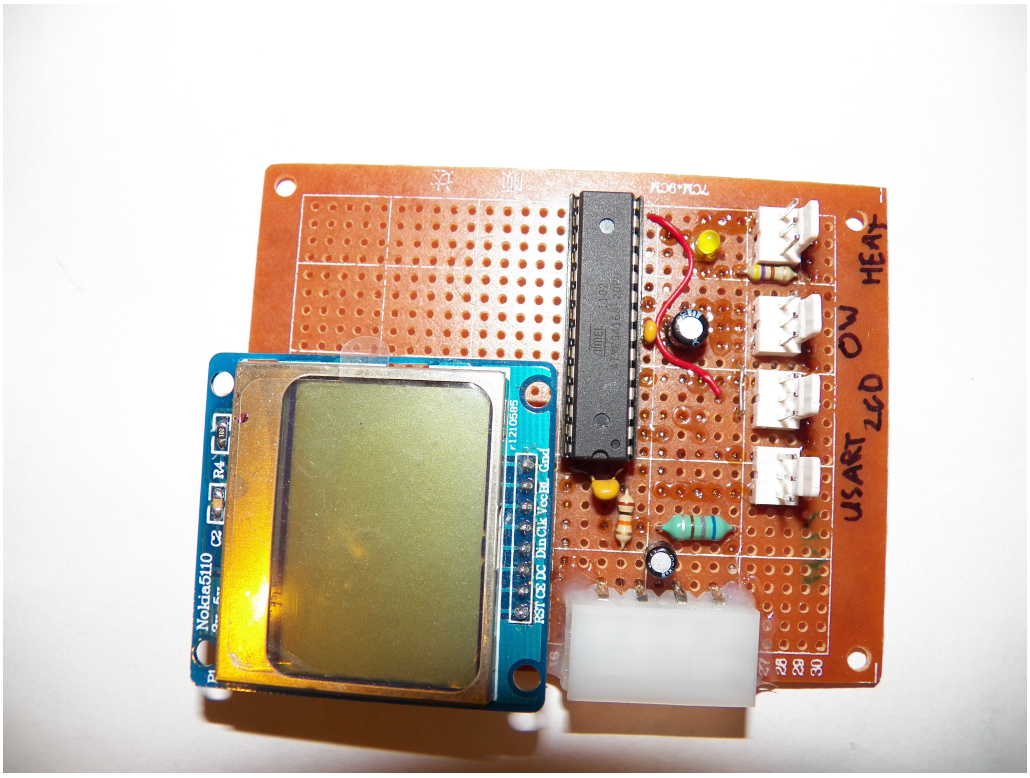
Multitasking Autotuning PID Controller in Heat Transfer Application



January 21, 2016

Contents

1 Hardware **1**

1.1 Theory 1

1.1.1 Bipolar transistor biasing 1

1.1.2 Phototransistor 1

1.2 Definitons 1

1.3 Goals 1

1.4 Layout 2

1.5 ZCD board 3

1.5.1 Schematic 3

1.5.2 PCB 4

1.5.3 Calculation 5

1.5.4 Measurements 6

1.6 Thermometer 7

1.6.1 Heater 7

1.6.2 Temperature sensor 8

1.7 SCR board 8

1.7.1 SChematic 8

1.7.2 Calculation 8

1.7.3 Measurements 9

1.8 Main board 9

1.8.1 Microcontroller 9

1.8.2 Power filtering 9
1.8.3 Connectors 10
1.8.4 Extendability 10

2 Software **11**

2.1 Theory 11
2.1.1 Radix tree 11
2.1.2 Cooperative multitasking 11
2.2 Goals 12
2.3 Language selection 12
2.4 Bootloader 12
2.4.1 Motivation 12
2.4.2 Previous work 13
2.4.3 Implementation 13
2.4.4 Communication protocol 14
2.4.5 Error checking 14
2.4.6 Use 14
2.5 External components 15
2.5.1 Onewire library 15
2.5.2 PID implementation 15
2.6 Architecture 15
2.6.1 Theory 15
2.6.2 Implementation 16
2.7 Modules 16

2.7.1 Configuration 16
2.7.2 Clock 16
2.7.3 Serial communication 17
2.7.4 Commands 17
2.7.5 Zero-cross detector 17
2.7.6 Triac control 18
2.7.7 Temperature measurement 18
2.8 Plotting tool 18

3 Algorithms **19**

3.1 Theory 19
3.1.1 Parallel PID controller 19
3.1.2 Anti-windup 19
3.1.3 Plant models 20
3.1.4 PID tuning algorithms 20
3.1.5 Digital control systems 21
3.1.6 Digital PID controller 21
3.2 PID implementation 22
3.3 Units of measure 22
3.4 Nonlinearity 22
3.5 Ziegler-Nichols sustained oscillations 23
3.6 Astrom-Hagglund sustained oscillations 23
3.7 Sampling rate correction 24
3.8 Peak detector 26

4 Conclusion 27

5 Bibliography 28

6 Appendix 1 - Source code A-1

6.1 rtplot A-1

6.2 megaboot A-7

6.3 micli A-16

1 Hardware

1.1 Theory

1.1.1 Bipolar transistor biasing

Saturation occurs when both PN junctions are forward-biased. In this mode the transistor is fully open, corresponding to logical "on" state. Cutting current to the base renders the device fully closed, or in logical "off" state.

1.1.2 Phototransistor

The phototransistor behaves similarly to an ordinary transistor with the exception that it's base region is sensitive to light. When no light is falling onto the base, the emitter current is [5, p. 418]:

$$I_T = I_{CB0} + \beta I_{CB0} = (1 + \beta) I_{CBO} \tag{1}$$

If then line is shone onto the base region, the concentration of electrons and holes increases. A similar photo current, as with photodiodes, flows. The difference is that the curent gets amplified $1 + \beta$ times compared to a photodiode.

Devices only two LEDs are available. This mode of operation is called 'floating base'. It is thermally unstable.

The alternative is to connect the (photosensitive) base to a suitable DC bias. In this configuration negative feedback is realized. As a result, the gain β decreeses. The negative feedback provides further benefits beyond thermal stability.

1.2 Definitons

device - the entirety of all physical components, resulting from this project.
CTR - current transfer ratio, optocoupler characteristic.

1.3 Goals

The device is intended as a learning project for the student, but also as a open software, open-hardwre project, which anyone can create and use. Thus, the following hardware design priorities have been identified, in order of decreasing importance:

- safety - the device shall not pose a fire or electric shock hazard to the end user
- reconstructability - the device shall be composed **only** of worldwide accessible components
- longivity - the device shall remain operational for 5 years of uninterrupted service with 95% confidence
- price - the BOM for the complete device shall not exceed 100BGN
- extendability - the number of input sensors and the number of output controllers, shall be trivially configurable
- ease of assembly - it shall be possible for a person with zero hardware experience to manufacture the device
- simplicity - each component shall fufill a specific purpose, and the number of components shall be the lowest possible

1.4 Layout

Due to the requirement of extendability, the device shall consist of a number of printed circuit boards, in contrast to a single monoliotic PCB. Each PCB shall fufill a sole purpose, and any number of different modules shall be able to mate together. The following distinct roles have been identified:

- high-voltage input stage - called zero-cross detector or ZCD board from now on
- low-voltage input stage - called temperature sensor or thermometer from now on
- computational stage - called main board from now on
- high-voltage output stage - called software controlled rectifier board or SCR board from now on

The resulting design exhibits the following characteristics.

Only a single ZCD board is required, because mains waveform is invariant across the device in it's entirety. Only a single main board is required, as the selected microcontroller, although inexpensive, provides plenty of resources for numerous control loops.

In order to satisfy the requirement for simplicity, the main board is configured for a single SCR output board. However, soldering aditional connectors to the main PCB is trivial, thus acheaving extensability. The SCR output board is long-life and supports loads of up to 1kW.

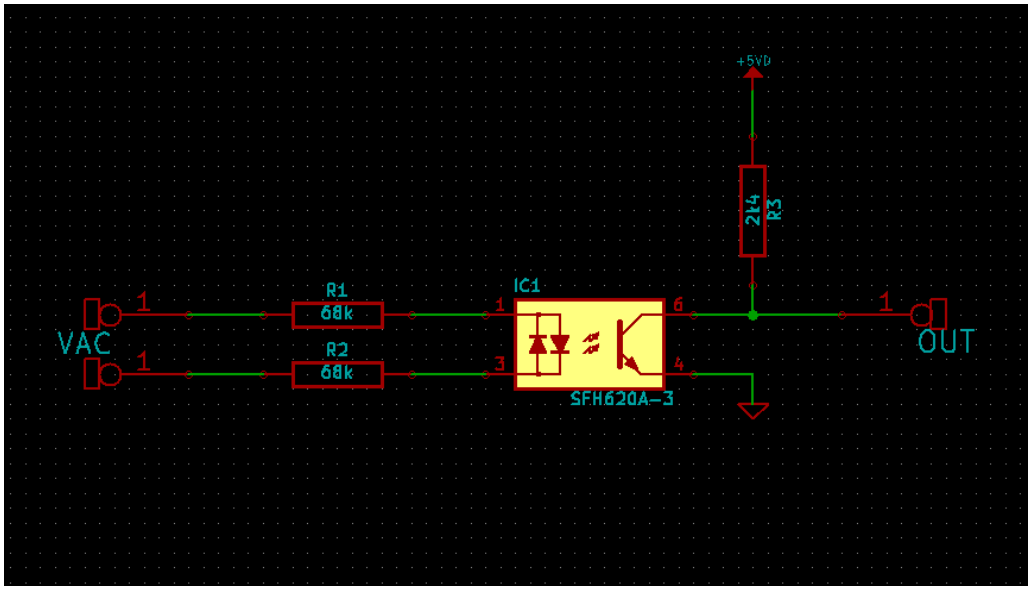
The most flexible part of the system is the thermometer configuration. Due to the selected temperature sensing IC, virtually unlimited (technically up to 2^{56}) devices are supported **without any hardware changes**.

1.5 ZCD board

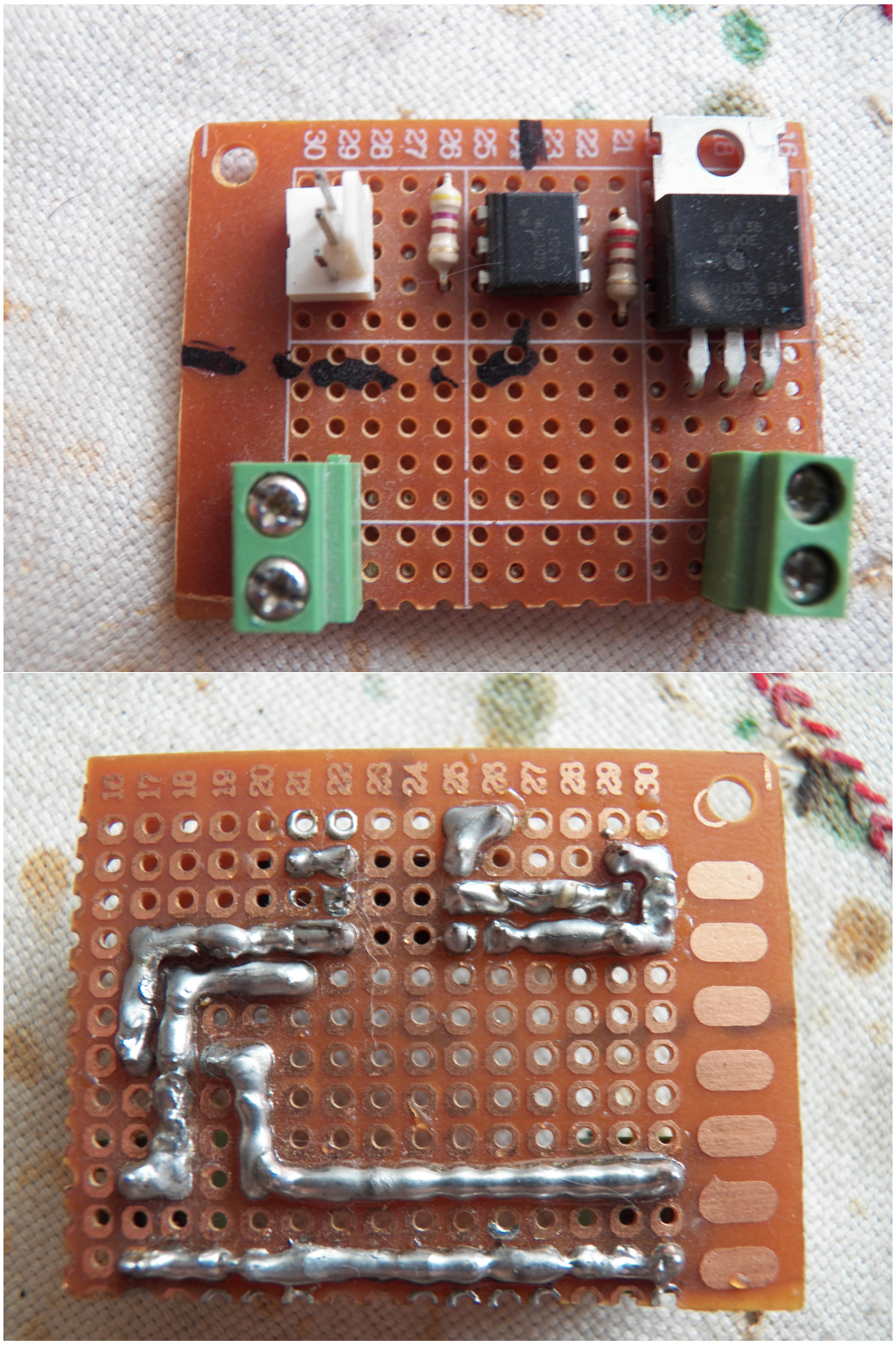
The ZCD board is a sensory input to the microcontroller.

Because the voltage of mains power is alternating, it is impossible to output precise amounts of power without knowing the phase of the waveform. The implementation of the ZCD board is straightforward and extremely simplified. In fact, an extensive internet search has demonstrated no other PCB has ever been designed with such a level of simplicity. In other words, **the designed PCB contains fewer elements than any known PCB for the same purpose**! This produces problems, which have deterred other designers. However, all artifats have been dealt with in software.

1.5.1 Schematic



1.5.2 PCB



1.5.3 Calculation

In order to protect the main board (and thus the user) from dangerous voltages, galvanic isolation is required. The standard means to this end are transformers and optocouplers. Optocouplers posess numerous advantages over transformers for our application:

- compactness
- low price
- neglegable phase shift

Furthermore, among optocouplers, the variation is considerable. We select a component with anti-parallel input LEDs, specifically designed for zero crossing - SFH620A-3. This is the most sensitive version of the IC (highest CTR), as input power is our greates concern.

