NTNU
Innovation and Creativity

# Developing Embedded Control System Platform using Atmel AVR32 Processor

Using Rapid Prototyping with Matlab Real-Time Workshop

**Øyvind Netland**

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

The goal of this master thesis is to use the new AVR32 processor architecture together with a developed I/O-card in a control system. The control system should be able use Matlab Real-Time Workshop generated code for rapid prototyping.

The assignment consists of:
- Learn how the AVR32 architecture and the STK1000 card works.
- Install developing- and debugging tools for the AVR32 architecture on a Linux workstation.
- Make a basic Linux system for the STK1000 card.
- Implement timers for controlling the periodic Matlab execution, and test how accurate these are.
- Design and implement a I/O-card using an 8-bit AVR microcontroller.
- Design the interface between the I/O-card and AVR32, and implement a driver the AVR32 can use to control the I/O-card.
- Find out how to compile and run Matlab Real-Time Workshop generated code under Linux on the AVR32 architecture.
- Make Matlab Simulink blocks for the I/O-card.
- Make it easy to use Matlab Real-Time Workshop code on AVR32 for rapid prototyping.
- Test and identify the efficiency and limits of the system.
- Make a control system platform with AVR32, that users can build their control system on.
- Make a user manual for the platform that contain the information a user needs to use it, so it can be used with minimal effort.

Assignment given: 08. January 2007
Supervisor: Amund Skavhaug, ITK

# Preface

This thesis has been very interesting, and a worthy end of my time at NTNU. It has been a privilege to work with a new processor architecture, and try to do something that nobody as far as I know have tried before. I would like to thank Amund Skavhaug, my supervisor, and Haavard Skinnemoen from Atmel Norway. Both of them have been helpful whenever I have needed some guidance.

# Abstract

AVR32 is a new processor architecture made by Atmel Norway, and in this thesis it has been used to make a control system platform. The hardware used in this platform is the STK1000, an AVR32 development board and an I/O-card. The I/O-card were developed as a part of the thesis. Software for the platform consists of I/O-card firmware, Linux device driver for the I/O-card and user mode drivers.

The platform supports usage of Matlab Real-Time Workshop as a rapid prototyping tool, that generates code from graphical visualization of mathematical models. S-functions were created so Matlab Real-Time Workshop can control the I/O-card.

The control system platform is documented in an user manual. This manual describes how to install development tools for the platform on a Linux or Windows computer, and how to use the it.

# Contents

# Chapter 1

# Introduction

Today, computer electronics are found almost everywhere. Many ordinary items contain small embedded computers and new cars have computer controlled breaks, fuel injection and stability control. A computer system that controls a physical system is called a control system, and control systems are continuously getting more complex and smaller in physical size.

In 2006, Atmel Norway released a new processor architecture called AVR32. This processor architecture is designed for use in embedded systems, specially small system with low power consumption. Small physical size and low power consumption are very important for many control systems, specially if the system is battery powered.

The main goal of this thesis was to make use of this new architecture to make a platform for control systems. This will use an AVR32 version of Linux as an operating system, since Linux is an operating system that is free, open source and suited for embedded development. Atmel Norway has ported the Linux kernel and many tools to the AVR32 architecture, so it's a natural choice.

A control system has to be able to observe and control the environment. To give the AVR32 this capability, an Input/Output-card has been developed. This card makes the AVR32 able to send and receive analog voltage signal from sensors and actuators, and are the AVR32s extension into the physical world.

To make development of control systems easier on the platform, Matlab Real-Time Workshop was adapted so it can be used as a rapid prototyping tool. This means that code can be generated from Matlab Simulink models, and it runs on the control system. This makes it easy to implement different control systems through the graphical interface of Simulink. The I/O-card can be used in Real-Time Workshop by using the developed Simulink S-function blocks.

To make it easier to use the control system platform, an user manual was written. This contain information about how to install the needed tools and use the control system platform from a Linux distribution or Windows. This is an important part of the thesis, since one of the goals were to make an user-friendly product. The manual should give enough information to use the control system platform. To develop the system further, this report should be read.

The chapters in this report are as follows:

- Chapter 2 Background
  This chapter describes some of the technologies and concepts behind the thesis.
- Chapter 3 AVR32 Linux on STK1000
  This chapter Describes how to install and use AVR32 Linux on STK1000.
- Chapter 4 Preliminary tests of AVR32
  This chapter performs tests to find out the capabilities of the STK1000 development board and the AVR32 processor architecture.
- Chapter 5 Preliminary tests of I/O-card
  This chapter performs tests to find out if correct solutions have been chosen for the I/O-card.
- Chapter 6 Prototype I/O-card
  This chapter describes the design and production of the prototype I/O-card.
- Chapter 7 Design of I/O-card software
  This chapter describes the design of the I/O-card drivers and firmware.
- Chapter 8 Implementation of I/O-card software
  This chapter describes the implementation of the I/O-card drivers and firmware.
- Chapter 9 Testing with prototype card
  This chapter testing the prototype card with the I/O-card drivers and firmware.
- Chapter 10 Final version of I/O-card
  This chapter describes the design and production of the final version of the I/O-card.
- Chapter 11 Matlab Real-Time Workshop
  This chapter adapts Matlab Real-Time Workshop to be used as a rapid prototyping tool for the control system platform.
- Chapter 12 User manual
  This chapter describes how to install the necessary tools and use the control system platform.

# Chapter 2

# Background

## 2.1 Embedded Control System

An embedded control system[14], or control system for short, is a computer system whose main task is to control a physical device or a system. These can vary in sizes, from large and complex systems like a ship or a factory, to smaller devices like a toy robot. A control system will have a predefined task, that won't change over time. This allows a control system to be more specialized than a workstation computer.

### 2.1.1 Real-time Constraints

Control systems often have real-time[18] constraints, which means that an operation has to be done both correctly and within a time limit. Hard real-time constraints means that the system will fail with possible disastrous result if a time limit isn't met. A soft real-time constraint is less serious, and will only lower the quality of the result if failing to meet a time limit.

### 2.1.2 Input/Output

A control system is useless unless it can measure and manipulate its environment, and to do this the system needs to use sensors and actuators. These instruments may have analog or digital interfaces. If they have analog interfaces, which is quite common, they are controlled with analog input and output (I/O). If the instruments are digital, they probably use a digital communication protocol, like RS-232 or USB.

To be able to use instruments with analog interfaces, a computer needs an I/O-card. These cards have a number of analog I/O channels that can be used with instruments with an analog interface. This normally implies that the card can measure the voltage of a signal (*analog input*) or make a signal with a given voltage (*analog output*). I/O-cards often have digital I/O as well, that can be used instead of analog I/O when only digital values (low or high) are used. This is often used together with an analog value, one example is control signals for electrical motor. An analog signal determine the speed or torque of the motor, while a digital signal determines its direction.

## 2.2 Atmel AVR32 Architecture

The new AVR32 microprocessor architecture from Atmel Norway claims to be an architecture for the 21st century[2]. Most other processors increase their throughput[1] by increasing the clock frequency. The AVR32 microprocessor aims to give high throughput with a slow clock. Since power consumption is directly related to clock frequency, this means that it can do the same work with less power. This will be an important characteristic in the future, since we will use more and more gadgets that use battery as a power source, like hand-held music- and video-players.

Low power consumption is also important for control systems. Many systems needs to have low weight, low cost and a long battery time. Since batteries normally are heavy and expensive, both weight and cost will decrease if the power consumption decrease.

### 2.2.1 STK1000

STK1000[8] is a development board for AVR32. STK1000 has some standard I/O connections, like Ethernet, RS-232, mouse and keyboard PS/2. It has an inboard LCD screen and general extension headers where it is possible to connect add-on cards. The STK1000 card comes with a SD flash card with a fully functional Linux system. This is described in more details in 2.3.4. Figure 2.1 on the facing page is an image of the STK1000 board.

### 2.2.2 AT32AP7000

The AT32AP7000 was the first microprocessor with the AVR32 architecture, and it's also mounted on the STK1002 daughter-board. that can be connected to the STK1000 board. The features of this processor are listed below.

- 32 KB on-chip SRAM.
- 16 KB instruction and 16 KB data caches.
- MMU and DMA controller
- Peripherals like audio DAC, LCD controller, USB 2.0 and two Ethernet MACs.
- Serial interfaces like RS-232/USART, TWI (I2C), SPI PS/2.

## 2.3 Linux

The Linux kernel[15] is an open-source and free[2] Unix-like operating system kernel. The kernel is a single binary program that controls the hardware resources of the computer and makes them available for other running programs. Most people think about Linux, they think about a Linux distribution. However, a distribution is actually a collection of

[1] Throughput is a measurement of how much work the processor is able to do.

[2] "*Free software* is a matter of liberty, not price. To understand the concept, you should think of *free* as in *free speech*, not as in *free beer*." is a good explanations on free software given in *The Free Software Definition*[23]

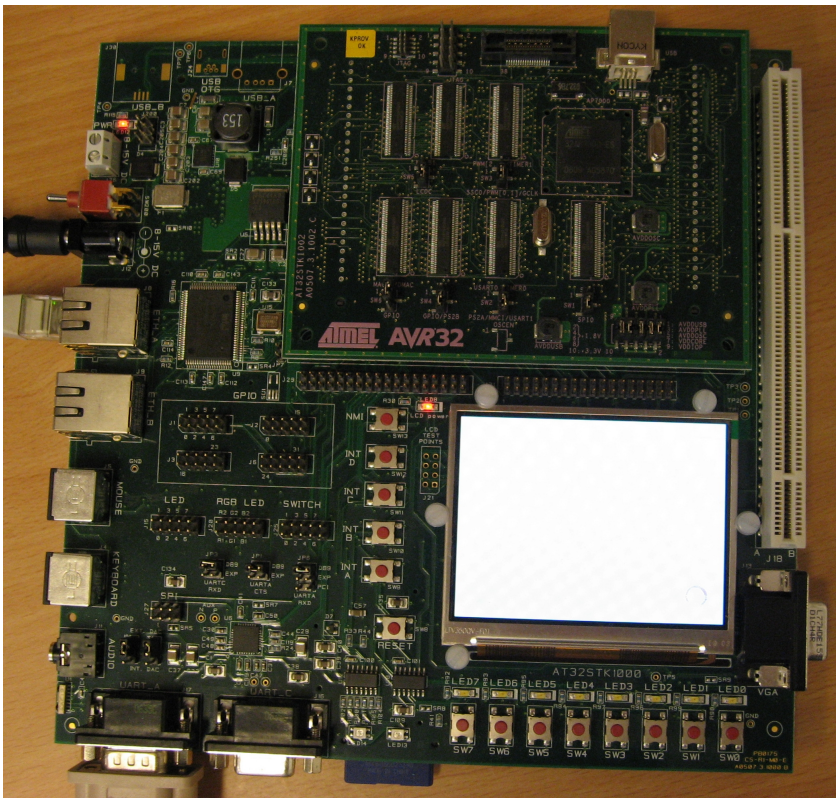


Figure 2.1: STK1000 development board

programs, including the Linux kernel, that can easily be installed on a computer. They may consist of thousands of programs, but it's only the kernel that is called Linux.

### 2.3.1 Linux Control Systems

Linux based operating systems are used in many different systems, from servers and workstations to embedded systems like control systems. In recent years they have been used more and more in embedded systems. This will most likely increase further in the future, as indicated in the article in Datarespons' magazine *Interrupt*[22]. Below is a list over reasons to use Linux in a control system, some of these are also mentioned in the article.

- Linux and many of the programs running on Linux are free and open-source.
- A Linux system is configurable, scalable and able to run on many different hardware platforms.
- Linux is a stable and well tested operating system kernel.
- Linux has many skilled users and developers who are often willing to help.

### 2.3.2 Kernel Mode and User Mode

Most processor architectures has the ability to run code in different levels of privilege. The `x86` architecture has four, called ring 0 to ring 3. The AVR32 architecture[11] has two normal modes, called *Supervisor* and *Application* (equivalent to ring 0 and ring 3). Other modes are reserved for interrupts and exceptions.

As described on page 19 of *Understanding Linux kernel*[20], Linux systems uses two of these levels, *kernel mode* and *user mode*. *Kernel mode* are often refereed to as *ring 0*[19] and is the highest privileged mode. A normal program executes in *user mode* but is able to switch to *kernel mode* when requesting a service that the kernel provides. *Kernel mode* is only used when necessary, and only when allowed by the kernel.

*User mode* programs can switch to *kernel mode* or communicate with the kernel through different interfaces. A *system call* are functions running in *kernel mode* that can be called from *user mode*, and they are described in chapter 10 of [20]. *Signals* are used to send notifications between *user mode* programs, or between the kernel and *user mode* programs, and are described in chapter 11 of [20]. The kernel works as a layer between hardware devices and *user mode* programs, and this is usually done with *Device Drivers*. Which are described in 2.3.3.

### 2.3.3 Linux Device Driver

A *device driver* is code that controls a device, normally this is some kind of hardware. How device drivers works are described in *Linux Device Driver*[21] and in chapter 13 of *Understanding the Linux kernel*[20]. A *device driver* has to be either a part of the kernel or a *kernel module* that can be linked into a running kernel. Devices are identified in the Linux kernel with a *major* and a *minor* number. These are numbers that the device driver has allocated for the devices it controls.

Device drivers are often used to make it possible for user mode programs to use the devices controlled by the driver. A *user mode* program access the driver through special files in the file system, called device nodes. These are normally located in the `/dev` directory. A user mode program can read, write or do other file operations on a node, just as if it was a normal file. When accessing a node with the same major and minor number as a device, the user-space program is actually accessing the driver of this device.

There are two types of device nodes, which corresponds to two different types of devices. Character or char devices are devices that receive or send data serially, while block devices receive or send big chunks of data at the time. Most devices except hard drives and RAM are char devices.

### 2.3.4 AVR32 Linux

AVR32 Linux[3] is a porting[3] of the Linux kernel done by Atmel Norway. This means that it's a version of the Linux kernel that are able to run on this processor architecture.

[3] Porting software is to do modifications so the software can run on other processors or operating systems, further description in [16].

AVR32 support was included in the mainstream Linux kernel 2.6.19 release, but it doesn't contain all the drivers for AVR32 and STK1000 board. To get these, it's necessary to use the AVR32 Linux kernel patches from the AVR32 Linux web page (http://avr32linux.org). Basic system utilities and development tools for the AVR32 architecture can also be found on this web page.

The STK1000 board comes with a SD flash card with a runnable Linux system. This system includes a patched 2.6.16 version of the Linux kernel and *Busybox*[4]. STK1000 uses *U-Boot*[5], which will boot from the SD card by default. By changing the boot arguments in *U-Boot*, it is also possible to boot from network. The advantage of network booting is that the kernel image, file system and programs that all have been developed on a workstation can be used by AVR32 Linux through the network. This makes development much faster since new files and programs don't have to be transferred to a physical medium like a SD card.

To communicate with the AVR32 Linux from a workstation, it's possible to use both a serial connection and telnet via Ethernet. Both of these solutions are available on the preinstalled Linux system on the SD flash card. They both give a text-based terminal just like a normal text-based Linux system would give. The AVR32 Linux system also has a small web server, that in an embedded control system can be used to display a simple web page with status information and measurement logging.

### 2.3.5 AVR32 Linux development tools

Atmel Norway has also ported development tools for use with the AVR32 Linux platform. These tools are available precompiled for different Linux distributions and for Windows, or they could be compiled from source. The development tools for AVR32 Linux includes the tools below.

- Binutils - GNU Binary utilities for handling object code.
- GCC - GNU compiler.
- uClibc - A lightweight version of the standard C-library.
- u-boot - A boot loader for many different processor architectures.
- GDB - GNU debugger that together with a GDB-server on AVR32 can debug code running on AVR32.
- GDB-server - A server version of GDB that can run on the AVR32.

## 2.4 Rapid Prototyping

In computer engineering, rapid prototyping is a development technique that rapidly make a simplified version of a system. This prototype usually implements a part of the complete

[4]Busybox is a lightweight program that includes the functionality of most small Linux applications that an embedded system will need in a single binary file.

[5]U-Boot (Universal bootloader) is a bootloader that can be used on many different platforms, among them the AVR32.

system, that are necessary to test thoroughly. These are often important parts of the system, and the result of the tests can be used to avoid premature decisions. A typical application is to design the user-interface with a rapid prototyping tool. This allows fast and cheap development of a prototype that the users of the system can test and evaluate. User evaluation can often give important information that is difficult to obtain by other means.

