NTNU
Innovation and Creativity

Microcontroller based fluid flow and image control system for biochemical analysis

Marius Bjørnstad Kotsbak

Master of Science in Engineering Cybernetics
Submission date: June 2007
Supervisor: Amund Skavhaug, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem Description

An image chip is going to be connected to a microcontroller to be able to capture the image and transfer it to a PC. This includes controlling the camera settings through the I2C protocol defined by Phillips and fast capture of the image data being transferred on a parallel bus. The receiving of data from the image chip must at least be 1 Megapixels/sec. but should be around 20 Megapixels/sec to make the image capture as fast as wanted for the analysis process. Light for the pictures is supposed to be supplied by a multicolor LED. There is already an image chip available with a circuit board to be used in this project although it might be changed to a better one later.

A reaction chamber is supposed to be temperature controlled. Heating is done by resistors outside the chamber. Temperature is measured around the resistors, but it is the inside temperature that is supposed to be controlled, but lacks measuring and thus needs to be estimated. Given temperature should be reached fast with little overshoot, as lowering the temperature is only possible by waiting for the chamber to cool down to the environment. Placing the system inside a refrigerator makes this easier. Overshoot might be eliminated by adding more of the cold fluid into the chamber but it is is undesirable.

Specific amounts of different fluids is to be added to the reaction chamber and then removed by adding water instead. It should be possible to control and know exactly how much of the chemicals that are added and how much is left in the storage.

Picture data and control should be possible to transfer through a USB connection between the microcontroller and a PC. To be able to transfer the pictures captured fast enough, at least 100 Mbits/s is required so that USB 1.1 is not fast enough, but at least USB 2.0 is required, or alternatively firewire or fast ethernet if available.

Possibility to easily create a sequence of commands, like specifying a temperature change, pause, open / close valves and picture capture, from a PC using simple graphical or text based programming.

Assignment given: 15. January 2007
Supervisor: Amund Skavhaug, ITK

Preface

This Master's thesis is written during spring 2007 at Department of Engineering Cybernetics at the Norwegian University of Science and Technology (NTNU). It is the final work for my master's degree in the specialisation Industrial Computer Systems.

This assignment was proposed by my father, Jarle Kotsbak, that has started a company Kotsbak Consulting to solve some parts needed for for development of a biochemical analysis instrument and make possible doing experiments that tests the ideas it is based on. Most of the work has been done in Kotsbak Consulting's office in Trondheim.using the company's equipment. Amund Skavhaug at Department of Engineering Cybernetics has been my supervisor at NTNU.

I would like to thank Jarle for this interesting assignment where I have been able to test a lot of equipment and work on new technology, his help with the electrical parts and useful discussions during the work. My supervisor, Amund Skavhaug, I would like to thank for taking the time to answer my questions and giving me tips when I was stuck with technical problems even when he was busy in his spare time. The people at support and development at Atmel I would thank for taking the time to help me solving my problems in more or less unreleased parts of their products.

Summary

Jarle Kotsbak started started the company Kotsbak Consulting to test an idea he had of a method of sequencing DNA material, that is reading the contents of the string of genes in a DNA. The method is supposed to be much faster and cheaper than the methods that exist today. For the analysis equipment there are a lot of parts that need to be developed, and the most important ones in the first phases of experimentation is given as problems for this thesis.

The analysis is going to take place in a relatively small reaction chamber where chemicals need to be automatically added and heated to the correct temperature. Then a high resolution image is going to be captured using a small image chip similar to those found inside digital cameras. As these have varying real time requirements, it is not possible to control from a ordinary PC. A microcontroller is needed to do the low level controls. Control of the experiments should be possible to set up from a PC connected to the equipment using a USB (Universal Serial Port) 2.0 port, which is also going to transfer the images captured to the PC for image and data analysis.

Atmel AVR32 microcontroller was suggested for use, and it was not found that any of the requirements was not in theory possible to solve using this microcontroller, so it was the one used in all parts. It was evaluated if it was best to program the microcontroller directly without an operating system, but it was found that using Linux, which the evaluation boards of the microcontroller was shipped with, had so many advantages over the small and uncertain advantages of programming the microcontroller directly. so Linux was used in all parts. As most of the needed drivers for Linux were new or unreleased, a lot of time was spent learning how to compile the Linux kernel with those patches and finding out how to use the undocumented parts of the drivers, as well as extending the drivers to support the hardware and specification needed.

The heating controller worked quite well after getting the connection to the sensors working, including estimation of the temperature inside the reaction chamber, which is the temperature that is going to be controlled, but can not be measured directly. If needed it is possible to tune the controller better for stricter requirements or change to a model based controller if needed and the parameters in the model can be found, which turned out to be hard.

A nice solution to the flow control was found using a stepping motor that is able to push the piston in a syringe containing one of the chemical mixes in exact amounts, controlled by an external stepping motor driver chip, which was easy to get working, connected using I2C. The solution was at least as good and much easier to implement than the one suggested in the assignment, using valves and pressure measurements to estimate the amount flowing.

It was shown that image capture of the required resolution, i.e. 2048 x 1536 and possibility for much higher, was possible using Linux and the hardware support for transferring images from image chips in the AVR32 microcontroller using kernel modules. The problem is that it was only working sometimes, so a little more debugging of the possible hardware/ software reason for this to be resolved, but it seems like it is only a small problem and not caused by any limitation in software or hardware.

It was attempted to get the USB part working without running an O/S on the microcontroller, but it did not succeed, partly because of incomplete documentation of the USB hardware module in the AVR32. Then the available driver systems for USB slave equipment under Linux was studied and it was found that gadgetfs seemed most promising to use. The driver did though first not load, just giving a stacktrace. In a newer version of the Linux version for the development board, it did though load and it seemed like it was working. Further development of this part was not done because of much delay in the other parts, and the fact that choosing Linux as platform, as well as the image being buffered in RAM, allows to use ethernet to transport the images during testing until later product development stages.

Overall most of the parts have been accomplished according to the specifications, although there was not too detailed practical testing of each part because of much time spent on solving problems, and the fact that many complex parts were outlined to be solved during the project. Solving the small problems left as described at the corresponding discussion chapters should, with small modifications/customization, make the practical solutions found usable for future prototypes of the product or for more experiments.

Contents

Chapter 1

Introduction

1.1 Background / motivation

This project was suggested by the company Kotsbak Consulting to solve some parts needed for a biochemical analysis instrument to do genetical analysis. This method is based on placing the genetical material in a chamber which is filled with some chemicals which makes the genes immobilized which attracts small magnetic beads (Ugelstad beads) which can then be identified using image analysis. A small image camera chip is going to be placed in one end of the chamber to do this. Since the reactions that are going to take place work best at specific temperatures, accurate heating of the fluids is required. The chamber is already designed, as seen in figure 1.2). It has pipes for fluid transport in and out. Normally there is always some fluid inside, and only water after cleaning between the experiments. Heating is going to be done by 3 resistors placed around the inner cube and a temperature sensor in the 4th side and then filled with epoxy through the open hole. The image chip is going to be placed on the side of the inner cube, with the glass in front of it removed, directly in contact with the fluids. An overview of the parts are seen in figure 1.1.

The image analysis, which is out of scope of this project, is probably going to be done on a PC, and the data is needed to be transferred to a PC anyway to do analysis of the registered position of the Ugelstad beads. Transfer of data is preferred to be done through a USB connection, which will also be used to control the experiments from the PC.

The parts made in this project are primarily going to be used to do experiments on the method, but solutions that is impossible to use in a final product and complicated or expensive to replace should be avoided. Some components are probably going to be changed for later experiments, for instance higher resolution of the image chip.

1.2 Problem areas

The following parts are going to be studied in this project. It is listed with the first as highest priority.

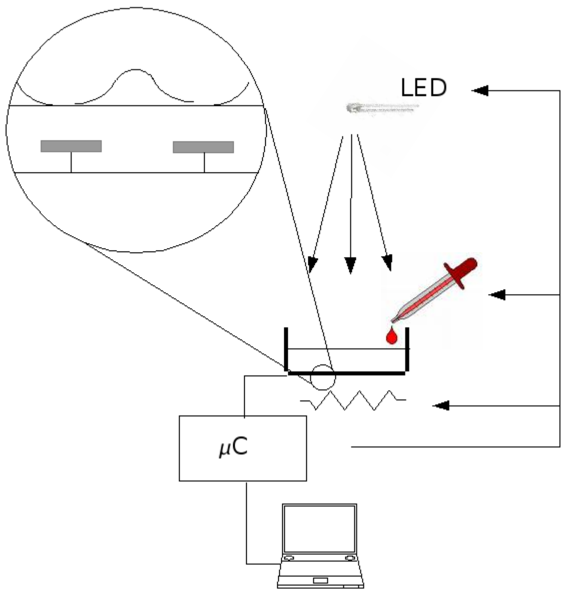

Figure 1.1: Overview of the system

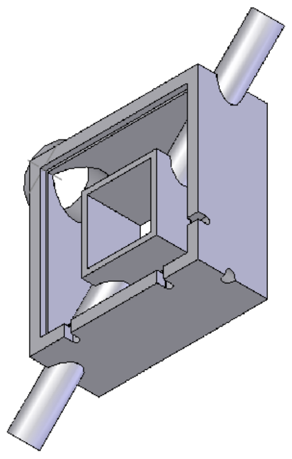

Figure 1.2: The reaction chamber

1.2.1 Communication between image chip and microcontroller

An image chip is going to be connected to a microcontroller to be able to capture the image and transfer it to a PC. This includes controlling the camera settings through the I^2C protocol defined by Phillips and fast capture of the image data being transferred on a parallel bus. The receiving of data from the image chip must at least be 1 Megapixels/sec. but should be around 20 Megapixels/sec to make the image capture as fast as wanted for the analysis process. Light for the pictures is supposed to be supplied by a multicolor LED. There is already an image chip available with a circuit board to be used in this project although it might be changed to a better one later.

1.2.2 Temperature control

A reaction chamber is supposed to be temperature controlled. Heating is done by resistors outside the chamber. Temperature is measured around the resistors, but it is the inside temperature that is supposed to be controlled, but lacks measuring and thus needs to be estimated. Given temperature should be reached fast with little overshoot, as lowering the temperature is only possible by waiting for the chamber to cool down to the environment. Placing the system inside a refrigerator makes this easier. Overshoot might be eliminated by adding more of the cold fluid into the chamber but it is is undesirable.

1.2.3 Fluid flow control

Specific amounts of different fluids is to be added to the reaction chamber and then removed by adding water instead. It should be possible to control and know exactly how much of the chemicals that are added and how much is left in the storage.

1.2.4 USB-communication between microcontroller and PC

Picture data and control should be possible to transfer through a USB connection between the microcontroller and a PC. To be able to transfer the pictures captured fast enough, at least 100 Mbits/s is required so that USB 1.1 is not fast enough, but at least USB 2.0 is required, or alternatively firewire or fast ethernet if available.

1.2.5 Sequence control from PC

Possibility to easily create a sequence of commands, like specifying a temperature change, pause, open / close valves and picture capture, from a PC using simple graphical or text based programming.

1.3 Chosen part of problem

Parts are going to be solved after the priority specified in section 1.2. When technical problems occurs or while waiting for equipment, other parts can be studied in the meantime. There will not be enough enough time to study all parts in great depth, but solutions should be found and described, and as much as possible tested.

1.4 Parts already solved

Before this project started, I worked for Kotsbak Consulting where I experimented with the use of the image chip interface of the AVR32 using embedded programming of the microcontroller without any operating system. It did not succeed, so then transferring the image data manually without any hardware support using the general I/O lines was attempted. That way it worked, but since the code was bit-banging the transfer and flow control, transfer rate possible without any errors was limited to around 1 MHz, just above the minimum specification of the image chip. Even at that rate some errors were observed. The data was then transferred to a PC using the serial port, which is very slow, requiring about 15 minutes of transfer time per image consisting of 3.2 MegaPixels, which is unacceptable. It was also suspected that the quality of the image is reduced by the slow reading of the image. On exposure of some evenly distributed light, it was observed darker colors on the side where transfer starts, fading from grey to white. That could be caused by the first pixels that was read getting less exposure time than the last pixels read, meaning that the slow reading from the chip to the microcontroller might be unacceptable.

1.5 Limitation of the project

Ordering of equipment, soldering of the circuits, wires etc. is the responsibility of the company, but some of the specifications of the equipment is decided by this work. The intention is not to create a fully working prototype of a product in this project, but show how the different parts can be solved and thus easily be possible to assemble and customize to a prototype of a lab equipment.

1.6 Disposition of report

Since there are several almost independent parts of this project, the report is sectioned after each part which is discussed and with a general conclusion at the end.

Chapter 2

Image chip connection and transfer to PC

2.1 Background

To do image analysis of the placements of the small Ugelstad beads inside the reaction chamber, an image chip is going to be inserted on one side of it. A high resolution is necessary, since the Ugelstad beads are small, around $2,8\mu m$. To avoid potentially slowing down the analysis, the image capture and transfer to a PC for analysis should be quite fast, completed in some seconds. Light for the chamber during the capture is necessary, but is easy to accomplished using a small LED controlled by a digital output line if needed, and is outside the scope of this project.

In addition to some tests using the Atmel AVR32 microcontroller, one from Cypress has also been tried to use for this task at Kotsbak Consulting, with little success because of unknown technical problems. It was based on sending the image data directly from the image chip through a small buffer in the microcontroller and then to the USB line. This assumed that the PC could receive and save the data at the same rate, which might not be the case on a non-realtime O/S normally used on a PC. The AVR32 has some advantages that made it look more promising. It has enough memory to buffer one complete image in RAM, so that it is not required to send it to the PC in real time. There is also a special hardware module made just for connecting image chips using hardware acceleration. Atmel's development section is also situated in Trondheim and many prior NTNU workers or students are now working there which should make the contact in case of problems easier.