Striving for minimal component count and price, the standard solution with a 10W input power resistor is dismissed. Thus, we need to work with 1/4W, E24 resistors. Due to the optocoupler's acceptable CTR, and extensive signal conditioning in software, this solution will prove to be viable!

It is worthy to note that the resistor rated voltage is of utmost importance. Because our resistors are rated to 200V peak, it would be a dangerous mistake to use a single resistor. Therefore, the $V_{AC} = 230\text{V}$, $V_{peak} = V_{AC} * \sqrt{2} = 325\text{V}$ is safely spread onto two identical resistors.

Let's suppose the line voltage varies from $V_{min} = 200VAC$ to $V_{max} = 250VAC$ rms.

$$P_{inputresistors} = \frac{V_{max}^2}{R_1 + R_2}$$

$$R_1 + R_2 \geq \frac{V_{max}^2}{P_{inputresistors,max}} = \frac{250^2}{0.25 + 0.25} = 125\text{k}\Omega$$

We select $R_1 = R_2 = 68\text{k}\Omega$.

$$i_{in,min} = \frac{V_{min} - 1.65}{2 * R_1 * 1.05} = \frac{198.35V}{142.8\text{k}\Omega} = 1.39\text{mA}$$

The output stage:

$$i_C \geq i_{in,min} * CTR_{min} \approx 1.39\text{mA} * 0.34 \approx 0.47\text{mA}$$

$$i_{leakage} \leq 1\mu\text{A}$$

$$V_{IL} = 0.3Vcc = 0.3 * 5\text{V} = 1.5\text{V}$$

$$V_{IH} = 0.6Vcc = 0.3 * 5\text{V} = 3\text{V}$$

If we strive to be below 1V for logic zero:

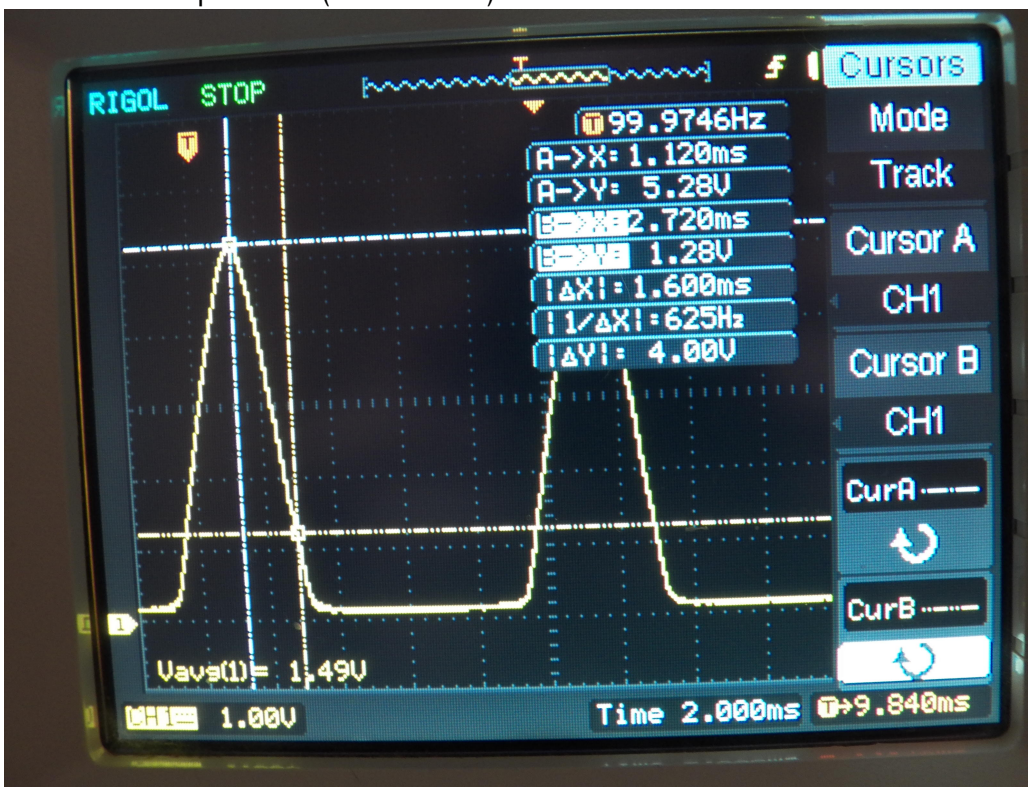
$$i_C * R_{output} = 1V$$

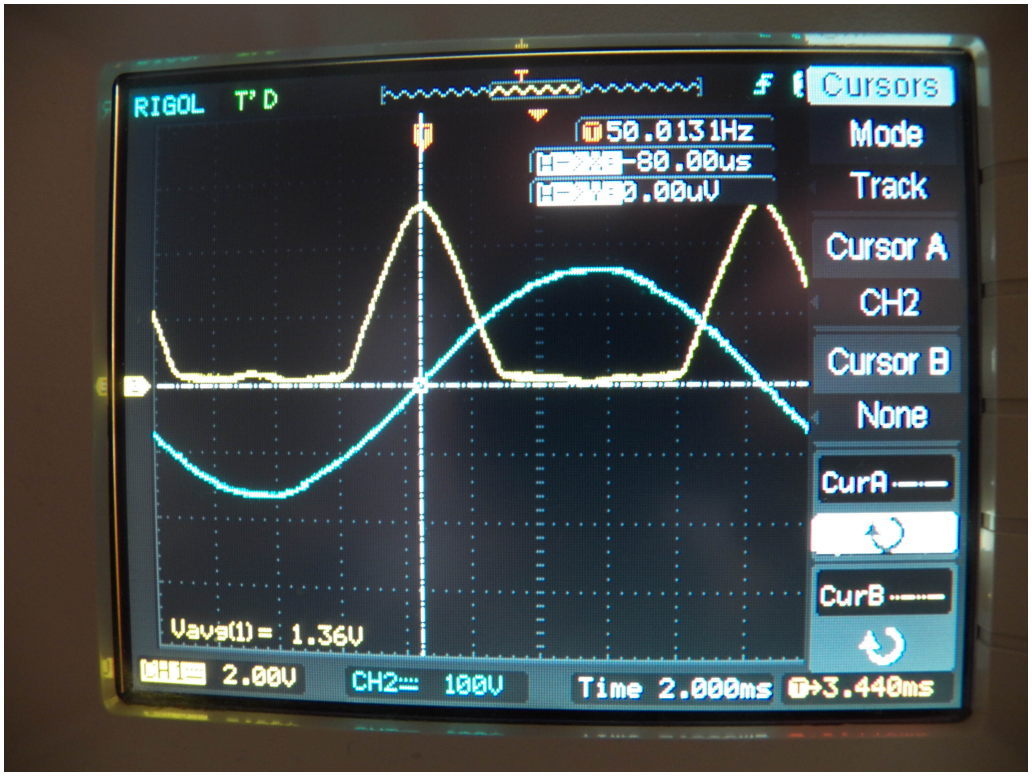
$$R_{output} = 1\text{V}/0.47\text{mA} = 2.13\text{k}\Omega$$

We select $R_3 = 2.4\text{k}\Omega$.

1.5.4 Measurements

Firstly, a temperature measurement is performed. The device is allowed to run for 10 minutes. Subsequently, each component is measured for overheating. Becuse component temperature measurement instrumentation is both expensive and dfficult to apply, the following rule of thunmb is used: *if a silicone component is too hot to keep your finger on it, it is too hot*. Althought the stated metod is vastly imprecise, it works well, because the skin pain temperature (about $60°$C) is far lower than silicone semicondutor Absolute Maximum Temperature (often $150°$C).





We observe that:

1. The pulse is very wide - about $3.2\text{ms} \equiv 32\%$ of the half-period.
2. The puslse is centered. This is great, because we can estimate the true zero crossing in software.

1.6 Thermometer

It is impossible to examine the temperature sensing element in isolation to the heater. Therefore, in this chapter, the complete plant, or in other words the combination of heater, thermal mass and thermometer, will be examined.

As this is an educational project, the quickest possible system responce is desired. Therefore, the thermometer is directly gued to the surface of the heater. Unfortunately, even in this setup, the maximum possible heating rate is about $6°\text{C/min}$ and cooling is even slower.

1.6.1 Heater

Initially, a 60W incandescent light bulb was selected. As European wall power exhibits frequency of 50Hz, the highest switching frequency is 100Hz by half-periods. Astonishing

to the experimenter:

- The fillament is not inert enough to integrate consecutive pulses, even if every odd half-wave is enabled.
- The human eye is unable to integrate the resulting 50Hz flicker.

As a result, looking in the controlled bulb is extremely annoying.

Consequenctly, a fish tank heater with nominal (maximum) power of 50W was selected. The observed temperature curves are equivelent sans the maddening light flicker. **The thermometer is glued to the surface to the heater for fastest response possible.**

1.6.2 Temperature sensor

The Dallas Semiconductor DS18S20 has been selected due to a variety of reasons:

- low price
- ease of interfacing to digital components
- ease of wireing
- extreme flexability of integration

This device is incredible. It can operate solely over two wires - including the ground wire. It performs digital temperature conversions, removing the need of an ADC (although our selected microcontroller features such). It can coexist on the same bus with as many as $2^{56} \equiv$ infinite number of other onewire sensors.

1.7 SCR board

1.7.1 SChematic

1.7.2 Calculation

Because we want to operate with more precision than an on-off controller, the following requirements are layed out:

- Life must exceed 3153600000 switches.

- Switching times must be in the microsecond range.
- Galvanic isolation.

In the light of this sepcification, it immediately becomes obvious that an electro-mechanical relay is not suitable. Again an optocoupler has been selected. The MOC3023 is canonical for power controll aplications. It is a triac output optocoupler, specifically designed to drive a power triac. The selected power triac is BT136 600E, providing control over applicances of up to $4\text{A} * 230\text{V} = 920\text{W}$.

1.7.3 Measurements

1.8 Main board

The main board houses the microcontroller, provides stable power to it and contains connector blocks to the other boards.

1.8.1 Microcontroller

The selected microcontroller is atmega168 from Atmel. It provides plenty of computing power and sufficient 16KB flash program memory at extremely modest cost. It can be programmed in assembler, C or C++ with widely available and free of charge tools. The datasheet is well written and a pletora of application notes are available from Atmel. Not to mention the extensive worldwide community, readily providing help to anyone new to the subject.

1.8.2 Power filtering

Three groups of components are responsible for providing clean and steady power to the microcontroller.

Firstly, an LC filter at the power connector of the board provides filtering of the external power. This procets against insufficiently filtered power supplies and allows powering the board from a low cost wall adapter "brick". Furthermore, any EMI picked over a long power supply coard, is eliminated. Moreover, the LC filter isolates the board from the power supply, allowing the PSU to power other devicecs in the same time, free of digital switching noise.

Secondly, a combination of two capacitors, in close physical proximity to the microcontroller power leads, further conditions the power supply rail. As the microcontroller draws

significant current for intervals far shorter tahn one microsecond, the PCB traces gain significant impedance. Consequently, this second group of capacitors needs to be located as close as possible to the IC. What's more, comodity electrolytic capacitors cannot operate at such high frequencies. Consequently, a ceramic capacitor is added, to handle the high frequency current pulses.

Lastly, the RESET pin of the controller is extremely sensitive to even short power line pulses (of the length of one clock cycle). Therefore, the exact RC circuit, recommended by Atmel, is used.

1.8.3 Connectors

The selected connector is NX5080-03SMR. This is a 3-pin, fixed orientation, low-current connector. The pin number is ideal for providing both power and a communication bus to connected boards. Because no terminal blocks are used, and all four connectors are wired in consistent fashion, incorrect wirein of the board is impossible.

1.8.4 Extendability

It has been paid attention to provide free PCB real estate, as well as conveniently located free processor pins. Consequently, it is trivial to add more connectors, or even a display, to the main board at a later moment of time.

2 Software

2.1 Theory

2.1.1 Radix tree

The radix tree is a data structure, suitable for storing and searching sets of strings. Each node, that is an only child, is merged with its parent. The operations, that can be performed on a radix tree, are as follows:

- insertion
- deletion
- searching
- find all strings with common prefix
- find predecessor
- find successor

All those operations exhibit complexity of O(k), where k is the length of the string.

On the other hand, radix trees can only be applied to strings, or to sets wich are directly mappable to strings. They also need serializable types to work on.

2.1.2 Cooperative multitasking

Cooperative multitasking[1] is a strategy for systems, which need to run separate tasks in parallel. The is different from cooperative multitasking, where the scheduler periodically interrupts the running process and initializes contex switch.

Using this strategy forces the programmer to design all the tasks in a specific way. Each task must return control to the environment after doing a small ammount of work. This can become extremely difficult is there is no obvious main loop, or for long-running functions. If a process does not return timely, all other processes become CPU starved and freeze.

One of the advantages of commperative multitasking is that context switches are controlled by the programmer. This way shared data can be proteced and fewer atomic sections need to be erected. Another consequence of this rather simple design is low overhead.

2.2 Goals

The device is a typical realtime embedded system. As such, mainframe programming tecniques are not applicable and embedded systems approach is required. Consequently, the following requirements are layed out to the program:

- Conveninet programming - during development, reprogramming the device shall be quick and easy, preferrably consisting of a single action.
- Convenient communication - both debug data and process information shall be easily accessible during device operation.
- Error tolerance - the device shall not freeze up upon an error, but should instead make best effort to keep the control loop active.
- Determinism - the programmer needs to have full control over what is executed when.
- Scriptability - the device shall be able to get reconfigured without a human at the other end id est a PC program shall be provided, which reconfigures the device in arbitrary ways.

In the oppinion of the author, all of these requirements have been fully fufilled with the current version of the software. The written software is free and open-source in it's entirety - it is protected under the MIT license.

2.3 Language selection

The selected processor can be programmed in avr-assembler, C or C++. Additionally, compilers exist for other less widespread languages, such as ADA. The author has selected the C language for it's simplicity, widespread use and compiler support for all of it's features.

2.4 Bootloader

2.4.1 Motivation

Several approaches were evaluated to reprogram the chip.

As the chip resides in a socket, instead of being soldered to the Main board, it could be removed from it's socket and inserted into a programmer. This approach is useful, but was discarded as extremely time-consuming.

The next possible approach is In-system programming. This constitutes soldering a header onto the Main board, and connecting it to the SPI bus of the microcontroller. This approach was also rejected for hte following reasons. The author was relucant to pay both the price in real estate on the PCB and the aditional complexity of wirein the header to the microcontroller. Not to mention using up some pins. Last but not least, this approach would significantly complicate the RESET pin nooise protection circuit.

The most complex, but in the same time best performing, approach was selected. Utilizing the fact that serial communication to a PC is needed for other reasons, the author decided to transfer new programs over the same bus. This way the device can be reprogrammed purely bu software, possibly even from a remote location.