### 2.4.1 Rapid Prototyping Control Systems

For control systems, rapid prototyping tools can be used to generate control algorithms from graphical visualization of mathematical systems. It's easier and faster to develop a control system with a tool like this than to write code. It's also easier to change and search for errors. These generated algorithms can be tested both against simulated and actual systems, which is useful during development. Parts of the algorithms can also be used in the complete system.

Two well known tools for simulating and prototyping of control systems are *Matlab* and *Labview*. Matlab was originally a program for matrix computation (the name stands for MATrix LABoratory), but it has evolved into a large collection of mathematical software. *Simulink* is one of the programs in this collection, and it's a program for simulating dynamical system represented by graphical block diagrams. Code can be generated from these block diagrams, making it ideal for rapid prototyping.

Labview uses a data flow language that describes how data flows between different nodes of a system, often a control system. This language is called *G* and are the basis of Labview. The nodes are representations of physical or logical components and can be placed and connected using a graphical interface. The *G* language is a parallel and platform independent[6] language and is well suited for prototyping.

[6]Except some special functions

# Chapter 3

# AVR32 Linux on STK1000

This chapter describes how to install development tools for AVR32 Linux, and how to configure it for development. Some of the problems encountered while working with this version of Linux are described as well.

## 3.1 Development Tools

Installing the development tools are very easy when using the Ubuntu[10] Linux distribution that was used during this thesis. This distribution uses the Apt[17] (Advanced Packaging Tool) package management tool, that can install software from servers defined in the `/etc/apt/sources.list` file. To add the server containing the AVR32 development tools, the following line should be added to this file.

```
deb http://www.atmel.no/beta_ware/avr32/ubuntu/dapper binary/
```

To install the tools, execute the following commands, and answer yes to all questions.

```
sudo aptitude update
sudo aptitude install avr32-linux-devel
```

If this approach doesn't work, or instructions for installing on other platforms, check the AVR freaks wiki `http://www.avrfreaks.net/wiki`.

### 3.1.1 Compiling Development Tools from Source

It's also possible to compile the development tools from source, and this was necessary during the start of this thesis. Then, the precompiled tools had a few bugs. Compiling the development tools from source is still the only option if special features is needed. Instructions for doing this can be found at `http://avr32linux.org/twiki/bin/view/Main/GettingStarted`.

## 3.2 AVR32 Linux Kernel Versions

Some of the early AVR32 versions of the Linux kernel had some problems. Both the 2.6.16, 2.6.18 and 2.6.19 versions were tested. During these tests the following problems was encountered.

- The 2.6.16 version had problems linking with the pthread library. The SPI driver worked, but had problems with stability and returned wrong data.
- The SPI driver did not work in the 2.6.18 version.
- The 2.6.19 version also had problems linking with the pthread library.

It's also possible that the problems with the pthread library were because of uClibc, the library used by AVR32 Linux. After switching to the 2.6.20 kernel version and new precompiled development tools, no kernel or library related problems were encountered.

## 3.3 Configure booting over Network

2.3.4 explains that the STK1000 can be configured to boot over network, and that this is a suitable solution during development.

### 3.3.1 Workstation Configuration

To boot the STK1000 over network, it's required that the workstation has two different servers. A TFTP-server (Trivial File Transfer Protocol) are a simple protocol for transferring small files and are used by the boot loader on STK1000 to download the kernel image. A NFS-server (Network File System) are a standard UNIX server for sharing files, and will be used to share a root filesystem with the STK1000 board. For a Ubuntu workstation these two servers are installed by using `aptitude`, and the following commands.

```
~$ sudo aptitude install tftpd xinetd nfs-kernel-server portmap
```

The TFTP-server are configured by adding the following text into `/etc/xinetd.conf`.

```
service tftp
{
protocol        = udp
port            = 69
dgram           = dgram
wait            = yes
user            = nobody
server          = /usr/sbin/in.tftpd
server_args     = /tftpboot
disable         = no
}
```

The following commands are used to start the server. This is only necessary to do once, afterwards the server starts by itself.

```
~$ sudo mkdir /tftpboot
~$ sudo chmod -R 777 /tftpboot
~$ sudo chown -R nobody /tftpboot
~$ sudo /etc/init.d/xinetd start
```

The TFTP-server are now started and shares the directory `/tftpboot`. If a kernel image are copied to this directory, the STK1000 are able to download and boot this image.

The NFS-server are configured by sharing a directory with NFS. This are done in the `/etc/exportfs` file, and below is an example of a line that will share a directory.

```
/home/oyvindne/master/fs 129.241.187.1/24(rw,no_root_squash,async)
```

To restart the NFS-server with new configuration, the following commands are used:

```
~$ sudo /etc/init.d/portmap restart
~$ sudo /etc/init.d/nfs-kernel-server restart
~$ sudo exportfs -a
```

### 3.3.2 Configure U-Boot

*U-Boot* (Universal Bootloader) are the boot loader used on STK1000. To configure it, the STK1000 has to be connected to a workstation with a serial cable. The Workstation should use a serial terminal program like *minicom* to connect to the STK1000. During start up of the STK1000 board, the `space bar` should be pressed to enter the *U-Boot* command line. Here, the `bootargs` variable has to be changed, and a `tftpip` variable has to be created. The following commands modify these variables. If used on other systems, the IP addresses to the host computer and the path to the NFS-server has to be changed.

```
setenv tftpip 129.241.187.212
setenv bootargs console=ttyUS0 ip=dhcp root=/dev/nfs nfsroot=129.241.187.212:/home/
    oyvindne/master/fs init=/sbin/init fbmem=900k
saveenv
```

# Chapter 4

# Preliminary tests of AVR32

This chapter describes some preliminary tests performed on the STK1000 board before continuing the work. It was important to reveal any weaknesses the AVR32 architecture may have, so the control system platform can be designed to avoid them as much as possible. It's also important to test that planned solutions can be used on AVR32.

## 4.1 Floating-point and Fixed-point operation Test

The AVR32 architecture don't have a hardware floating-point unit (FPU). This means that floating-point operations have to be done with software emulation. Since Matlab RTW generated code normally has many floating-point operations, it's important to know the processors ability to calculate these. The test referred to in this chapter was set up to reveal this. A test for fixed-point calculations was performed as well, and the two results were compared.

The tests consists of 100 million either floating-point or fixed-point multiplications. The test program measures the time used by these operations, and calculate the number of microseconds and clock cycles one operation takes (on average). To measure the time, the `stopwatch` library was used. This library was developed during the thesis, to make time measurements easier, and consists of `stopwatch.h` and `stopwatch.c` that can be found in the appendix C.3.5 and C.3.6.

### 4.1.1 Code

```
1  #include <stdio.h>
2
3  #include "stopwatch.h"
4
5  #define COUNT 100000000
6  #define CPU_CLOCK 140 // Clock frequency of AVR32 on STK1000.
7
8  int main()
9  {
10    long a;
11    float b;
12    long usec, i;
13    float usecPerCalc, ticksPerCalc;
```

```
14      struct sStopWatch watch;
15
16      StartStopWatch(&watch);
17      for(i=0; i<COUNT; i++){
18          a = 64343*52342; // Random integers.
19      }
20      usec = StopStopWatch(&watch);
21
22      usecPerCalc = (float)(usec) / (float)COUNT;
23      ticksPerCalc = usecPerCalc * (float)CPU_CLOCK;
24
25      printf("total time: %i\n", usec);
26      printf("microseconds per calculation: %f\n", usecPerCalc);
27      printf("clock tick per calculation: %f\n", ticksPerCalc);
28
29      StartStopWatch(&watch);
30      for(i=0; i<COUNT; i++){
31          b = 234092098437498.249872398472398*209347802947239.290834732984723; //
                Random doubles.
32      }
33      usec = StopStopWatch(&watch);
34
35      usecPerCalc = (float)(usec) / (float)COUNT;
36      ticksPerCalc = usecPerCalc * (float)CPU_CLOCK;
37
38      printf("total time: %i\n", usec);
39      printf("microseconds per calculation: %f\n", usecPerCalc);
40      printf("clock tick per calculation: %f\n", ticksPerCalc);
41  }
```

### 4.1.2 Result

Table 4.1 shows how the STK1000 preformed on these tests compared to a Dell workstation with a Pentium 4 CPU.

Table 4.1: Results of floating-point (FP) and fixed-point tests

| Platform | CPU | $\mu s$/FP | cycles/FP | $\mu s$/fixed | cycles/fixed |
|---|---|---|---|---|---|
| Dell PC with Ubuntu | $2.4GHz$ | 0.00273 | 6.55 | 0.00269 | 6.47 |
| STK1000 | $140MHz$ | 0.492 | 68.8 | 0.123 | 17.2 |

The Pentium is considerable faster than the AVR32, as expected. It uses the same time for both the floating-point and fixed-point operations. This is because it has a floating-point unit. The AVR32 uses about 4 times longer time on a floating-point than on a fixed-point which was regarded as a positive result. It was a bit concerning that the AVR32 uses about 3 times as many clock cycles per fixed-point operation than the Pentium 4, specially since Atmel Norway promise more throughput with lower clock. This may be due to the employed test program inability to compare one fast and one slow processor with each other.

## 4.2 Timer precision Test

Most control systems have tasks that have to be carried out with regular intervals, every millisecond or shorter. This can be implemented using timers. An easy solution was to use the internal Linux timers, that has a maximum resolution of $1000Hz$. `setitimer()` will start a timer that sends a `SIGALRM` signal periodically. A `signal()` function can register

this signal and select a function that runs each time the `SIGALRM` is received. This is all done inside a user mode program.

The test consists of 10000 periods of $10ms$ each. The reason for not choosing $1ms$ which is the lowest available timer, is that the standard Ubuntu kernel doesn't have the highest precision timer (This can be changed by recompiling the kernel, but the test is just as good with $10ms$). The periodic tasks was created using the `periodicTask` library, that was developed to make it easy to make and start periodic tasks. This library consists of `periodicTask.h` and `periodicTask.c`, that are in the appendix C.3.3 and C.3.4. The `stopwatch` library mentioned in 4.1 was also used.

### 4.2.1 Code

```
1  #include <stdio.h>
2  #include <math.h>
3
4  #include "periodictask.h"
5  #include "stopwatch.h"
6
7  #define COUNT 10000
8  #define PERIODE 1
9
10 int main()
11 {
12    int i, usec[COUNT], totUsec;
13    float totVar, avrUsec, avrVar, avrStd;
14    struct sPeriodicTask task;
15    struct sStopWatch watch;
16
17    InitPeriodicTasks();
18    StartPeriodicTask(&task, PERIODE);
19
20    StartStopWatch(&watch);
21
22    for(i=0; i<COUNT; i++){
23       WaitPeriodicTask(&task);
24       usec[i] = StopStopWatch(&watch);
25       if(i != COUNT - 1)  StartStopWatch(&watch);
26    }
27
28    totUsec = 0;
29    for(i=5; i<COUNT-5; i++){
30       totUsec = totUsec + usec[i];
31       totVar = totVar + pow(((float)usec[i] - 10000*PERIODE),2);
32    }
33    avrUsec = ((float)(totUsec))/(COUNT-10);
34    avrVar = totVar/(COUNT-10);
35    avrStd = sqrt(avrVar);
36
37    printf("Average periode time: %f\n" , avrUsec);
38    printf("Variance: %f\n" , avrVar);
39    printf("Standard derivate: %f\n" , avrStd);
40 }
```

### 4.2.2 Result

In table 4.2 on the next page the results of the test when performed on different platforms are shown.

STK1000 has the best timer precision of the two machines. The average period is less than the correct value, but the difference is not significant. The standard deviation describes

Table 4.2: Results of timer tests

| Platform | CPU | Average | Standard deviation |
|---|---|---|---|
| Dell Workstation running Ubuntu 7.04 | $2.4GHz$ | $10013.09\mu s$ | $2057\mu s$ |
| STK1000 running avr32 Linux | $150MHz$ | $9994.91\mu s$ | $7.25\mu s$ |

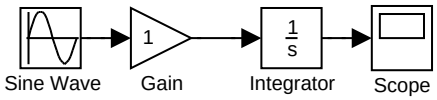


Figure 4.1: First order low-pass passive filter

how much the results differ from the expected value, and it considerable lower for the STK1000. This is because the workstation computer has many programs running, and these will disturb the timer.

## 4.3 Test of Matlab Real-Time Workshop Generated Code

One of the goals of this thesis was to configure Matlab RTW to be used as a rapid prototyping tool for the control system. This made it important to check that it was possible to run RTW code on AVR32. A simple test system was created, consisting of a gain block and an integration block, and generated code from it. The GRT system target was used when generating code, see 11.1.1. Figure 4.1 describes the simple test system.

To compile the code for AVR32, it was necessary to change the generated makefile. Adding `CC=avr32-linux-gcc` to the Makefile (see 11.2.2 for the exact location) specifies that the AVR32 compiler should be used. When running the Makefile with this addition, the Makefile returned the following output.

```
avr32-linux-gcc -c -O -ffloat-store -fPIC -m32    -DUSE_RTMODEL -ansi -pedantic -
    DMODEL=test -DRT -DNUMST=2 -DTID01EQ=1 -DNCSTATES=1 -DUNIX -DMT=0 -DHAVESTDIO
    -I. -I.. -I/usr/local/matlab/simulink/include -I/usr/local/matlab/extern/
    include -I/usr/local/matlab/rtw/c/src -I/usr/local/matlab/rtw/c/src/ext_mode/
    common -I/home/oyvindne/diplom/matlab/simpleTest/test_grt_rtw -I/home/oyvindne/
    diplom/matlab/simpleTest -I/usr/local/matlab/rtw/c/libsrc        rt_nonfinite.c
cc1: error: invalid option '32'
make: *** [rt_nonfinite.o] Error 1
```

The compilation ends with an error, since the AVR32 compiler doesn't understand the `-m32` flag. When removing this from the Makefile, the compilation is successful, and the compiled program runs on the STK1000 card with no problems. Notice that during the first AVR32 compilation in a folder, many object files are compiled, since the default ones from Matlab are all `i386` specific. These files are only compiled once, so future compilations will take much shorter time.

## 4.4 Preliminary Test Conclusion

After doing these tests it was concluded that the AVR32 processor can be used in a control system, even if the floating-point performance wasn't impressive. This weakness implies

that use of floating-point has to be minimized whenever possible. In situations where it's impossible or hard to avoid using floating-point, it can be used, even if it means that the effectiveness of the program is reduced.

The timer test gave a good result, and these timers will be used through the thesis. They give good precision for timers down to $1ms$. The AVR32 will probably be best suited for control systems with higher periods than $1ms$, because of the slow clock and poor floating-point performance. The last test in this chapter confirmed that the AVR32 can run code generated by Matlab RTW, after a few modifications.

# Chapter 5

# Preliminary tests of I/O-card

2.1.2 describes why control systems need I/O, specially analog I/O. Since the STK1000 don't have any analog I/O, a I/O-card was developed. This chapther performs test to find out if the chosen ATmega128 microcontroller is able to work as a controller for the I/O-card, and if the SPI-bus is useable as communication between STK1000 and the I/O-card.

## 5.1 AVR Microcontroller

### 5.1.1 ATmega128

An ATmega128[1] microcontroller was used to control the I/O-card. This is a 8-bit microcontroller from the AVR family from Atmel Norway. ATmega128 is a 64-pin microcontroller with $128kB$ flash and $4096B$ SRAM, making it one of the AVRs with highest specification. Some of the features of the ATmega128 are shown below.

- 8-channel 10-bit ADC (analog input).
- 2-channel 16-bit PWM (Pulse Width Modulator) timers with 3 subchannels each. These can be used as analog output.
- 2 8-bit timers.
- 2 USARTs that can be converted RS-232.
- Several external interrupts.
- SPI (Serial Peripheral Interface).
- TWI (Two-Wire Interface).
- Several general digital I/O pins (digital I/O).

With these built-in features, there was no need for extra analog I/O components. This microcontroller could do both the communication with the STK1000 board and the actual I/O. This made the design of an I/O card easier.

### 5.1.2 STK500/501

The STK500[9] is an AVR development board that is well suited for early stage development of AVR microcontrollers. To use this board with the ATmega128 model, it's necessary to use the expansion module STK501, which supports 64 pins surface mounted AVRs. When using STK500/501 all the pins of the used AVR are available as headers, and all the basic components that the AVR often uses like clock source and RS-232 circuit are ready to use.