2.2 Chosen solution

An image chip from Micron, MT9T001, which has maximum resolution 2048x1548, is going to be used in this project. Kotsbak Consulting already has made a board with the chip and support components connectable via a flat cable. See diagram for this board in figure 2.1. It should be possible to exchange it with an image chip with higher resolution later.

The image chip is controlled using the I^2C protocol, such as changing the image capture settings, image size and trigger image capture. Image data is transferred using a simple parallel protocol. Ten bits are transferred simultaneously on the data lines, synchronized with a pixel clock from the image chip. There is just one way flow control; the receiver must capture the data as it appears on the bus, and there are no error control in the data

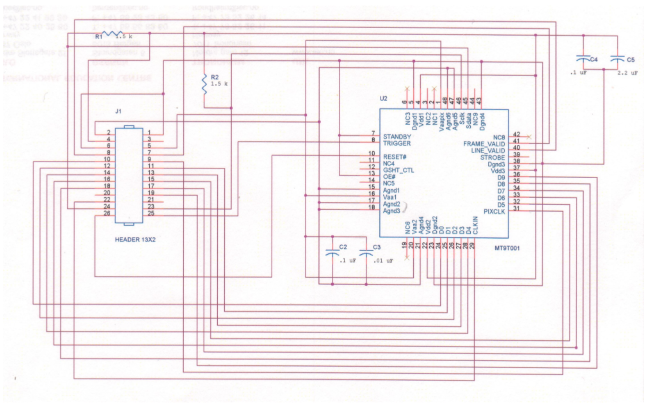

Figure 2.1: Image chip diagram

or redundancy. In addition there are control lines for signalling new lines in the image and start of a new image. You can however control the frequency of the pixel clock between 1 MHz and the master clock sent to the chip by setting a divider register.

The image chip is supposed to be connected to the AVR32 microcontroller from Atmel, using the development board STK1000, which has an image chip interface called ISI (Image Chip Interface) module built into the microcontroller. It supports DMA transfers of image data directly from the image chip to SD-RAM connected to the microcontroller. Before this project started using this programming the AVR32 manually without OS to use the ISI module has been tried but not succeeded.

We then have to choose between embedded programming of the microcontroller directly without an O/S, or using Linux on the AVR32. The main advantage of using embedded programming is the possibility to program real time, optionally using a real time O/S or system like FreeRTOS. Also it might be easier to program controllers for peripherals on the chip manually instead of developing drivers for Linux. That is not true in all cases, as often a hardware independent part of a driver already exists for Linux, like most of the TCP/IP and USB protocols. As reading of image data is accomplished without Linux, it could be a solution to get USB working in order to send the images fast to the PC. But slow reading of the image could be unacceptable, see section 1.4.

Programming for Linux has a lot of advantages. It is more convenient to have a file system where you can save your data and programs and thus have more programs available to run. Your own programs can easily be transferred through ethernet network instead of flashing the internal memory using the AVR JTAG ICE converter. Manual capture of image data using a user space program is probably not possible. Although there exists a /dev GPIO (Genereal purpose I/O) driver, it is probably too slow as an extra layer, since the receiving is just barely working using an optimized native program.

The latest patches for the AVR32 Linux includes driver for I^2C support, which was hard to get working without O/S. The GPIO driver should make it easy to use digital input and outputs. The only thing that can be a important problem by using Linux is the real time requirements as the Linux version supplied is not a real time O/S, although some low latency enhancements in the 2.6.x kernels probably help. As long as no other applications are running, the current process should get most of the CPU time, only disturbed by system tasks and possibly system functions that could have variable response time.

In this project no parts have very strict real time requirements. Temperature control is a quite slow system compared to how large delays can be experienced. Using threads (supported on AVR32 by the pthreads library) enables different tasks to run better simultaneously. Parts that are time critical can if necessary be made as kernel modules, thereby avoiding interruption. Also parts using DMA (direct memory access) and IRQ (interrupts) are sure that they can act in time, which includes USB, ethernet, TWI and the hardware ISI image chip module. The conclusion is that the best in this case is to try to get all the parts running on Linux on AVR32.

Atmel is working on a Video4Linux2 driver for the ISI module. It is a goal to try getting this driver working in the project with the available image chip. That will ensure more error free transfer of image at full speed using hardware support. The image can then be sent to a PC for processing through USB, ethernet network or SD card, which all work much faster than through a serial port. When running under Linux, drivers are quite easily available and easy to use for such equipment.

At the start of the project, in the beginning of January 2007, Atmel supplied an incomplete ISI driver they were working on. The linux kernel needed to be recompiled after applying their ISI patch, which was not that easy. After building the kernel and installing it on the board, the driver seemed to load successfully, but it was no clock from the AVR32 to the image chip and no video device created. In end of March 2007, a new patch of the ISI driver was sent from Atmel, which looked more comprehensive and more promising by looking at the source code.

This new version consists of a general driver for the ISI module called "atmel-isi", which does the configuration of the ISI settings, and follows the video4linux2 standard, and a camera driver initializing the actual image chip and triggering capture. Driver for the Micron image chip MT9V011 was supplied, which was used as basis for developing a driver for our MT9T001 image chip. Many of the registers were similar or equal so it was quite easy to make this new module, "mt9t001".

After the first experiments (see section 2.3), it was clear that it was too little RAM on the STK1000/STK1002 board (8 MB). It is enough for the default max resolution in the driver (320 x 240), using 4 bytes pr. pixels occupying approx. 307 KB. With the maximum resolution of the image chip that I am using, 2048 x 1536, required RAM for the buffer is around 12,5 MB for full colors (4 bytes pr. pixel), and 6,3 MB for grayscale/raw data (2 bytes pr. pixel) from the image chip. We need the latter, but that leaves less than 2 MB for Linux to use, which seems too little, as it is using around 4 MB right after boot. As it is required at a later stage to use a 8 mega pixels image chip, that will require 8 MB * 2 = 16 MB RAM, which makes the STK1000/1002 developer board unusable.

It was confirmed by Atmel that it was possible to exchange the memory module, but it is surface soldered on the board, thus not that easy to replace. Another solution might be to use swap partition mounted on NFS, which is supported. As long as the network is fast and the PC the mounting is done from has enough memory to cache the swap file in RAM, it should be quite fast, but probably considerably slower than on board RAM. It also requires a dirty trick to free the memory to get enough free contiguous memory, allocating large blocks, forcing used memory to be moved to the swap, and then free them and hope that the memory left is usable. For later usage in a product it is also not able to depend on ethernet, but connect the equipment to a PC only through USB, and not requiring NFS server on the connected PC.

During the project another development board for the AVR32 microcontroller was released, ATNGW100 "Network Gateway Kit". Though first seen as a tool to experiment with networking applications, after more detailed study it was found that it has all the same equipment that this project uses on the STK1000 kit, but without screen, sound connector etc. It also has 32 MB SD-RAM included on the board, instead of the 8 MB on the STK1000. That makes it much better for connecting to image chips, as it also has enough

RAM for a 8 Mega pixel camera, leaving 16 MB. for Linux and applications. See a picture of the board connected to the image chip board in figure 2.2. The board is also booting and mounting all file systems from one 8 MB serial (slower, used for /usr filesystem) flash and one parallell flash (faster used for the root filesystem) both soldered on the board, avoiding to use a SD-CARD for normal use as long as the space available is sufficient. NFS (Network file system) mounting was used for easy transferring of program and data to and from a PC. It is also nice that they have replaced the PCI style connector with 3 ordinary connectors used on flat cables, making connection of the image chip easier and safer and also gives more general I/O lines easily connectable. An extra bonus is that this board is smaller and has much lower costs. It might even be possible to use in a final product.

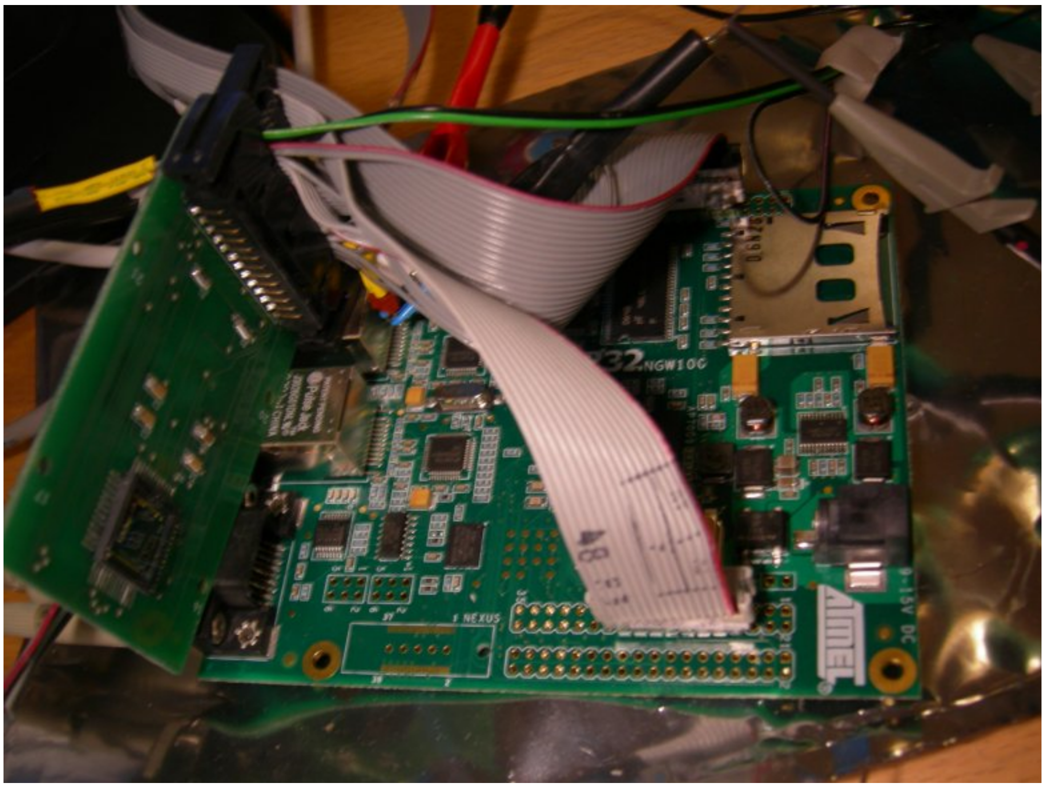

Figure 2.2: NGW100 board connected to the image chip board

2.3 Testing of solution

The first incomplete driver did not work at all (see above). The second driver supplied by Atmel had problems loading. The "atmel-isi" module complained by problems allocating enough buffer memory for the image saving. The problem is that this memory has to be contiguous in physical memory, as the ISI hardware module skips the memory manager, writing directly to the physical memory. After Linux on STK1000 board has loaded, there is not even enough free memory for the quite low default max. resolution 320 x 240. Trying to stop ftp, www, ssh etc. servers on the board also did not help, as it probably just created fragmented free memory. Disabling it from the boot sequence did help. Then the module could load, but almost nothing appeared in the syslog, it did not print anything of the debug information found in the driver code. Compiling the kernel with "debug filesystem" ("debugfs") in addition to enabling more debug information for Video4Linux and device drivers in general was done. The driver is using debugfs for outputting debug information in a directory structure (typically mounted at /debug), but nothing appeared there. Also the extra debug information was not very useful, just showing some more general debug messages when the module was loaded.

This part was postponed a bit, and some support requests regarding this problem sent to Atmel. When the NGW100 development board was delivered, TWI (I^2C) was first tried again to see if it was also working on this board, as it is required for both the image chip communication and the other parts. It was not working at all, giving less output in the system log when loading the module (i2c-atmeltwi.ko) compared to the same loading on STK1000. A support request to Atmel solved the problem. The reason it did not work was that Atmel had removed TWI support on the Linux build for NGW100, because it was meant to be used for network and I/O applications. Although the driver was there, the architecture specific initialization code at boot for TWI was removed. The following line was then re-added to to arch/avr32/boards/atngw/setup.c (in the function atngw_init(void)):

at32_add_device_twi(0);

That line adds initialization of the internal routing to the TWI controller in the AVR32, needed for it to work. When comparing to the same initialization for STK1000, it was found that the ISI patch added the ISI hardware routing initialization in some STK1002 specific file which is not used on NGW100, so at the same place this line was also added:

at32_add_device_isi(0);

The kernel was then built again (changing the board type from STK1000 to NGW100), and tried to exchange the included kernel by that one. The result was that it did not boot (maybe some wrong kernel config options). The board has a bit complex procedure for replacing the root filesystem (which includes the kernel), using a SD-card described on the AVR32 wiki on the Internet, though it worked. As I predicted this kind of kernel boot problems to happen again (which turned out to be true), I afterwards found a better solution to try out new build kernels; copy the new kernel to /uImage.new (the original called /uImage), and then instruct u-boot (which is the boot loader used, running from flash included in the microcontroller) during booting in order to boot the new kernel by this command line:

setenv bootcmd 'fsload uImage.new;bootm' boot

If it then does not work, you can just let it boot normally without any options to load the original and then try a new Linux build.

After some trying, I got a kernel that booted and seemed to work. The TWI driver then gave the same output on loading of the module as on the STK100 board, but it did not work with the temperature reading part (section 3.2.1).

Then the ISI modules were tried on that board. They actually gave much more debugging output in the system log than on the STK1000 board, so that you could see how far it was going in the source code. The camera driver failed because of the TWI not working (it tries to detect the camera and configure it though TWI). Those parts were removed so that it should load anyway. The reason it should still be able to capture images is that the default option on the image chip is to continuously capture images and send them on the parallell interface as fast as possible, and the ISI module determines when an image should be captured and enables the ISI module only during that capture and transfer. The /dev/video0 device then appeared, but no data was returned by reading from it.