2.4.2 Previous work

Many bootloader programs exist for the selected chip. Unfortunately, all of them are either:

- bloated with unnececery functionality, thus large in size,
- do not work out of the box and thus require manipulation of the source code to work or
- are licensed under un incompatible license than the veri permissive MIT license.

2.4.3 Implementation

The bootloader was implemented in standard-conforming C language, and written by the author in it's entirety. It is nearly impossible to create smaller in size bootloader for this device. The program code is simple and straighforward, lacking complicated preprocessor directives. Consequencly, it is in the oppinion of the author, easy to read and modify by anyone. The code is of cource publically visible and licensed in a way, that any person can freely and legally modify and redistribute it.

The bootloader instructions reside at the high end of the flash memory, while the application resides in the low end. Consequencly, the application never knows that a bootloader program was installed.

The finished product was called 'megaboot' - a bootloader for atmega devices. The source code in its entirety is provided as appendix A1.

2.4.4 Communication protocol

The XMODEM protocol has been selected. This protocol is not used in modern high-bandwidth, high-complexity networking. To the contrary. this protocol was popular when computers posessed resources comparable to the selected microcontroller. It is a simple protocol with very little overhead (channel efficiency is 97%). Furthermore, it is easy to implement in few program instruction, using up small amount of flash memory.

2.4.5 Error checking

Erorr checking is provided throughtout the program:

- CRC sums onto each received XMODEM packet
- "magic character" starts each packet
- addressing wrong (inexistient) flash pages is protected against
- receiving packets out of order is also checked agains

All the error checking code is conditionally included via the only preprocessor definition. This serves twin purpose. Not only does it clearly indicate error checking code apart from the actual program logic, but also provides the option to remove all error checking, should a person attempt to reduce the program size to the next smaller option.

2.4.6 Use

The build system passes the symbol BOOTLOAD to the application program. This symbol holds the address of the first instruction of the bootloader. The application is then free to jump to that address whenever required. In an assembler program, this would have been performed via a RJMP instruction. On the other hand, higher level languages cast the pointer to a function call address (optionally with the _Noreturn attribute) and call it.

After the bootloader has been invoked, it expects program data over the communication channel. Fortunately, many applications are available, which support the XMODEM protocol. One example is the popular and free serial terminal 'minicom'.

2.5 External components

2.5.1 Onewire library

A C library, written by Peter Dannegger, Martin Thomas and others, is being used for communication with the thermometer.

2.5.2 PID implementation

A professional implementation of a parallel pid with integral saturation, implemented by Atmel employees, is used.

2.6 Architecture

2.6.1 Theory

Numerous program architectures have been published throughout embedded programming books. However, they all fall in one of the following three groups:

- Continuous execution - suitable for very simple programs or devices, which do exactly one thing. The software performs it's tasks one after the other, waiting as long as needed for the tasks to complete.
- Cooperative multitasking - used for complex embedded applications in resource-constrained emvironments. Timer tick is defined and upon each tick, the software performs all scheduled tasks. The differences with the above are:
    1. The tasks are performed once per specified time period, and not continuosly. This is important for the PID algorithm.
    2. As a consequence, tasks must be written with multitasking in mind. Even a single process, which blocks for longer than the timer tick, would violate the real-time operation of the system.
- True preemptive multitasking - used for complex systems, such as personal computer operating systems. This approach is far superior to all others, because programs are easily written, and it is the operating system's responsibility to schedule them efficiently. Unfortunately, this is a complex taks, utilizing expensive (id est slow and memory consuming) algorithms. Nevertheless, there exist lightweight multitasking operating systems for atmega devices.

The author has selected the secodnd approach.

2.6.2 Implementation

The file 'src/main.c' performs scheduling of the services, provided by the remaining modules. The author has put effort into making all other modules independent of each other. This is expected to provide the following benefits:

1. The code is simple and easy to understand.

2. The implementation is not prone to error, due to chaning an unrelated portion of the code.

3. Software modules are easy to remove from the current project and insert into another, unrelated, project.

2.7 Modules

2.7.1 Configuration

Following best programming practices, the author has exported global project setting to the file 'src/config.h'. From there it is easy to control:

- all serial communication options
- hardware bindings id est which pins fufill which tasks
- thermometer library settings
- is error checking to be performed

2.7.2 Clock

This module utilizes a separate hardware timer to provide timekeeping and event management services. The module keeps track of relative time, that is seconds elapsed from turn-on. The clock wrap period is 136 years.

Event management is acheaved through the function clock_sleep_until_next_second(). This is the essence of the multitasking system.

2.7.3 Serial communication

The built-in 'Universal sinchronous and asynchronous receive and transmit unit' or USART module is utilized in an elegant way. By providing implementation to functions put_char() and get_char(), the usart software modules actually binds 'stdin', 'stderr' and 'stdout' streams to the USART unit. This fasciliates usage of standard stream operations, such as 'printf()' and 'scanf()'. An added advantage is that the serial communiation implementation can be changed - e.g. to flow over USB or Bluetooth, and zero client code will need to be modified.

2.7.4 Commands

Utilizing the multitasking architecture of the software, a 'commands' module is implemented, which accepts user instructions at all possible time points, irrespective of currently executed tasks. An added advantage of the commands system is that the device is fully scriptable. Elaborating, a 'bash' script can issue commands to the system and read status back. Such a script can, for example, log temperature data and performed control actions by the device. As a matter of fact, such a script is provided, together with a Qt4 implementation of a temperature data visualization tool. For more details, see the section Plotting tool

2.7.5 Zero-cross detector

An elaborate algorithm for mains voltage control was implemented. The following options were considered:

- Perform pulse width modulation on the control signal to the heater. The sinusoidal nature of mains voltage introduces an absolute timing error of up to $2 * 10$ms. In the author's oppinion, with this simple scheme, only on-off control is possible.
- Perform classical phase control. This approach has proven itself over the years. Furthermore, extensive literature is available, concerning this well-understood phenomenon.
- Measure AC phase and turn on only specific integer number of half-waves.

As a PID control algorithm is desired, the first approach is immediately discarded. The author moved away from the second approach for the following reasons. The device in it's entirety is connected to the mains in a single point. Phase control not only produces significant electro-magnetic interference, but also requires a zero-cross detector. Such a detector relies on the clan waveform of the measured signal. Consequenctly, in such a

scheme, the triac would be injecting noise into the zero-cross sensor, nessetiating extensive filtering, with all of the associated complexity and latency.

Consequencly, the author has selected the final approach. By paying the price of less resolution than actual analog phase control, the following benefits have been attained:

- Significantly reduced EMI - the selective activation of separate half-periods introduces a harmonic of 100Hz frequency. However, it's power is believed to be significantly lower than the combined power of the infinite spectrum ofa phase-controlled system.
- Simplified control - phase control would have nesseciated floating point numbers. Linking in the floating point math library is expensive (both execution speed and storage space wise).
- Indeterminate - the IEEE standard for floating point implementation in computers highlights numerous cases, where the result of a computation is implementation-dependent. In contrast, an integer number implementation is always fully deterministic.

But wait! The author has identified an algorithm to increase the precision to theoretically infinite value. By utilizing a variation of Bresenham's line algorithm, comulative error is eliminated given enough time. Furthermore, the algorithm supports safe modification of the setpoint, without relying on the fact, that the system tick is of known duration. The implementation of the algorithm is quite simple and can be observed in file src/zcd.c, being contained in the function shoudl_turn_on();

2.7.6 Triac control

2.7.7 Temperature measurement

2.8 Plotting tool

Plotting a signal in real time can be acheaved by numerous approaches. Among them are MATLAB, Scylab, Mathematica, Qt etc. However, those that are not proprietery, are still badly suited to real-time operation. Thus were selected the 'python' and 'matplotlib' technologies for their versability.

The written plotting tool is called 'rtplot' - RealTime Plotting utility. It operates on a vector of signals, plotting them against different time periods. The longer time periods obtain their datapoints via algebraic averaging of the next shortest plot. The program output covers the whole display and presents axis labels and units of measure. Inidividual datapoints are also displayed, in addition to the interpolating line between them.

3 Algorithms

3.1 Theory

3.1.1 Parallel PID controller

The analog PID controller is an enchancement of the PI controller. An aditional compoennt is assed to the PI sum. It depends on the speed of change of the error but is expected to be zero at steady-state. Here is how PID control looks like in the time domain:

$$u(t) = K_p e(t) + K_p \frac{1}{T_i s} \int_0^{t_1} e(t)dt + K_p T_d \frac{de(t)}{dt} = u_p(t) + u_i(t) + u_d(t) \tag{2}$$

where T_d is caleld derivative time constant.

In state space the PID control law looks like this:

$$u(s) = K_p e(s) + K_p \frac{1}{T_i s} + K_p T_d s e(s) = u_p(s) + u_i(s) + u_d(s) \tag{3}$$

And for the transfer function (this is a parallel structure PID) we have:

$$W_{pid}(s) = \frac{u(s)}{e(s)} = K_p + K_p \frac{1}{T_i s} + K_p T_d s = K_p \left[\frac{T_i T_d s^2 + T_i s + 1}{T_i s} \right] \tag{4}$$

The advantages of the PID controller include swift transient processes and reduced overshoot. The main disadvantages is that when either the setpoint $u(t)$ or the Load Disturbance changes, the effect is opposite - the derivative term strives for overshoot. This is called 'set point kick' and is malitious [4, p. 33].

3.1.2 Anti-windup

Among numerous practical applications, the permissible range of values for the controller output is limited. For example, a mechanical valve cannot be opened wider than it's physical construction allows. A triac-controlled heater cannot supply negative thermal energy. When the control signal $u(t)$ exceeds the maximum possible control, the integral factor becomes ineffective.

The most popular algorithms for aveliating this problem are [4, p. 49]:

- Limmiting the magnitude of the integral sum in $[u^{max}, u^{min}]$.
- Summing only on valid computed control (not clipped).

3.1.3 Plant models

Some algorithms for tuning vontrollers rely on a model of the plant. Others rely on observing a step response or stable oscilaltions state. Thus, firstly the most commonly used models are examined [4, p55].

A two parameter model accounts for pure delay and first order plat dynamics. It is used for integrating plants.

$$W(p) = \frac{a}{Lp} e^{-Lp} \tag{5}$$

,where:
L - pure time delay
a - coefficient.

A three parameter model is insted of aperiodic type.

$$W(p) = \frac{K_0}{1 + Tp} e^{-Lp} \tag{6}$$

where:
L - pure time delay,
K_0 - gain,
T - dominant time constant.

3.1.4 PID tuning algorithms

The **tangent method** is a graphical or computational algorithm. A tangent line is constructed through the inflection point of the plant's step response. By measuring the coordinats of the itnersection of this tangent with y or x axis, the parameters of 2 or 3 parameter model are estimated [4, p. 55].

The **area-based method** for estimating coefficietnts of a 3 coefficient plat is a computational method. Measured is the area between the setpoint and step response. Also, measured is the area between the step response and x axis. Solely by using those two measurements, the area-based method estimates 3 parameter models.

Optimization-based methods can fit models of unspecified order.

The **first method of Zigler-Nichols** is significant both for its simplicity and for its hystory value. The algorithm defines mapping from coefficients in (5) to recommend p, i and d gains for the closed system. Even among the plethora of modern methods, Zeighle-Nichols I remains a useful starting point for more complex methods, such as optimizing or self-tuning.

The **second method of Ziegler-Nichols** is based on steering the plant into sustained oscillations. The period and magnitude of the resultant oscillations are measured. Together

with the loop time and the amplitude of the relay, a second table provides suggestions on K_p, T_i. T_d. This method is considered an improvement on the Ziegler-Nighols I method.

Astrom-Hagglund also utilize the self-oscilating mode of target plant. However, this method advies a different way of controlling the system into oscilations. This relay-based method is believed to be more effective tahn ZN for stable or inert plants.

Of course there exist numerous core complicated tuning algorithms, many of which propriatery. Just to name a few: optimization of value function, neural networs, robust parameters for controlling stochastic plant.

3.1.5 Digital control systems

Computer control systems sample their imputs[2, p. 904] at discrete intervals of time. Assuming that the analog signal is constant in the unmeasureable moments is suitable approximation. The (logical) devices, which perform conversion between analog and discrete time domain are sampler and respectively n-th level hold.

3.1.6 Digital PID controller

One method [4] relies on the following assumptions [4, p. 88]:

- $u_i(t)$ is estimated using second rectangle method
- $u_b(t)$ is estimated via first backwards finate difference.

This is the non-recursive descrtiption of the controller:

$$u(k) = K_p\left[e(k) + \frac{T_0}{T_i}\sum_{i=1}^{k} e(i) + \frac{T_d}{T_0}(e(k) - e(k-1))\right] \tag{7}$$

If the control law is to be computed by a digital device, a more efficient approach would be as follows. Let us assume

$$K_i\frac{K_pT_0}{T_i}$$

$$K_d = \frac{K_pT_d}{T_0}$$

Then the PID controlelr can be written in the following rocursive format:

$$u(k) = K_pe(k) + I(k) + K_d(e(k) - e(k-1)) \tag{8}$$

$$I(k) = I(k-1) + K_ie(k), I(0) = 0 \tag{9}$$

3.2 PID implementation

Used is a discret pid algorithm with parallel structure and integral windup. The computer implementation expects proportional integral and derivative gains, as opposed to time constants. Consequencly, the responsibility for translating from T_i, T_d to K_i, K_d falls to the client software, written by the author.

3.3 Units of measure

Because various physical and computational resources are performed in the system, unit conversions need to be carefully considered.

We are constrained by the PID algorithm software to use 'int16_t' values for input, output and coefficients. Furthermore, coefficients are represented in 9s6 format i.e. $128 == 1.0$.

Secondly, the temperature sensor interface library returns 'decicelsius_t'. This is an integral value of measured degrees celsius, multiplied by 10. The accuracy of the thermometer is $0.5°C$

Last constraint is the input to the triac control. In an attempt to increase the resolution of the triac control algorithm as far as possible, the author has used the whole range of 'uint16_t' values. In other words, 0 corresponds to 0% or 0V at the heater, and $2^16 \equiv 65536$ corresponds to 100% or 230V at the heater.

Apparently, the PID algorithm needs to read 'decicelsius_t' and output 'uint16_t', which is impossible. One of the two units needs to be selected for the algorithm to work with, and the other - converted to/from. The author has selected 'decicelsius_t' as the more human-readable format. Furthermore, no aditional precision is lost by using this shriked type.