This development board was used to test how different features of ATmega128 could be used before making a prototype I/O-card. These test helped minimizing the number of design errors on the prototype.

## 5.2 Analog input

Analog input are used to measure the voltage of signals, and are done with ADCs (Analog-Digital Converter). These are important for a control system, to receive measurement data from sensors. The I/O-card will use the ADC on the ATmega128. This is a 8-channel 10-bit ADC that can measure the voltage between two pins or between a pin and ground. This ADC can convert voltages between $GND$ and $AV_{CC}$ (analog supply).

### 5.2.1 Test of ATmega128 ADC

The ADC of ATmega128 was tested by connecting different voltages from STK1000 ($2.5V$, $3.3V$ and $5V$) to one of the analog in channels. The code below starts an ADC conversion and shows the result (8 MSB) on the LEDs after it was completed. Table 5.1 on the next page shows the results.

```
1  // Includes.
2  #include <avr/io.h>
3
4  // Macros for bit operations.
5  #define set_bit(reg, bit) (reg |= (1 << bit))
6  #define clear_bit(reg, bit) (reg &= ~(1 << bit))
7  #define test_bit(reg, bit) (reg & (1 << bit))
8
9  int main()
10 {
11     // Direction of ports.
12     DDRC = 0xff;
13     clear_bit(DDRF, 0);
14
15     // Initialize ADC.
16     ADMUX = (1<<REFS0) | (1<<ADLAR);
17     ADCSRA = (1<<ADEN) | (1<<ADPS2) | (1<<ADPS1);
18
19     // Start conversion.
20     set_bit(ADCSRA, ADSC);
21
22     // Wait for conversion.
23     while(!test_bit(ADCSRA, ADIF));
24
25     // Set result as output to LEDs.
26     PORTC= ADCH;
27 }
```

Table 5.1: Results of ADC test

| Voltage | Multimeter voltage | ADC result (8 bit) | ADC result in voltage |
|---|---|---|---|
| $2.5V$ | $2.50V$ | 113 | $2.21V$ |
| $3.3V$ | $3.27V$ | 136 | $2.66V$ |
| $5V$ | $5.00V$ | 210 | $4.12V$ |

As seen in table 5.1, the results from the ADC were not very accurate. All results were $80\% - 90\%$ of the correct value. It might be because of an error with the STK500/501 card or maybe the ATmega128.

## 5.3 Analog Output

Analog output is used to make an analog voltage. This is important for control system, so they are able to control actuators. The I/O-card will use PWM (Pulse-Width Modulation) of the ATmega128 to generate an analog signal. PWM is a digital signal with a constant frequency and a controllable duty-cycle. The duty-cycle is a value that describes how much of the period the digital signal is high. A 50% duty-cycle means that the signal is high half of the periode, then low the rest of the periode, and will look like square-wawe signal.

### 5.3.1 Generating Analog Signal from PWM

A low-pass filter is also called an "averaging" filter, and it will convert the PWM signal into an analog voltage equal to the average voltage of the PWM-signal. This analog voltage can be set by varying the duty-cycle of the PWM, making this a digital-to-analog conversion (DAC). An example of a PWM-signal before and after a first-order low-pass filter are shown in figure 5.1 on the next page. As all the figures describing PWM signals shows what happens when a PWM duty-cycle increases from 0% to 50%, which is the same as the DAC increases it's value from $0V$ to $2.5V$.

### 5.3.2 Quality of Analog Signal

The quality of an analog signal made with a PWM can be described with two values. The ripple is how much the output varies, and the response-time is how fast the output change when the duty-cycle change. The second-order filter in figure 5.2 on the following page has ripple marked with green. The ripple are measures as the amplitude of the signal between the green dotted lines, in the figure about $0.15V$. The response time are defined throughout this report as the time used to increase from $0V$ to $2V$ when the DAC increases from $0V$ to $2.5V$. This are marked with red in the figure and is about $300\mu s$. There are many factors that influence the ripple and response time of the DAC, and these are explained below.

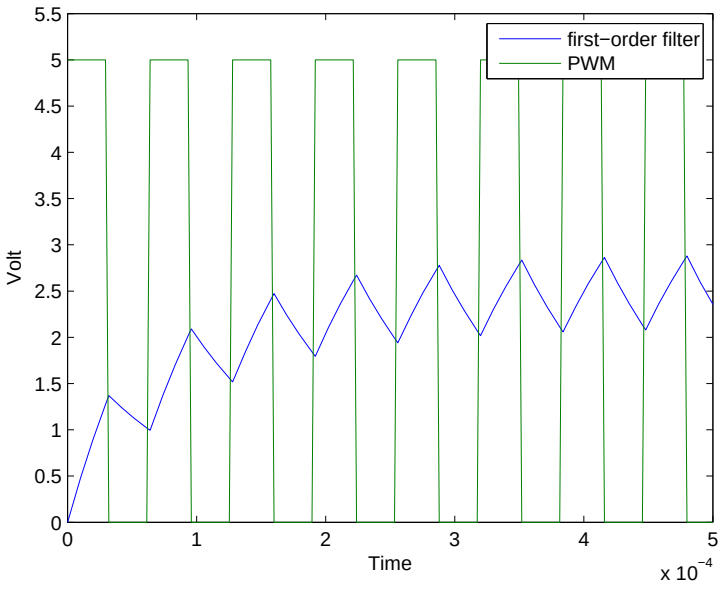


Figure 5.1: PWM signal and PWM signal with first-order low-pass filter.

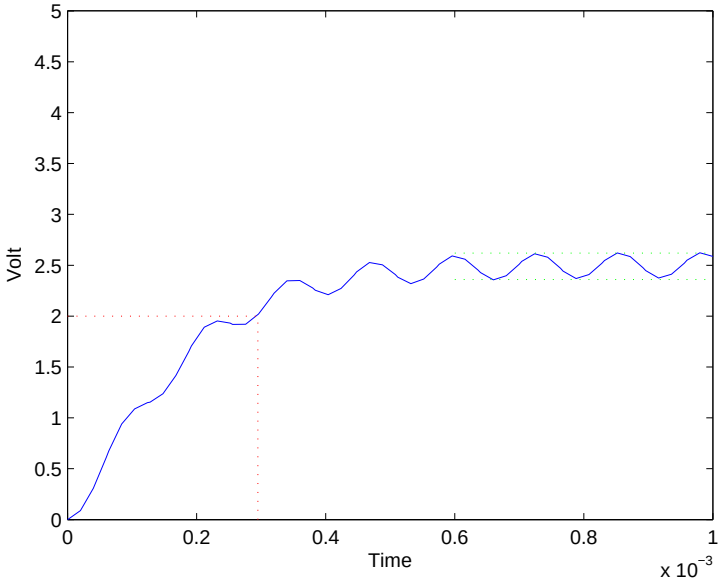


Figure 5.2: PWM signal and PWM signal with first-order low-pass filter.

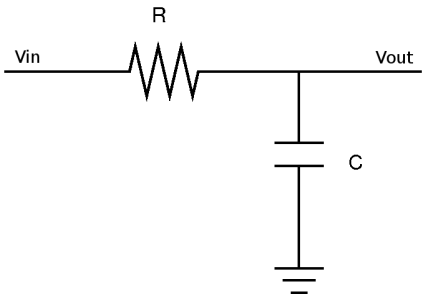


Figure 5.3: First order low-pass passive filter

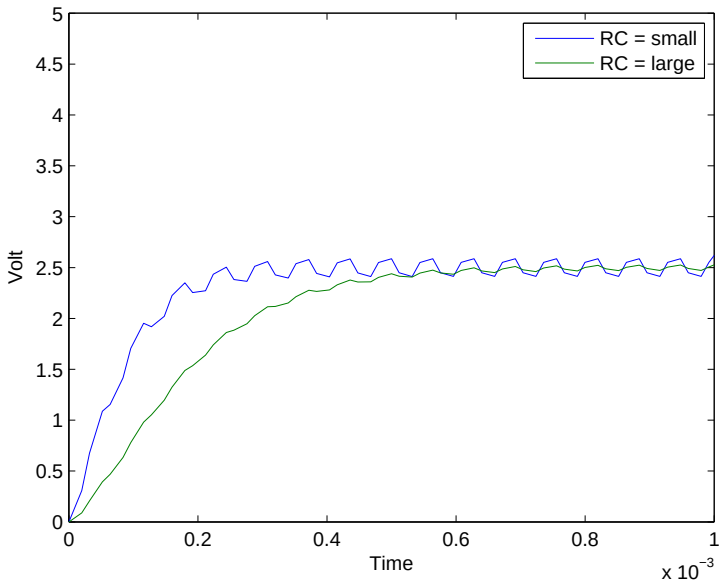


Figure 5.4: PWM signal filtered with filters with different Time constants (RC)

### 5.3.3 Time constant (RC)

A simple passive[1] low-pass filter consists of a resistor and capacitors as shown in figure 5.3. The time constant (RC) of the filter is the product of the resistance and capacitance of these two components, and it describes how the filter works.

When using a low-pass filter to convert a PWM-signal to an analog signal, the time constant defines the response time of the DAC. A low time constant will give a "fast" filter, while a high time constant will give a "slow" filter. A fast filter will increase the ripple of the analog signal. Figure 5.4 shows how two second-order filters with different time constants (RC) filters the same PWM-signal.

[1] A filter is passive if it only consists of passive components like resistors, capacitors and inductors. Passive filters can't amplify signals, only filter out the unwanted frequencies.

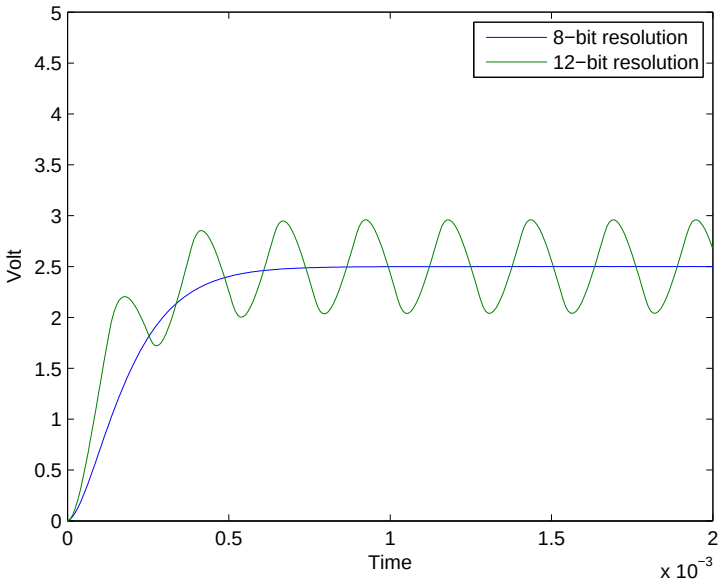


Figure 5.5: 8-bit and 12-bit DAC with equal filters.

### 5.3.4 Frequency and Resolution

The frequency of the PWM-signal depends on the clock frequency of the device (the microprocessor) that makes the PWM and the resolution of the DAC. Equation 5.1 describes how the frequency of the PWM-signal can be calculated.

$$f_{PWM} = \frac{f_{clock}}{2^{bits}} \tag{5.1}$$

A high PWM frequency will give lower ripple than a low PWM frequency. This means that increasing the resolution will increase the ripple. Figure 5.5 shows a 8-bit and a 12-bit resolution DAC filtered with the same filter. It shows that the response time is about the same for both, but the 12-bit resolution result has significant ripple.

### 5.3.5 Order of low-pass Filter

By adding several first-order filters after each other, a higher order filter is created. The second filter will then filter the result of the first filter and so on. Figure 5.6 on the next page shows a DAC with equal (same RC) filters of different orders, and it shows that each filter reduces ripple but increases response time.

### 5.3.6 Test of ATmega128 PWM as DAC

5.3.1 explains how a PWM-signal can be used to make an analog voltage. This should be possible to do with the ATmega128 since it has several timers that can be used as PWM. Timer 1 and 3 was used for this, since these have changeable resolutions up to 16 bits.

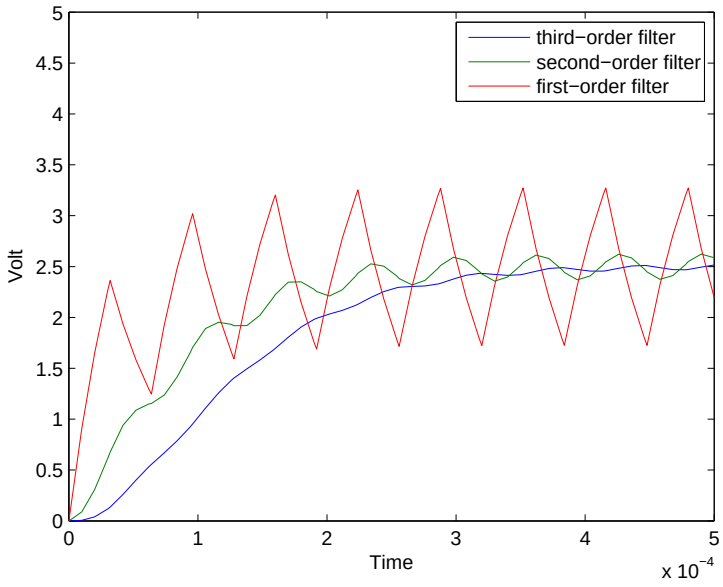


Figure 5.6: DAC with equal (same RC) first, second and third-order filters.

The STK500/501 board was used to test the PWM. This was done by starting PWM on the different outputs, while probing them with an oscilloscope.

The code below starts a 10-bit PWM-signal with 50% duty-cycle to the first output of timer 1. This should give a square-signal with a frequency given by the equation 5.2 on the following page.

```
1  // Includes.
2  #include <avr/io.h>
3
4  int main()
5  {
6      //Set the timer 1A as output.
7      set_bit(DDRB, 5);
8
9      // Set the control registers for PWM to fast PWM, with no prescaler and clear
           output
10     // on compare match (non-inverting mode).
11     TCCR1A =(1<<COM1A1) | (1<<COM1B1) | (1<<COM1C1) | (1<<WGM11);
12     TCCR1B = (1<<WGM13) | (1<<WGM12) | (1<<CS10);
13
14     // Set resolution of PWM to 10.
15     ICR1H = 0x03;
16     ICR1L = 0xFF;
17
18     // Set 50% duty cycle.
19     OCR1AH = 0x02;
20     OCR1AL = 0x00;
21 }
```

With a 10-bit PWM-timer and a clock frequency of $8MHz$, the PWM-frequency are calculated in equation 5.2 on the next page. When running this test, the period of the PWM signal was measured to $130\mu s$ on an oscilloscope, that according to equation 5.3 on the following page is almost the same as the theoretical value.

$$f_{PWM} = \frac{8MHz}{2^{10}} = 7.81kHz \tag{5.2}$$

$$f_{PWM} = \frac{1}{130\mu s} = 7.69kHz \tag{5.3}$$

## 5.4 SPI

To be able to use the I/O-card, the AVR32 needs a way to communicate with it. SPI is a serial bus that's often used for communication between different integrated circuits on the same board. It's an simple yet effective bus that is full duplex, meaning that data are transferred in both directions at the same time. Both AVR32 Linux and the ATmega128 supports SPI, and this bus will be used for data transfer between the STK1000 board and the I/O-card.

Another serial bus called I2C (Inter-Integrated Circuit) or TWI (Two-Wire Interface) was also considered, and was also supported by AVR32 Linux and ATmega128. This bus does about the same as SPI, but the differences described in 5.4.1 made the SPI a better choice.

### 5.4.1 SPI bus

A SPI bus consists of one master and one or more slaves. The master can communicate with one slave at the time by using the slaves "chip-select" signal. This is different than I2C, that uses addressing to decide which device that should receive the message. By using "chip-select" signals the SPI doesn't need to start each data package with an address, meaning that a SPI package only contain data, while a I2C package contain an address and data. This makes SPI more effective when only a few devices uses the bus.

When more devices uses the bus, SPI won't be a good choice. This is because each device needs its own "chip-select" signal, meaning a dedicated pin on the master for each of the slaves. It's also possible to use an additional component that decodes an address to several slave select signals.

The SPI bus consists of four types of signals describes below. All the devices on the bus also needs common ground.

- MOSI (Master Out Slave In) sends data from the master to the slave.
- MISO (Master In Slave Out) sends data from the slave to the master.
- SCK (Serial ClocK) is the common clock sent by the master. Since it's a common clock on the bus, the bus is synchronous.
- CS (Chip Select) or SS (Slave Select) is a signal that master send to start a transfer with a slave.