From the output in the log and the fact that the Video4Linux2 client gave TIMEOUT, it was found that it stopped due to failing to reset the ISI module in this part of the atmel-isi.c code:

```
 /*
* Reset the controller and wait for completion. The
* reset will only succeed if we have a pixel clock
```

```
 * from the camera.
 */
init_completion(&isi->reset_complete);
isi_writel(isi, CR1, ISI_BIT(RST) | ISI_BIT(DIS));
isi_writel(isi, IER, ISI_BIT(SOFTRST));
timeout = wait_for_completion_timeout(&isi->reset_complete,
      msecs_to_jiffies(5000)); // Marius change: longer timeout (was 100)
if (timeout) {
// Marius test: skip this temporary:
        //ret = -ETIMEDOUT;
   // isi->camera = 0; // Marius change: deselect camera
// goto out;
   pr_debug( "Timout but ignoring for testing..." );
}
isi_writel(isi, IDR, ~0UL);
```

Not even changing to a much higher timeout for resetting (5 seconds instead of 100 ms) was of any help. Checking the source code of the wait_for_completion_timeout() function (no documentation found) seems to indicate that the function will return the remaining of the timeout or zero if it timed out waiting for the completion. Thus this check seemed to be the opposite of what it should. Skipping the test lets it continue running and actually completing all the initialization.

The next problem then was that the capture of image was about to start, but no start of frame (SOF) interrupt was received, i.e. no image data was received from the image chip. After checking the pins to the image chip it was discovered that in was a clock to the image chip (master clock), but no pixel clock coming back. As it then was the image chip not doing what it should, I checked its circuit and found that the reset button was not correctly connected, a pull up resistor was missing, causing it to just reset itself.

Fixing that problem actually gave promising debug information in the log:

```
inside isi\_read: starting capture
isi: waiting for SOF
isi: starting capture
isi: finished capture
inside isi\_read: copy count: 0, ret: 0
```

saying it starts capturing, getting start of frame interrupt and end of frame. The only problem was that capturing data using a simple Video4Linux2-client or just by command line “cat /dev/video0 > /tmp/capture.data” gave a zero bytes file (also seen from the debug information). That was caused by the dummy I^2C code in the camera driver returning 0 resolution. After fixing that problem, some data was actually received from the driver, containing data that could be correct. At least something was written to the memory, as it was tried to add an ’A’ in every byte of the image buffer. It was also checked that the ’A’s were not replaced when trying the same test and disabling the image chip in the initialization.

Then a higher resolution was to be tried, as the default resolution of the driver was quite low (320 x 240). Also since the TWI was not working, the unconfigured image chip sends its maximum resolution, only a part of the image will be captured and the ISI module might also fail when it receives more data when it is expecting end of frame (it is not clear from the documentation). Therefore the ISI module should match the default resolution of the image chip until the TWI problem is fixed.

The image chip maximum resolution was tried to set as maximum in the Video4Linux2 driver, but resulted in memory allocation error (as on the STK1000 board). Removing daemons in the startup did not help, only giving the possibility to use a little higher resolution, 1024x768, still too little for the image chip. Following a tip from Atmel support, doing the memory allocation at early boot time like it is done for the frame buffer (for the screen) on STK1000 was tried. That involves inserting code early in the Linux booting before the memory paging system and normal allocation is started, giving the possibility to reserve a contiguous memory block that is not used for normal memory allocation in Linux for system and applications. A boot option “isimem” was added, where you specify how much memory to preallocate and optionally where to place it. I.e. to allocate enough for the image chip the following command was given to u-boot:

```
setenv bootargs console=ttyS0 root=/dev/mtdblock1 rootfstype=jffs2 isimem=6300k
```

which instructs this code added to linux-2.6.18/arch/avr32/kernel/setup.c to parse the kernel option:

```
/*
```

```
 * Early isi buffer allocation. Works as follows:
 *   - If isimem_size is zero, nothing will be allocated or reserved.
 *   - If isimem_start is zero when setup_bootmem() is called,
 *     isimem_size bytes will be allocated from the bootmem allocator.
 *   - If isimem_start is nonzero, an area of size isimem_size will be
 *     reserved at the physical address isimem_start if necessary. If
 *     the area isn't in a memory region known to the kernel, it will
 *     be left alone.
 *
 * Board-specific code may use these variables to set up platform data
 * for the framebuffer driver if isimem_size is nonzero.
 */

static unsigned long __initdata isimem_start;
static unsigned long __initdata isimem_size;

/*
 * "isimem=xxx[kKmM]" allocates the specified amount of boot memory for
 * use as isi buffer.
 *
 * "isimem=xxx[kKmM]@yyy[kKmM]" defines a memory region of size xxx and
 * starting at yyy to be reserved for use as isi buffer.
 *
 * The kernel won't verify that the memory region starting at yyy
 * actually contains usable RAM.
 */
static int __init early_parse_isimem(char *p)
{
  printk( "Parsing isimem kernel parameter...\n" );

isimem_size = memparse(p, &p);
if (*p == '@') {
p++;
isimem_start = memparse(p, &p);
}
return 0;
}
early_param("isimem", early_parse_isimem);
```

and this line added to the function setup_arch() in the same file

```
board_setup_isimem(isimem_start, isimem_size);
```

calls this function in linux-2.6.18/arch/avr32/boards/atngw/setup.c:

```
void __init board_setup_isimem(unsigned long isimem_start,
      unsigned long isimem_size)
{
if (!isimem_size)
return;

if (!isimem_start) {
void *isimem;

isimem = alloc_bootmem_low_pages(isimem_size);
isimem_start = __pa(isimem);
} else {
pg_data_t *pgdat;

for_each_online_pgdat(pgdat) {
if (isimem_start >= pgdat->bdata->node_boot_start
    && isimem_start <= (pgdat->bdata->node_low_pfn
    * PAGE_SIZE))
reserve_bootmem_node(pgdat, isimem_start,
     isimem_size);
}
}

printk("%luKiB isi memory at address 0x%08lx\n",
       isimem_size >> 10, isimem_start);
atmel_isi0_data.isimem_start = isimem_start;
atmel_isi0_data.isimem_size = isimem_size;
}
```

which does the actual allocation easily using the alloc_bootmem_low_pages() function (given that you don't bother where the memory is allocated).

The location of the allocated memory is saved in this struct defined in linux-2.6.18/include/asm/arch-at32ap/board.h:

```
struct atmel_isi_platform_data {
unsigned long isimem_start;
unsigned long isimem_size;
};

struct platform_device *at32_add_device_isi(unsigned int id, struct atmel_isi_pl
```

which is then used as parameter to the function call to at32_add_device_isi() in linux-2.6.18/arch/avr32/boards/atngw/setup.c:

```
 static int __init atngw_init(void)
{
at32_add_system_devices();

at32_add_device_usart(1);

if (eth_data[0].valid)
at32_add_device_eth(0, &eth_data[0]);
if (eth_data[1].valid)
at32_add_device_eth(1, &eth_data[1]);

spi_register_board_info(spi_board_info, ARRAY_SIZE(spi_board_info));

at32_add_device_spi(0);
at32_add_device_mmci(0, &mmci0_data);
at32_add_device_usb(0);

at32_add_device_twi(0);

at32_add_device_isi(0, &atmel_isi0_data );

return 0;
}
```

which then saves it to the platform device in linux-2.6.18/arch/avr32/mach-at32ap/at32ap7000.c:

```
struct platform_device *__init
at32_add_device_isi(unsigned int id, struct atmel_isi_platform_data *data)
{
struct platform_device *pdev;

switch (id) {
case 0:
pdev = &atmel_isi0_device;
select_peripheral(PB(0),  PERIPH_A, 0); /* DATA0  */
select_peripheral(PB(1),  PERIPH_A, 0); /* DATA1  */
select_peripheral(PB(2),  PERIPH_A, 0); /* DATA2  */
select_peripheral(PB(3),  PERIPH_A, 0); /* DATA3  */
select_peripheral(PB(4),  PERIPH_A, 0); /* DATA4  */
select_peripheral(PB(5),  PERIPH_A, 0); /* DATA5  */
select_peripheral(PB(6),  PERIPH_A, 0); /* DATA6  */
select_peripheral(PB(7),  PERIPH_A, 0); /* DATA7  */
select_peripheral(PB(11), PERIPH_B, 0); /* DATA8  */
select_peripheral(PB(12), PERIPH_B, 0); /* DATA9  */
select_peripheral(PB(13), PERIPH_B, 0); /* DATA10 */
select_peripheral(PB(14), PERIPH_B, 0); /* DATA11 */
select_peripheral(PB(8),  PERIPH_A, 0); /* HSYNC  */
select_peripheral(PB(9),  PERIPH_A, 0); /* VSYNC  */
select_peripheral(PB(10), PERIPH_A, 0); /* PCLK   */

/* Master clock for the camera: GCLK0 */
select_peripheral(PA(30), PERIPH_A, 0);
clk_set_parent(&mt9v011_mclk, &pll0);
break;

default:
return NULL;
}

memcpy(pdev->dev.platform_data, data,
       sizeof(struct atmel_isi_platform_data));
platform_device_register(pdev);

return pdev;
}
```

In the same file a macro creates the struct instance “atmel_isi0_device”: “DEFINE_DEV_DATA(atmel_isi, 0);” with the same name as the module “atmel_isi.ko” connecting them together. In the module the following code was added to avr32_isi_probe(struct platform_device *pdev):

```
struct atmel_isi_platform_data *isi_mem_data = pdev->dev.platform_data;

        /* Use platform-supplied framebuffer memory if available */
        if (isi_mem_data && isi_mem_data->isimem_size != 0) {
  printk( "Using preallocated isimem.\n" );

  isi->capture_phys = isi_mem_data->isimem_start;
  isi->capture_buf_size = isi_mem_data->isimem_size;

  isi->capture_buf = ioremap( isi->capture_phys, isi->capture_buf_size );
        } else {
printk( "No preallocated isimem found, trying to allocate ISI buffer now\n" );

        /* Allocate capture buffer */
isi->capture_buf = dma_alloc_coherent(&pdev->dev,
      isi->capture_buf_size,
      &isi->capture_phys,
      GFP_KERNEL);

        }

if (!isi->capture_buf) {
ret = -ENOMEM;
dev_err(&pdev->dev, "failed to allocate capture buffer\n");
goto err_alloc_buf;
}
```

which using the platform device tries to check if buffer memory for ISI is preallocated and in that case use that instead of allocating at module loading.

Booting with this code included and with the new kernel option, one can see using the command “free” that the amount of total memory visible to Linux was decreased by the amount that was specified in the boot option. The system was tested using preallocated memory and seemed to work, but no data was received from the driver, the data from the driver was left at the same value as set by the initialization of the memory. Testing the original method of allocating for lower resolution at module load was also tried again and now without any result, even though it worked before.

It is not easy to find the reason that this seems to work very unstable. Atmel reported some problems in their experiments by noise from long cables between the image chip and the microcontroller. It should be checked with them if it causes the transfer to fail or just gives errors in the transfer. The transfer speed should also be decreased. It was tried to lower the frequency of the master clock on the microcontroller side, which also gives lower pixel clock, without any better result. The problem can be caused by broken cables or noise to the cables. One way to check if there are cable problems is to run again the embedded program that is bit-banging the transfer. The buffer preallocated should also be verified that is not cached.

When TWI is working again it is possible to decrease the speed of the pixel clock, which is done in the driver for the similar image chip Atmel has used for testing ISI. A new NGW100 is ordered to see if the TWI problem could be caused by hardware problems before using too much time on debugging it. It is unlikely, but possible that the same can be the case for ISI that stopped working.

2.4 Discussion

This part has taken a lot of time to solve. In some degree time has been used to figure out how Linux works on the boards, as it was neccesary to patch on the kernel shipped with the board. Also developing Linux kernel and kernel modules takes some time to learn as the documentation is in many parts hard to find, so you are left with examples and source code. It might exist some books about this, but some of this driver systems are so new and quickly changing that nobody has probably written any books about it either. I am not sure what the big blocker problem here was, as the output did not tell, but it might have been a memory allocation or hardware initialization that failed silently. Maybe some more time could have been spent debugging kernel code, but it is not easy.

One could also ask if there are other solutions to solve the same, that is capturing a high resolution image and transferring it to a PC using USB. About the same is happening in an ordinary digital compact camera at the same resolutions. Web cameras for PC are even easier to use as it is already based on USB transfer of the images, but the maximum resolution available is too little. A problem in both of these cases is that it is not easy to modify the equipment to be used in this case, as the chamber is small and it is not much more space left than for the image chip itself. Some camera equipment for this case might be available, but later the image chip also is supposed to be specially made without the glass and without the color filters, as highest possible resolution greyscale is what is needed. Using another microcontroller when the unsolvable problems appeared is no guarantee that would have given a solution faster, as another is already tried (see background about the Cypress microcontroller).

Embedded programming of this could have been worked more on, but then also USB or ethernet would have been required to get working, which is ready for Linux. Developing using Linux also has other advantages, so I think it was a good choice to use Linux.

Chapter 3

Temperature control

3.1 Background

The reaction chamber should be possible to heat up to a predefined temperature which is ideal for the fluids. To allow the whole process to finish as quickly as possible the target temperature should be reached as fast as possible. The only possibilities for lowering the temperature are waiting for heat to disappear or exchange with new fluids at storage temperature, which causes waste of possible expensive chemicals that we should avoid. Therefore temperature should not get much higher than the target to allow fast operation without extra cost of lost chemicals.

One challenge here is that temperature is only going to be measured outside the inner chamber containing the fluid to be temperature controlled. That means that the temperature inside has to be estimated based on the temperatures outside (near the heating resistors), the ambient temperature and the power to the resistors. In the physical model used we placed a small temperature measurement NTC inside the chamber too, to be able to check how well the estimation is working.