3.4 Nonlinearity

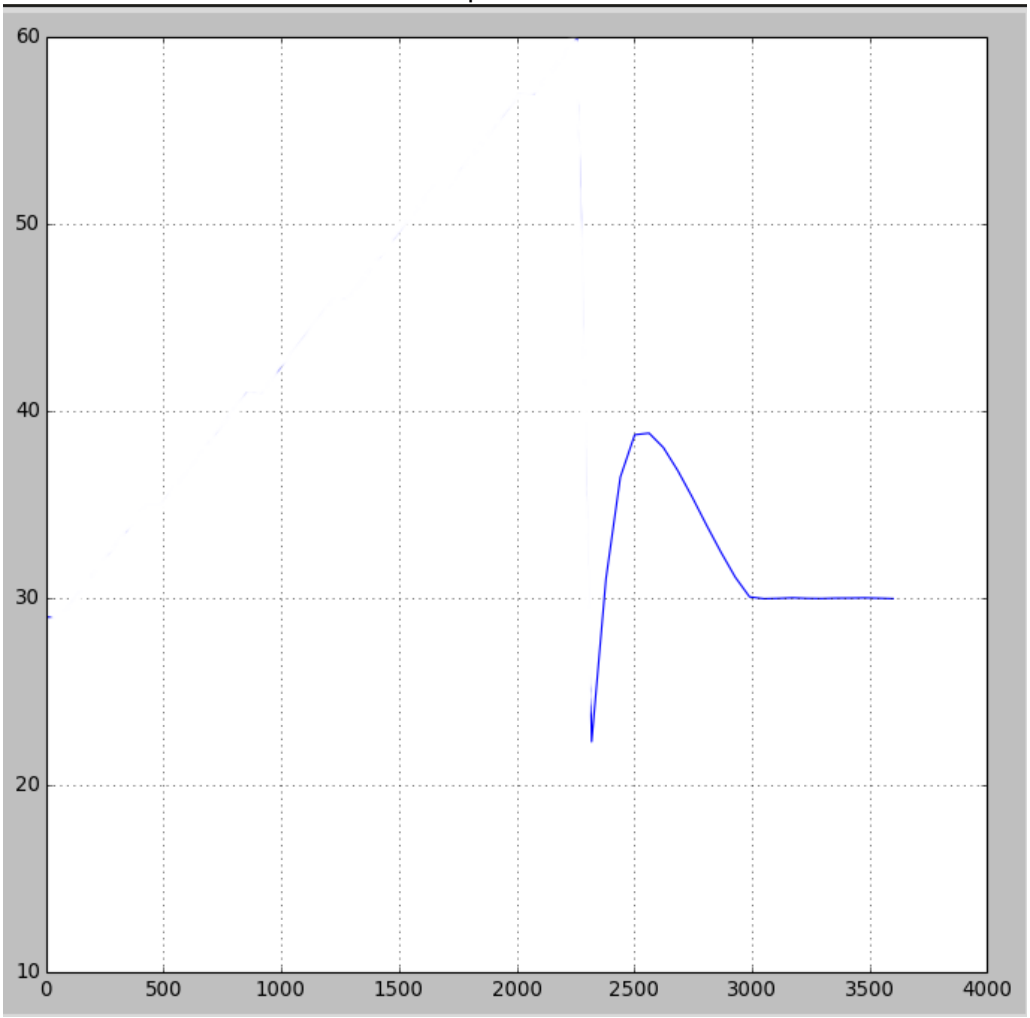
The heating circuit is expected to exhibit good linearity. However, the cooling process is both not under the control of the regulator, and extremely non-linear. The author expects cooling to conform to the Stefan-Bolzman law of radiated thermal energy in free space:

$$L = \frac{\sigma}{\pi} T^4$$

If this is true, estimating a transfer function of the whole system is pointless. The resulting linear system will be close to reality only for a very limited range of setpoints.

3.5 Ziegler-Nichols sustained oscillations

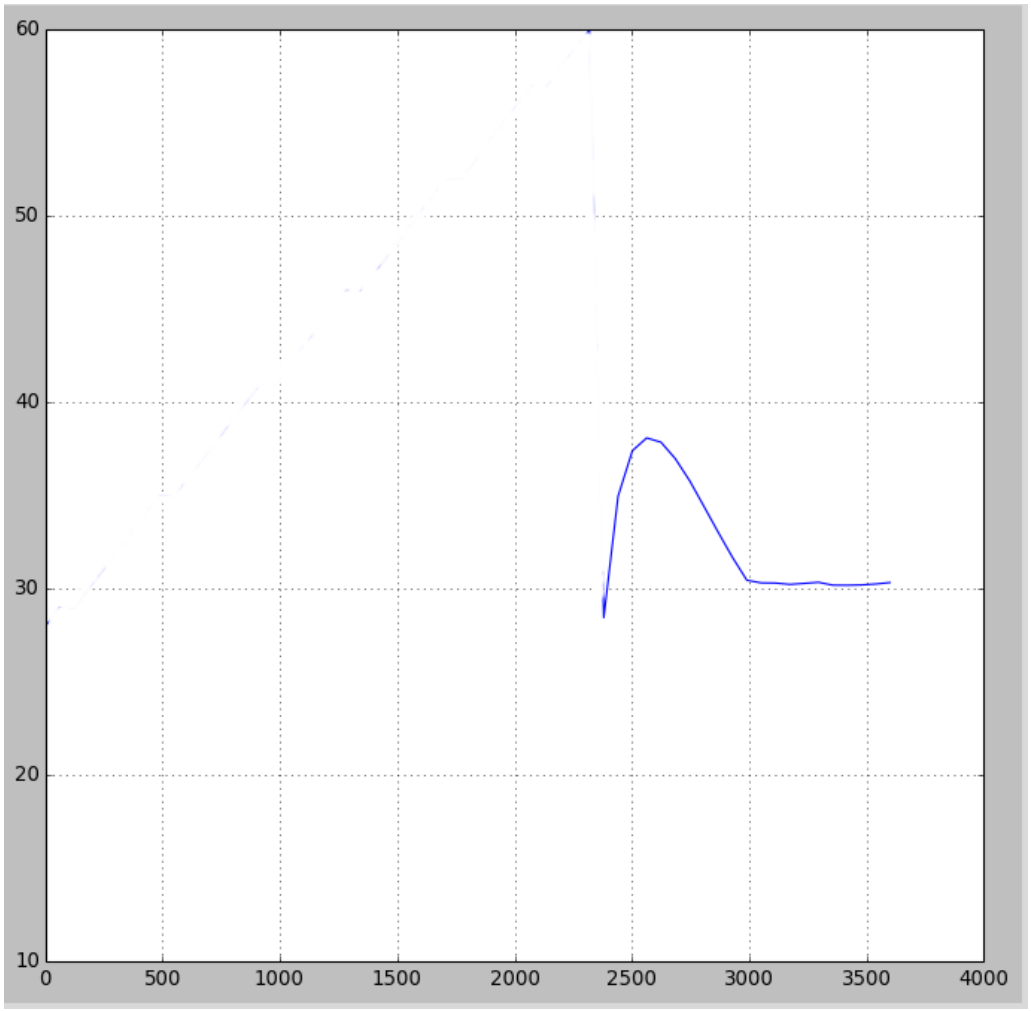
Initially, the Ziegler-Nichols algorithm for obtaining sustained oscillations and thus critical gain K_u and critical period T_u was attempted. It was observed that if any oscillations are observed at all, their amplitude is lower than the absolute error in the system.



3.6 Astrom-Hagglund sustained oscillations

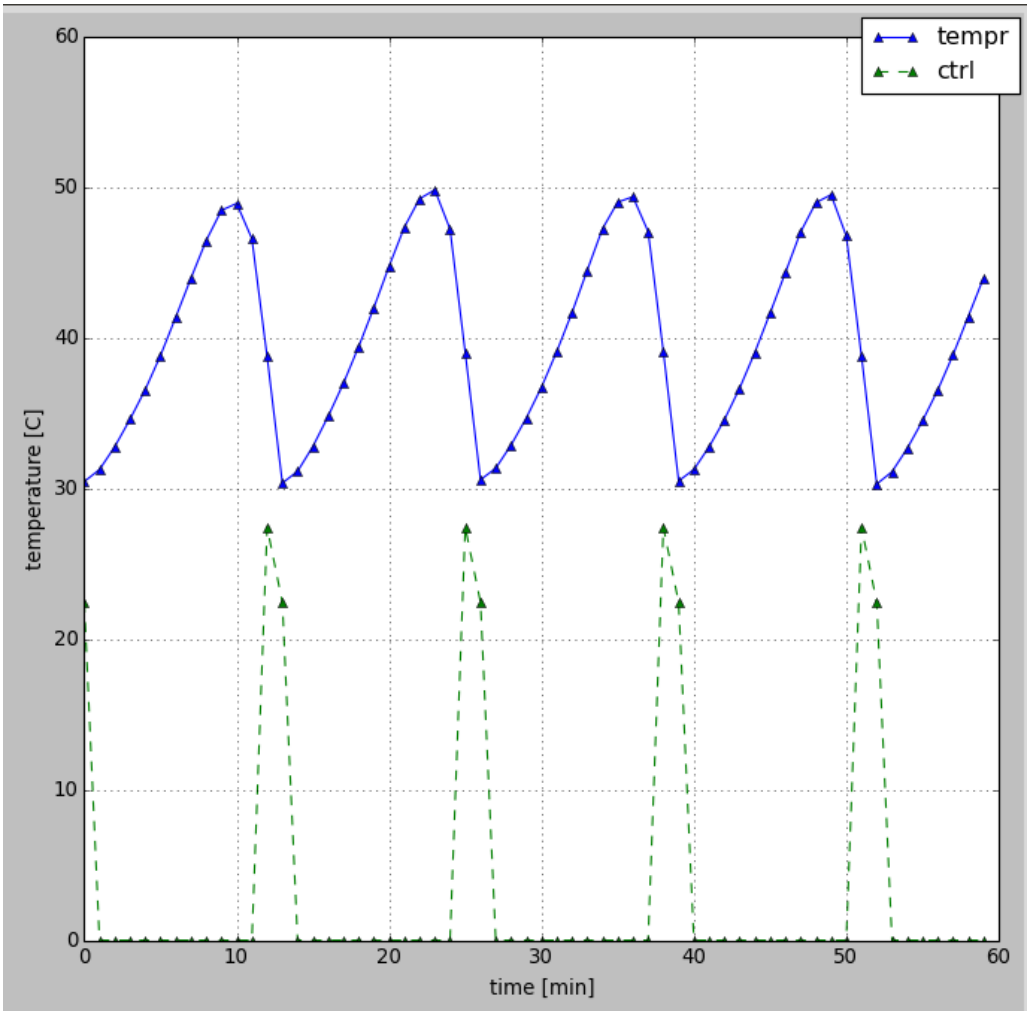
Secondly, replacing the PID controller with a relay was attempted. The relay acts as a sign function. It was observed that again no sustained oscillations are exhibited by the

system.



3.7 Sampling rate correction

Following advice in the literature[3], the author increased the control algorithm period to roughly $1/10$'th of the dominant open loop system time constant, namely to 60 seconds. Finally sustained oscillations were observed.



We observe 10 minute rise time and 3 mintute fall time. The oscillation amplitude is 29 C.

$$K_u = \frac{4d}{\pi a} = \frac{4 * 50}{\pi 29} \approx 9.90 \tag{10}$$

$$P_u = P_r + P_f = 13 * 0.6 * 60 = 780 seconds \tag{11}$$

By the second method of Ziegler-Nichols, we calculate [4]

$$K_p = 0.6 K_u = 5.94 \tag{12}$$

$$T_i = 0.5 T_u = 390\text{s} \tag{13}$$

$$T_d = 0.125 T_u = 97.5\text{s} \tag{14}$$

The calcualted values correspond to an analog PID controller. Normally, discretization would need to be performed on the model and the obtained coefficietns. However, in the current application, the control loop time is about two orders of magnitude faster than the dominant time cosntant.

Another consequence of running the experiment with different time constant than the real controller is that time is scaled. Consequently, after running a tuning loop with period of 60s in order to tune a controller, running with loop time of 1s, all obtained time cosntat coefficients need to be devided by 60.

3.8 Peak detector

A peak detector software module identifies local maxima and minima via 2 sample memory. It is equivelend to 1-dimentional 2nd order casual digital filter. During experiments it was established taht regardless of the inertial properties of the process, small local extrema (noise) twart the measurement.

As the peak detector circuit is not expected to be low-latency, non-casual filter can be used for increased noise stability. Proposed is a 6-sample memory filter of the type:

$y(k) = (x(k-3) \geq x(k)) \wedge (x(k-2) \geq x(k)) \wedge (x(k-1) \geq x(k)) \wedge (x(k+1) < x(k)) \wedge (x(k+2) < x(k)) \wedge (x(k+3) < x(k))$

where,
$y(k)$ - this function is 1 (true) during a local maximum, 0 (false) otherwise.
$x(k)$ - measured process output at k-th time period.
This solution is superior to classical low-frequency because it does not introduce error in signal timestamps.

4 Conclusion

It has been (accidentaly) demonstrated that too tight controller loop can prevent steering of intert plants into sustained oscillations. Realizing the overshoot is roughly proportional to the control loop duration, suitable sustained oscillation modes can be invoked.

Another practical problem encountered were the local extrema along the plant output, regardless of the plant's intertia. A proposed solution is designing a more sophisiticated, noncasual digital filter.

The following experimental points were reported by the device during recording of graph 3.7.1:

extr. type	deci	seconds
min	287	428
max	452	569
min	271	711
max	449	859

Those fit well the graph and specification, most within 10% relative error.

With the ever accelerating technological growth, RospberryPi products are overwhelming the low-price segment. AVR, PIC, MSP and other families of microcontrollers are still essential for low-power applications or mission-critical hardware.

5 Bibliography

References

[1] McGraw-Hill. Dictionary of scientific and technical terms, 6e. http://encyclopedia2.thefreedictionary.com/cooperative+multitasking. Online; accessed January 20 2016.

[2] Robert H. Bishop Richard C. Dorf. Modern control systems, 11th edition, 2007.

[3] Емил Гарипов. Идентификация на системи, част 1: Идентификация чрез непрекъснати модели, 2007.

[4] Емил Гарипов. Цифрови системи за управление, част 1: Проектиране на ПИД регулатори, 2007.

[5] Атанас Шишков. Полупроводникова техника, част 1, 1990.

6 Appendix 1 - Source code

Only source code, belonging to the author, is listed.