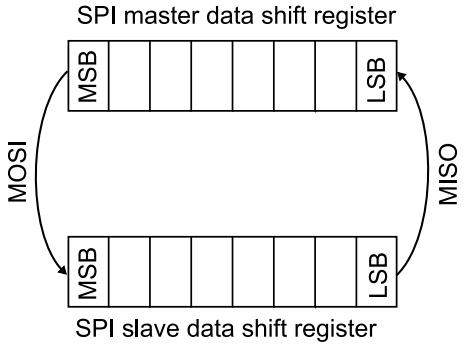


Figure 5.7: How data are transferred between SPI master and slave.

### 5.4.2 SPI transfer

When using SPI it's important to know how the data is transferred. Each device on the bus has a SPI data register, and during a transfer between two devices the content of their SPI data registers is exchanged. This is described in figure 5.7. It means that before the transfer both devices need to place what they want to send in their data register, and after the transfer the data in the register is the received data. Since data is sent both ways at the same time, SPI is full duplex.

If the master just wants to send data, not receive, the received data can be dropped. If the master wants to receive data, a dummy byte[2] has to be sent, since the slave can't send anything when the master doesn't.

The slave can't initialize a data transfer, and this may give some problems. Specially if the slave has to complete something, and then report back to the master. This can be solved by letting the master poll[3] the slave until it is finished. This will slow down both the master and slave. A more effective solution is to let the slave trigger an external interrupt on the master when it's finished.

## 5.5 Test of SPI communication between AVR32 and ATmega128

By connecting the STK1000 and STK500 boards, it was possible to test SPI communication between the two processors. It was important to find out if SPI would work before designing the I/O-card.

[2]A dummy byte is a byte that contain data of no value, and is just used to request data from the SPI slave. The dummy byte can have any value, but should be chosen to a value that the slave won't believe is something else.

[3]In this context, polling is to periodically ask if something is finished until the answer is yes. This is not a very sophisticated method, but it's usually the easiest to use.

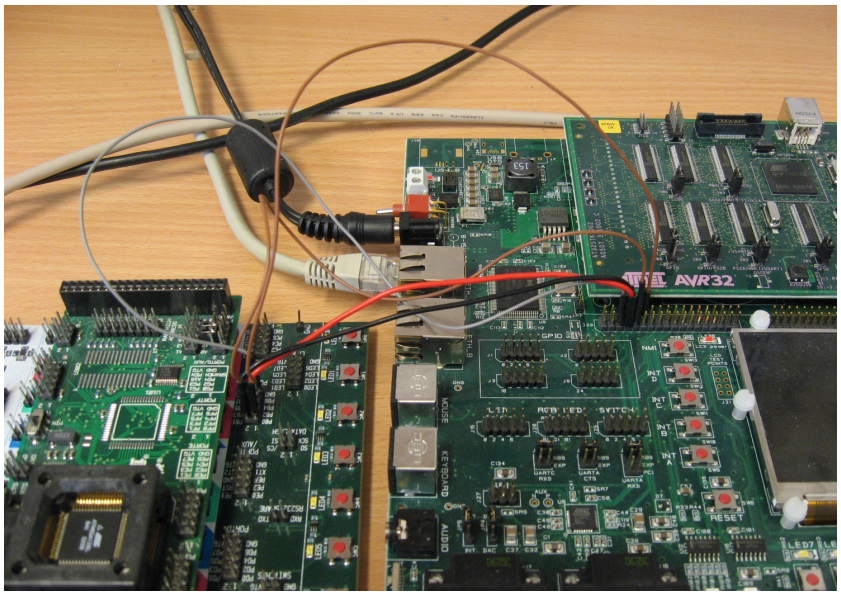

Figure 5.8: Image of STK500 and STK1000 with SPI connection

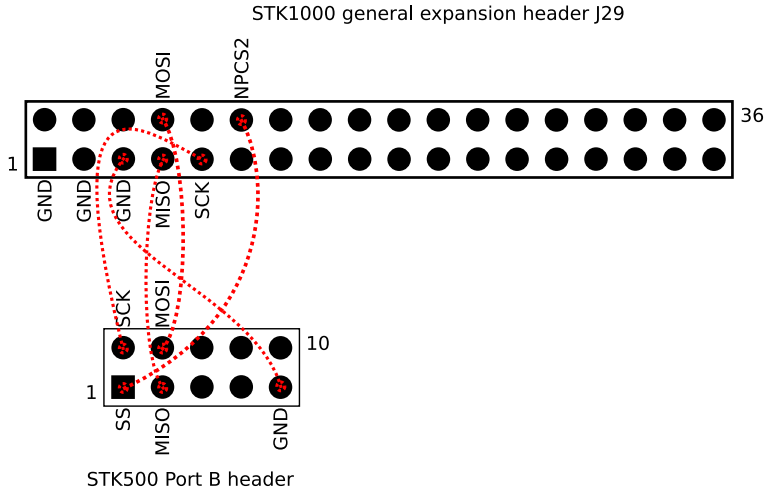


Figure 5.9: SPI wires between STK500 and STK1000.

### 5.5.1 Hardware setup

SPI signals are available on headers on both STK cards, and they were connected with short cables, like on the figure 5.8. Figure 5.9 shows which pins on the header that are used. The signals are connected as follows: MOSI, MISO and SCK should be connected to the same signal, while the NPCS2 on STK1000 should be connected to the SS signal on STK500. The two processors must also have common ground to communicate with SPI.

### 5.5.2 AVR32 as SPI master

As most hardware, the SPI driver can't be accessed directly from user mode. To make it possible, a device driver has to be made. 8.2 describes how the device driver for the

I/O-card was implemented, and the device driver used for this test was similar. During the test the device driver sends the value 222 to the SPI slave, and it prints out the returned value.

### 5.5.3 ATmega128 as SPI slave

SPI with ATmega128 is used by reading and writing three SPI-specific registers. The control register (SPCR), the status register (SPSR) and the data register (SPDR). How these registers are used in slave mode is described below.

- SPCR
  - Bit 0 (SPR0) and bit 1 (SPR1) aren't used in slave mode
  - Bit 2 (CPHA) and bit 3 (CPOL) defines the SPI-mode. The different SPI-modes have different timing between the clock signal and the data signals. All devices using the same SPI-bus need to use the same mode.
  - Bit 4 (MSTR) selects slave mode when written to zero.
  - Bit 5 (DORD) selects if the LSB or MSB should be sent first during transfer. All devices using the same SPI-bus should use the same data order, or data needs to be reversed in software.
  - Bit 6 (SPE) turns on SPI for this device.
  - Bit 7 (SPIE) turns on SPI-interrupts for this device.
- SPSR
  - Bit 0 (SPI2X) isn't used in slave mode
  - Bit 1 to 5 are reserved bits.
  - Bit 6 (WCOL) is set if the data register is written to during transfer.
  - Bit 7 (SPIF) is set when a SPI-transfer is completed. Will trigger an interrupt if bit 7 of SPCR is high.
- SPDR is used both to send and receive data. The data in the register before transfer will be sent to the master, while the data in the register after transfer is the data received from the master. Writing or reading to this register clears bit 7 of SPSR.

The simple program below was used, to test the SPI communication. It waits for a SPI transfer initialized by a SPI master. The slave will send the value 111 to the master and it will use the LEDs on the STK500 to display the received data. This confirms that the SPI communication works both ways.

```
1  // Includes.
2  #include <avr/io.h>
3
4  int main()
5  {
6      //Initialize SPI.
7      SPCR = (1<<SPE);
8      DDRB = (1<<3);
9
10     //Initialize LEDs.
```

```
11      DDRD = 0xff;
12      PORTD = 0;
13
14      //Set data.
15      SPDR = 111;
16
17      //Wait for transfer.
18      while(!(SPSR & (1<<SPIF)));
19
20      //Send received data to LEDs and loop forever.
21      PORTD = SPDR;
22      while(1);
23  }
```

### 5.5.4 Result

The test of a simple SPI transfer between the two STK boards was successful. This means that it will be possible to use SPI as communication between AVR32 and an I/O-card. The highest SPI frequency that worked was measured to about $1.90MHz$ with an oscilloscope. According to the ATmega128 data sheet[1] both the low and high period of the SCK signal has to be longer than 2 clock cycles. When the clock frequency is $8MHz$ this means that the highest theoretical frequency is $2MHz$, that are close to the measured value.

# Chapter 6

# Prototype I/O-card

This chapter desribes the design and production of the first of two I/O-card versions produced during this thesis. The first card is called the prototype card, and was designed to be ideal for testing and debugging.

## 6.1 PCB Software

A PCB (Printed-Circuit Board) is an epoxy bonded fiberglass sheet with copper layers that can be etched or milled off and leave electrical circuits. Most electronics are implemented on PCBs, and they may have different numbers of layers, and each layer can contain circuits.

There are a number of different software solutions for designing PCBs. In this thesis the freeware version of Eagle[4] was used. This program has a limitation on the size of the finished card, and it can't design cards with more than two layers. This wasn't a problem, since the maximum size ($8x10cm$) and two layers were enough for the purpose.

## 6.2 Features

This card was designed to have these features:

- ATmega128 with 16 MHz crystal as controller.
- JTAG-interface for ATmega128.
- Reset button for ATmega128.
- RS-232 connection for debugging to a serial port on a workstation.
- Header ready to connect to general expansion header on STK1000 with power-supply, SPI and interrupt signals.
- Jumper for choosing between external power-supply and power-supply from STK1000.
- 2 AVR32 interrupts that can be triggered from ATmega128.

- 2 AVR32 interrupts that can be triggered externally.
- 2 ATmega128 interrupts that can be triggered externally.
- 6 channel 8 to 12-bits analog output.
- 8 channel 10-bits analog input.
- Common ground for both STK1000 and STK500.
- 8 channel digital output.
- 8 channel digital input.
- SPI signals available on ”debug” header.

## 6.3 Components

The components used for this card:

- ATmega128 microcontroller ([1]).
- MAX233 UART to RS-232 IC ([5]).
- Two $100nF$ capacitors for decoupling ATmega128 and MAX233).
- Linear voltage regulator (7805) with capacitors ([7]).
- $100nF$ capacitor and $10\mu H$ inductor for LC filter on analog supply.
- $16MHz$ crystal and two $18pF$ capacitors for crystal circuit for ATmega128.
- One button, $10k\Omega$ resistor and $100nF$ capacitor for reset button for ATmega128.
- Six second-order RC-filter for filtering PWM signal to analog value. Two resistors and two capacitors for each filter.
- 3 operational amplifiers (MC1458) ([6]).
- Screw clamps for connecting different signals.
- 2x5 male header for JTAG connection.
- 2x18 male header for STK1000 connection.
- 1x3 male header with jumper for power-supply selection.
- 1x8 header for debug signals.

## 6.4 Schematic

The schematic of a circuit is a logical representation of how the different parts connects to each other. When designing a circuit, it's normally best to start with this. The whole schematic are in the appendix B.1. Different sections of the circuit are presented below. Be aware that these are the orignal schematics of the prototype, and have errors described in 6.7.

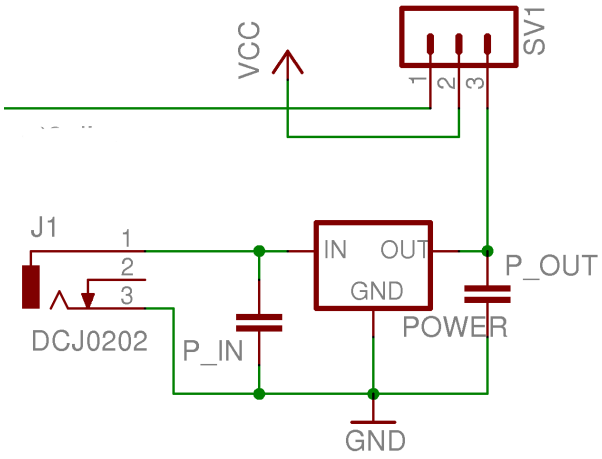


Figure 6.1: Power circuit

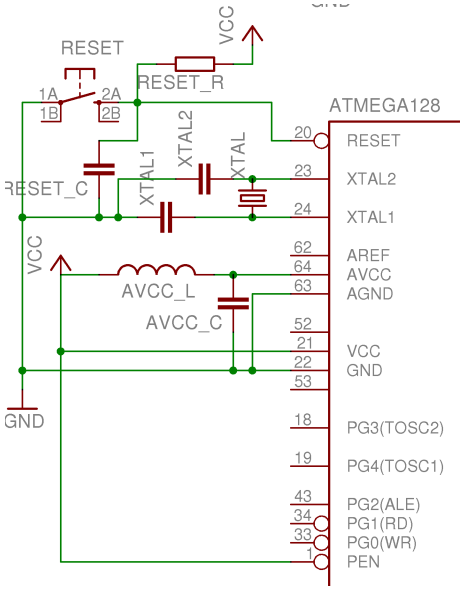


Figure 6.2: Reset, crystal and analog reference circuits for ATmega128

### 6.4.1 Power Circuit

Figure 6.1 shows the Power circuit used for the I/O-card. To the left are the power jack connector, for connecting a AC/DC adapter. In the bottom right corner, there is a linear voltage regulator `7805`[7] that gives out a stable $5V$ voltage source. This component needs a capacitor on both input and output to work properly. On the top there is a 3-pin header. A jumper is used to choose what kind of voltage source the rest of the card will use, either the output from the voltage regulator or the $5V$ signal from the STK1000 card.

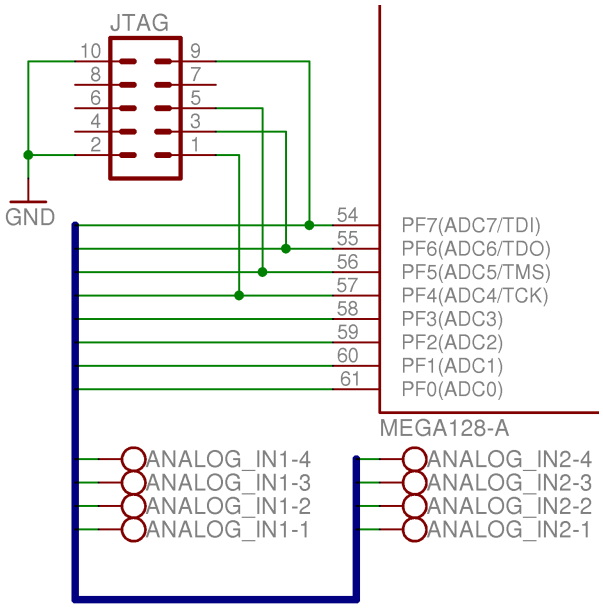


Figure 6.3: JTAG header and analog input connectors

### 6.4.2 Reset, Crystal and Analog Supply Circuit

Figure 6.2 on the previous page shows the reset, crystal and analog reference circuits for the ATmega128. The reset circuit provides a button that will "pull-down" the active-low reset port of the microcontroller when pushed. This will reset the processor. This circuit also has a "pull-up" resistor that makes sure the reset port is high when the button isn't pushed.

The crystal circuit was made to give the ATmega128 a stable and high frequency clock source. The main reason for this is that a higher clock frequency will make the PWM signal better suited as a base for making analog out signals.

The analog supply is necessary to make the ATmega128 able to read analog input signals. The crystal and analog supply circuits are implemented after specification given in the ATmega128 data sheet [1].

### 6.4.3 JTAG header and Analog Input Connectors

Figure 6.3 shows the JTAG header for the ATmega128 and screw clamps for the analog input signals. The JTAG header was added to allow a AVR JTAG ICE (either original or mk2) to connect, program and debug the ATmega128. The analog input screw clamps was mounted at the side of the I/O card, and makes it easy to connect wires with analog signals to the ADC of the ATmega128. Four of ATmega128s ports was used both by the JTAG header as analog input. This shouldn't be a problem as long as no analog signals are connected to these ports when using the JTAG header.