The requirements of the regulation are:

- Uncertainty of maximum +/- 0.5 - 1.0 C when temperature stable, which should happen as fast as possible.
- Max overshoot 3.0 K for max 30 seconds
- Setpoint range is between room temperature (around 20 C) and 80 C

A similar demo reaction chamber having the same structure is made to avoid using the expensive real chambers. See a picture of the board with the chamber in figure 3.1. Therefore it is important to show how the regulation is made so it can be done in the same way when moving to the real chamber.

3.1.1 Physical modelling

The demo reaction chamber consists of an outer part containing the resistors with a temperature (which is measured) called T_o and an inner part containing water with a temperature T_i. The heat flow from outer part to inner part is

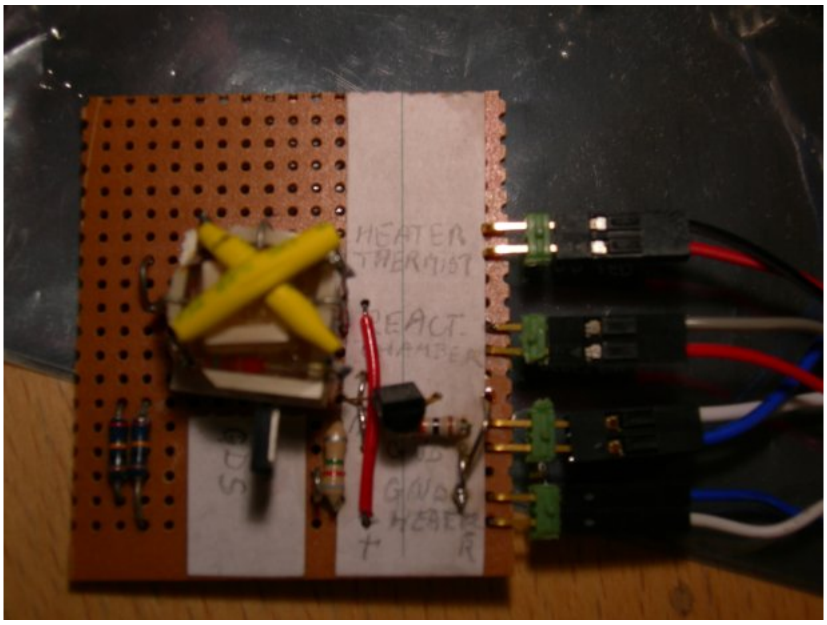

Figure 3.1: Board with the test reaction chamber

$$P = \lambda \frac{A}{h}(T_o - T_i)$$

Change of T_i is given by the heat capacity:

$$P = C_w \Delta T_i$$

Combining these 2 formulas gives:

$$\Delta T_i = \frac{\lambda}{C_w}\frac{A}{h}(T_o - T_i)$$

Most of the formula are constants:

Symbol	Description	Estimated value
λ	Heat conduction	$5,00 * 10^{-3} \frac{W}{Cm*K}$ (Epoxy)
A	Area heat passing	Ca. $1cm^2$
h	Length heat is conducted	Ca. 0.5 cm
C_w	Heat capacity of the inner water	0.4181 J/K (at 20 C)

C_w was calculated using measured 1 ml = 0.1 g water multiplied with the constant for specific heat capacity of water $(0,4181 J/gK)$. There is a lot of uncertainty in these numbers in addition to the model being 3-dimensional requiring to use complex vectorized calculation based on the placement of the resistors etc. to get the correct values. Heat leaking from outer part to the room is not included. As we can see from the formula, the change of inner temperature is proportional to the difference between the outer and inner temperature.

Combining these into unknown constants:

- k_1 - heat flow constant between outer and inner part of chamber.
- k_2 - heat flow constant between environment and inner part of chamber.
- k_3 - heat flow constant between environment and outer part of chamber.

for each heat flow we can construct differential equations for the change of outer and inner temperatures

$$\frac{dT_i}{dt} = k_1(T_o - T_i) - k_2(T_e - T_i) \tag{3.1}$$

$$= k_1 T_o - T_i k_1 - k_2 T_e + T_i k_2$$

and

$$\frac{dT_o}{dt} = \frac{P}{C_o} - k_3(T_o - T_e) - k_i(T_o - T_i)$$

$$= \frac{P}{C_o} - k_3 T_o + T_e k_3 - k_1(T_o - T_i)$$

where T_e is environment temperature and C_o is unknown heat capacity of the outer part of the chamber.

The corresponding system matrixes are then:

$$\begin{bmatrix}\dot{T}_i\\ \dot{T}_o\\ \dot{T}_e\end{bmatrix} = \begin{bmatrix}(k_2-k_1) & k_1 & -k_2\\ k_1 & -k_3-k_1 & k_3\\ 0 & 0 & 0\end{bmatrix}\begin{bmatrix}T_i\\ T_o\\ T_e\end{bmatrix} + \begin{bmatrix}Ca.0^1\\ \frac{1}{C_o}\\ Ca.0^2\end{bmatrix} u$$

Here we have 4 unknown constants which are hard to calculate from the demo model and also hard to measure because they are related and thus hard to isolate each constant.

3.2 Chosen solution

We need to measure the temperature, which is easy by using NTC thermistors and use digital I/O output from the microcontroller to control the heating.

3.2.1 Temperature measurement

The AVR32 microcontroller used has no A/D-converter included, so that has to be added outside the microcontroller. Since I^2C-communication is working with linux, it would be easiest to find one A/D-converter using that. Also multiple channels, at least 4 would be nice to have if more measurements are needed in other parts. A suitable product, AD7993 from "Analog Devices", was found which was fast to acquire. See datasheet in digital attachments in "AD-converter/AD7993_7994.pdf". It uses I^2C and has 4 input channels at 10 bits resolution. As it was surface mountable, connection to a Veroboard was done using a small conversion circuit board. The board was then connected to the microcontroller board by I^2C. The 3 NTC-resistors (environment, inner and outside of chamber) were connected to separate input channels using a voltage divider with a 10 kOhm resistor and the reference voltage at 3,3 V. The circuit itself is operated at 5 Volts found at pin 6 on connector J29 on the the STK1000 board. Ground and 3,3 V is found at pins 3 and 4 on J29 respectively. TWI lines SDA and SCL is connected to pins 25 and 26 of connector J31 respectively. See a picture of this board in figure 3.2.

To convert the input values received from the A/D-converter we first need a relationship between the input value and voltage at the NTC. The input is quantities of $2^{10} = 1024$ (10 bit resolution) of the reference voltage:

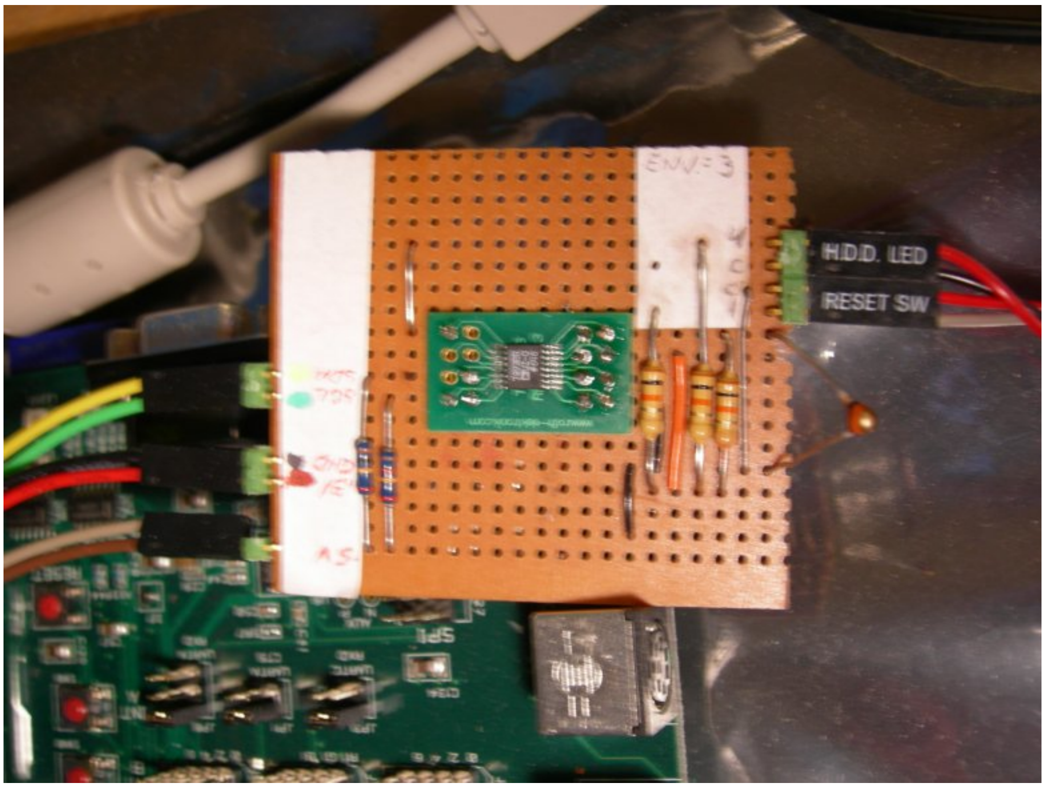

Figure 3.2: A/D converter mounted on a Veroboard. The texts “H.D.D. LED“ and ”RESET SW” are there because the cables are from a PC. They are connected to the inner and outer chamber NTC thermistors.

$$Untc = \frac{In * Vref}{2^{10}}$$

We need the resistance in the NTC, R_{ntc}, which is found from:

$$U_{ntc} = \frac{R_{ntc} * U_r}{R_r} = \frac{R_{ntc} * (U_{ref} - U_{ntc})}{R_r}$$

$$R_{ntc} = \frac{U_{ntc} R_r}{U_{ref} - U_{ntc}}$$

The the temperature in Kelvin is found from this formula from datasheet.

See directory NTC_thermistor in digital attachments.

$$abstemp = \frac{1}{A_1 + B_1 * ln\frac{R_{ntc}}{R_r} + C_1 ln^2\frac{R_{ntc}}{R_{ref}} + D_1 * ln^3\frac{R_{ntc}}{R_{ref}}}$$

Then temperature in Celsius is abstemp + 273,15. The constants X_1 are found in the datasheet for the specific NTC product.

The A/D-converter has different modes of operation. It can either trigger measurement from an external pin, do measurement periodically or by request. Since the temperature control is a slow system, the delays for A/D-converting is in another magnitude than is needed, so it does not matter how the converter is used, as long as we get at least one measurement per second. All channels were enabled by writing to the configuration register (and the other option in the configuration byte at the default). Then first periodical reading was tried by setting the cycle timer register to 256 times the convertion time. The read value is then found in the convertion result register. Afterwards the reading by request was tried, by reading from the convertion result register, but with the requested channel(s) set in the upper 4 bits.

In order to use the TWI driver, the module has to be loaded in addition to the general /dev/i2c driver by running these commands on the Linux command line:

```
modprobe i2c-atmeltwi
modprobe i2c-dev
```

The calculation was implemented in a program for the AVR32 using the C math library. See source code of adc.c in the appendix B or in digital attachments.

3.2.2 Heat generation

For heating we are using 3 resistors in the reaction chamber. They are driven by an external power source controlled by a MOSFET. This is to be controlled by the microcontroller. Since the physical system is slow, analog output is not needed, it is sufficient to use a digital output to turn the heating on and off in a duty cycle.

The easiest way would probably be to use a PWM (pulse width modulation) output, but since that currently has no Linux driver available, we can set up a similar system by using ordinary digital outputs, which is supported by a userspace GPIO /dev-device driver

Digital I/O ports on the AVR32 STK1000 development board are shared by different equipment. Most port can be used either by one of two different hardware modules or used as general I/O. There are four 8 bits connectors easily available on the STK1000 board, but by checking the ports they are connected to, we find that connector J1 is used totally by ISI which we want to use simultaneously, J3 and J6 are used for Ethernet, and most of J2 is also used for ISI. There are though three free lines (GPIO13-15) on J2 which we can use. In addition some lines on J29 and J31 could have been used.

The /dev-driver is configured by configfs usually mounted at /config. By adding a directory below /config/gpio, some files appear for controlling the access to the gpio. The standard distribution adds /config/gpio/switches, so I add a new one, /config/gpio/J2. Inside the following files appear: enabled, gpio_id, oe_mask and pin_mask. It is initialized by this shell code:

```
Set pin mask, we only use the 3 free pins, bits 5-7 => E0:
‘‘echo 0xE0 > /config/gpio/J2/pin_mask’’

Set the same pins as output lines:
‘‘echo 0xE0 > /config/gpio/J2/oe_mask’’

Set to the second GPIO

echo 2 > /config/gpio/J2/gpio_id

Enable:
‘‘echo 1 > /config/gpio/J2/enabled’’
```

3.2.3 Temperature estimation

The formula 3.1 gives the change of the inner temperature, which is going to be estimated. That means we can save the current estimated temperature in a variable and then add the derivative of the inner temperature given in the formula multiplied with the time elapsed since the last estimation. The algorithm can be seen in function "estimate_temp()" in appendix B. The unknown constants are a problem. Some calculation has been made, but they can also be found by trying different values and then adjust them. For this to be correct it is necessary to know exactly the values of these constants and other effects like radiation directly from the resistors to the inner part.

3.2.4 Temperature controller

As we can see from the model of the system in the physical modelling chapter, this is a quite slow system with heat capacity making an integration effect in the system so a P or PD controller should be enough to make it stable. Ideally a model based controller be used, but that is not possible without knowing at least the four unknown constants. Not enough independent experiments were found to find all of these constants. Without these, the point of simulating the system or checking the stability of it using for instance bode diagram is lost.