6.1 rtplot

Listing 1: 'hwcomm.py'

```
#!/usr/bin/env python

"""Receive line strings from device. Parse them and return
   numeric temperatures.

"""

import serial
import string
import sys
import time

MAX_TEMP = 60
MIN_TEMP = 0
NEWLINE = "\n"

class Serial(object):
    """Serial duplex communication."""
    def __init__(self, raw=False, simulate=False):
        self.simulate = simulate
        self.raw = raw
        if not simulate:
            self.comm = serial.Serial(port='/dev/ttyUSB0',
                                      baudrate=38400,
                                      bytesize=serial.
                                         EIGHTBITS,
                                      parity=serial.
                                         PARITY_NONE,
                                      stopbits=serial.
                                         STOPBITS_ONE,
                                      timeout=None,
                                      xonxoff=False,
                                      rtscts=False,
                                      writeTimeout=None,
                                      dsrdtr=False,
                                      interCharTimeout=None,
```

```
                                    )

    def run(self):
        if self.simulate:
            self._generate_random_data()
        else:
            self._listen_forever()

    def _listen_forever(self):
        """Blocking! Forever!"""
        def rl(size=None, eol=NEWLINE):
            """pyserial's implementation does not support the
               eol parameter"""
            ret = ""
            while True:
                x = self.comm.read()
                if x == eol:
                    return ret
                ret = ret + x
        self.comm.readline = rl

        while True:
            measurement = self.comm.readline()
            if self.raw:
                print measurement
            else:
                temp = self._parse_line_return_temp(
                   measurement)
                if temp is not None:
                    print temp[0], temp[1]

    def _generate_random_data(self):
        """For debug purposes."""
        import random
        while True:
            measurement = str(random.randint(MIN_TEMP,
               MAX_TEMP)) + ".0"
            print 0, measurement, float(measurement) / 2 #
               TODO: adapt _parse_line_return_temp()
            time.sleep(0.3)

    def _parse_line_return_temp(self, line):
        try:
            s1 = string.split(s=line, sep=' ')  # time
               decicelsius
            return (float(s1[1]) / 10), (float(s1[2]) / 10)
```

```
        except ValueError:
            return None


def main():
    import sys
    raw = False
    simulate = False
    for arg in sys.argv:
        if arg == '--raw':
            raw = True
        if arg == '--sim':
            simulate = True
    comm = Serial(raw=raw, simulate=simulate)
    comm.run()

if __name__ == "__main__":
    main()
```

Listing 2: 'plot.py'

```
#!/usr/bin/env python

"""A colelction of classes for dynamically updated XY-plots.

"""

# Constants
MINUTE_S = 60
HOUR_S = MINUTE_S * 60
DAY_S = HOUR_S * 24
WEEK_S = DAY_S * 7
MONTH_S = DAY_S * 30
YEAR_S = DAY_S * 365


# Settings
MAX_TEMP = 60
MIN_TEMP = 0
DATAPOINTS_PER_GRAPH = 60
TIME_INTERVALS = [MINUTE_S,
                  HOUR_S,
#                   DAY_S,
#                   WEEK_S,
#                   MONTH_S,
#                   YEAR_S,
                  ]
```

```
# Imports
import matplotlib.pyplot as plt
import collections as col
import numpy as np

class Demuxer(object):
    """Based on input temperature, update plots as needed"""
    def __init__(self):
        self.w = Window()
        self._counter_samples = 0

    def handle_new_value(self, vals):
        """Update relevant plots."""
        self._counter_samples += 1

        # Update shortest interval plot. This always happens.
        self.w.add_datapoint(plot_number=0, y=vals)

        # Calculate the impact of a new point on the averaging
            plots.
        # We skip the first plot, because it is already
           covered above.
        for i, v in enumerate(TIME_INTERVALS[:-1]):
            target_plot = i + 1
            if self._counter_samples % v == 0:
                avv_vals = self._get_average(plot_num =
                   target_plot - 1) # next shorter interval
                self.w.add_datapoint(plot_number=target_plot,
                                     y=avv_vals)

    def _get_average(self, plot_num):  # TODO: vectorize!
        data = self.w.get_yaxis(plot_num=plot_num)
        average = np.mean(data, axis=1)
        return list(average)

class Window(object):
    """Holds a collection of equally-sized, static x-axis,
       dynamic
     y-axis plots. Temperature range aware. Takes up the whole
         screen.

    """
    def __init__(self):
        # Redraw plots as soon as self.fig.canvas.draw() is
           called.
```

```
        plt.ion()

        # Create the window surface
        dpi=80  # default value
        screen = get_screen_resolution()
        width = screen[0] / dpi
        height = screen[1] / dpi
        self.fig = plt.figure(figsize=(width, height), dpi=dpi
           )  # the main window

        # Create the individual plots
        self.plots = []
        plots_map = 100 + len(TIME_INTERVALS) * 10   # 234
           means 2x3 grid, 4th subplot
        for i, d in enumerate(TIME_INTERVALS):
            x_axis = np.linspace(0, d, DATAPOINTS_PER_GRAPH)
            graph = Graph(window=self.fig, subplot_num=
               plots_map + i + 1, x_axis=x_axis)
            self.plots.append(graph)

    # Redrawing belongs here for fine control over this time-
       consuming operation.
    # Note that fig.canvas.draw() redraws the whole window!
    def add_datapoint(self, plot_number, y):
        self.plots[plot_number].add_datapoint(y)
        self.fig.canvas.draw()

    def get_yaxis(self, plot_num):
        return self.plots[plot_num].y_data


class Graph(object):
    # Datapoints 'y' can contain arbitrary number of elements.
    # Each index in 'y' is treated as separate signal.
    # Signal hystory is kept as list of deques.
    # Each deque represents1 a circular buffer of a single
       signal.
    def __init__(self, window, subplot_num, x_axis, dim=2):
        ax = window.add_subplot(subplot_num)
        l = len(x_axis)
        self.y = ax.plot(range(l), l*[-1,], '-',       #
           Obtain handle to y axis.
                         range(l), l*[-1,], '--',
                         marker='^'
                         )
```

```
        # Hack: because initially the graph has too few y
           points, compared to x points,
        # nothing should be shown on the graph.
        # The hack is that initial y axis is seto be be below
           in hell.
        self.y_data = []
        for i in range(dim):
            self.y_data.append( col.deque(len(x_axis)
               *[-9999,],          # Circular buffer.
                                       maxlen=len(x_axis)
                                       )
                               )

        # Make plot prettier
        plt.grid(True)
        plt.tight_layout()
        ax.set_ylim(MIN_TEMP, MAX_TEMP)
        plt.figlegend(self.y, ('tempr', 'ctrl'), 'upper right
           ');
        ax.set_ylabel('temperature [C]')
        # Terrible hack!
        if subplot_num == 121:
            ax.set_xlabel('time [s]')
        else:
            ax.set_xlabel('time [min]')

    def add_datapoint(self, y):
        if not isinstance(y, col.Sequence):
            y = [y,]

        print "y is ", y
        for i in range( len(y) ):
            self.y_data[i].appendleft(y[i])
            self.y[i].set_ydata( self.y_data[i] )


def get_screen_resolution():
    return 1366, 768


def main():
    import sys
    d = Demuxer()
    while True:
        # Plot space delimited line of measurements.
        try:
```

```
            line = sys.stdin.readline()
            values = line.split()
            cast = (values[1], values[2])
            d.handle_new_value(cast)
        except:
            pass


if __name__ == "__main__":
    main()
```

Listing 3: 'run.sh'

```
#!/bin/bash
LOGFILE=/tmp/tempr_log

trap "kill -- -$$" EXIT          # Kill all children on exit.

stdbuf -o0 ./hw_comm.py --raw --sim  > "$LOGFILE"  & #
   Unbuffered logging.
tail -f "$LOGFILE" | ./plot.py              &
tail -f "$LOGFILE"
```

6.2 megaboot

Listing 4: 'makefile'

```
PROJNAME         = megaboot
UC               = atmega168
BOOTLOAD         = 0x3E00 # byte address, start of bootlaoder
HEXFORMAT        = ihex

LDFLAGS          = -lm -lc -Wall -mmcu=$(UC)
LDFLAGS_LOADER   = -nostartfiles
LDFLAGS_LOADER  += -Wl,-Map,build/bootloader.map
LDFLAGS_LOADER  += -Wl,--section-start=.text=$(BOOTLOAD)
LDFLAGS_APP      = -Wl,-Map,build/test.map
LDFLAGS_APP     += -Wl,--section-start=.text=0
CFLAGS           = -fpack-struct -Os -mcall-prologues -mmcu=$(
   UC)
CFLAGS          += -finline-functions --std=c11
CFLAGS          += -Wall -Winline -Wstrict-prototypes -Wno-main
    -Wfatal-errors -Wpedantic
CFLAGS          += -DBOOTLOAD=$(BOOTLOAD)

all:
        # Compile.
```

```
	avr-gcc $(CFLAGS) src/bootloader.c -c -o build/
	   bootloader.o
	avr-gcc $(CFLAGS) src/test.c -c -o build/test.o

	# Link.
	avr-gcc $(LDFLAGS) $(LDFLAGS_LOADER) build/bootloader.
	   o -o build/bootloader.elf
	avr-gcc $(LDFLAGS) $(LDFLAGS_APP) build/test.o -o
	   build/test.elf
	avr-objcopy -j .text -j .data -O $(HEXFORMAT) build/
	   bootloader.elf build/bootloader.hex
	avr-objcopy -j .text -j .data -O $(HEXFORMAT) build/
	   test.elf build/test.hex
	avr-objcopy -j .text -j .data -O binary build/test.elf
	    build/test.bin

	# Combine.
	srec_cat build/test.hex -I build/bootloader.hex -I -o
	   build/$(PROJNAME).hex -I

	# Report.
	# If bootloader .text size exceeds 512 bytes, it will
	   no longer fit in the NRWW section!
	avr-size -B build/bootloader.elf build/test.elf

upload:
	sudo avrdude -p $(UC) -c usbasp -e -U flash:w:build/$(
	   PROJNAME).hex

fuses:
	sudo avrdude -p $(UC) -c usbasp -U lfuse:w:0xE2:m -U
	   hfuse:w:0xDF:m -U efuse:w:0x05:m
	# Default for the atmega168 is lfuse:62, hfuse:df,
	   efuse:01
	# Low fuse for 8MHz clock: 0xE2
	# Extended fuse with 512 bytes bootloader, start at
	   application start: 0x05

disasm:
	avr-gcc $(CFLAGS) bootloader.c -S -o build/bootloader.
	   S && nano build/bootloader.S

clean:
	@mv build/empty.txt .
	rm -f build/*
	@mv empty.txt build/
```

Listing 5: 'src/config.h'

```
#ifndef _CONFIG_H_
#define _CONFIG_H_

#include <stdint.h>

// If undefined, error checking on the XMODEM protocol is
   disabled.
// Also, the flash writing routine is not protected form
   overwriting the bootloader section.
// Furthermore, flash memory verification after write is not
   performed.
#define ERROR_CHECKING

// CPU nominal frequency, Hz.
#define F_CPU 8000000

// USART baud rate.
#define BAUD 38400

// This macro is passed from the makefile.
// It contains the byte address of the start of the bootloader
    section.
// This depends upon device and the BOOTSZ[1, 0] bits.
// YOU MUST SET THIS MANUALLY IN THE MAKEFILE.
// #define BOOTLOAD <byte_address>

enum
{
    ERROR_NONE                          =  0,
    ERROR_PROTOCOL_FIRST_CHARACTER      = -1,
    ERROR_PROTOCOL_PACKET_NUMBER_INVERSION = -2,
    ERROR_PROTOCOL_PACKET_NUMBER_ORDER  = -3,
    ERROR_PROTOCOL_CRC                  = -4,
};
typedef int8_t error_t;

#endif  //#ifndef _CONFIG_H_
```

Listing 6: 'src/bootloader.c'

```
#include <avr/io.h>
#include <avr/boot.h>
#include <avr/interrupt.h>
#include <avr/pgmspace.h>
#include <stdint.h>
```

```
#include "usart.h"
#include "config.h"

#define XMODEM_PAYLOAD_BYTES 128

// Last page available to the application section. Next page
   is part of the bootloader section.
#define APPLICATION_SECTION_END_PAGES (BOOTLOAD / SPM_PAGESIZE
   )

// The flash writing routines expect an unsigned char.
typedef unsigned char buff_t;

// Custom runtime assert statement.
#undef assert
#ifdef ERROR_CHECKING
    #undef NDEBUG
#else
    #define NDEBUG
#endif

#ifdef NDEBUG
#define assert(expr)
#else
//void fatal_error(int err_num, char *fname, int line_num,
   const char *foo)
inline void fatal_error(void)
{
    usart_transmit('E');
    while(1);
}
//#define assert(expr) (void)((expr) || (fatal_error(0,
   __FILE__, __LINE__, __func__), 0))
#define assert(expr) (void)((expr) || (fatal_error(), 0))
#endif

// Remove interrupt vector table.
// Required because we are linking with -nostartfiles.
__attribute__ ((section(".init9"))) void initstack(void)
{
  __asm volatile (".set __stack, %0" :: "i" (RAMEND));  // Set
     stack.
  __asm volatile ("in r1, 0");                          // R1
     must at all times contain 0.
  __asm volatile ("rjmp main");                         //
     Jump to main().
```

```
}

// Write one Flash page.
// 'buffer' needs to be 'SPM_PAGESIZE' bytes long.
// 'page_offset' is the byte offset of the page
// We CAN'T jump to the application later because we did not
   call 'boot_rww_enable()'.
void program_flash_page(unsigned page_offset, const buff_t
   buffer[])
{
    assert(page_offset < BOOTLOAD);                          //
       Guard against overwriting the bootloader section.

    boot_page_erase(page_offset);
    boot_spm_busy_wait();                                    //
       Wait until the memory is erased.

    const buff_t *buff = buffer;
    for(int i = 0; i < SPM_PAGESIZE; i += 2)
    {
        // Set up little-endian word.
        uint16_t w = *buff++;
       w += (*buff++) << 8;

        boot_page_fill (page_offset + i, w);
    }

    boot_page_write(page_offset);
    boot_spm_busy_wait();                                    //
       Wait until the memory is written.

#ifdef ERROR_CHECKING
    // Verify flash content. Unfortunately this exceeds the
       512 byte target size, so is commented out.
//    boot_rww_enable();
//    buff = buffer;
//    for(unsigned i = 0; i < SPM_PAGESIZE; i+=2)
//    {
//        uint16_t w1 = *buff++;
//        w1 += (*buff++) << 8;
//
//        uint16_t w2 = pgm_read_word_near(i + page_offset);
//        assert(w1 == w2);
//    }
#endif
}
```

```
// Receive one XMODEM packet.
// Check for errors as provided by the protocol.
// Parameter: 'char payload_buffer[XMODEM_PAYLOAD_BYTES]' is
   an output.
// Return: 1 - EOF was received, 0 - normal packet, <0 - error
// See: https://en.wikipedia.org/wiki/XMODEM
// See: http://www.atmel.com/Images/doc1472.pdf
error_t receive_xmodem_packet(buff_t payload_buffer[])
{
    uint8_t first_byte = usart_receive();
    switch(first_byte)
    {
        case ASCII_EOT:
            return 1;
        case ASCII_SOH:
            break;        // Execute body of the function.
        default:
            return ERROR_PROTOCOL_FIRST_CHARACTER;
    }

    uint8_t packet_number = usart_receive();                ;
        (void)packet_number;
    uint8_t packet_number_inverted = usart_receive();       ;
        (void)packet_number_inverted;
#ifdef ERROR_CHECKING
    static unsigned packet_counter = 1;
       // First packet is number 1, not 0.
    if((uint8_t)(~packet_number_inverted) != packet_number)
       // Cast because of integer promotion.
    {
        return ERROR_PROTOCOL_PACKET_NUMBER_INVERSION;
    }
    if(packet_counter != packet_number)
       // TODO: send ACK and ignore duplicate packet
    {
        return ERROR_PROTOCOL_PACKET_NUMBER_ORDER;
    }
#endif

    uint8_t checksum = 0;                                   ;
        (void)checksum;
    for(unsigned i = 0; i < XMODEM_PAYLOAD_BYTES; ++i)
    {
        payload_buffer[i] = usart_receive();
```

```
#ifdef ERROR_CHECKING
        checksum += payload_buffer[i];
           // Unsigned overflow is deterministic and safe.
#endif
    }

    const uint8_t expected_checksum = usart_receive();
       // 1-byte if initial request was NACK, 2-byte if initial
        request was 'C'.
    (void)expected_checksum;
#ifdef ERROR_CHECKING
    if(expected_checksum != checksum)
    {
        return ERROR_PROTOCOL_CRC;
    }

    ++packet_counter;
#endif
    return 0;
       // Packet received successfully.
}


// We are supposed to arrive here after a jump from the
   application section.
// Next we anticipate XMODEM transmission of new application
   data.
// In the end we state success of failure via the serial
   conenction and wait for a cold reset.
void main(void)
{
    buff_t xmodem_payload[XMODEM_PAYLOAD_BYTES];
    const buff_t *page_ptr;                          //
       SPM_PAGESIZE == 64 bytes for an atmega8
    unsigned page_offset = 0;

    cli();
    usart_init();
    usart_transmit(ASCII_NACK);                    // Request
       file transfer.

    while(1)
    {
        int packet_type = receive_xmodem_packet(xmodem_payload
           );
        switch(packet_type)
```

```
        {
            case 1:                                     // This is
               the last packet. It does not contain data.
                usart_transmit(ASCII_ACK);
                goto FINISHED;

            case 0:                                     // Packet
               accepted correctly, proceed to next one.
                page_ptr = xmodem_payload;
                for(int i = 0; i < (XMODEM_PAYLOAD_BYTES /
                   SPM_PAGESIZE); ++i)
                {
                    program_flash_page(page_offset, page_ptr);
                    page_offset += SPM_PAGESIZE;
                    page_ptr += SPM_PAGESIZE;
                }
                usart_transmit(ASCII_ACK);          // Request
                   another packet AFTER we have processed the
                   previous.
                break;

            default:                                    // Error in
               transmission, request resend of packet.
                usart_transmit(ASCII_NACK);
                break;
        }
    }

FINISHED:
    while(1);
}
```