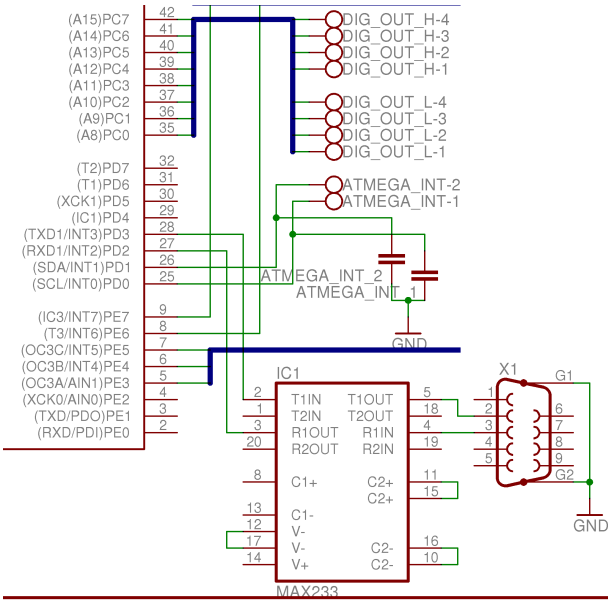


Figure 6.4: RS-232 circuit and digital output connectors

### 6.4.4 RS-232 circuit and Digital Output Connectors

Figure 6.4 shows a circuit for RS-232 interface for ATmega128 so it can connect to a serial cable on a PC. The microcontroller has two UART signals, and one of these was converted into a RS-232 signal with the MAX233 ([5]) from Maxim. This signal was made available on a 9-dsub connector, that connects to a normal serial cable.

The figure also shows screw clamps for digital out signals of the I/O-card. Like the analog in, these will be placed on the edge of the card.

### 6.4.5 STK1000 headers and Digital Input Connectors

Figure 6.5 on the following page shows the circuit for STK1000 headers and digital in signals (same as for digital output). A 36 pin header are used to connect to one of the expansion headers on the STK1000 board. This header includes signals for SPI, interrupts and ground. The I/O-card can also draw power from the $5V$ source from STK1000. Two of the interrupts are connected directly to the ATmega128, and two are available on screw clamps.

To make debugging simpler, a header was placed on the card that had pins for the different SPI signals. These pins makes it easy to probe these signals and look at them on an oscilloscope.

### 6.4.6 Decoupling Capacitors and VCC/GND Connectors

Figure 6.6 on the next page shows a small but important part of the I/O-card. This is the two capacitors that are used for decoupling the ATmega128 and MAX233. Decoupling is important since it ensures that these ICs gets a stable power supply. The figure also shows

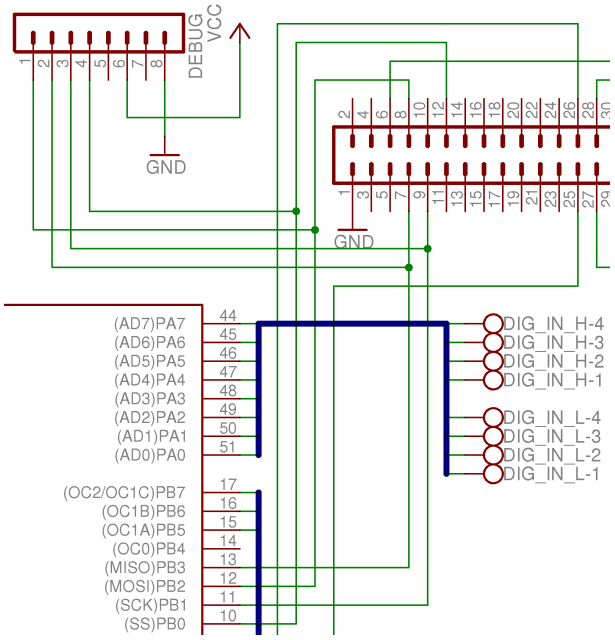


Figure 6.5: STK1000 headers and digital input connectors

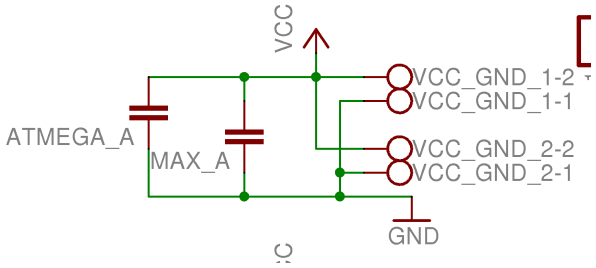


Figure 6.6: Decoupling capacitors and VCC/GND connectors

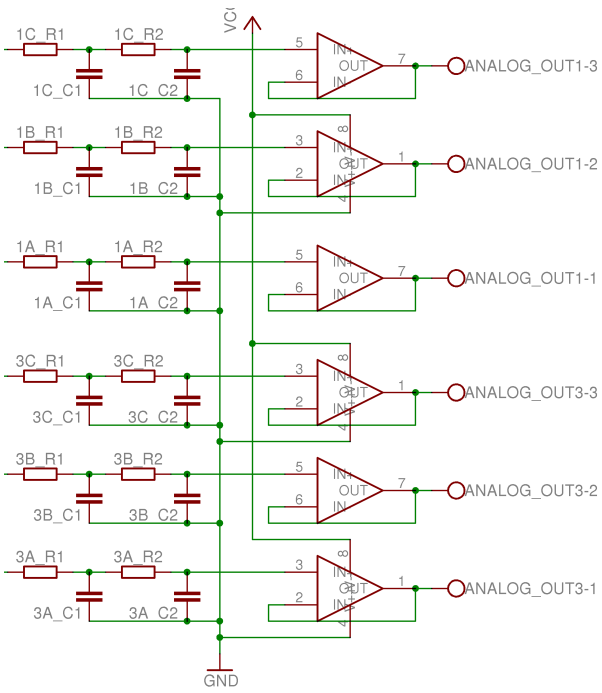


Figure 6.7: Analog out circuit

two screw clamps that has VCC and GND signals. These signals are available for devices connecting to the I/O card, and are necessary because all I/O signal needs a reference.

### 6.4.7 Analog Output Circuit

Figure 6.7 shows the filters and operational amplifiers (op-amp) that are used to convert the PWM-signal into an analog voltage. Each of the 6 analog output subchannels has its own second-order passive low-pass filter, that smooth the PWM signal to a constant analog voltage between $0V$ and $5V$. These filters consists of two resistors and two capacitors each. The 6 subchannels can have different filters, and since this is a prototype card, it's imperative that several different filters are tested.

The reason for choosing a second-order filters are that it will give a good compromise between ripple, response and physical size. A first-order filter would give too much ripple, while a higher order filer would take too much physical space. Figure 6.8 on the next page shows that by using different cut-off frequencies it's possible to get similar behavior from a second and a third order filter. This means that higher order filter not necessarily gives a better result.

After each filter, an op-amp circuit is placed. These are voltage followers, which means that their output voltage is the same as their input voltage. They are not used for amplifying the voltage, but they are used to amplify the current, since an op-amp can supply a lot more current than an ATmega128 can. The op-amps high input impedance, means that the current flowing from the ATmega128, through the filter and to the op-amp are minimal. This is important since the properties of passive filters depends on the current flowing through. High current, will mean a high voltage loss in the resistors of the filter.

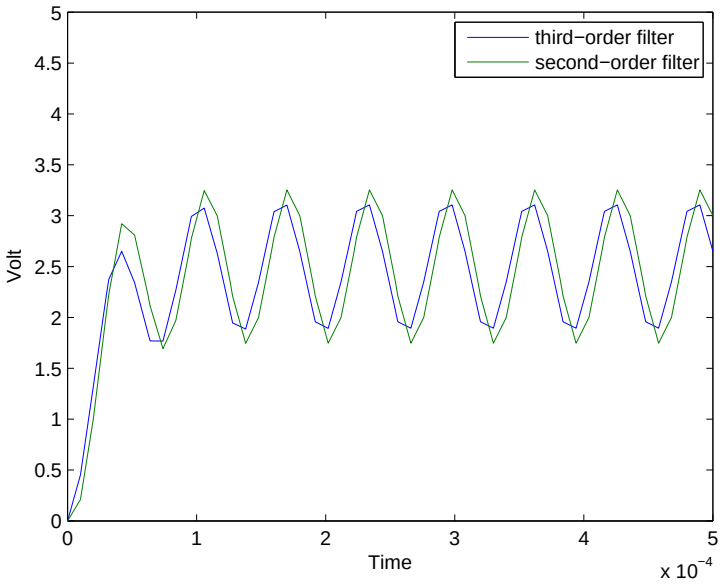


Figure 6.8: Second-order and third order low-pass filters with equal ripple and response time

## 6.5 Layout

The layout of a circuit is a representation of how the different parts are placed on a PCB, and how the copper layers should be etched. To ensure that the parts are connected the same way as in the schematics, Eagle uses the schematics as a template. When making the layout, it's important to make the connections between the different parts as straight and short as possible. This will make assembly easier and it will minimize noise problems.

Figure 6.9 on the facing page shows the top side of the first prototype, and figure 6.10 on the next page shows the bottom side. Below are some important design choices are commented.

- The ATmega128 was placed on the bottom layer. This was because it's a surface-mounted device, and many of its pins were connected to parts that isn't surface mounted. By placing it on the bottom, the number of vias [1] was reduced, since most signals started and ended up on the same layer.
- All the digital I/O, analog I/O and external interrupts was connected to screw clamps, to give easy access to them and the ability to connect wires easily. This did unfortunately take a lot of space on the card.
- Only the ATmega128 was a SMD. This was done to make it easier to debug and to make small changes to the board. The final version of the card, should use more SMD components, since it will allow a smaller layout.

[1] Vias are small metal cylinders that can be placed inside a hole on the PCB. This allows electrical connection between top and bottom layer. Vias should be avoided when possible, since they increase assembly time and noise.

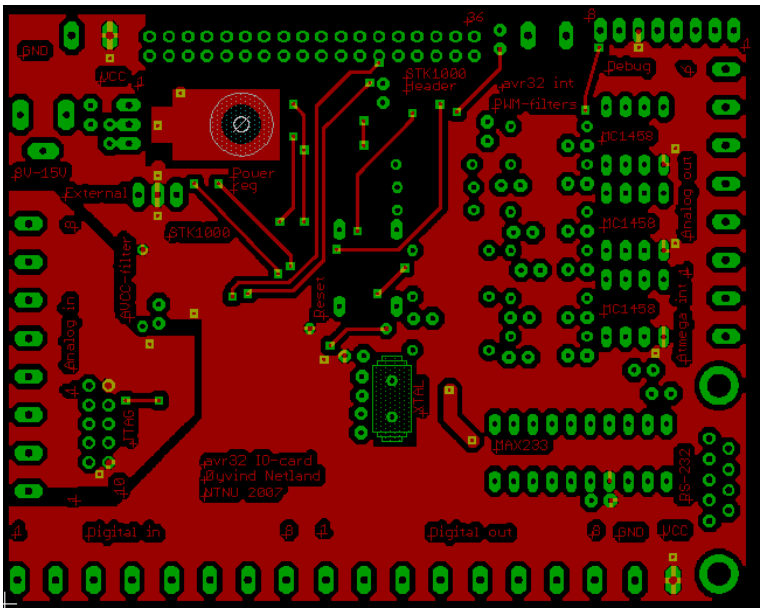


Figure 6.9: Top layer of prototype card. (Not to scale)

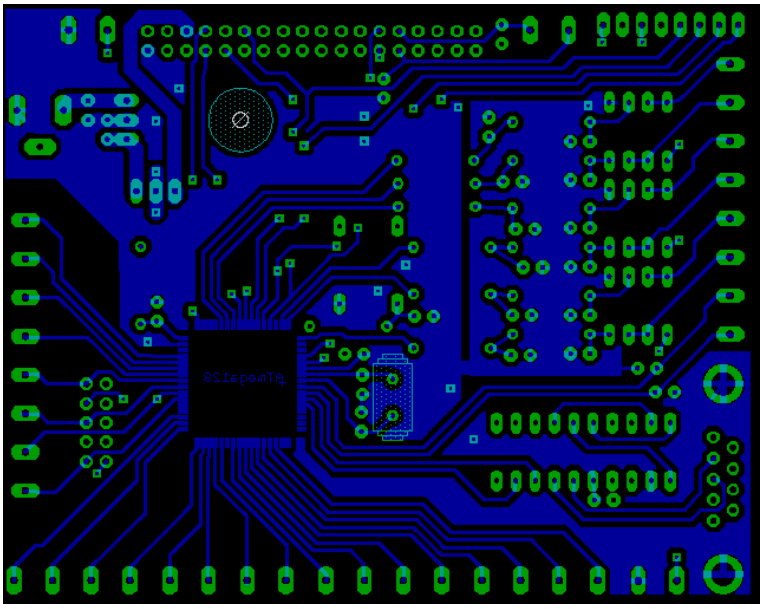

Figure 6.10: Bottom layer of prototype card. (Not to scale)

- The analog signals were kept as isolated as possible from the digital signals. This is because fast changing digital signals can induce noise on analog signals. The analog signals was also placed directly above or below a ground plane to protect them. These ground planes only had a minimal connection to the rest of the ground plane.

## 6.6 Assembly and test

The layout from 6.5 was used to etch a PCB card. The rest of the card was assembled by soldering the components on to the PCB. The assembly of the card should be done step by step, and each step should be tested before moving to next step. By doing this, each part of the circuit can be tested as isolated as possible. And it's also possible to know, to a certain degree, that the components that are assembled and tested are working as planned.

6.6.1 to 6.6.6 describes the steps of he assembly, and the tests done after completing each step. Before soldering anything to the card, the card should be disconnected from the STK1000 card and from power supply. While soldering it's easy to make short circuits, which should be avoided.

### 6.6.1 Power Regulator Circuit

The power regulator circuit consists of a power jack, a voltage regulator, two capacitors and a 3 pin header used for selecting power supply. It was tested by connecting a $9V$ source to the power jack, and selecting external power supply. The output voltage was measured to $5.0V$, which means that the voltage regulator are working.

### 6.6.2 ATmega128 and JTAG Interface

After soldering the ATmega128, it's important to check that there are no short circuits between its pins. After checking this thoroughly, the JTAG header and decoupling capacitor was connected. The different functions of the ATmega128 will be tested when they are needed. The first test was to connect to the ATmega128 through the JTAG-interface. The `avrdude` command below connects to the ATmega128 and writes some info about it. This command returned correct information about the ATmega128, which means that basic operations of the ATmega128 works, and that the JTAG interface is able to program it.

```
sudo avrdude -P usb -pm128 -cjtag2 -v
```

### 6.6.3 Crystal Circuit

The crystal circuit consists of a crystal (16MHz) and two capacitors. To use this circuit, the fuse bits[2] of the ATmega128 has to be changed. The `avrdude` command below does this.

[2]Fuse bits are bits in AVR microcontrollers memory that defines clock source and other important characterisitcs of the AVR. These can be written are read with a programming toll like the JTAG.

When starting up the ATmega128 with the crystal circuit and new fuse bits, the crystal pins were probed with an oscilloscope. The result was a sinus signal with a frequency of $16MHz$, which means that the crystal circuit works.

```
sudo avrdude -P usb -pm128 -cjtag2 -U lfuse:w:0xFE:m -U hfuse:w:0x89:m -U efuse:w:0
    xFF:m
```

### 6.6.4 RS-232 and Reset Circuits

These two elements were best to test together. The RS-232 circuit consists of the MAX233 IC and a 9 dsub connectors (connector used by serial port of a standard computer). The reset circuit consists of a button, a resistor and a capacitor. The RS-232 circuit was tested by sending a byte of data from the card, when it was connected to the workstation with a serial-cable. The code below sends one byte with a baud rate of $19.2kbps$. When running this code, the workstation received the byte when using a program like *minicom*. The reset circuit worked since the same byte was received by the workstation each time the reset button was pressed.

```
1  // Includes.
2  #include <avr/io.h>
3
4  int main()
5  {
6  // Set direction registers
7  set_bit(DDRD, 3);
8  clear_bit(DDRD, 4);
9
10 // BAUD rate = 19.2k
11 UBBR1H = 0;
12 UBBR1L = 51;
13
14 // Enable both transmit and receive
15 UCSR1B = (1<<RXEN1) | (1<<TXEN1);
16
17 // Use 8 bit data, no parity and 2 stop bits.
18 UCSR1C = (1<<UCSZ11) | (1<<UCSZ10) | (1<<USBS1);
19
20 // Start sending by writing to the send data register.
21 UDR1 = 'A';
```

### 6.6.5 Headers for STK1000 and debugging

These headers were tested by connecting the header to the general expansion header (J29) on the STK1000 board, and try the same test as in 5.5. Since the SPI communication also worked through the header, it meant that the header are connected properly. To check if the STK1000 can be used as power supply for the I/O card, the jumper was from external power supply to STK1000. This worked, which meant that the STK1000 can supply the ATmega128 with power.