3.3 Testing of solution

The /dev device driver was not documented at all when this was tested, so searching on the Internet after API specification for /dev i2c drivers for Linux was tried. Following a couple of them did not succeed, the driver did not respond like it was supposed to. Thinking about the example code for the first version of the ISI driver actually used TWI to set up the image chip, that code was studied, and the same method used for this part. It was actually more complex than the other drivers found, allowing you to send structs describing the I^2C transfers wanted, so that you by one function call both specify sending of register you want to read, and then a read request of the register afterwards.

The A/D-converter was first tried using periodical reading. By outputting the values and warm up one of the NTCs, it was clear that the order of the channels read was random, even when reading all in one read operation. Although the read value also contain 2 bits that tells which channel that is read, logic for handling this was rather avoided by using the mode where A/D-conversion happens when you request it. A function ad7993_read_input(short channel) was made, which requests a specific channel to be read and strips off channel number and alert flag from the data. One could have made a function which reads all channels in one to save some traffic on the bus, but it was not done as the running time spent on this part was not a problem. Then it was easy to get the correct data of the channels that were requested and the A/D-chip sleep between each reading.

The reading and converting of the temperatures first seemed to work great, but stopped after some running. Reading the temperatures faster did make it to stop faster, leading me to believe some resource was used for every reading. Studying the code it was discovered that the /dev/i2c-0 file was opened for each reading and not closed, resulting in the Linux system to consume all file descriptors after some running. Adding a closing of the file did solve this problem. The file could also be kept open between each reading, but it did work without problems anyway, so it was not done.

When reading was working ok, the heat generation was also included, letting the chamber heat. After manually checking that it worked, a simple P-controller based on the inner measurement was added and the K-value tuned. What was then discovered was that when the I/O port was on, the power to the resistors was not as much as when giving 5 volts on the control input to the heat generator. The MOSFET was not completely turned on by the 3,3 volts output from the AVR32. This was solved by adding an ordinary transistor in front of it, letting much higher heating power, also resulting in much higher overshoot.

Then the estimator was tried, first initialized by constants given from the physical model. The estimated temperature was a lot lower than the real temperature, so it seemed it was something wrong with the estimation of the physical properties. Highering the constant for the heat flow from outer to the inner part of the chamber so it should match the real temperature gave a quite good result of heating from room temperature (20-25 C) to 30 C, but it was worse for higher temperatures (35-45 C). Thus it was suspected that some heat also flows direct from inner chamber to the outside, which is possible at one of the ends. A negative factor of the difference between the inner temperature (estimated) and the room temperature (measured) was added, giving much better results for higher temperatures, while preserving good results for lower temperatures.

A test of the estimator and controller is shown in figure 3.3 where K=0,2, $T_i = 0,1$ and setpoint 35 C. What we can see is though the estimator is quite good, we have a quite large overshoot which takes too much time too move back from 40 C to the desired 35 C. This is probably caused by the control based on the inner temperature, which is hanging behind the outer temperature. We can see that the outer temperature is moving very high in the start before it is stopping when the desired temperature is crossed. But then the outer temperature is too high, which causes the inner temperature too get too high. Tuning down the K constants makes the overshoot less, but makes the temperature change slower. What we need is a limit of the outer temperature, which is not used is the regulator at all, just as part of the estimator.

It would be possible to limit the outer temperature by just turning the heater off if the temperature becomes too high, but a much better way to do it is to make 2 controllers, one for the inner temperature which controls a controller of the outside temperature instead of just going directly to the heater, by giving it a relative temperature to the inner like this line of code:

```
setpoint_outer_temp = est_temp + ( 3.0 * (setpoint - est_temp) );
```

which gives an outer P-controller. Testing this shows that it avoids the high outside temperature, the outer temperature is decreasing before the inner temperature setpoint is reached, and thus gives much less overshoot, though uses some time to reach the target temperature. See figure 3.3. Although the estimator is somewhat lower than then real value, the overshoot is nothing (by looking at the estimator) compared to the large overshoot seen in figure 3.3. It was here running with 15V and 0.3 A = 4,5 W through the resistors.

3.4 Discussion

In this testing model the water inside is closed inside and not possible to replace, thus it has a temperature near the measured outer temperature, which is the same as the environment temperature when starting. In a real case the fluid flowing in could be stored at a low temperature (to preserve it). Then we know what temperature the fluid has when it flows into the chamber. It would then be natural to update the estimate to this temperature (if known, probably environment temperature or some storage temperature).

One problem with this measurement if we need very exact measurement is the uncertainties of the constants in the formula. They are given for a NTC product, not for each NTC resistor that you buy. That means there

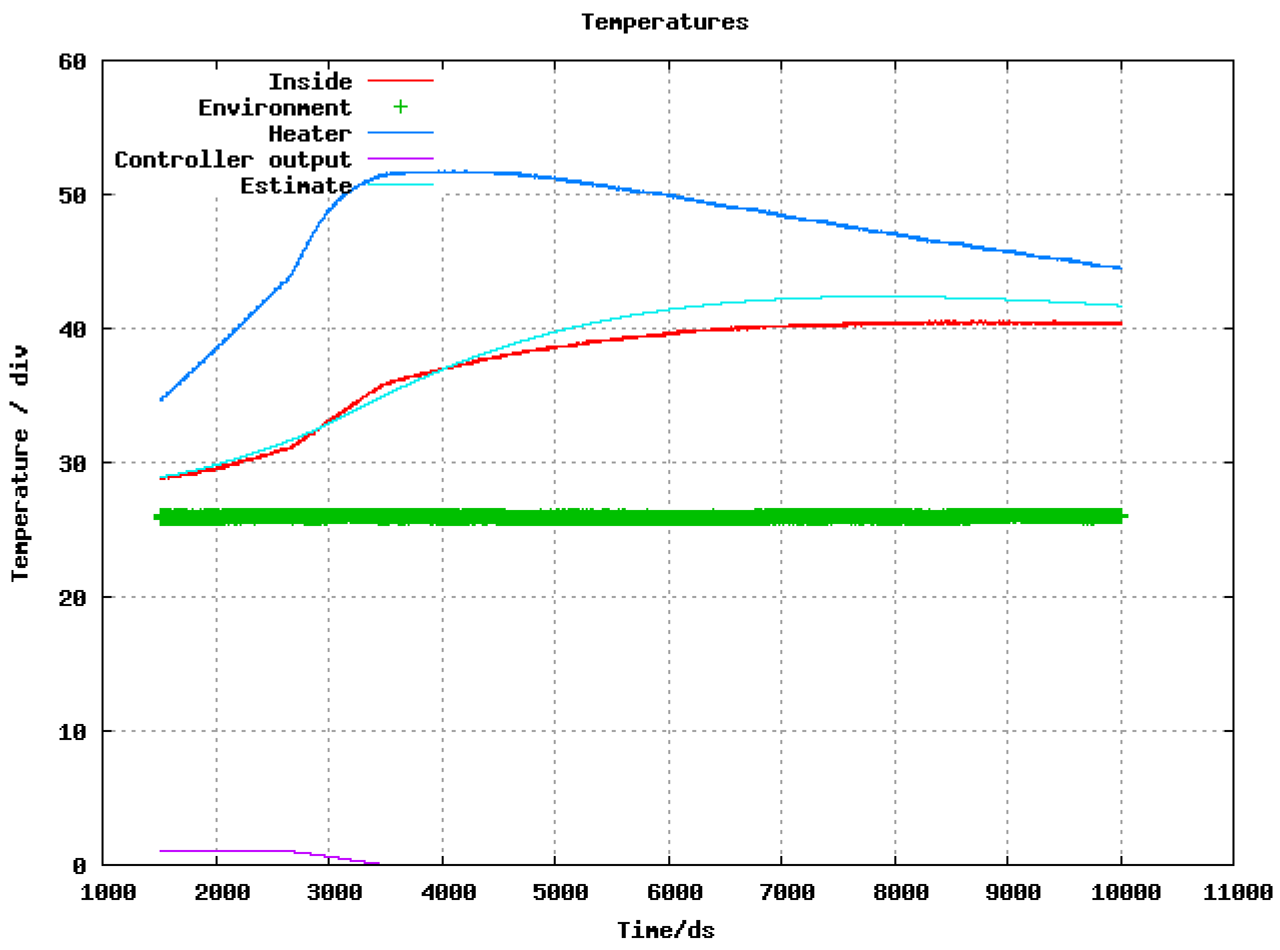

Figure 3.3: Temperature estimation and control. K=0.2, Ti=0.1, set-point=35 C

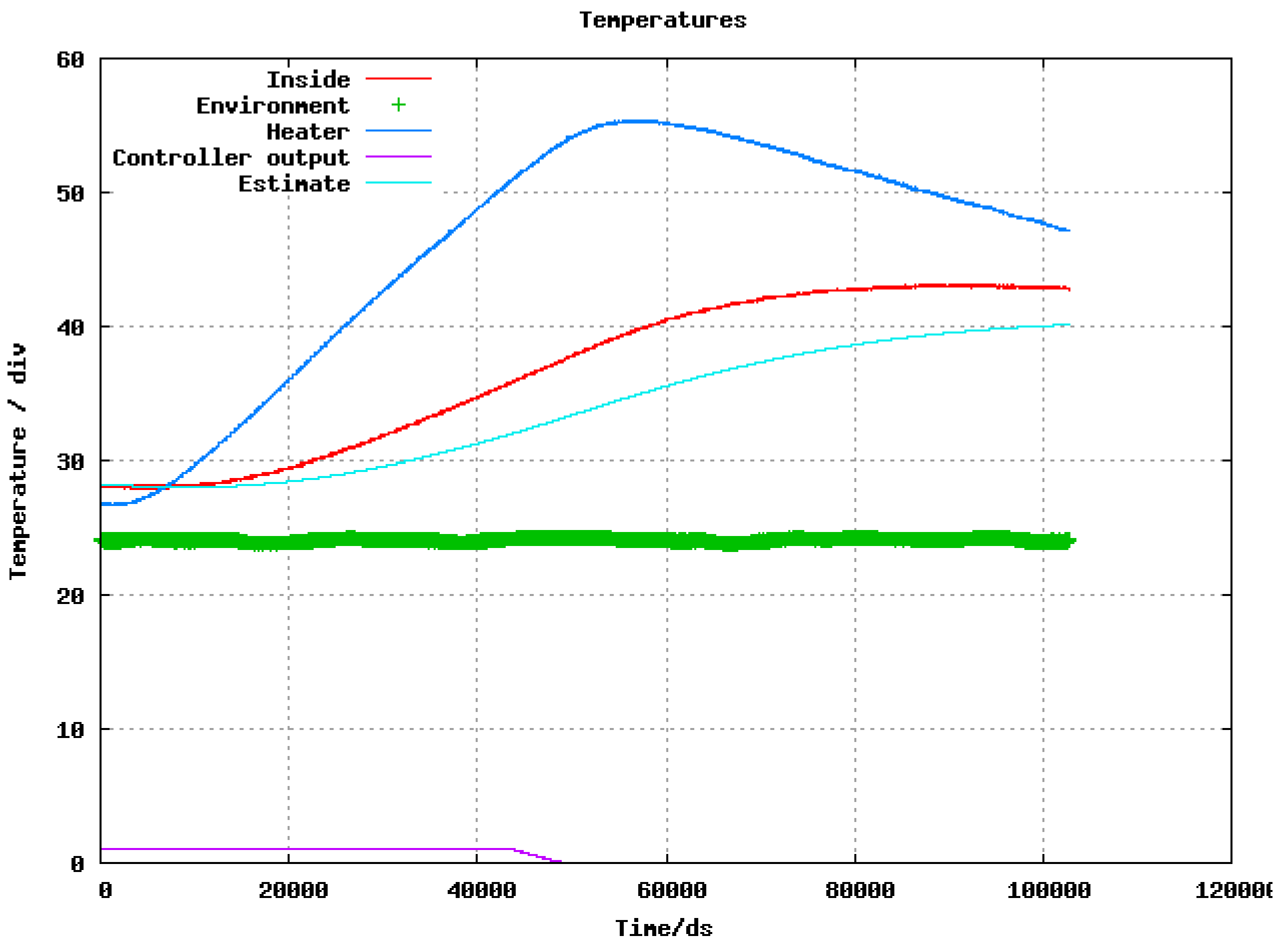

Figure 3.4: Temperature estimation and control using both outer and inner cotroller. K=0.1, Ti=0.1, setpoint=40 C

could be depending on the product typically around 1% uncertanty of the measurement. This can to a large degree be avoided by calibrating the actual NTC resistors before selling the product, by either filling the chamber with water of known temperature or measuring the temperature after heating to a specific temperature and then calibrate the constants until it is heating correct. If not only linear adjustments are going to be made, measuring with different temperatures are neccessary. For this to work it is required that the estimation is working perfectly. Another (possibly simpler and more exact) solution to this is to place the chamber in 2 known temperatures (using refrigerators) and then measure the temperature after some time.

Because of the added transistor to the circuit, the logics of the heating is no longer safe, it is heating if the the input line is not pulled down to zero. This should be avoided in the next prototype to reduce the risk of overheating in case of a cable problem. Anyway it might be smart to include some physical limitation too, either some measurement outside the chamber that is stopping heating using some simple electronic circuit if overheating occurs, or ensuring that the maximum power delivered if the heater is left on is not enough to cause damage, just give a high temperature.

Chapter 4

Flow control

4.1 Background

In the reaction chamber it should be possible to add certain amounts of different fluids to make reactions, then later let them out of the chamber. Water should be one of these fluids, to clean the equipment between each analysis. Mixing of the fluids is assumed to happen through a common pipe into the reaction chamber or inside it and is out of scope of the project.

4.2 Chosen solution

First a solution using syringes pressured by springs and controlled using valves and pressure sensors were thought of. The problems with that solution were that it was complex, hard to measure flow and amount of fluid left in the syringe, requiring a lot of estimators based on remote measurements.