Listing 7: 'src/usart.h'

```
#ifndef _USART_H_
#define _USART_H_

#include "usart.h"
#include "config.h"

#include <avr/io.h>
#include <util/setbaud.h>     // F_CPU and BAUD defined in
   config.h
#include <stdint.h>

#define NEWLINE "\r\n"
enum
```

```
{
    ASCII_SOH  = 0x01,
    ASCII_EOT  = 0x04,
    ASCII_ACK  = 0x06,
    ASCII_NACK = 0x15,
};


inline void usart_init(void)
{
    UBRR0H = UBRRH_VALUE;                                   // Set baud
       rate.
    UBRR0L = UBRRL_VALUE;
#if USE_2X                                                  // We are
   operating in asychronous mode: we need this consideration.
    UCSR0A |= (1 << U2X0);
#else
    UCSR0A &= ~(1 << U2X0);
#endif
    UCSR0B = (1<<RXEN0)|(1<<TXEN0);                         // Enable
       receiver and transmitter
    UCSR0C = (3<<UCSZ00);                                   // Set frame
       format: 8 data bits, 1 stop bit, no parity aka 8N1
}

inline void usart_transmit( unsigned char data )
{
    while ( !( UCSR0A & (1<<UDRE0)) );                      // Wait for
       empty transmit buffer
    UDR0 = data;                                             // Put data
       into buffer, sends the data
}

inline char usart_receive(void)
{
    while ( !(UCSR0A & (1<<RXC0)) );                        // Wait for
       data to be received.
    char c = UDR0;                                          // Get and
       return received data from buffer.
    return c;
}

#endif //#ifndef _USART_H_
```

Listing 8: 'test.c'

```
#include "config.h"
```

```
#include <util/delay.h>

#include "usart.h"

void send_block(unsigned bytes, char *buff)
{
    for(int i = 0; i < bytes; ++i)
    {
        usart_transmit(buff[i]);
    }
}

int main(int argc, char **argv)
{
    usart_init();

    char msg[] = "Loading bootloader in 3 seconds.";
    send_block(sizeof(msg), msg);
    _delay_ms(3000);

    typedef void (* fn_ptr_t) (void);
    fn_ptr_t my_ptr = (fn_ptr_t)BOOTLOAD;
    my_ptr();
}
```

6.3 micli

Listing 9: 'makefile'

```
PROJNAME        = micli
UC              = atmega168
HEXFORMAT       = binary
BOOTLOAD        = 0x3E00 # byte address, start of bootlaoder

LDFLAGS         = -lm -lc -Wall -mmcu=$(UC)
LDFLAGS        += -Wl,-Map,build/$(PROJNAME).map
CFLAGS          = -fpack-struct -Os -mcall-prologues -mmcu=$(
   UC)
CFLAGS         += -finline-functions --std=c11
CFLAGS         += -Wall -Winline -Wstrict-prototypes -Wno-main
    -Wfatal-errors -Wpedantic
CFLAGS         += -DBOOTLOAD=$(BOOTLOAD)

all:
        # Compile.
        avr-gcc $(CFLAGS) src/main.c -c -o  build/main.o
```

```
	avr-gcc $(CFLAGS) src/clock.c -c -o build/clock.o
	avr-gcc $(CFLAGS) src/commands.c -c -o build/commands.
	  o
	avr-gcc $(CFLAGS) src/pid_tune.c -c -o build/pid_tune.
	  o
	avr-gcc $(CFLAGS) src/strings.c -c -o build/strings.o
	avr-gcc $(CFLAGS) src/usart.c -c -o build/usart.o
	avr-gcc $(CFLAGS) src/tempr.c -c -o build/tempr.o
	avr-gcc $(CFLAGS) src/zcd.c -c -o build/zcd.o

	avr-gcc $(CFLAGS) libs/crc/crc8.c -c -o build/crc8.o
	avr-gcc $(CFLAGS) libs/ds18x20/ds18x20.c -c -o build/
	  ds18x20.o
	avr-gcc $(CFLAGS) libs/onewire/onewire.c -c -o build/
	  onewire.o
	avr-gcc $(CFLAGS) libs/pid/pid.c -c -o build/pid.o

	# Link.
	avr-gcc $(LDFLAGS) build/main.o build/clock.o build/
	  commands.o build/pid_tune.o build/strings.o build/
	  usart.o build/tempr.o build/zcd.o build/crc8.o build
	  /ds18x20.o build/onewire.o build/pid.o -o build/$(
	  PROJNAME).elf
	avr-objcopy -j .text -j .data -O $(HEXFORMAT) build/$(
	  PROJNAME).elf build/$(PROJNAME).bin

	# Report.
	avr-size -B build/$(PROJNAME).elf

clean:
	@mv build/empty.txt .
	rm -f build/*
	@mv empty.txt build/
```

Listing 10: 'config.h'

```
#ifndef _CONFIG_H_
#define _CONFIG_H_

#include <stdint.h>

// The main loop runs in 1 second.
// As the pid control loop period determines a lot of the math
   ,
// it can be run at any integer number of seconds, from 1 to
   UINT16_MAX.
#define PID_CONTROL_LOOP_SECONDS 1
```

```
// CPU nominal frequency, Hz.
#define F_CPU 8000000

// USART baud rate.
#define BAUD 38400

// This macro is passed from the makefile.
// It contains the byte address of the start of the bootloader
    section.
// This depends upon device and the BOOTSZ[1, 0] bits.
// YOU MUST SET THIS MANUALLY IN THE MAKEFILE.
// #define BOOTLOAD <byte_address>

// Maximum temperature in 'decicelsius', that should not be
   exceeded.
// Note that due to overshoot this can be exceeded by as much
   as 100 decicelsius.
#define PROC_VAL_CRITICAL (550)

// Control output is scaled to (MAX_TEMP-MIN_TEMP) and
   reported.
// This way the pid algorithm operates in a consistend unit of
    measure:
// 'decicelsius_t'.
#define MIN_TEMPR (0)
#define MAX_TEMPR (640)


// Heater hardware
#define PIN_HEATER  PD6
#define PORT_HEATER PORTD
#define DDR_HEATER  DDRD


// Thermometer configuration.
#define DS18X20_EEPROMSUPPORT     0
#define DS18X20_DECICELSIUS       1
#define DS18X20_MAX_RESOLUTION    0
#define DS18X20_VERBOSE           0

#define PORT_OW PORTB
#define PIN_OW  PB6
#define DDR_OW  DDRB
#define PINPORT_OW PINB
```

```
#define OW_ONE_BUS
#define OW_PIN PIN_OW
#define OW_IN PINPORT_OW
#define OW_OUT PORT_OW
#define OW_DDR DDR_OW


#endif  //#ifndef _CONFIG_H_
```

Listing 11: 'main.c'

```
#include "config.h"
#include "clock.h"
#include "commands.h"
#include "pid_tune.h"
#include "strings.h"
#include "tempr.h"
#include "usart.h"
#include "zcd.h"

#include <assert.h>


// Quieck hack for assert() to work.
#define abort() printf("CRITICAL ERROR")


decicelsius_t to_deci(zcd_proc_val_t val)
{
    return (((int32_t)val * (MAX_TEMPR-MIN_TEMPR)) /
       ZCD_PROC_VAL_MAX);
}
zcd_proc_val_t to_zcd(decicelsius_t val)
{
    if (val < 0) return 0;
    else return (((int32_t)val * ZCD_PROC_VAL_MAX) / (
       MAX_TEMPR-MIN_TEMPR));
}


void task_parse_cmd(void)
{
    int len;
    char cmd_buff[MAX_CMD_LEN];
    if(listen_for_command(cmd_buff, &len))
    {
        int err = execute_command(cmd_buff, len);
```

```
        if(err)
        {
            cmd_buff[len] = '\0';
            printf( strings_get(STR_UNKNOWN_COMMAND), cmd_buff
                );
        }
    }
}


void task_pid_run(void)
{
    static enum state_e
    {
        TUNING,
        CREATING,
        RUNNING,
    } state = TUNING;

    pid_inout_t proc_val = (pid_inout_t)tempr_get();
    pid_inout_t ctrl = 0;

    // Safety.
    if(proc_val >= PROC_VAL_CRITICAL)
    {
        printf( strings_get(STR_PROCESS_UNSTABLE) );
        zcd_proc_val_t cast = to_zcd(0);
        zcd_set(cast);
        return;
    }

    switch(state)
    {
        case TUNING:
            ctrl = pid_tune_Ziegler_Nichols( proc_val,
               clock_get() );
            if( pid_tune_finished() )  state = CREATING;
            break;

        case CREATING:
            ctrl = 0;
            pid_coeff_t p, i, d;
            pid_get_coeffs(&p, &i, &d);
            printf("Configuring pid with p=%i, i=%i, d=%i in 9
               s6 format." NEWLINE, p, i, d);
```

```
//              printf( strings_get(STR_PID_CALIBRATION), p, i,
   d );
            pid_config(p, i, d);
            state = RUNNING;
            break;

        case RUNNING:
            ctrl = pid_run(proc_val);
            break;
    }

    // Cast [0, 640] decicelsius to [2^0, 2^16].
    zcd_proc_val_t cast = to_zcd(ctrl);
    zcd_set(cast);
}


void task_report(void)
{
    clock_seconds_t time = clock_get();
    decicelsius_t temperature = tempr_get();
    zcd_proc_val_t triac = to_deci( zcd_get() );
    printf("%lu %i %u" NEWLINE, time, temperature, triac);
}


void task_tempr(void)
{
    tempr_measure();
}


void main(void)
{
    usart_init();
    clock_init();
    tempr_init();
    zcd_init();

    printf( strings_get(STR_PROGRAM_START) );
    for( ; ; clock_sleep_until_next_second() )
    {
        task_parse_cmd();
        task_tempr();
        task_pid_run();
        task_report();
```

```
    }
}
```

Listing 12: 'pid_tune.h'

```
// Used hardware: none.
// Used interrupts: none.

#ifndef _PID_TUNE_H_
#define _PID_TUNE_H_


#include "clock.h"
#include "tempr.h"

#include <stdbool.h>
#include <stdint.h>


#define PID_COEFF_MAX (INT16_MAX)
#define PID_INOUT_MAX (INT16_MAX)
#define PID_INOUT_MIN (INT16_MIN)


typedef int16_t pid_coeff_t;                    // 9s6 format
    i.e. 128 == 1.0
typedef int16_t pid_inout_t;


pid_coeff_t to_pid_coeff(int8_t coeff);
void pid_config(pid_coeff_t p, pid_coeff_t i, pid_coeff_t d);
void pid_setpoint(pid_inout_t sp);
pid_inout_t pid_run(pid_inout_t proc_val);

pid_inout_t pid_tune_Ziegler_Nichols(pid_inout_t proc_val,
   clock_seconds_t now);
bool pid_tune_finished(void);
void pid_get_coeffs( pid_coeff_t *p, pid_coeff_t *i,
   pid_coeff_t *d);


#endif  // defined(_PID_TUNE_H_)
```