### 6.6.6 ADC Supply Filter

The filter for the ADC supply is a LC-filer, and it's used to filter out digital noise, so the analog reference is kept as stable as possible. This is to ensure that the analog in is

as accurate as possible. On this prototype card, the inductor of the LC-filter was short circuited, since no inductor of correct value was available. The inductor works as a short circuit for DC current, but it stops AC current. So replacing the inductor with a short circuit will just make the filter stop high frequency noise less effective.

### 6.6.7 Analog Output Circuit

The analog output circuit was assembled with different filters, that were used to find a filter for the final I/O-card. 9.2.2 describes the different filters that were tested and the results.

## 6.7 Errors in Schematic

During the assembly and test of the prototype, the following errors was found in the design.

- $V_{CC}$ of the JTAG-interface wasn't connected to the $V_{CC}$ of the board.
- The operational amplifiers don't work as they should, see 9.2.
- The 9-dsub connector for the RS-232 connection weren't connected correctly. Pin number 5 should be connected to ground, and the cable shield shouldn't.

# Chapter 7

# Design of I/O-card Software

This chapter describes the design of the different parts of the system, and how these should work togeter. The system consists of firmware running on the I/O-card, a device driver running in kernel mode in AVR32 Linux and a user mode driver that makes user mode programs able to use the I/O-card.

## 7.1 Communication Protocol

The communication between the AVR32 (master) and the ATmega128 (slave) uses SPI, and it's a very central part of the system. Before designing the software a protocol was designed to define the rules of communication between the master and slave. Below are some terms that are used in the description of the protocol.

- A "transfer" is a single exchange of data between the master and the slave.
- A "command" is a task the master wants the slave to do.
- A "message" is a set of transfers that makes sure that the slave receives a command, and returns data if necessary.
- A "message structure" is the way the set of transfers are organized in a message. The message structure also makes sure that an error in one of the transfers will be detected.

### 7.1.1 Commands

The protocol has a number of defined commands. Some of them are data commands, that are used by the master to either read an input or write to an output of the I/O-card. Another type of commands are control command that change the internal settings of the I/O-card.

One of the commands listed here are marked with "buffering only", which means that it will only be used in one of the two possible solutions defined in 7.2.2.

Data commands:

- *Get analog input* is the command for reading an analog value.
- *Set analog output* is the command for writing an analog value.
- *Get digital input* is the command for reading all of the 8 digital input channels at once.
- *Set digital output* is the command for writing one or more of the 8 digital output channels.

Control commands:

- *Set analog input enable* is the command for enabling/disabling an analog input channel. (buffering only)
- *Set analog output resolution* is the command for changing the resolution of an analog output channel.

### 7.1.2 Message Structure

The communication protocol also defines two message structures that a message containing a command has to follow. There are two structures because commands that send data to the slave (set digital output) will need a slightly different message structure than those requesting data from the slave (get digital input). None of the commands are required to both send and request data from the slave.

If either the master or the slave fail to send the bytes required during all the transfers of a message, the part that detected the error will stop sending, and the other part will then be aware the error. This ensures that both parts will know if something went wrong while sending the message.

Figure 7.1 describes the message structure for commands where the master sends data. The message structure is further described below.

- The synchronization phase has to make sure that both the master and slave are ready for the command, and that they are synchronized. If not synchronized it's possible that the slave will interpret the data the master sends wrongly. To synchronize, the master initially sends the command code from 7.1.4. If the slave returns a code that confirms it's ready for transfer, then the synchronization was successful. If it returns something different, then the master has to keep sending the code until the slave returns the ready code. In figure 7.1 this is represented by the first message, which is unsuccessful, since the slave returns something unknown. The second transfer is successful, since the slave returns the ready code.
- In the master data phase, the master sends data to the slave. Every time the master sends a new byte to the slave, the slave will return an acknowledgment of the previous byte sent by the master. How an acknowledgment works is explained in 7.1.3. When the master sends the first data byte, then the slave will return an acknowledgment of the code it received during the synchronization period. When the master has sent all the data defined by the command, it sends a dummy byte to retrieve acknowledgment of the last data byte. If any of the acknowledgments were wrong, the master will abort the command.

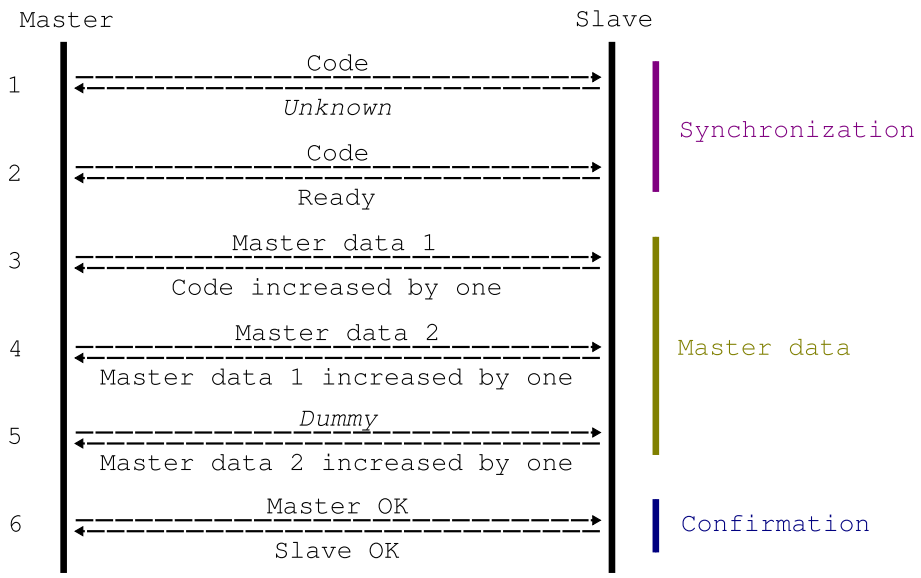


Figure 7.1:

- If nothing goes wrong during the message, both master and slave will get to the confirmation phase at the same time. Then the master will send an "master OK" code, and the slave will return an "slave OK" code. After a successful exchange of "OK" codes, both the master and the slave can be sure that the message was successfully transferred. For commands where the master sends data, it's especially important that the slave get confirmation that the data it received was correct.

The interface for requesting data from the slave is similar to that just described. It is described in figure 7.2, and has three phases that are described below:

- The synchronization phase is almost identical to that in the sending structure. The difference is that after the master sends the code, it has to send a dummy byte to retrieve the acknowledgment of the code.
- The slave data phase are similar to the master data phase in the sending structure. But since the slave sends data, the master has to retrieve the first byte by sending a dummy byte. Then it will send an acknowledgment for the first byte, and the slave will return the second byte, and so on. When the master sends an acknowledgment for the last byte, the slave will return a dummy byte, and the slave data phase is ready.
- The confirmation phase is the same as for the sending interface.

Both of these structures will have the following advantages:

- All bytes sent by both master and slave are sent back as acknowledgments and checked.
- Since the communication is full-duplex, the interfaces defined here won't use more than one or two SPI transfers than similar interfaces without acknowledgments would use.
- Several of the bytes received both by the master and slave are known, and can

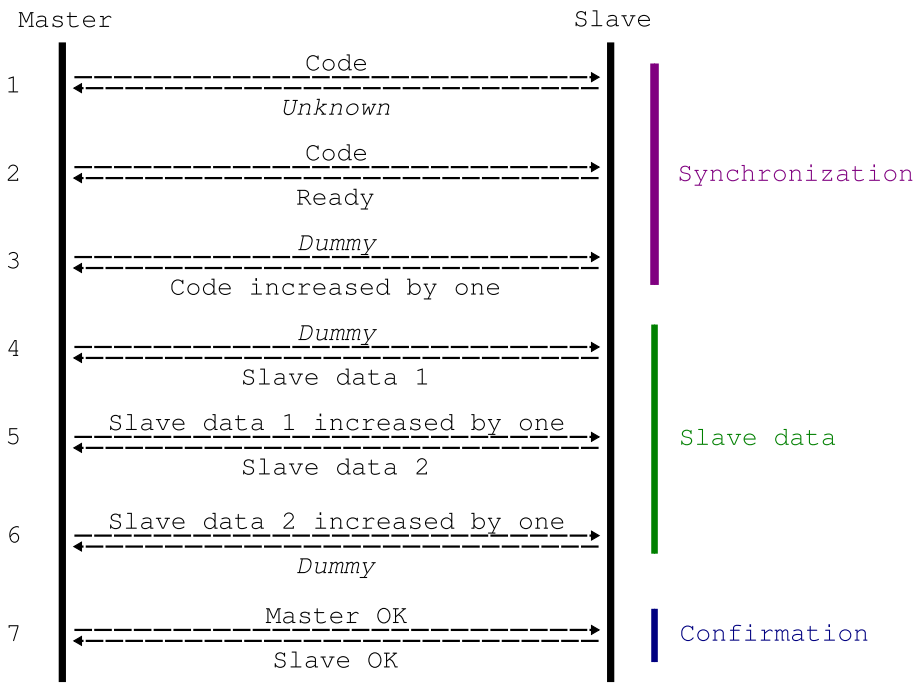


Figure 7.2:

therefor be checked for errors.

### 7.1.3 Acknowledgments

When either the master or the slave receives data, they will return an acknowledgment during the next transfer as explained in 7.1.2. This is to detect errors that either breaks the synchronization between the master and the slave, or errors during transfer of data.

To be able to tell if the correct data was received, the acknowledgment returns the received data back the the sender. The natural choice would be that the acknowledgment would have the same value as the original data. This proved to be a bad idea, because of the way SPI works. SPI uses one data register for both sending and receiving. If this register isn't deleted after a transfer, the previously received byte will be sent during the next transfer. This means that an acknowledgment consisting of the original data would fail. Therefore the acknowledgment is always the original byte increased by one. If the original byte had the value 255, then the acknowledgment would be zero.

### 7.1.4 SPI codes

Each message begins with the master sending a code to the slave. This code tells the slave what kind of command the message contain, and information about channel and subchannel if needed. The code consists of 1 byte (8 bits), where the first 5 bits are reserved for identifying the command, and the last three bits are for identifying channels and subchannels. The header file `spiByteCodes.h` below has constants and macros for

easy usage of these codes. It also contains comments that explain what the different bits in the code means.

```
1  #ifndef SPI_BYTES_CODES
2  #define SPI_BYTES_CODES
3
4  /*********************************************************************
5   * The code bytes are generated the following rules.
6   *
7   * - No code can have the value 0, because thats used as dummy byte.
8   * - Bit 7 (MSB):
9   *   - Equals 0 for commands that sends or receives data.
10  *   - Equals 1 for control commands.
11  * - Bit 6:
12  *   - Equals 0 for commands for digital IO.
13  *   - Equals 1 for commands for analog IO.
14  * - Bit 5:
15  *   - Equals 0 for commands for output.
16  *   - Equals 1 for commands for input.
17  * - Bit 4 and 3:
18  *   - Equals normally 11.
19  *   - Can have other values if bytes 5-7 is the same for multiple commands.
20  * - Bit 2 to 0 (LSB):
21  *   - For analog out, bit 2 equals channel and bit 0-1 equals subchannel.
22  *   - For analog in, bit 0-2 equals channel.
23  *   - For digital IO, bit 0-2 equals 000.
24  *********************************************************************/
25
26 // codes
27 #define SLAVE_READY              0xaa /* 10101010 */
28 #define SLAVE_OK                 0x11 /* 00010001 */
29 #define MASTER_OK                0x22 /* 00100010 */
30 #define DUMMY                    0x00 /* 00000000 */
31
32 // Data command codes.
33 #define CODE_AIN                 0x78 /* 01111000 + ch */
34 #define CODE_AIN_REQ             0x68 /* 01101000 + ch */
35 #define CODE_AIN_RET             0x70 /* 01110000 + ch */
36 #define CODE_AOUT                0x58 /* 01011000 + ch<<2 + sch */
37 #define CODE_DIN                 0x38 /* 00111000 */
38 #define CODE_DOUT                0x18 /* 00011000 */
39
40 // Control command codes.
41 #define CODE_AIN_EN              0xf8 /* 11111000 + ch */
42 #define CODE_AOUT_RES            0xd8 /* 11011000 + ch>>2 */
43
44 // Macros for finding channels from codes.
45 #define CODE_TO_CH_AIN(code)     (code & 0x7)
46 #define CODE_TO_CH_AOUT(code)    ((code & 0x4) >> 2)
47 #define CODE_TO_SCH_AOUT(code)   (code & 0x3)
48
49 // Macros for finding codes from channels.
50 #define CH_TO_CODE_AIN(ch)       (CODE_AIN + ch)
51 #define CH_TO_CODE_AIN_EN(ch)    (CODE_AIN_EN + ch)
52 #define CH_TO_CODE_AIN_REQ(ch)   (CODE_AIN_REQ + ch)
53 #define CH_TO_CODE_AIN_RET(ch)   (CODE_AIN_RET + ch)
54 #define CH_TO_CODE_AOUT(ch, sch) (CODE_AOUT + (ch << 2) + sch)
55 #define CH_TO_CODE_AOUT_RES(ch)  (CODE_AOUT_RES + (ch << 2))
56
57 #endif //SPI_BYTES_CODES
```

All the codes are unique, and they are also different from the other special bytes that are used during SPI messages. This includes the `SLAVE_READY`, `MASTER_OK`, `SLAVE_OK` and `DUMMY`. This means that the error-checking used in the message will be more effective, since each code only has one meaning.

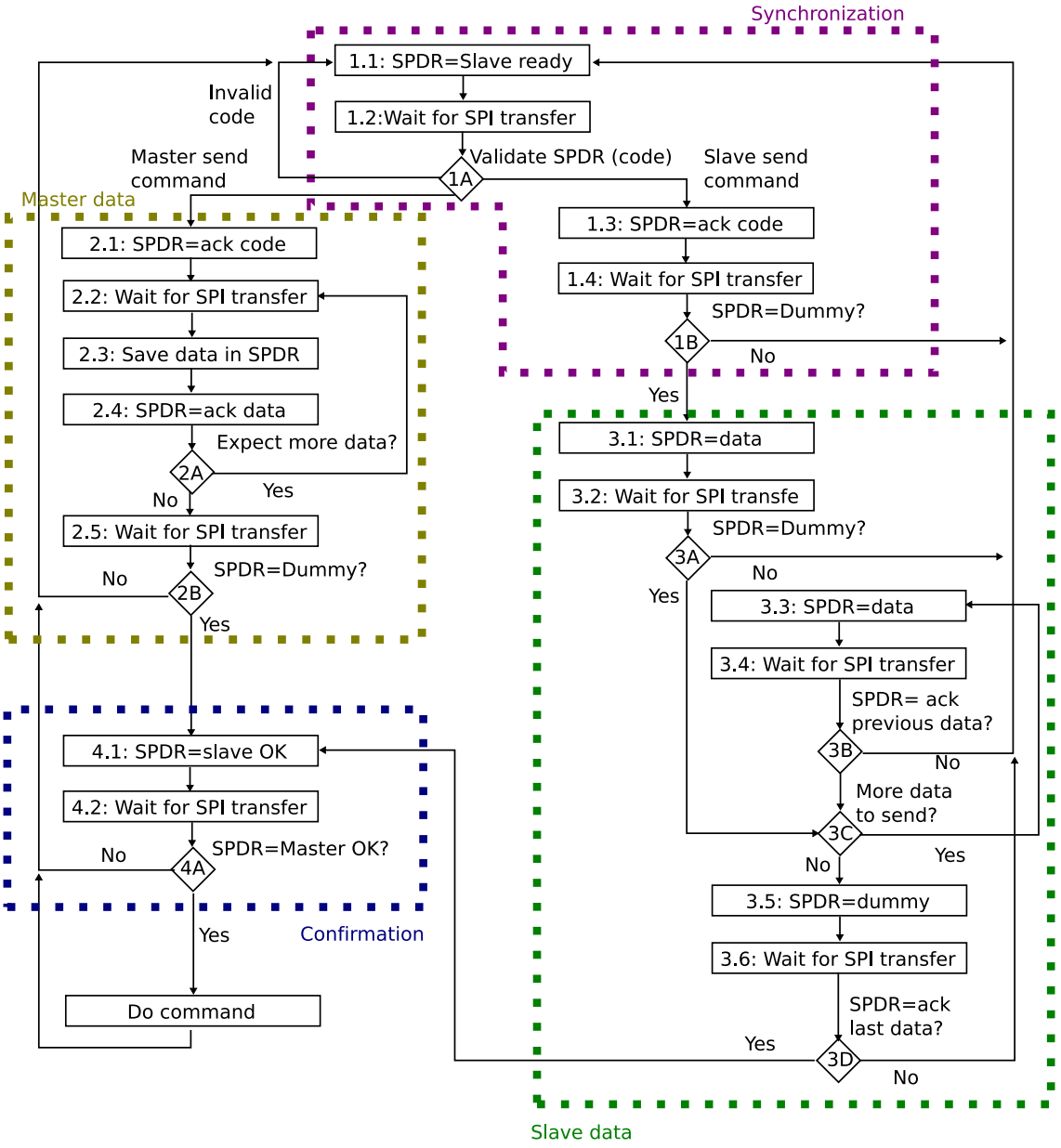


Figure 7.3: Activity diagram for SPI on ATmega128

## 7.2 ATmega128 Firmware

### 7.2.1 SPI commands

The main task for the ATmega128 is to listen for commands from the AVR32, and do what these commands tells it to do. 7.1.2 defines the message structure used by AVR32 to give commands. Figure 7.3 is an activity diagram that describe what the ATmega128 has to do when receiving a message. This activity diagram is divided in four phases, each corresponds to one of the different phases of the structure.