Then a simpler solution was found, using a screw that pushes into the syringe starting fluid flow. That gives much easier control of the amount of fluid applied to the reaction chamber and we also know how much fluid is left. Control of the screw was first tested using Lego (r) parts, a motor connected to the screw connected to a rotation sensor. It was found to be unpractical size and unreliable connection to the screw. Thus a better solution using a stepping motor was explored. In fact it was possible to get stepping motors with an internal screw system so that you get linear pushing without rotation outside the motor package. With such a stepping motor mounted on the syringe, the moving shaft is pushing the piston which then pushes out the fluid. This gives a precise control of the flow by stepping one and one step with the motor, resulting in a small amount of fluid flow for each step. By counting the steps, it is also possible to know how much is left, since the total number of steps is known. See figure 4.1 of the stepping motor mounted on the syringe.

The stepping motor found from Haydon also had available a driver board that handles the different stepping motor phases, and smooth switch between them as well as half steps. It is controlled either by microswitches on the board or external 0/5V input lines to turn motor on/off, set direction and triggering steps, making it easy to connect to the AVR32 board's i/o lines. The board was easy to get up and running, connected to the motor and use the button on the board for manual stepping motion. But when the documentation of the input lines was studied in more detail, it was clear

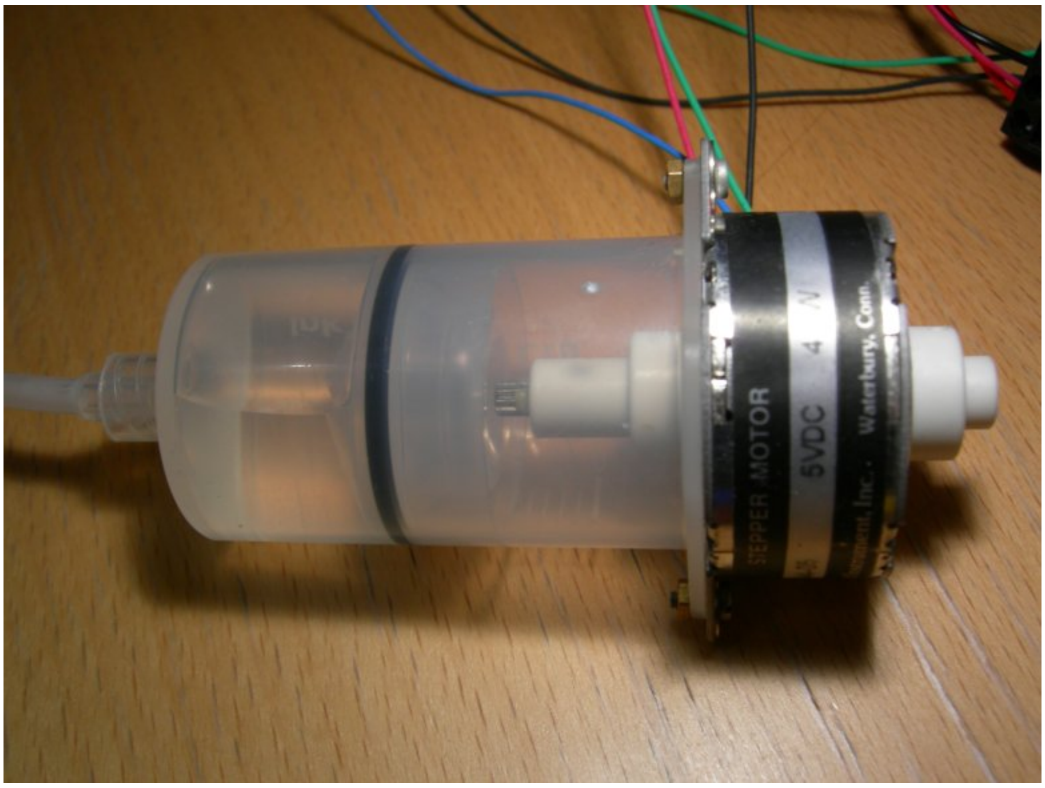

Figure 4.1: Stepping motor used for the flow control

that the stepping input line was supposed to be short-circuited by a button, relay or something like that, which did not seem safe to connect to one of the I/O lines of the AVR32, as it possibly could cause current to flow into the microcontroller with a risk for damage or malfunction.

Thus a new solution was searched for, ideally using I^2C interface, as that was working great with Linux on AVR32. One from AMIS, AMIS-30624, was found suitable. It has a lot of features like automatic positioning with adjustable acceleration and speed, current etc. It can also make smooth movement using 1/2, 1/4, 1/8 and 1/16 steps, and since it is using pulse-width modulation for controlling the average current through the coils, it generates little heat, allowing it to be a normal surface mountable integrated circuit, though light cooling on circuit board is recommended. Anyway it measures its own temperature and shuts down when it becomes too high, in addition to detecting error condition in the motor or the connection to it. A board was made following the recommended setup of external capacitors etc. it required. See images 4.2 and 4.3.

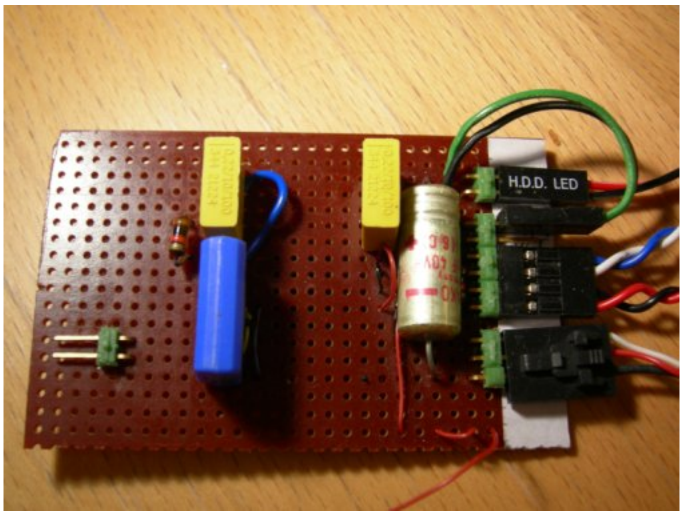

Figure 4.2: Front of flow controller board

The address of the chip is one byte, of which the two upper bits are 1, the next 4 bits are defined by OTP-memory, then follows one bit which is dependent on an input line called HW, and the lowest bit is always 1. OTP is one time programmable memory, meaning it is only possible to zap them from 0 to 1 but not the opposite. This addressing makes it possible to use up to 32 such chips on the same bus. One has to set the OTP memory if more than 2 chips are used, or there is an address conflict with some other chip on the same bus. This is done using a special command to the chip and ensuring that the supply voltage is in a specified range. We only use one here during the testing, so using the default address is fine.

Using the i2c /dev driver some utility function for the relevant messages to send to the controller was made. For less error prone and easier readable programming of the bit fields in the messages to be sent and received, corresponding c-structs were made, see the complete source code in appendix A.

The position given to the controller is relative to where it starts, and when the power is turned on you don't know were the shaft of the motor is placed. It means that you somehow have to get it running to a known position or measure the position by some external sensor. In this case it is easiest to let it run to the end first as it does not damage the piston (they are not connected). To avoid too much wear and tear on the motor, it should be stopped at the end. It is possible to get a motor with end switch, but the testing motor for this project did not have one. Also the controller supports

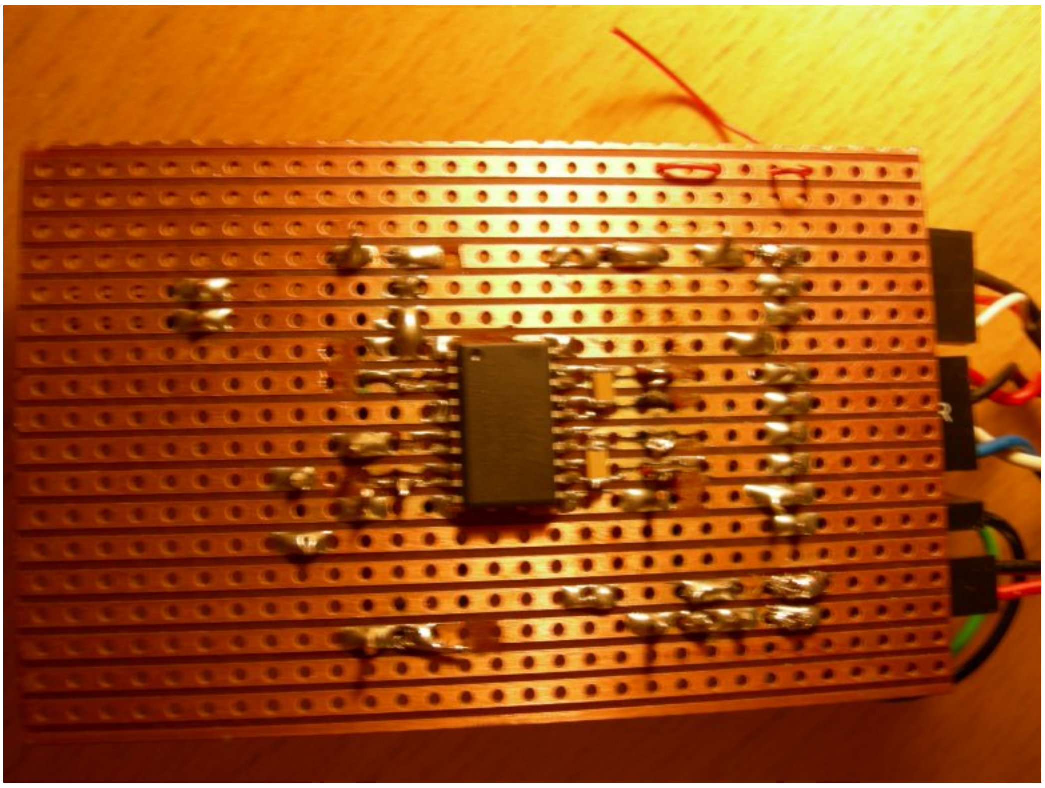

Figure 4.3: Back of flow controller board

measuring out of the returned EMF that the shaft has met some obstacle or the end (stall detection), which is supposed to be possible to use instead of the end switch. It is then possbible to know how much fluid is pushed out of the syringe through the position given to the controller. To avoid problems If the power supply is broken during the use of some fluid, the position should be saved frequently during the movement to some safe storage (like flash ram). A replacement of syringe should only be done when the shaft is moved to the innermost position.

4.3 Testing of solution

The motor and controller chip was first tested using the default settings and the minimum voltage 8 V to the chip. Sending the SetPosition command was tried with different values. Apparently nothing happened, but listening closely to the motor it was possible to hear some noise when sending a SetPosition command with a new position. Then it was clear that the motor was supplied with too little power. Maximum current during stepping, “Irun” register, is as default 0 which means 50 mA (see table in datasheet). The motor is specified to 5 VDC and 4,6 W, and the chip must have at least 8 V supply voltage. Given that voltage, the maximum current should be set to 4,6 W / 8 V = 0,575 mA. Nearest settings are D=566 and E=673. But since the motor in this case is not going to be run more than for short times, and during movement the power to each coil is varying (we do not need much holding current here), a somewhat higher power is possible. Therefore “Imax” was set to the maximum F=800 mA and input voltage to 10 V. This gives 10 * 0,8 = 8 W to the motor. This value is set by the “SetMotorParam“ command, which also sets other parameters. These parameters were initially set to their default values.

After the change, the shaft was moving, but it was clear that most of the steps were lost, so it did not move much. Then the maximum and minimum speed was lowered by setting both register ”Vmax“ to the lowest (0=99 full steps/s) and Vmin to the lowest (1= 1/32 of Vmax). The movement was then better, but it still lost a lot of steps in the beginning of the motion. I then began wondering if the motor was too weak for the controller, but trying higher voltages did not help either. Then the acceleration register was tried to change to the lowest value (0 = 49 fullsteps/s^2). Using these settings it seemed to work great, getting all the steps using a fast enough movement.

Then the movement to the end position was tried, using stall detection. It first seemed to work pretty well, stopping almost at the end. There were somewhat varying how close to the end it stopped because of some running at the end before stopping. At the shaft’s outer end it worked sometimes to stop when the syringe was empty, but not that good, because the pressure against the fluid made the stall detection detect obstacle before the end. The test program was changed to count the steps moved from the start position, and stop when the measured total number of steps were moved.

During that testing a new problem was discovered. The stall detection during the movement to the start position both detected the end and a position where the shaft was halfway out. It probably means that there are something inside the stepping motor making it a bit harder to move at one place. Since I don’t have more motors to test this on, it is hard to know if it is something wrong with this motor, or it is possible that every motor can have a similar problem. Beside this problem (forcing it over the problem point), the interactive program works nice, allowing you to enter the number of steps you want it to move forward after automatically moving to the start.

4.4 Discussion

This part did work quite good without much time spent on technical problems with the TWI drivers already working from the temperature reading part. It was probably good that the original proposal to use pressure measurements was avoided as it probably would have been a lot more work to get working and with less accurate operation. The example program is almost ready to use, but the motor should be replaced with one containing an end switch to avoid the problem with the end detection.

Chapter 5

USB communication

5.1 Background

USB communication between a PC and the board should be tested to be able to transfer pictures and control the equipment through USB, avoiding the use of ethernet and allowing very fast transfer of images. This makes it easy for customers of the product to use, just plug in the USB cable instead of doing network configuration to use it.

5.2 Chosen solution

The USB standard defines a lot of standard profiles to allow using for instance a random mouse without a corresponding driver. By looking at the available profiles with corresponding protocols, some might look usable. ”Picture Transfer Protocol“ (PTP) is what seems to be closest to what we need, and avoids the need for device driver at the PC. Many programs already exists that supports this protocol. At least during testing it might be nice to use this protocol, although transfer using NFS through ethernet is then also easy. What is the disadvantage of using such profile/protocol in a final product is that it might not be possible to add extra possibilities in the protocol for other transfers than pictures.

