedia.org/wiki/Code_Composer_Studio.
- [8] FreeRTOS home page. URL: <https://www.freertos.org/>.
- [9] lwIP Wiki. URL: https://lwip.fandom.com/wiki/LwIP_Wiki.
- [10] M-Labs - Wikipedia. URL: <https://en.wikipedia.org/wiki/M-Labs>.
- [11] MAX1968/MAX1969 Data Sheet Rev. 3, 5/15. URL: <https://datasheets.maximintegrated.com/en/ds/MAX1968-MAX1969.pdf>.
- [12] MCU Market on Migration Path to 32-bit and ARM-based Devices. URL: <http://www.icinsights.com/data/articles/documents/541.pdf>.
- [13] Mosquitto home page. URL: <https://mosquitto.org/>.
- [14] Mqtt essentials part 5: Mqtt topics & best practices. URL: <https://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices/>.
- [15] Realterm home page. URL: <https://realterm.sourceforge.io/>.
- [16] Sinara - An open-source hardware ecosystem for quantum physics. URL: <https://sinara-hw.github.io/>.
- [17] Sinara hardware. URL: <https://m-labs.hk/artiq/sinara.html>.
- [18] Sinara Open Hardware Project. URL: <https://github.com/sinara-hw>.
- [19] Texas Instruments, Tiva C Series TM4C1294NCPDT Microcontroller Data Sheet (Rev. B). URL: <http://www.ti.com/lit/ds/symlink/tm4c1294ncpdt.pdf>.
- [20] Thermostat's github. URL: <https://github.com/sinara-hw/Thermostat>.

- [21] thermostat_thorlabs github. URL: https://github.com/sinara-hw/thermostat_thorlabs.
- [22] Thorlabs, MTD415T Data Sheet Rev. 1.0. URL: <https://www.thorlabs.com/drawings/86e5095db682f685-F39DDF82-06B5-5331-4CF5B79D7E9DE6A2/MTD415T-DataSheet.pdf>.
- [23] Tiva C Series TM4C1294 Connected LaunchPad Evaluation Kit (Rev. C). URL: <http://www.ti.com/lit/ug/spmu365c/spmu365c.pdf>.
- [24] UNIPLEX III Data Sheet. URL: https://www.kloepper-therm.de/fileadmin/pdf/Uniplex-Flyer_EN.pdf.
- [25] Using the Stellaris Ethernet Controller With Lightweight IP (lwIP). URL: <http://www.ti.com/lit/an/spma025c/spma025c.pdf>.
- [26] Eric Brown. Linux and Open Source on the Move in Embedded, Says Survey. URL: <https://www.linux.com/news/event/elce/2017/linux-and-open-source-move-embedded-says-survey>.
- [27] D.R. White et al. Guide on secondary thermometry – thermistor thermometry, 2014. URL: <https://www.bipm.org/utils/common/pdf/ITS-90/Guide-SecTh-Thermistor-Thermometry.pdf>.
- [28] Richard M. Murray. Karl Johan Astrom. *PID Control*. 2019. URL: http://www.cds.caltech.edu/~murray/books/AM08/pdf/fbs-pid_01Jan19.pdf.
- [29] Krzysztof Paprocki. Mikrokontrolery stm32. praca pod kontrolą freertos. URL: <https://ep.com.pl/files/2434.pdf>.
- [30] Michał Gąska Paweł Kulik, Grzegorz Kasprowicz. Driver module for quantum computer experiments: Kasli, 2018. URL: <https://doi.org/10.1117/12.2501709>, doi:10.1117/12.2501709.
- [31] HiveMQ Team. MQTT Essentials: Part 1 – Introducing MQTT. URL: <https://www.hivemq.com/blog/mqtt-essentials-part-1-introducing-mqtt/>.
- [32] Joseph Wu. A basic guide to rtd measurements, 2018. URL: <http://www.ti.com/lit/an/sbaa275/sbaa275.pdf>.