- Synchronization phase

- Activity 1.1 sets the SPI data register (SPDR) to the "Slave ready" code. The data in this register will be sends to the master during the next SPI transfer.
- Activity 1.2 waits until the next SPI transfer.
- Decision point 1A validates the received data, and check if it's one of the codes defined in 7.1.4. This code tells the slave what kind of command the message contains. Depending on the type of message, the next activity will either be 1.3 (slave sends) or 2.1 (master sends).
- Activity 1.3 sets the SPI data register to an acknowledgment of the code.
- Activity 1.4 waits until the next SPI transfer.
- Decision point 1B checks if the received byte is a dummy byte. No dummy byte means that the message failed.

◦ Master data phase

- Activity 2.1 sets the SPI data register to an acknowledgment of the code.
- Activity 2.2 waits until the next SPI transfer.
- Activity 2.3 saves the received data in the appropriate variable.
- Activity 2.4 sets the SPI data register to an acknowledgment of the received data.
- Decision point 2A checks if the slave expects more data. The slave knows this, because the number of data bytes are always the same for the same type of command. Further, the type of command is known because of the code.
- Activity 2.5 waits until the next SPI transfer.
- Decision point 2B checks if the received data was a dummy byte. No dummy byte means that receiving the message failed.

◦ Slave data phase

- Activity 3.1 sets the SPI data register to the first data the slave wants to send.
- Activity 3.2 waits until the next SPI transfer.
- Decision point 3A checks if the received data was a dummy byte. No dummy byte means that receiving the message failed.
- Decision point 3C checks if there is more data that should be sent, as in the master data phase, the number of bytes is defined by the type of command.
- Activity 3.3 sets the SPI data register to the next data the slave wants to send.
- Activity 3.4 waits until the next SPI transfer.
- Decision point 3B checks if the received data was an acknowledgment of the data the slave sent during the transfer before the recent transfer. No acknowledgment means that receiving the message failed.
- Activity 3.5 sets the SPI data register to be a dummy byte, since no data should be sent from the slave during the next transfer.

- Activity 3.6 waits until the next SPI transfer.
- Decision point 3D checks if the received data was an acknowledgment of the last data the slave sent. No acknowledgment means that receiving the message failed.

◦ Confirmation phase

- Activity 4.1 sets the SPI data register to the "Slave OK" code. This tells the master that the slave didn't discover any errors during the message.
- Activity 4.2 waits until the next SPI transfer.
- Decision point 4A checks if the received data is the "Master OK" code, which will confirm that the message was sent without errors.

### 7.2.2 Analog Input

The analog to digital conversion inside the ATmega128 is an operation that takes 13 ADC clock cycles. The ADC clock has to be between $50kHz$ and $200kHz$, to do a 10 bits conversion. By using a prescaler of 128 the ADC will get a frequency calculated in equation 7.1. With this frequency an ADC conversion will take $104\mu s$ as shown in equation 7.2 and 7.3. This means that the ADC conversion will take a considerable amount of time, and the ATmega128 will not be able to do the conversion during a SPI message if the AVR32 doesn't wait for it.

$$f_{ADC} = \frac{f_{\mu C}}{128} = 125kHz \tag{7.1}$$

$$T_{ADC} = \frac{1}{f_{ADC}} = 8\mu s \tag{7.2}$$

$$T_{conversion} = 13 * T_{ADC} = 104\mu s \tag{7.3}$$

To solve this problem, two solutions were developed. The first solution is to use interrupts. This is normally the best way to solve problems when a processor has to wait while another (often I/O) computer component completes its task. When the task is completed an interrupt is sent to the processor, and the it can retrieve the result.

The second solution is to let the ATmega128 know which analog input channels that are in use. It can then continuously convert the analog values of these channels into digital values and store them in a buffer in memory. When the master wants to read an analog input channel, the slave can just return the previously converted result. This solution is possible to use, since the analog to digital conversion is a separate part of the ATmega128. Meaning that the normal code will continue to run as usual during the analog conversion. The only overhead is saving the results and start new conversions.

The two solutions, called interrupt mode and buffering mode, will both work, but have some different advantages, shown below.

◦ Interrupt mode will always return a newly converted result, while buffering mode may return an older result.

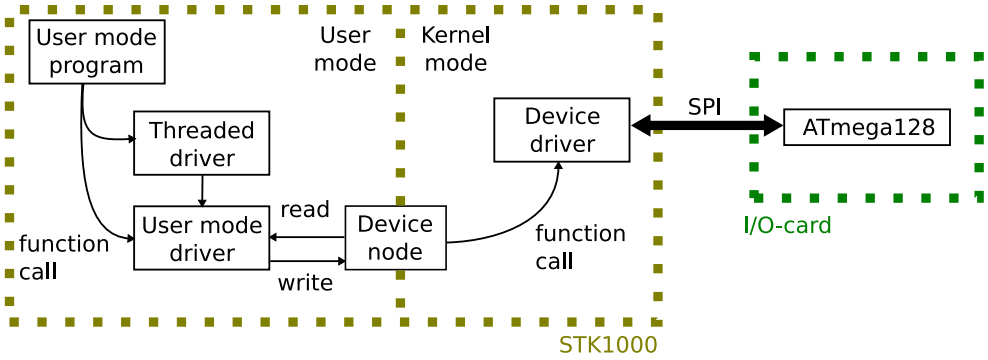


Figure 7.4: Describing the different components

- Buffering mode will return a result immediately, while interrupt mode has to wait for the new conversion.
- Interrupt mode will make the implementation of the AVR32 kernel module more complex, since it will need to react to interrupts. Buffering mode won't need this.
- Buffering mode will make the implementation of the ATmega128 more complex, since it continuously has to start new analog to digital conversions and save the results. This has to be implemented so it doesn't affect the normal operation. Interrupt mode won't need this.

### 7.2.3 Timeout

7.2.1 describes one way the ATmega128 can manage SPI communication with the AVR32. One problem with this solution is that an error can leave the slave waiting for an SPI transfer that never shows up. This means that it's impossible or hard to know in what state the slave will end up if it was an error during the message. This wouldn't break future messages, since the master starts all messages by synchronizing with the slave. But it's generally a bad idea that the ATmega128 can end up in an unknown state.

The reason that the ATmega128 risks to end up in an unknown state are the "Wait for SPI transfers" in figure 7.2.3. These could possible wait forever, which is the reason for the problem. If all of these (except for activity 1.1) could have timeouts that would send them back to activity 1.1, then the problem should be solved. The trick is to select a correct time out period. It should probably be something like 2-3 times higher than the normal pause between SPI transfers.

## 7.3 Device Driver

The purpose of the *device driver* is to act as a link between a user mode program and the I/O-card. The user mode program access the device driver through a *device node*, and it access the I/O-card through SPI communication. This is shown in figure 7.4.

The Linux kernel is a complicated piece of software. Hence, to make a module, make it react to device nodes, use SPI configure interrupts is not a trivial task. How to program

the *device driver* to carry out these tasks is described in 8.2.

### 7.3.1 Device Nodes

*Device nodes* are used by user mode programs to access the device through the device driver. A simple device driver has often only one node associated with it. More complex drivers may have several, where each node represents a specific function. Several nodes was used for this driver, one for each of the analog channels, one for digital input and one for digital output. Notice that analog output only has two nodes, so each node will control three analog output subchannels.

The reason for this choice was to have one node for each of the smallest controllable parts of the I/O-card. For example the two analog output channels can have different resolutions, but all the subchannels of one channel has to have the same. Since the analog input channels aren't dependent on each other in the same way, they have one node each. This will make it easier to control them individually. The I/O-card is a typical the char device, since only small amounts of data are transferred serially with SPI.

### 7.3.2 Master send Data

The activity diagram in figure 7.5 on the facing page describes how the device driver sends a message containing data to the slave. It does not describe how the type of command are decided, or how to read and write user mode data. This are described in more detail in 8.2.7

- Synchronization phase
  - Activity 1.1 sends the appropriate code to the slave.
  - Decision point 1A checks if the received data was a slave ready signal. If it wasn't the ready signal, the code has to be sent again.
- Master data phase
  - Activity 2.1 sends the first data.
  - Decision point 2A checks if the received data was an acknowledgment of the code. No acknowledgment means that sending the message failed.
  - Decision point 2C checks if there is more data to send.
  - Activity 2.2 sends the next data.
  - Decision point 2B checks if the received data was an acknowledgment of the previously sent data. No acknowledgment means that sending the message failed.
  - Activity 2.3 sends a dummy byte to retrieve the acknowledgment of the last data.
  - Decision point 2D checks if the received data was an acknowledgment of the last sent data. No acknowledgment means that sending the message failed.

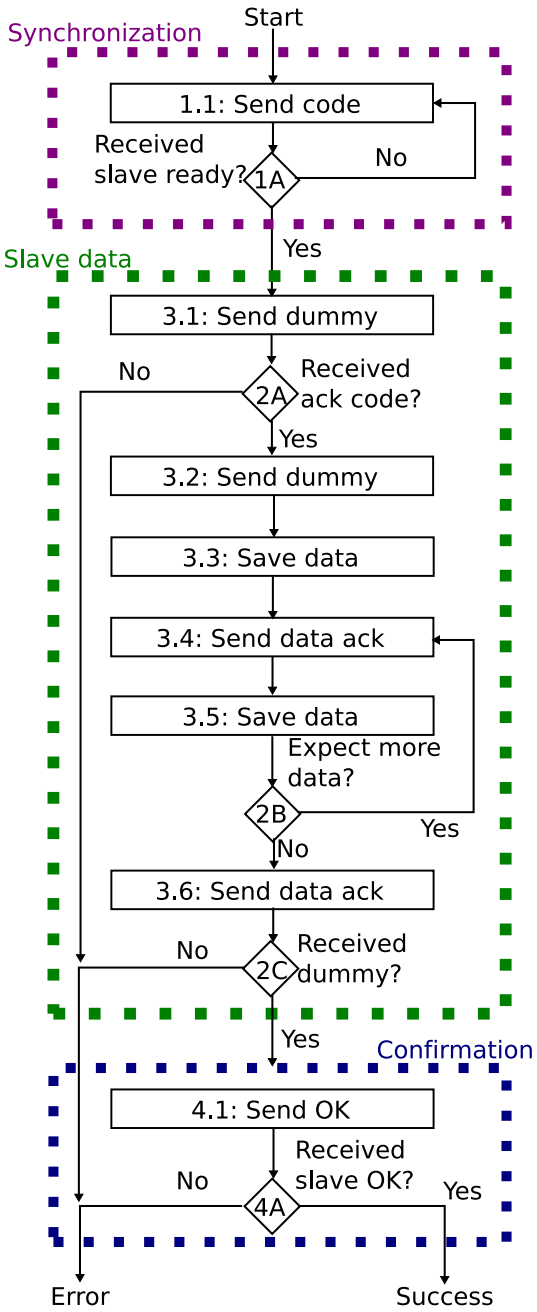


Figure 7.5: Activity diagram for device driver when master sends data

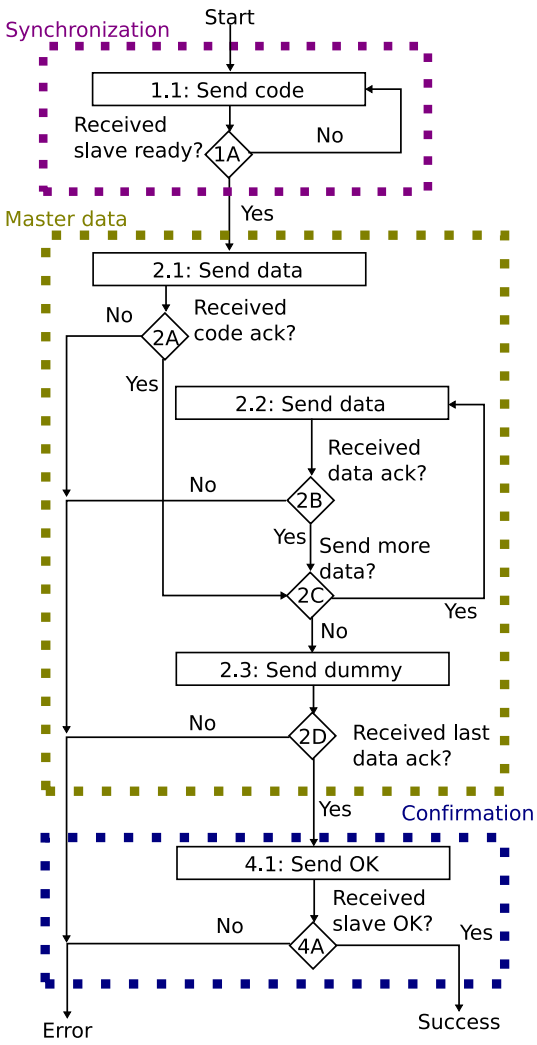


Figure 7.6: Activity diagram for device driver when master requests data

- Confirmation phase
  - Activity 4.1 sends the master OK signal.
  - Decision point 4A checks if the received data was a slave OK signal. If it was a slave OK signal, the message was sent successfully.

### 7.3.3 Request Data from Slave

The activity diagram in figure 7.6 describes how the *device driver* sends a message that retrives data from the slave. The Synchronization and Confirmation phases are identical as in 7.3.2.

- Slave data phase
  - Activity 3.1 sends a dummy byte.

- Decision point 2A checks if the received data was an acknowledgment of the code. No acknowledgment means that sending the message failed.
- Activity 3.2 sends a new dummy byte.
- Activity 3.3 saves the received data.
- Activity 3.4 sends an acknowledgment of the received data.
- Decision point 2B checks if more data is expected.
- Activity 3.6 sends an acknowledgment of the last received data.
- Decision point 2C checks if the received data was a dummy byte. No dummy byte means that sending the message failed.

## 7.4 User Mode Driver

Each of the commands that can be given to the I/O-card have one or more functions provided by the user mode driver. These functions makes it possible to give the I/O-cards commands just with a single function call, and this function call should return the result of the message. This will make it very easy to use the different features of the I/O-card. The reason that some commands should have more than one function, is that they can be used in different ways. The user mode driver should have the following functionality:

- Read from different analog in channels.
- Write values to different analog out channels.
- Read all digital in channels at once.
- Write to all or a chosen number of the digital out channels.
- Change different properties of the I/O card, like analog resolution or enable/disable functionality.

## 7.5 Threaded User Mode Driver

Since communicating with I/O often includes waiting, commands are often executed in threads. To make this possible, another layer of software was used on top of the user mode driver. This layer uses threads to send and receive data from the I/O-card, making the total system more effective. This has only been designed for analog I/O.

### 7.5.1 Threaded Analog Input

The analog input uses threads to sample the analog input values. Each channel has its own periodic thread that gets an analog input value and store it in a buffer. When a user mode application needs the value of an analog channel, it can quickly get the buffered value.

This solution works best when the I/O-card uses interrupt mode, since the value the thread is putting into a buffer should be as new as possible. If using buffering mode, the data will be buffered twice, and this should be avoided. The main disadvantage of interrupt mode is that the command takes longer time, is less of a problem when it's performed by a thread. It will not slow down the rest of the program.