This could either be done by embedded programming without O/S or using Linux. Embedded programming was first tried, and then using Linux. Using Linux there exists something called USB gadget API framework. Several protocols are supported. One is ”Ethernet over USB“ which enables the use of the usual TCP/IP protocol on both PC side and microcontroller side, but requires some configuration on the PC side and might also require network allocation, routing etc., making it unsuitable for a consumer product. Then we have a serial emulation driver, but it does not supports speeds higher than the serial port, which is way too slow. File backed storage sounds promising but the problem here is that it then follows the USB mass storage profile, which operates at the level below the file system, making it impossible to access the files (the whole file system exposed) from both the microcontroller and the PC at the same time. Unloading the driver and remounting the file system between each picture and other transfer does not sound like a good solution.

What is left then is a general system called ”Gadget Filesystem” (GadgetFS), which is a way of programming a Linux USB slave device from userspace by accessing special files. Asynchronous I/O is also supported for high speed transfer of data, so it seems usable for this task. The protocols, type and number of endpoints is then up to the program to configure. For the image transfer either isochronous or bulk transfer endpoints should be used. Isochronous is the fastest by skipping error correction, but here bulk should probably be used instead if the speed is fast enough, as we should have no transfer errors of the pictures.

5.3 Testing of solution

The embedded program did not succeed, as the USB controller module did not behave like described in the documentation, which was also quite incomplete. Also since the other parts were decided to develop using Linux, that was then tried. An example program for the gadgetfs was tried after mounting the gadget filesystem using:

```
  mount -t gadgetfs gadgetfs /dev/gadget
```

after creating the directory /dev/gadget (it is a file system, not a device node) and loading the module "gadgetfs.ko".

When this was first tested, the loading of the module just gave a segmentation fault and a stack trace which did not indicate what the problem was. Testing this again after the Board support package 2.0 for STK1000 board was released, this problem had disappeared and the example program did seemed to halfway work:

```
 /dev/gadget/husb2_udc ep0 configured
serial="agpzt0gnj47lrfmqp4u9bw8j61a2jihh29ninvvpv1ecnp1jhrf315ov9rygv4p"
? illegal config
ep0 stall: No such process

ep0 read after poll: Interrupted system call
```

and the device connection could be seen from the PC:

```
[568045.847407] usb 5-2: new high speed USB device using ehci_hcd and address 2
[568045.981130] usb 5-2: string descriptor 0 read error: -32
[568045.982711] usb 5-2: string descriptor 0 read error: -32
[568045.984125] usb 5-2: string descriptor 0 read error: -32
[568045.984295] usb 5-2: configuration #3 chosen from 1 choice
[568050.982696] usb 5-2: can't set config #3, error -110
```

although something seemed to be wrong.

5.4 Discussion

As it was decided that the USB part was less important than the other parts, as the images could be transferred using ethernet or SD cards, more testing of this part was not conducted. The gadgetfs system seemed to work. Just some small problems needed to be solved, and it seems flexible enough to follow the requirements of both image transfer and control of experiments from the PC, so as long as Linux is going to be used on the microcontroller, this system should be the first to try when the USB transfer is needed. Avoiding making a new kernel module for the USB transfer should let the development be much easier and faster.

Chapter 6

Conclusion

At the start of the project it was stated several parts that was to be solved. This was done because it was hard to estimate how much work was required to complete them, as much of the time is often spent solving problems and you can not know in advance how many problems will appear and how hard it is to solve them. It is better to have more to work on when one part fails, or vti'e waiting for ordered parts, and always have something to solve if all of the tasks are completed.

The image transfer was very hard to solve until the breakthrough with the test on the NGW100 card, but also required a lot of work because of the need to patch and recompile the Linux kernel right, and understand the kernel module code to be able to debug it, fix problems and extend it. The result is code that almost is working and patches that fix problems, extend the maximum resolutions possible and add support for one additional image chip that Atmel is probably interested in including into their packages and the normal Linux kernel.

After getting the TWI drivers to work, which required manual patching of the kernel before the Board support package 2.0 was released, the temperature control was possible to do with only small technical problems that were relatively easy to solve, thus allowing to use time on the actual estimation and controlling algorithms and parameters. The resulting controlling is also following the requirements, so more tuning is only needed if the operation should be faster.

The flow control was relatively straight forward, as the TWI driver system already was well tested, it just took some time to wait for the motor and the controller chip. It is now working pretty well and almost usable, there is only one problem moving the shaft to the initial position, which probably will be solved when a new motor using an end switch is received. Anyway it is possible to use it in experiments by manually moving the shaft to the start position before connecting fluid content on the stepping motor (which now must be done manually anyway).

In was done some experiments with the USB part in the beginning, but as it failed on some stack traces when loading the driver itself, it was postponed and time spent on the other parts which turned out to be more important when Linux was chosen, as there are then other ways of transferring data during the experiment phase of the product development. Though after the Board Support Package 2.0 was released it turned out to be possible to continue on this part, although then time was already spent on solving the other parts.

Sequence control from PC was also a low priority task as it is used for controlling the other parts behavior, therefore getting them working first was of higher priority. The other parts are though based on functions that triggers the actions so it should be easy to make this a system of function calls to other modules first. To make it more flexible it could read a data file that is transferred using NFS containing a simple command language to trigger the different operations. To avoid doing anything on the microcontroller board, the program could check if a new file is uploaded to a specific directory and then automatically executing the files it finds. If necessary these files can be made using a graphical user interface on a PC and later they may be transferred using USB instead of ethernet.

Overall most of the parts have been accomplished according to the specifications, although not too detailed practical testing of each part because of much time spent on solving problems and the fact that many complex parts were outlined to be solved during the project. Solving the small problems left as described at the corresponding discussion chapters should, with small modifications/customization, make the practical solutions found usable for future prototypes of the product or for more experiments.

Chapter 7

Future work

The priority list of tasks to perform to complete the parts required in this project is as follows:

1. Solve the problem with TWI on the NGW100 board to avoid using the big, complex and expensive STK1000 card and allow all parts (image capture and temperature control etc.) to run on the same board. It is hard to estimate how much time is required to fix this. It could be a simple problem that is solved in one hour or require a week of debugging. The cost of this is just the work needed.

2. Debug the image capture problems more. Having solved the TWI problems could make this easier. Since it is almost working it is probably caused by a problem like noise in the cables which can be reduced or a broken cable. It is also hard to estimate the time this require as it is probably one simple error that has to be fixed for it to suddenly work.

3. When a new stepping motor is delivered, test stopping the motor by polling the end stop switch. It seems also interrupt on change of the I/O pins is supported in Linux so that it might be used instead of polling, if required. Adding this capability seems like a trivial task, as digital output is working, and then a digital input connected to the switch should also work.

4. Move the temperature controlling system to a real reaction chamber and adjust the parameters so that it works adequate on that system too. If faster temperature stabilization is needed, calculating or measuring the unknown parameters in the model probably has to be done to make a better regulator. Depending on how good the controller is to be made, this can be time consuming but should not be delayed by technical problems. It might require finding an application to model the three-dimensional physical model, in order to get the exact constants, or hire someone to do the calculation.

Appendix A

Source of flow control

```
#include "stepper.h"
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#include <math.h>

#define DEBUG 1

//typedef short boolean;
typedef enum {false = 0, true = 1} boolean;

typedef struct stepper_full_status1_t {
  unsigned int address         : 8;

  unsigned int i_run           : 4;
  unsigned int i_hold          : 4;

  unsigned int v_max           : 4;
  unsigned int v_min           : 4;

  unsigned int acc_shape       : 1;
  unsigned int step_mode       : 2;
  unsigned int shaft           : 1;
  unsigned int acceleration    : 4;
  unsigned int vdd_reset       : 1;
  unsigned int step_loss       : 1;
  unsigned int electrical_def  : 1;
  unsigned int uv2             : 1;
  unsigned int thermal_shutd   : 1;
  unsigned int thermal_warn    : 1;
  unsigned int temp_info       : 2;

  unsigned int motion_status   : 3;
  unsigned int ext_sw_status   : 1;
  unsigned int over_current_1  : 1;
  unsigned int over_current_2  : 1;
  unsigned int stall_detected  : 1;
  unsigned int charge_pump_err : 1;

  unsigned int                 : 8;
  unsigned int abs_threshold   : 4;
  unsigned int delta_threshold : 4;
} stepper_full_status1_t;

/**
Buf min 8 bytes
**/
short stepper_read_status1( stepper_full_status1_t* status1 )
{
int file;
int reg = 0x81; // GetFullStatus1
unsigned char wbuf[1] = { (unsigned char)reg };
struct i2c_msg t_testSet[2] = {{STEPPER_ADDRESS, 0, 1, wbuf},{STEPPER_ADDRESS, 1
struct i2c_rdwr_ioctl_data t_msgSet;
t_msgSet.msgs = t_testSet;
t_msgSet.nmsgs = 2;

        if ((file = open(TWI_DEVICE,O_RDWR)) < 0) return -ENODEV;

if( ioctl(file,I2C_RDWR,&t_msgSet) < 0 ) return -1;

return close(file);
}

typedef struct stepper_full_status2_t {
  unsigned int address         : 8;
  signed   int act_pos         : 16;
  signed   int target_pos      : 16;

  unsigned int secure_pos_low  : 8;

  unsigned int FS2StallEn      : 3;
  unsigned int                 : 1;
  unsigned int DC100           : 1;
  unsigned int secure_pos_high : 3;

  unsigned int abs_stall        : 1;
  unsigned int delta_stall_low  : 1;
  unsigned int delta_stall_high : 1;
  unsigned int min_samples      : 3;
  unsigned int dc100_stall_en   : 1;
  unsigned int pwmj_en          : 1;
} stepper_full_status2_t;

short stepper_read_status2( stepper_full_status2_t* status )
{
int file;
int reg = 0xFC; // GetFullStatus2
unsigned char wbuf[1] = { (unsigned char)reg };
struct i2c_msg t_testSet[2] = {{STEPPER_ADDRESS, 0, 1, wbuf},{STEPPER_ADDRESS, 1
struct i2c_rdwr_ioctl_data t_msgSet;
t_msgSet.msgs = t_testSet;
t_msgSet.nmsgs = 2;

        if ((file = open(TWI_DEVICE,O_RDWR)) < 0) return -ENODEV;

if( ioctl(file,I2C_RDWR,&t_msgSet) < 0 ) return -1;

return close(file);
}

int stepper_set_position(int pos)
{
        int file;
        unsigned char msg_data[5]= {0};
        struct i2c_msg msg = {STEPPER_ADDRESS, 0, 5, msg_data};
        struct i2c_rdwr_ioctl_data wr = { &msg, 1 };
short reg = 0x8B; // SetPosition

pos <<= 3; // half stepping

        /* open handle */
        if ((file = open(TWI_DEVICE,O_RDWR)) < 0)
                return -ENODEV;

msg_data[0] = (unsigned char) reg;

msg_data[1] = 0xFF;
msg_data[2] = 0xFF;
msg_data[3] = (unsigned char) ((pos&0xFF00)>>8);
if ( pos < 0 ) msg_data[3] |= 0x80; // set sign bit
msg_data[4] = (unsigned char) (pos&0x00FF);

        if ( ioctl(file, I2C_RDWR, &wr) <0 )
                return -1;
close(file);

return 0;
}

int stepper_set_motor_params(unsigned char iRun, unsigned char iHold)
{
        int file;
        unsigned char msg_data[8]= {0};
        struct i2c_msg msg = {STEPPER_ADDRESS, 0, 8, msg_data};
        struct i2c_rdwr_ioctl_data wr = { &msg, 1 };
        short reg = 0x89; // SetMotorParams
        /* open handle */
        if ((file = open(TWI_DEVICE,O_RDWR)) < 0)
                return -ENODEV;


msg_data[0] = (unsigned char) reg;
msg_data[1] = 0xFF;
msg_data[2] = 0xFF;
msg_data[3] = (unsigned char) (((iRun&0x0F) << 4) | ((iHold & 0x0F)));
msg_data[4] = 0x01; // Vmin/Vmax to min.
```

```
msg_data[5] = 0x00; // Sec position, acc to min.
msg_data[6] = 0x00; // Sec pos.
msg_data[7] = 0xA2; // Div. set to default

        if ( ioctl(file, I2C_RDWR, &wr) <0 )
                return -1;
close(file);

return 0;
}

typedef struct set_stall_param_t {
  unsigned int command         : 8; // = 0x96
  unsigned int data1           : 8;
  unsigned int data2           : 8;

  unsigned int Irun            : 4;
  unsigned int Ihold           : 4;

  unsigned int Vmax            : 4;
  unsigned int Vmin            : 4;

  unsigned int min_samples     : 3;
  unsigned int shaft           : 1;
  unsigned int acc             : 4;

  unsigned int abs_threshold   : 4;
  unsigned int delta_threshold : 4;

  unsigned int fs2stall_en     : 3;
  unsigned int acc_shape       : 1;
  unsigned int step_mode       : 2;
  unsigned int dc100_st_en     : 1;
  unsigned int pwmj_en         : 1;
} set_stall_param_t;

int stepper_set_stall_params(unsigned char iRun, unsigned char iHold)
{
        int file;
        set_stall_param_t msg_data = {0};
        struct i2c_msg msg = {STEPPER_ADDRESS, 0, 8, (unsigned char*)&msg_data};
        struct i2c_rdwr_ioctl_data wr = { &msg, 1 };
        short reg = 0x96; // SetStallParam
        /* open handle */
        if ((file = open(TWI_DEVICE,O_RDWR)) < 0)
                return -ENODEV;

msg_data.command = (unsigned char) reg;
msg_data.data1 = 0xFF;
msg_data.data2 = 0xFF;

msg_data.Irun = iRun;
msg_data.Ihold = iHold;

msg_data.Vmax = 0; // Vmax to min.
msg_data.Vmin = 1; // Vmin to min. part of Vmax

msg_data.min_samples = 0; // (?)
msg_data.shaft = 1; // shaft direction
msg_data.acc = 0x00; // acceleration to min.

msg_data.abs_threshold = 0x2; // Enable and try highest (sensitivity)
msg_data.delta_threshold = 0x00; // Disable...  // Enable and try highest thresh

msg_data.fs2stall_en = 0; // trying lowest
msg_data.acc_shape = 0; // normal accelecation (enable)
msg_data.step_mode = 0; // half steps // 1/16 steps  // default: half steps
msg_data.dc100_st_en = 1; // disable of motion detection on 100% duty cycle
msg_data.pwmj_en = 1; // enable pwm jitter to avoid resonance harmonics

        if ( ioctl(file, I2C_RDWR, &wr) <0 )
                return -1;
close(file);

return 0;
}

int stepper_reset_position()
{
        int file;
        unsigned char msg_data[1]= {0};
        struct i2c_msg msg = {STEPPER_ADDRESS, 0, 1, (unsigned char*)&msg_data};
        struct i2c_rdwr_ioctl_data wr = { &msg, 1 };
        short reg = 0x86; // ResetPosition
        /* open handle */
        if ((file = open(TWI_DEVICE,O_RDWR)) < 0)
                return -ENODEV;

msg_data[0] = (unsigned char) reg;

        if ( ioctl(file, I2C_RDWR, &wr) <0 )
                return -1;
close(file);

return 0;
}

#define STEPPER_MS_OUTER_MASK 0x4
#define STEPPER_MS_ACC 0x1
#define STEPPER_MS_DEC 0x2
#define STEPPER_MS_MAX 0x3

int busyWaitUntilStopped()
{
  stepper_full_status1_t status1;
    do {
      int res = stepper_read_status1( &status1 );
      if ( res < 0 ) {
printf( "Error reading status: %i", res);
return res;
      }
    } while ( status1.motion_status != 0x00 ); // wait until motor is stopped
}

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

  if ( argc == 3 ) { // test
    res = stepper_set_stall_params( 0x0F /* C=800 mA*/,  0x00 /* hold E=673 mA*/
    stepper_full_status1_t status1;
    char input[80];
    int pos = 0;
    if ( res < 0 ) {
      printf( "Error setting stall parameters: %i", res);
      return res;
    }

    // reset so we are sure the next command moves complete to the end
    stepper_reset_position();

    // ensure at innermost position
    res = stepper_set_position( -1000 );
    if ( res < 0 ) {
      printf( "Error setting position: %i", res);
      return res;
    }

    busyWaitUntilStopped();

    //    return -1;

    stepper_reset_position();


    do {
      stepper_full_status1_t status1;
      printf( "\nPos: %i. New position (relative), 0 to end: ", pos );
      gets( input );

      pos += atoi( input );
      res = stepper_set_position( pos );

      if ( res < 0 ) {
printf( "Error setting position: %i", res);
return res;
      }

      printf ( "Waiting for movement...." );
      busyWaitUntilStopped();
      printf( "Movement completed!" );

      /*      res = stepper_read_status1( &status1 );
      if ( res < 0 ) {
printf( "Error reading status 2: %i", res);
return res;
      }

      if ( status1.stall_detected ) {
printf( "\n\n Empty!\n" );

break;
}*/

      if (pos > 350 ) {
printf( "\n\n Empty!\n" );

break;
      }
    } while (atoi(input) != 0 );
  }

  if ( argc == 2 ) {
    stepper_full_status2_t status2;
    //    res = stepper_set_motor_params( 0x0C /* C=476 mA*/,  0x00 /* hold */);
    //res = stepper_set_motor_params( 0x0F /* C=800 mA*/,  0x00 /* hold E=673 mA

    res = stepper_set_stall_params( 0x0F /* C=800 mA*/,  0x00 /* hold E=673 mA*/

    if ( res < 0 ) {
      printf( "Error setting stall parameters: %i", res);
      return res;
    }

    res = stepper_set_position( atoi(argv[1]) );
    if ( res < 0 ) {
      printf( "Error setting position: %i", res);
      return res;
    }

    res = stepper_read_status2( &status2 );
    if ( res < 0 ) {
      printf( "Error reading status 2: %i", res);
      return res;
    }

    printf( "Act_pos: %i, Target_pos: %i, secure_pos: %i, asb_stall: %i, delta_st
    status2.act_pos >> 3, status2.target_pos >> 3, status2.secure_pos_low | (sta
    status2.abs_stall, status2.delta_stall_low, status2.delta_stall_high );
  }

    // read status
    char buf[7];
    int i;
    stepper_full_status1_t status1;

    res = stepper_read_status1( &status1 );
    if ( res < 0 ) {
      printf( "Error reading status: %i", res);
      return res;
    }

    printf( "Stall: %i, step loss: %i, electrical defect: %i, under voltage: %i,
    status1.stall_detected, status1.step_loss, status1.electrical_def, status1.u

    if ( 0x00 == status1.motion_status ) {
      printf( "Motor stopped\n" );
    }
    else {
      if ( status1.motion_status & STEPPER_MS_OUTER_MASK ) printf( "Outer motion
      else printf( "Inner motion " );

      switch (status1.motion_status & 0x3) {
      case STEPPER_MS_ACC:
printf( "acceleration." );
break;

      case STEPPER_MS_DEC:
printf( "deceleration." );
break;

      case STEPPER_MS_MAX:
printf( "max. speed." );
break;
      }
    }

    printf( "\n" );

} // main
```