Listing 13: 'pid_tune.c'

```
#include "pid_tune.h"
#include "usart.h"
```

```
#include "../libs/ds18x20/ds18x20.h"
#include "../libs/onewire/onewire.h"
#include "../libs/pid/pid.h"

#include <assert.h>
#include <stdbool.h>
#include <stdlib.h>
#include <util/atomic.h>
#include <util/delay.h>


#define ATOMIC ATOMIC_BLOCK(ATOMIC_FORCEON)
typedef enum pid_state_e
{
    SETTLED,
    OSCILLATING,
    UNSTABLE,
} pid_state_t;


typedef struct
{
    const bool is_min;
    pid_inout_t val;
    clock_seconds_t when;
} extremum_t;
typedef struct
{
    bool ready;
    extremum_t min, max;
    extremum_t *curr, *prev;
    pid_coeff_t p, i, d;
} pid_tune_t;


pid_inout_t pid_onoff_controller(pid_inout_t proc_val,
   pid_inout_t sp, pid_inout_t ampl);
pid_state_t pid_wait_to_settle(pid_inout_t proc_val,
   pid_inout_t critical, pid_inout_t treshold, clock_seconds_t
   now);


pid_inout_t g_setpoint = 0;
struct PID_DATA g_pid;
pid_tune_t g_pid_tune =
{
```

```
    .ready = false,
    .min = {.is_min=true, .val=PID_INOUT_MAX, .when=0},
    .max = {.is_min=false, .val=PID_INOUT_MIN, .when=0},
    .curr = &g_pid_tune.min,
    .prev = &g_pid_tune.max,
    .p = 0,
    .i = 0,
    .d = 0,
};

void pid_config(pid_coeff_t p, pid_coeff_t i, pid_coeff_t d)
{
    pid_Init(p, i, d, &g_pid);
}


void pid_setpoint(pid_inout_t sp)
{
    g_setpoint = sp;
}


pid_inout_t pid_run(pid_inout_t proc_val)
{
    pid_inout_t control = pid_Controller(g_setpoint, proc_val,
        &g_pid);
    return control;
}


pid_coeff_t to_pid_coeff(int8_t coeff)
{
    return coeff * SCALING_FACTOR;
}


bool pid_tune_finished(void)
{
    return g_pid_tune.ready;
}


void pid_get_coeffs(pid_coeff_t *p, pid_coeff_t *i,
   pid_coeff_t *d)
{
    assert(g_pid_tune.ready);
```

```
    *p = g_pid_tune.p;
    *i = g_pid_tune.i;
    *d = g_pid_tune.d;
}


// Astrom-Hagglund
// Second method of Ziegler-Nichols
pid_inout_t pid_tune_Ziegler_Nichols(pid_inout_t proc_val,
   clock_seconds_t now)
{
    assert(!g_pid_tune.ready);

    pid_inout_t ctrl;

    // Evaluate relay control only each
       PID_CONTROL_LOOP_SECONDS.
    const pid_inout_t relay_ampl = 400;
    static unsigned loop_counter = 0;
    static pid_inout_t prev_ctrl = 0;
    if(loop_counter++ % PID_CONTROL_LOOP_SECONDS)
    {
        ctrl = prev_ctrl;
    }
    else
    {
        ctrl = pid_onoff_controller(proc_val, 300, relay_ampl)
           ;
        prev_ctrl = ctrl;
    }

    pid_state_t osc = pid_wait_to_settle(proc_val, 600, 10,
       now);
    (void)osc;
//    assert(osc == OSCILLATING);  // we don't have asserts
   yet i.e. just hangs

    clock_seconds_t patience = 6000;
    if(now > patience)
    {
        g_pid_tune.ready = true;
        pid_inout_t a = g_pid_tune.max.val - g_pid_tune.min.
           val;
        pid_inout_t pi = 31;  // decicelsius?
        pid_inout_t Ku = (4 * relay_ampl) / (pi * a);
```

```
        clock_seconds_t Tu = 2 * (g_pid_tune.curr->when -
           g_pid_tune.prev->when);
        g_pid_tune.p = 6 * Ku;  // decicelsius?
        g_pid_tune.i = g_pid_tune.p / (5 * Tu);
        g_pid_tune.d = g_pid_tune.p / (1 * Tu);
printf("NOW max = %i at %lu, min = %i at %lu" NEWLINE,
   g_pid_tune.max.val, g_pid_tune.max.when, g_pid_tune.min.val,
    g_pid_tune.min.when);
    }

    return ctrl;
}


pid_inout_t pid_onoff_controller(pid_inout_t proc_val,
   pid_inout_t sp, pid_inout_t ampl)
{
    if(proc_val > sp)
    {
        return 0;
    }
    else
    {
        return ampl;
    }
}


pid_state_t pid_wait_to_settle(pid_inout_t proc_val,
   pid_inout_t critical, pid_inout_t treshold, clock_seconds_t
   now)
{
    extremum_t *min = &g_pid_tune.min;
    extremum_t *max = &g_pid_tune.max;
    extremum_t **curr = &g_pid_tune.curr;
    extremum_t **prev_ = &g_pid_tune.prev;

    static pid_inout_t prev, prevprev;

    if(*curr == min)
    {
        // Look for a local maximum.
        if(prevprev <= prev && prev > proc_val)
        {
            (*max).val = proc_val;
            (*max).when = now;
```

```
            *curr = max;
            *prev_ = min;
printf("new max: %i %lu" NEWLINE, (*max).val, (*max).when);
        }
    }
    else
    {
        if(prevprev >= prev && prev < proc_val)
        {
            (*min).val = proc_val;
            (*min).when = now;
            *curr = min;
            *prev_ = max;
printf("new min: %i %lu" NEWLINE, (*min).val, (*min).when);
        }
    }

    prevprev = prev;
    prev = proc_val;

    if(proc_val >= critical)  return UNSTABLE;
    else if((*min).val + treshold < (*max).val)  return
       SETTLED;
    else return OSCILLATING;
}
```

Listing 14: 'clock.h'

```
// Used hardware: watchdog.
// Used interrupts: WDT_vect.
// This module uses the watchdog 128kH timer to issue 1 second
    interrupts.
// It also keeps track of real time since start-up.

#ifndef CLOCK_H_
#define CLOCK_H


#include <stdint.h>


typedef uint32_t clock_seconds_t;
void clock_init(void);
void clock_sleep_until_next_second(void);
clock_seconds_t clock_get(void);
```

```
#endif  // #ifdef CLOCK_H_
```

Listing 15: 'clock.c'

```
#include "clock.h"

#include <stdbool.h>
#include <avr/interrupt.h>
#include <avr/sleep.h>
#include <avr/wdt.h>
#include <util/atomic.h>


#define ATOMIC ATOMIC_BLOCK(ATOMIC_FORCEON)


typedef clock_seconds_t time_t;
time_t g_time_sec;


ISR(WDT_vect)
{
    clock_init();
    ++g_time_sec;
}


void go_to_sleep(void)
{
    set_sleep_mode(SLEEP_MODE_IDLE);
    cli();
    sleep_enable();
    sei();
    sleep_cpu();
    sleep_disable();
}


void clock_init(void)
{
    wdt_reset();
    ATOMIC
    {
        wdt_enable(WDTO_1S);
        WDTCSR |= (1<<WDCE);
        WDTCSR = (1<<WDIE) | (1<<WDP2) | (1<<WDP1);   //
            Interrupt mode, 1 second cycle.
```

```
    }
}


// Sleep until a watchdog interrupt is executed.
// If any other interrupt weakes us up, we know it by
   g_time_sec.
void clock_sleep_until_next_second(void)
{
    time_t before, now;
    ATOMIC{ before = g_time_sec; }
    do
    {
        go_to_sleep();
        ATOMIC{ now = g_time_sec; }
    } while( before == now );
}


clock_seconds_t clock_get()
{
    time_t now;
    ATOMIC{ now = g_time_sec;  }
    return now;
}
```

Listing 16: 'commands.h'

```
// Used hardware: none.
// Used interrupts: none.

#ifndef _COMMANDS_H_
#define _COMMANDS_H_


#include <stdbool.h>


#define MAX_CMD_LEN 32                              // In bytes,
   includeing whitespaces and parameters.
bool listen_for_command(char cmd_buff[], int *bytes);
int execute_command(char cmd_buff[], int bytes);


#endif  // ifndef _COMMANDS_H_
```

Listing 17: 'commands.c'

```
#include "commands.h"
#include "config.h"
#include "pid_tune.h"
#include "strings.h"
#include "usart.h"
#include "zcd.h"

#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <util/delay.h>


// Quieck hack for assert() to work.
#define abort() printf("CRITICAL ERROR")


// Function declarations.
void cmd_help(char*, int);
void cmd_reprogram(char*, int);
void cmd_pid_coeffs(char*, int);
void cmd_pid_setpoint(char*, int);
void cmd_zcd_set(char*, int);


// List of user commands.
typedef void (* fn_ptr_t) (char *cmdline, int bytes);
typedef struct command
{
    const fn_ptr_t handler;
    const char *msg;
    const unsigned len;                         // Without
       the terminating '\0'.
} command_t;
#define DECLARE_COMMAND(name) {cmd_##name, #name, sizeof(#name
   )-1},
const command_t Commands[] =
{
    DECLARE_COMMAND(help)
    DECLARE_COMMAND(reprogram)
    DECLARE_COMMAND(pid_coeffs)
    DECLARE_COMMAND(pid_setpoint)
    DECLARE_COMMAND(zcd_set)
};
```

```
// Command handlers.
void cmd_help(char *cmdline, int bytes)
{
    printf( strings_get(STR_HELP), MAX_CMD_LEN );

    const command_t *it = Commands;
    const command_t *it_end = Commands + (sizeof(Commands) /
       sizeof(command_t));
    for(; it < it_end ; ++it)
    {
        printf("%s, ", it->msg);
    }
    printf(NEWLINE NEWLINE);                          // One blank
       line after output.
}


void cmd_reprogram(char *cmdline, int bytes)
{
    printf( strings_get(STR_JUMPING_TO_BOOTLOADER) );
    _delay_ms(3000);

    typedef void (* fn_ptr_t) (void);
    fn_ptr_t my_ptr = (fn_ptr_t)BOOTLOAD;
    my_ptr();
}


void cmd_pid_coeffs(char *cmdline, int bytes)
{
    // Parse 'pid_coeffs <p> <i> <d>'.
    cmdline[bytes] = '\0';
    char *it = cmdline+sizeof("pid_coeffs");
    int num[3];
    for(int i = 0; i < 3; ++i)
    {
        num[i] = atoi(it);
        it += (num[i] / 10) + 2;                      // Skip all
           digits and a whitespace.
        if(num[i] < 0)  ++it;
    }

    // Unsofrunately, we support only integer gains currently.
```

```
    pid_config( to_pid_coeff(num[0]), to_pid_coeff(num[1]),
       to_pid_coeff(num[2]) );
    printf(NEWLINE NEWLINE);
}


void cmd_pid_setpoint(char *cmdline, int bytes)
{
    cmdline[bytes] = '\0';
    int number = atoi(cmdline+sizeof("pid_setpoint"));
    pid_setpoint(number);
    printf( strings_get(STR_PID_SETPOINT) , number);
}


void cmd_zcd_set(char *cmdline, int bytes)
{
    cmdline[bytes] = '\0';
    int number = atoi(cmdline+sizeof("zcd_run"));
    zcd_set(number);
    printf( strings_get(STR_TRIAC_OUTPUT), number,
       ZCD_PROC_VAL_MAX);
}


// Listens to stdin until a valid command is received.
// Commands have the format:
// !command parameters ENTER
// buff[] must be of size MAX_CMD_LEN
bool listen_for_command(char cmd_buff[], int *bytes)
{
    memset(cmd_buff, 0, MAX_CMD_LEN);
    char c;

    // Wait for a string of the type "!.....\n" and record
        it in a buffer.
    do
    {
        if( !(UCSR0A & (1<<RXC0)) )
        {
            return false;                                  // End of
                input buffer.
        }
        c = getchar();
    } while(c != '!');
```

```
    for(int i = 0;; ++i)
    {
        c = getchar();
        if(c == '\r' || c == '\n')
        {
            *bytes = i + 1;                         // Without
               leading '!'.
            return true;
        }
        else
        {
            assert(i < MAX_CMD_LEN);
            cmd_buff[i] = c;
        }
    }
}


// buff[] must be of size MAX_CMD_LEN
// Returns 0 if command handler was called and non-zero in
   case of error.
int execute_command(char cmd_buff[], int bytes)
{
    // Try to match each known command against the buffer.
    // Stop on first match.
    // Call command handler and return.
    // A more efficient algorithms would be: https://en.
       wikipedia.org/wiki/Radix_tree
    for(int i = 0; i < sizeof(Commands) / sizeof(command_t);
      ++i)
    {
        for(int j = 0; j < Commands[i].len; ++j)
        {
            char c = cmd_buff[j];
            if(c != Commands[i].msg[j]) break;   // Continue
               to test next command.
            if(j == Commands[i].len - 1)         // j is
               counted from 0, while len is counter from 1.
            {
                Commands[i].handler(cmd_buff, bytes);
                return 0;
            }
        }
    }
    return -1;                                          // No command
        matched.
```

```
}
```

Listing 18: 'error.h'

```
#ifndef ERROR_H_
#define ERROR_H_


#include "usart.h"


enum tempr_errors_e
{
        SUCCESS = 0,
        ERROR_NO_DEVICE_FOUND = 1,
        ERROR_START_MEASUREMENT,
        ERROR_READ_TEMPERATURE,
};


// #define NDEBUG  // Uncomment for release build.

// Error checking macro. If any function invocation fails, the
    program halts.
// Foo is the function to be evaluated safely. If the return
   value is nonzero,
// error message is logged and the program stalls.
// The while(0) part is there to make sure the semicolon after
    the macro always means the same
// thing in cases of
//if(a)
//  MACRO;
//else
//  somethingElse()
#define CHECK_FOR_ERRORS(foo)
                                                              \
    do

       \
    {

       \
        int ret = foo;

           \
        if(ret != 0)
```

```
            \
        {
            \
             handle_error_combined(ret, __FILE__, __LINE__,
                __func__);      \
        }
            \
    } while(0)

// Crash the program. A program path, leading to critical
   error occured.
#define CRASH_CRITICAL_ERROR() handle_error_extended(__FILE__,
    __LINE__, __func__);

// Custom runtime assert statement.
#ifdef NDEBUG
#define ASSERT(expr)
#else
#define ASSERT(expr) (void)((expr) || (handle_error_extended(
   __FILE__, __LINE__, __func__), 0))
#endif

// Compile-time assert with message.
// Use like: STATIC_ASSERT(exp1 > exp2,
   error_message_must_be_a_token);
#define STATIC_ASSERT4(COND,MSG) typedef char MSG[(!!(COND))
   *2-1]
#define STATIC_ASSERT3(X,M,L)  STATIC_ASSERT4(X,
   static_assertion_at_line_##L##__##M)
#define STATIC_ASSERT2(X,M,L)  STATIC_ASSERT3(X,M,L)
#define STATIC_ASSERT(X,M)     STATIC_ASSERT2(X,M,__LINE__)


typedef int8_t error_t;


inline void handle_error(error_t err)
{
    printf("Error %i" NEWLINE, err);
}


inline void handle_error_extended(char *file, int line, char *
   func)
```

```
{
    printf("assert() failed in %s:%i at %s()." NEWLINE, file,
       line, func);
}


inline void handle_error_combined(error_t err, char *file, int
    line, char *func)
{
    printf("%s returned error %i at %s:%i." NEWLINE, func, err
       , file, line);
}


#endif  // defined(ERROR_H_)
```