Appendix B

Source of temperature control

```
//
// temp_ctrl.c
//

#include <pthread.h>

#include "adc.h"
#include "gpio.h"

int duty_cycle_on = 0;  // up to regulation_time
#define regulating_time 50 // ms

void *Duty_func(void *threadid)
{
  while(1) {
    if ( duty_cycle_on > 0 ) {
      gpio_set_output( 0 ); // on
      usleep( duty_cycle_on );
    }

    if ( duty_cycle_on < regulating_time ) {
      gpio_set_output( 1 ); // off
      usleep( regulating_time - duty_cycle_on );
    }
  }
}
```

```
/**
Returns new estimated temperature
**/
float estimate_temp( float outside_temp, float inside_temp, float environment_te
{
  const float constant = 0.0239178 / 8; // try and fail...
  const float constant_environment = constant / 3.4; // utifra maaling

  float delta_temp_pr_sec = constant * (outside_temp - inside_temp)
    - constant_environment * (inside_temp - environment_temp); // try to compens

  return inside_temp + delta_temp_pr_sec * secs_elapsed;
}


/** Returns control output **/
float regulate_outer_temp( float heater_temp, float setpoint, float Kp, float Ti
    float paadrag;
    static float lastPaadrag = 0.0;

    //float diff = (setpoint - est_temp);
    float diff = (setpoint - heater_temp); // - regulerer etter estimert temp.
    // P-reg
    //    paadrag = Kp * diff; // 0-1.0

    // PI-reg
    paadrag = lastPaadrag + Kp*diff + Ti*diff;


    if (paadrag > 1.0) paadrag = 1.0;
    else if (paadrag <= 0) paadrag = 0.0;

    int duty_on = regulating_time * paadrag;
    int duty_off = regulating_time - duty_on;

    duty_cycle_on = duty_on; // set global variabel for duty thread


    /*    if ( duty_on > 0.001 ) {
      gpio_set_output( 0 ); // on
      usleep( duty_on );
    }

    if ( duty_off > 0.001 ) {
      gpio_set_output( 1 ); // off
      usleep( duty_off );
    }
    */
    return paadrag;
}

int main (int argc, char **argv)
{
  ad7993_init();
  gpio_init();

  printf( "Temperatures: \nReaction chamber, environment, heater\n" );

  //#define Kp 0.5

#define Kp 0.1
#define Ti 0.1


#define setpoint 40.0

  pthread_t duty_thread;
  printf("Creating thread\n");
  pthread_create(&duty_thread, NULL, Duty_func, NULL);

  struct timeval tv;
  gettimeofday (&tv, NULL);
  long int startSecs = tv.tv_sec;
  long int startMillis = tv.tv_usec / 1000;

  int i = 0;

  float est_temp = read_temp(1); // Set to real value during testing. Should be

  while (1 ) {
    float heater_temp = read_temp(4) - 0.5; // calibrating

    float setpoint_outer_temp;
    setpoint_outer_temp = est_temp + ( 3.0 * (setpoint - est_temp) );

    float paadrag = regulate_outer_temp( heater_temp, setpoint_outer_temp, Kp, T

    gettimeofday (&tv, NULL);
    int millis = (tv.tv_sec - startSecs) * 1000 + (tv.tv_usec/1000 - startMillis

    est_temp = estimate_temp( heater_temp, est_temp, read_temp(3) /*envir.*/, re

    printf ( "%i %f %f %f %f %f\n", millis, read_temp(1), read_temp(3), heater_t

    usleep( 100 );
  }

   pthread_exit(NULL);

  return 0;
}
```

```
//
// adc.c
//

#include "adc.h"
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#include <math.h>

#define DEBUG 1

//typedef short boolean;
typedef enum {false = 0, true = 1} boolean;

unsigned short ad7993_read(unsigned char reg, boolean two_bytes)
{
int file;
unsigned char wbuf[8] = {reg};
unsigned char rbuf[8] = {0,0,0,0,0,0,0,0};
struct i2c_msg t_testSet[2] = {{AD7993_ADDRESS, 0, 1, wbuf},{AD7993_ADDRESS, 1,
struct i2c_rdwr_ioctl_data t_msgSet;
t_msgSet.msgs = t_testSet;
t_msgSet.nmsgs = 2;

        if ((file = open(TWI_DEVICE,O_RDWR)) < 0) return -ENODEV;

if( ioctl(file,I2C_RDWR,&t_msgSet) < 0 ) return -1;

close(file);

return two_bytes ? ((rbuf[0]<<8) | rbuf[1] ) : rbuf[0];
}

/**
Buf min 8 bytes
**/
short ad7993_read_channels( unsigned char* buf )
{
int file;
```

```
unsigned char wbuf[8] = { 0x0 };
// unsigned char rbuf[8] = {0,0,0,0,0,0,0,0};
struct i2c_msg t_testSet[2] = {{AD7993_ADDRESS, 0, 1, wbuf},{AD7993_ADDRESS, 1,
struct i2c_rdwr_ioctl_data t_msgSet;
t_msgSet.msgs = t_testSet;
t_msgSet.nmsgs = 2;

        if ((file = open(TWI_DEVICE,O_RDWR)) < 0) return -ENODEV;

if( ioctl(file,I2C_RDWR,&t_msgSet) < 0 ) return -1;

return close(file);
}

int ad7993_write(unsigned char reg, unsigned short val, boolean two_bytes )
{
        int file;
        unsigned char msg_data[2]= {0};
        struct i2c_msg msg = {AD7993_ADDRESS, 0, two_bytes ? 3 : 2, msg_data};
        struct i2c_rdwr_ioctl_data wr = { &msg, 1 };

        /* open handle */
        if ((file = open(TWI_DEVICE,O_RDWR)) < 0)
                return -ENODEV;

msg_data[0] = (unsigned char) reg;

if ( two_bytes ) {
  msg_data[1] = (unsigned char) ((val&0xFF00)>>8);
  msg_data[2] = (unsigned char) (val&0x00FF);
}
else {
    msg_data[1] = (unsigned char) (val&0x00FF);
}

        if ( ioctl(file, I2C_RDWR, &wr) <0 )
                return -1;
#if DEBUG
//        printf("0x%02x: 0x%04x\n", reg, ad7933_read(reg, two_bytes) );
#endif
```

```
close(file);

return 0;
}


int ad7993_init( void )
{
int retval;

printf("Initializing ad7993...");

    // Enable only channel 3: 48h = 72:
//    retval |= ad7993_write(0x02, 0x48, false);

// Channel 4
//retval |= ad7993_write(0x02, 0x88, false);

// Channel 1
//retval |= ad7993_write(0x02, 0x18, false);


// Channel 1 / 3
//retval |= ad7993_write(0x02, 0x58, false);

// Channel 1-4
retval |= ad7993_write(0x02, 0xF8, false);

    // Enable cycle timer register to Tconvert * 256,
    //retval |= ad7993_write(0x03, 0x4, false); /* Soft reset */

    return retval;
}

int ad7993_read_input(short channel)
{
  //  printf( "\nRequested channel %i, read channel: %i", channel, (ad7993_read(

  // read value
  return ((ad7993_read( (1 << (3+channel)) | 0x00, true) >> 2) & 0x3FF); // remo
}
```

```
#define Uref 3.3
#define Rconst 10000.0

#define A1 0.003354016
#define B1 0.0002569355
#define C1 0.000002626311
#define D1 0.0000000675278
#define Rref 10000

float convert_temp( int input )
{
  float Untc = (input * Uref) / 1024.0;
  float Rntc = (Untc * Rconst) / (Uref - Untc);
  float temp = 1/((A1 + B1*log(Rntc/Rref) + C1*pow(log(Rntc/Rref), 2) + D1*pow(l

  return temp - 273.15; // return in celcius
}

float read_temp( short channel)
{
  int input = ad7993_read_input(channel);

  return convert_temp(input);
}

/*float read_temps()
{
  unsigned char rbuf[8] = {0,0,0,0,0,0,0,0};

  ad7993_read_channels( rbuf );

  for ( int i = 0; i < 4; i++ ) {
    int input = (rbuf[i*2] << 8) | rbuf[i*2+1];
    printf( "\nTemp channel %i: %f", (input >> 12) & 0x03 , convert_temp((input 
  }
}
*/
```