Listing 19: 'strings.h'

```
#ifndef STRINGS_H_
#define STRINGS_H_


typedef enum
{
    STR_HELP                    = 0,
    STR_PROGRAM_START           = 1,
    STR_UNKNOWN_COMMAND         = 2,
    STR_TRIAC_CALIBRATION       = 3,
    STR_JUMPING_TO_BOOTLOADER   = 4,
    STR_PID_SETPOINT            = 5,
    STR_PID_CALIBRATION         = 6,
    STR_TRIAC_OUTPUT            = 7,
    STR_PROCESS_UNSTABLE        = 8,
    STR_COUNT_TOTAL__
} strings_e;

// Returns an SRAM buffer with the required string.
// The buffer is guaranteed to exist unaltered unti
// a new call to strings_get().
const char* strings_get(strings_e);


#endif  // defined(STRINGS_H_)
```

Listing 20: 'strings.c'

```
#include "strings.h"
#include "usart.h"
```

```
#include <avr/pgmspace.h>


#define MAX_STR_LEN (80)


// Holds a copy of the last string requested.
char g_strings_buff[MAX_STR_LEN];

const char string_1[] PROGMEM =
    "!command parameters ENTER" NEWLINE
    "Do not exceed %u characters." NEWLINE;

const char string_2[] PROGMEM =
    "Program start!" NEWLINE;

const char string_3[] PROGMEM =
    "Unknown command: %s." NEWLINE;

const char string_4[] PROGMEM =
    "Running triac with calibration of %li us." NEWLINE;

const char string_5[] PROGMEM =
    "Jumping to bootloader in 3 seconds." NEWLINE;

const char string_6[] =
    "PID setpoint is now %u decicelsius." NEWLINE NEWLINE;

const char string_7[] =
    "Configuring pid with p=%i, i=%i, d=%i in 9s6 format."
      NEWLINE;

const char string_8[] =
    "Triac output configuread at %u from %u." NEWLINE NEWLINE;

const char string_9[] =
    "Process unstable!!!" NEWLINE;


PGM_P const g_strings_table[STR_COUNT_TOTAL__] PROGMEM =
{
    string_1,
    string_2,
    string_3,
    string_4,
```

```
    string_5,
    string_6,
    string_7,
    string_8,
    string_9,
};

const char* strings_get(strings_e n)
{
    strcpy_PF( g_strings_buff, pgm_read_word(&(g_strings_table
       [n])) );
    return g_strings_buff;
}
```

Listing 21: 'tempr.h'

```
#ifndef TEMPR_H_
#define TEMPR_H_


#include <stdint.h>


typedef int16_t decicelsius_t;


void tempr_init(void);
void tempr_measure(void);
decicelsius_t tempr_get(void);


#endif  // defined(TEMPR_H_)
```

Listing 22: 'tempr.c'

```
#include "error.h"
#include "tempr.h"

#include "../libs/onewire/onewire.h"
#include "../libs/ds18x20/ds18x20.h"

#include <util/delay.h>


#define MAX_INIT_RETRIES (20)
```

```
// Because the output waveform is computed synchronously with
   the mains zero-crossing,
// but temperature measurement and pid calculation happen
   synchronously with the cpu,
// there is a need to synchronise between those processes.
// This variable stores the latest measured value of the
   temperature.
static decicelsius_t g_temperature;


// Start temperature measurement.
// Before reading, we need to wait at least
   DS18B20_TCONV_12BIT == 750ms.
void start_meas(void)
{
    error_t err = DS18X20_start_meas(DS18X20_POWER_EXTERN,
       NULL);
    if(err != DS18X20_OK)  handle_error(
       ERROR_START_MEASUREMENT);
}


// Read temperature measurement.
void read_tempr(void)
{
    decicelsius_t temperature;
    error_t err = DS18X20_read_decicelsius_single(
       DS18S20_FAMILY_CODE, &temperature);
    if(err != DS18X20_OK)  handle_error(ERROR_READ_TEMPERATURE
       );
    g_temperature = temperature;
}


void tempr_init(void)
{
    // Initialize the temperature sensor.
    uint8_t id;
    error_t err;
    for(int i = 0, err = 0;
        i < MAX_INIT_RETRIES && err == OW_PRESENCE_ERR;
       ++i)
    {
        err = ow_rom_search(OW_SEARCH_FIRST, &id);
    };
    if(err == OW_PRESENCE_ERR || err == OW_DATA_ERR)
```

```
    {
        handle_error(ERROR_NO_DEVICE_FOUND);
    }

    start_meas();
    _delay_ms( DS18B20_TCONV_12BIT );
}


void tempr_measure(void)
{
    // We assume more than 750ms have elapsed since
       measurement was started.
    // We read to the global variable, then start the next
       measurement.
    read_tempr();
    start_meas();
}


decicelsius_t tempr_get(void)
{
    return g_temperature;
}
```

Listing 23: 'usart.h'

```
#include "usart.h"


int put_char(char c, FILE *out)
{
    usart_transmit(c);
    return 0;                                    //SUCCESS
}


int get_char(FILE *in)
{
    unsigned char c;
    c = usart_receive();
    return (int)c;                               //SUCCESS
}
```

Listing 24: 'usart.c'

```
// Call usart_init() to make stdin and stdout point to the
   debug usart.
```

```
// From then on printf(), getchar() etc. functions read/write
   to the usart.
// Use constant NEWLINE for cross-patform compatability.
// Used hardware: TX and RX pins, USART0.
// Used interrupts: none.

#ifndef _USART_H_
#define _USART_H_

#include "usart.h"
#include "config.h"

#include <avr/io.h>
#include <util/setbaud.h>                          // F_CPU and
   BAUD defined in config.h
#include <stdint.h>
#include <stdio.h>

#define NEWLINE "\r\n"
enum
{
    ASCII_SOH  = 0x01,
    ASCII_EOT  = 0x04,
    ASCII_ACK  = 0x06,
    ASCII_NACK = 0x15,
};


int put_char(char c, FILE *out);
int get_char(FILE *in);
inline void usart_init(void)
{
    UBRR0H = UBRRH_VALUE;                          //Set baud
       rate.
    UBRR0L = UBRRL_VALUE;
#if USE_2X                                         // We are
   operating in asychronous mode: we need this consideration.
    UCSR0A |= (1 << U2X0);
#else
    UCSR0A &= ~(1 << U2X0);
#endif
    UCSR0B = (1<<RXEN0)|(1<<TXEN0);                // Enable
       receiver and transmitter
    UCSR0C = (3<<UCSZ00);                          // Set frame
       format: 8 data bits, 1 stop bit, no parity aka 8N1
```

```
    // Make stdin, stdout and stderr point to the debug usart.
    static FILE fileInOutErr = FDEV_SETUP_STREAM(put_char,
       get_char, _FDEV_SETUP_RW);
    stdin = &fileInOutErr;
    stdout = &fileInOutErr;
}


inline void usart_transmit( unsigned char data )
{
    while ( !( UCSR0A & (1<<UDRE0)) );            // Wait for
       empty transmit buffer
    UDR0 = data;                                  // Put data
       into buffer, sends the data
}


inline char usart_receive(void)
{
    while ( !(UCSR0A & (1<<RXC0)) );              // Wait for
       data to be received.

/*      if(UCSR0A & (1 << FE0))
                return ERROR_USART_FRAME_ERROR;
        if(UCSR0A & (1 << DOR0))
                return ERROR_USART_DATA_OVERRUN_ERROR;
        if(UCSR0A & (1 << UPE0))
                return ERROR_USART_PARITY_ERROR;
*/
    char c = UDR0;                                // Get and
       return received data from buffer.
    return c;
}


inline void usart_flush( void )
{
    unsigned char dummy;
    (void) dummy;
    while ( UCSR0A & (1<<RXC0) ) dummy = UDR0;
}


#endif //#ifndef _USART_H_
```

Listing 25: 'zcd.h'

```
// Used hardware: ICP1 pin. timer1.
// Used interrupts: TIMER1_CAPT_vect, TIMER1_COMPA_vect.

#ifndef _ZCD_H_
#define _ZCD_H_


#include <stdint.h>


#define ZCD_PROC_VAL_MAX (UINT16_MAX)


typedef uint16_t zcd_time_t;
typedef uint16_t zcd_proc_val_t;

// Interrupts are enabled upon return.
void zcd_init(void);

// Values are in range [0, 2^16] and map to [0, 100]% .
void zcd_set(zcd_proc_val_t setpoint);

// Get current setpoint in [0, 2^16] range.
zcd_proc_val_t zcd_get(void);

#endif  // ifndef _ZCD_H_
```

Listing 26: 'zcd.c'

```
#include "config.h"
#include "strings.h"
#include "usart.h"
#include "zcd.h"

#include <stdbool.h>
#include <stdint.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/atomic.h>


#define PROC_VAL_MAX (ZCD_PROC_VAL_MAX)
#define ATOMIC ATOMIC_BLOCK(ATOMIC_FORCEON)


typedef zcd_time_t time_t;
```

```
typedef zcd_proc_val_t proc_val_t;


static volatile bool g_event_captured = false;     // TODO:
   maybe merge with g_rising_detected
static volatile bool g_calibration_mode = false;
static time_t g_calibration;
static proc_val_t g_setpoint;

// This is a horrible hack.
// For some reason, when reprogramming TIMSK1 inside an
   interrupt,
// even if the timer is stopped, triggers an immediate CTC
   interrupt.
// So we enable the interrupts in public api, and then use
   this variable
// to select the correct CTC interrupt.
static volatile bool g_rising_detected = false;


zcd_time_t zcd_calibrate(void);
void zcd_run(zcd_time_t calibration);

// Get current setpoint in [0, 2^16] range.
zcd_proc_val_t zcd_get(void);
bool should_turn_on(void);
void start_timer1_ctc(time_t max);
void start_timer1_capture_rising(void);


// We use the rising edge interrupt both in calibration and in
    live run.
ISR(TIMER1_CAPT_vect)
{
    if(g_calibration_mode)
    {
        g_event_captured = true;
    }
    else
    {
        // Make sure the triac control is off before the zc
        // to avoid accidentally enabling it for the next half
           -wave.
        // Detect the true zero crossing and make a decision
           for the next half-wave.
        PORT_HEATER &= ~(1<<PIN_HEATER);
```

```
        g_rising_detected = true;
        start_timer1_ctc(g_calibration);          // Will call
           TIMER1_COMPA_vect() at next true ZC.
    }
}


// We use the CTC interrupt only in live run mode.
ISR(TIMER1_COMPA_vect)
{
    if(!g_rising_detected)
    {
        return;
    }
    g_rising_detected = false;
    if(should_turn_on())
    {
        PORT_HEATER |= (1<<PIN_HEATER);
    }
    else
    {
        PORT_HEATER &= ~(1<<PIN_HEATER);
    }
    start_timer1_capture_rising();
}


inline void start_timer1_capture_rising(void)
{
    TCCR1B = (1<<ICNC1) | (1<<ICES1) | (1<<CS10);  // Enable
       noise canceler, interrupt on rising edge, clock
       prescaler == 1.
}


inline void start_timer1_capture_falling(void)
{
    TCCR1B = (1<<ICNC1) | (1<<CS10);              // Enable
       noise canceler, interrupt on falling edge, clock
       prescaler == 1.
}


void start_timer1_ctc(time_t max)
{
    TCNT1 = 0;
```

```
    OCR1A = max;
    TCCR1B = (1<<WGM12) | (1<<CS10);              // CTC mode,
       clock prescaler == 1.
}


void stop_timer1(void)
{
    TCCR1B = 0;                                   // Stop clock
       .
    TIMSK1 = 0;                                   // Disable
       interrupts.
}


// Decide if this particular half-wave should be turned on.
   Further discussion:
// http://programmers.stackexchange.com/questions/304546/
   algorithm-to-express-an-integer-as-a-sum-of-some-binary-
   numbers
// The function simulates an uint16_t overflow.
bool should_turn_on()
{
    static uint32_t acu = 0;
    acu += g_setpoint;
    if(acu > PROC_VAL_MAX)
    {
        acu &= PROC_VAL_MAX;
        return true;
    }
    else
    {
        return false;
    }
}


// Initialize the triac control.
void zcd_init(void)
{
    zcd_time_t zcd_calibration = zcd_calibrate();
    zcd_set(0);
    zcd_run(zcd_calibration);
    printf( strings_get(STR_TRIAC_CALIBRATION),
       zcd_calibration / (F_CPU / 1000000) );
}
```

```
// Measure the offset from the rising edge of the ZCD to the
   true zero crossing.
// An oscilloscope trace of the ZCD pcb, used in the project
   can be found here:
// http://electronics.stackexchange.com/questions/201605/
   automatic-calibration-of-a-zero-cross-detector-latency
// It is evident, that the pulse is wide, but centered around
   the ZC.
// Therefore, we measure its width and divide it by two to get
    the offset from the rising edge to the true ZC.
time_t zcd_calibrate(void)
{
    // Restore interrupt status after exit.
    // Enable all interrupts here, because there were glitches
        when altering TIMSK1 form an ISR.
    char sreg = SREG;
    sei();
    TIMSK1 = (1<<ICIE1);                          // Enable
       edge interrupts.

    // Wait for rising edge of the zcd pulse.
    g_calibration_mode = true;
    start_timer1_capture_rising();
    while(!g_event_captured);
    g_event_captured = false;
    ATOMIC{ TCNT1 = 0; }   // Clear timer1.

    // Measure pulse width.
    ATOMIC{ start_timer1_capture_falling(); }
    while(!g_event_captured);
    volatile time_t count;
    ATOMIC{ count = ICR1; } // Width of the zcd pulse, in CPU
       cycles.
    stop_timer1();

    // Estimate offset from measured to real zc.
    time_t rising_to_zc = count / 2;              // In CPU
       cycles.

    SREG = sreg;
    return rising_to_zc;
}
```

```
void zcd_set(proc_val_t setpoint)
{
    ATOMIC_BLOCK(ATOMIC_RESTORESTATE)
    {
        g_setpoint = setpoint;
    }
}


// Wait for rising edge (input capture mode).
// Then wait for half a pulse width duration (ctc mode).
// Then set or clear the triac triger pin.
// Repeat indeffinetely.
void zcd_run(time_t calibration)
{
    sei();
    TIMSK1 = (1<<ICIE1) | (1<<OCIE1A);            // Enable
       level and CTC interruprs.
    DDR_HEATER |= (1<<PIN_HEATER);

    g_calibration_mode = false;
    ATOMIC{ g_calibration = calibration; }
    start_timer1_capture_rising();
}


zcd_proc_val_t zcd_get(void)
{
    zcd_proc_val_t ret;
    ATOMIC { ret = g_setpoint; }
    return ret;
}
```