### 7.5.2 Threaded Analog Output

The analog output uses a worker thread. This thread waits until it receives an analog output command from the main thread. Then it executes this command and waits for the next. This means that the main thread gives a short command to the worker thread, and then it can continue executing the rest of its code.

# Chapter 8

# Implementation of I/O-card Software

This chapter describes how the design from chapter 7 was implemented in several files of C-code. Only the header files are shown here. The code files can be found in the appendix. All the functions definitions and variables are commented in the header files, which makes it possible to know the purpose of the different functions by looking at the headers. Some parts of the code files are shown here and used as examples.

## 8.1 ATmega128 Firmware

The code for ATmega128 are divided into several pairs of headers and code files. Each of these pairs provides the code for controlling a part of the ATmega128.

### 8.1.1 am128main

This file contain the main function of the ATmega128, which is where the code execution starts. This main function will initialize the I/O-card, with the help of the other code files and then start the main program-loop. This loop will run as long as the ATmega128 has power, and will listen for SPI messages from AVR32.

The code below is the header file `am128main.h`, and the code file `am128main.c` is in the appendix C.1.2. The header file has some `#define` statements that enables or disables different features in the code. These are used to change how the I/O-card works without having to make changes in the rest of the code.

```
1  // Defines the clock frequency of the device. Needed for util/delay.h.
2  #define F_CPU 16000000
3
4  // Includes.
5  #include <avr/io.h>
6  #include <util/delay.h>
7  #include <avr/interrupt.h>
8
9  // Macros for bit operations.
10 #define set_bit(reg, bit) (reg |= (1 << bit))
```

```
11 #define clear_bit(reg, bit) (reg &= ~(1 << bit))
12 #define test_bit(reg, bit) (reg & (1 << bit))
13 #define loop_bit_is_set(reg, bit) while(!test_bit(reg, bit))
14 #define loop_bit_is_clear(reg, bit) while(test_bit(reg, bit))
15
16 // Enables debugging over RS-232.
17 // This will give problems, since the debugging takes a long time.
18 // #define RS232_DEBUG
19
20 // Enables timout of SPI transfers.
21 // Timeout period = TIMEOUT * 20 microseconds.
22 //#define TIMEOUT_ENABLE
23 #define TIMEOUT 25
24
25 // Defines if buffering mode or interrupt mode will be used for ADC.
26 #define BUFFERING_MODE
27 //#define INTERRUPT_MODE
28
29 // Main function that initialize the I/O-card and then starts the main program loop
      , which continuously checks for new messages from the AVR32.
30 int main();
31
32 #include "am128uart.h"
33 #include "am128io.h"
```

The code below is the main program loop, which will wait for new commands. On line 14 the SPI data register are set to the `SLAVE_READY` code. This makes sure that the next byte the it sends to the master will be this code. Line 17 waits for a new SPI transfer, and the code received in this SPI transfer is sent to the function at line 24. This function will validate the code and continue the SPI command.

```
11    // Main program loop.
12    while(1){
13       // Prepare to send ready signal, and wait for code from master.
14       SPDR = SLAVE_READY;
15
16       // Wait for SPI transfer.
17       while(!test_bit(SPSR,SPIF)){
18 #ifdef BUFFERING_MODE
19          // Checks if conversion is finished if using bufering mode.
20          ADC_CheckBuffer();
21 #endif
22       }
23       // Start receiving new command.
24       IO_NewCommand(SPDR);
25    }
```

When in buffering mode, the ATmega128 checks for completed ADC conversions while waiting for SPI messages. This is only checked inside this loop, because checking it during a SPI message can disturb the synchronization between ATmega128 and AVR32. It would be more logic to use interrupts when a ADC conversion was completed. However, when doing this about 6% of the SPI messages returned errors. This was because these interrupts happend in the middle of a SPI command, and it made the ATmega128 unable to return the correct data at the correct time.

By polling the ADC whenever the ATmega128 waits for a new message (not new transfer) the work of managing ADC will only be done when the ATmega128 normally would do a busy wait for SPI. For this to work robustly, it's important that all the other "wait for SPI" functions in the program has timeouts as described in 7.2.3. This is because the ATmega128 will be unable to check the ADC conversions if it doesn't return to the the loop described here.

### 8.1.2 am128io

`am128io.h` and `am128io.c` contains functions that receive a code that the main function has received from AVR32. These functions will receive the rest of the message. The code below is the header file `am128io.h`, and the code file `am128io.c` is in the appendix C.1.3. This header file is quite long, but it also defines how all of the different messages are implemented.

```
1  #ifndef AM128_IO
2  #define AM128_IO
3
4  // Initialize the different parts of the I/O-card.
5  void IO_Init();
6
7  // Function that checks if an received data (rx) was a correct ack of the
       transmitted data (tx).
8  signed char IO_TestAck(unsigned char tx, unsigned char rx);
9
10 // Checks the received code and starts receiving a new command if code is valid.
11 signed char IO_NewCommand(unsigned char code);
12
13 /*
14 Analog in SPI message.
15
16 Number   Received from master    Returned to master
17 1        DUMMY                  code + 1
18 Wait for interrupt if in interrupt mode.
19 2        DUMMY                  vauleH
20 3        valueH + 1             vauleL
21 4        valueL + 1             DUMMY
22 5        MASTER_OK              SLAVE_OK
23 */
24 signed char IO_AnalogIn(unsigned char code);
25
26 /*
27 Analog out SPI message.
28
29 Number   Received from master    Returned to master
30 1        valueH                 code + 1
31 2        valueL                 vauleH + 1
32 3        DUMMY                  vauleL + 1
33 4        MASTER_OK              SLAVE_OK
34 */
35 signed char IO_AnalogOut(unsigned char code);
36
37 /*
38 Digital in SPI message.
39
40 Number   Received from master    Returned to master
41 1        DUMMY                  code + 1
42 2        DUMMY                  value
43 3        value + 1              DUMMY
44 4        MASTER_OK              SLAVE_OK
45 */
46 signed char IO_DigitalIn(unsigned char code);
47
48 /*
49 Digital out SPI message.
50
51 Number   Received from master    Returned to master
52 1        mask                   code + 1
53 2        value                  mask + 1
54 3        DUMMY                  vaule + 1
55 4        MASTER_OK              SLAVE_OK
56 */
57 signed char IO_DigitalOut(unsigned char code);
```

```
58
59 #ifdef BUFFERING_MODE
60 /*
61 Analog in enable SPI message. (Only used in buffering mode)
62
63 Number   Received from master    Returned to master
64 1        enable                 code + 1
65 2        DUMMY                  enable + 1
66 3        MASTER_OK              SLAVE_OK
67 */
68 signed char IO_AnalogInEnable(unsigned char code);
69 #endif
70
71 /*
72 Analog out resolution SPI message.
73
74 Number   Received from master    Returned to master
75 1        resolution             code + 1
76 2        DUMMY                  resolution + 1
77 4        MASTER_OK              SLAVE_OK
78 */
79 signed char IO_AnalogOutRes(unsigned char code);
80
81 // Includes.
82 #include "../spiByteCodes.h"
83 #include "am128spiSlave.h"
84 #include "am128pwm.h"
85 #include "am128adc.h"
86
87 #include "am128io.c"
88
89 #endif //AM128_IO
```

Below is the function `IO_AnalogOut()` from `am128io.c`. This function is a good example on how the functions in this file works. The other functions are all similar.

```
113 signed char IO_AnalogOut(unsigned char code)
114 {
115    unsigned char valueH, valueL, rx;
116    char ret;
117
118    // 1st byte: store received valueH, return code + 1.
119    ret = SPI_SlaveTransferByte(code + 1, &valueH, TIMEOUT);
120    if(ret < 0) return ret;
121
122    // 2nd byte: store received valueL, return valueH + 1.
123    ret = SPI_SlaveTransferByte(valueH + 1, &valueL, TIMEOUT);
124    if(ret < 0) return ret;
125
126    // 3rd byte: check received DUMMY, return valueL + 1.
127    ret = SPI_SlaveTransferByte(valueL + 1, &rx, TIMEOUT);
128    if(ret < 0) return ret;
129    if(rx != DUMMY) return -1;
130
131    // 4th byte: check received MASTER_OK, return SLAVE_OK.
132    ret = SPI_SlaveTransferByte(SLAVE_OK, &rx, TIMEOUT);
133    if(ret < 0) return ret;
134    if(rx != MASTER_OK) return -1;
135
136    // When the message is confirmed the mask and data are applied to digital out.
137    PWM_SetOutValue(CODE_TO_CH_AOUT(code), CODE_TO_SCH_AOUT(code), valueH, valueL);
138
139    // Debug printing, if enabled.
140    UART_Print('c', code);
141    UART_Print('h', valueH);
142    UART_Print('l', valueL);
143 }
```

Each of the transfers is marked with a comment explaining what ATmega128 sends and what it expects to receive. After each transfer the return value is checked. If this is negative it means that the function timed out before a transfer was received. If the return value is negative or the received data wasn't the expected data, the function returns an error.

After the 4th transfer is controlled, the received data is written to PWM. The channel and subchannel of the PWM is defined by the code. Before the function returns, it also prints some debugging information via UART, if this is enabled.

### 8.1.3 am128slaveSpi

`am128slaveSpi.h` and `am128slaveSpi.c` contain functions for using the ATmega128 as a SPI slave device. During the development several different slave transfer functions were used. The end result was just one function that sends and receive one byte and has timeout if this is enabled.

The functions that receives the SPI messages does this one transfer at the time. As opposed to the AVR32 SPI driver, the ATmega128 is able to handle each SPI transfer individually without losing effectivity. This gives code that's easier to read and understand.

The code below is the header file `am128spiSlave.h`, and the code file `am128spiSlave.c` is in the appendix C.1.4.

```
1  #ifndef AM128_SPI_SLAVE
2  #define AM128_SPI_SLAVE
3
4  // Initialize the ATmega128 as a SPI slave. If timeout are enabled, then this
       function will initialize a timer aswell.
5  void SPI_SlaveInit(void);
6
7  // Function for transfer one byte of data. It will wait for an SPI transfer from
       the master. If timeouts are enabled, the function will return an error after
       some time. Since SPI is fully duplex, this function will both send and receive
       data.
8  signed char SPI_SlaveTransferByte(unsigned char tx, unsigned char *rx, unsigned
       char usec);
9
10 #include "am128spiSlave.c"
11
12 #endif //AM128_SPI_SLAVE
```

Below is the `while` loop that is used to wait for a SPI transfer. To wait for something to happen with a loop like this, is called a busy-wait, and normally it's something that should be avoided. This is because a busy-wait wastes many processor cycles that could be used by other processes. For the ATmega128 this isn't a problem since it can only have one process. It would also be possible to use interrupts instead of busy-wait, but it would make the code more complex and probably not more effective.

```
while(!(SPSR & (1<<SPIF)))
```

It's also important to notice that the SPI data register (SPDR) should always be read after a transfer (after the `while` loop is finished). The status bit for received SPI transfer will only be cleared if the SPDR is either read or written to. Failing to clear the status bit makes it impossible to know if a new transfer has been received or not.

### 8.1.4 am128ADC

`am128ADC.h` and `am128ADC.c` contain functions for using the ADC (analog to digital converter). I had decided to test two different modes to use analog input, and both of them are implemented in the same code file. This means that much of the code are inside `#ifdef` statements so only the code for the selected mode will be compiled.

```
1  #ifndef AM128_ADC
2  #define AM128_ADC
3
4  /*
5  Initialize the ADC part of the ATmega128.
6  */
7  void ADC_Init();
8
9  #ifdef BUFFERING_MODE
10 // Data structures and functions for buffering mode.
11
12 // Struct for making a circular array for storing the channels that are enabled.
13 struct chanList{
14         struct chanList *next;
15         struct chanList *prev;
16         char channel;
17 };
18
19 struct chanList channeltab[8]; // Static decleration of list elements.
20 struct chanList *channels; // Pointer to active element of the circular list.
21 char chCount; // Number of elements in the list.
22
23 // tables for storing the latest value for each channel.
24 unsigned char dataH[8], dataL[8];
25
26 // Enable one analog channel, if it's not already enabled. Enabled means that the
       ATmega128 will continuously convert analog values on this channel and store the
       digital values in a buffer.
27 void ADC_EnableChan(char channel);
28
29 // Disable one analog channel, if it's not already disabled.
30 void ADC_DisableChan(char channel);
31
32 // Check if the ongoing conversion is finished, if so store the data and start a
       new one.
33 void ADC_CheckBuffer();
34
35 // Starts conversion of the next channel in the array.
36 void ADC_NewConversion();
37
38 // Stops and ongoing conversions.
39 void ADC_StopConversion();
40
41 #endif
42
43 #ifdef INTERRUPT_MODE
44 // Interrupt mode.
45
46 // Starts a conversion of a given channel.
47 void ADC_StartConversion(char channel);
48
49 // Checks if a conversion is finished, if so the result are stored, and an
       interrupt is sent to the AVR32.
50 signed char ADC_CheckConversion(unsigned char *data);
51 #endif
52
53 #include "am128adc.c"
54
55 #endif //AM128_ADC
```

The code for interrupt mode is very simple, it consists of one function that starts conversion on a channel, and one that checks if the ongoing conversion is completed. The last one will also return the conversion result in an array.

The code for buffering mode is more complex. The idea is that the ATmega128 should convert one analog input channel, save the result in a buffer, before starting to convert the next channel. Since each conversion takes some time, it would be a good idea to only convert the channels that are in use. A doubly-circularly-linked list[1] was designed to contain all the active channels. Since the list is circular, the program can always move to the next channel in the list whenever a new conversion should be started.

### 8.1.5 am128pwm

`am128pwm.h` and `am128pwm.c` contain functions for using the PWM (Pulse-width modulation) timers as analog output signals. With these functions it's possible to change the output value of the subchannels and the resolution of the channels. All subchannels of the same channel will always have the same resolution.

The code below is the header file `am128pwm.h`, and the code file `am128pwm.c` is in the appendix C.1.6.

```
1  #ifndef AM128_PWM
2  #define AM128_PWM
3
4  // Initialize 2 PWM channels with 3 subchannels each. By default all PWM are 8-bit
       and has an output value of zero.
5  void PWM_Init();
6
7  // Changes the output value of one of the subchannel. The new value are given with
       two 8 bit values.
8  void PWM_SetOutValue(char ch, char sch, char valueH, char valueL);
9
10 // Changes the resolution of one of the channels (affecting all subchannel). When
       resolution are changed, the output value of all subchannels affected are set to
        zero.
11 void PWM_SetResolution(char ch, char resolution);
12
13 #include "am128pwm.c"
14
15 #endif //AM128_PWM
```

### 8.1.6 am128uart

`am128uart.h` and `am128uart.c` contain functions for using the UART. With the MAX233 IC on the I/O-card it's possible to use to UART to communicate with a workstation through a serial cable. This is very useful for debugging, since it's possible to print small messages to the screen of a computer.

The code below is the header file `am128uart.h`, and the code file `am128uart.c` is in the appendix C.1.7.

```
1  #ifndef AM128_UART
2  #define AM128_UART
```

[1] A doubly-circularly-linked list is a list where each item points both to its next and previous item, and the last item in the list points to the first.

```
3
4  // Initialize the UART with baud rate of 19.2k.
5  void UART_Init();
6
7  // Transmit one byte of data via UART.
8  void UART_TxByte(char data);
9
10 // Send a byte containing a <space>.
11 void UART_TxSpace();
12
13 // Send two bytes that togeter makes a new line.
14 void UART_TxNewLine();
15
16 // Convert a decimal number to a string of three characters and send them.
17 void UART_TxDecimal(unsigned char value);
18
19 // Send a character and a value.
20 void UART_Print(char symbol, unsigned char value);
21
22 #include "am128uart.c"
23
24 #endif //AM128_UART
```

## 8.2 Device driver

The device driver has to be coded in kernel mode, and during development it's implemented as a kernel module. Programming kernel modules are not the same as programming normal C-programs, even if both use the C-programming language. To start with, the kernel module doesn't have a `main()` function, but does have different functions that runs in different situations. Another difference is that functions defined by user mode libraries like glibc aren't available. Instead of a user mode library, it's possible to include header files from the kernel source to get more functionality. A third difference is that there are no floating-point operations or data types in the kernel.

### 8.2.1 Init and Exit Functions

As mentioned, different functions in a kernel module runs in different situations. The two most basic functions are the init and exit functions. These are functions that are run when the module is loaded in and out of the kernel. The macros `module_init` and `module_exit` are used to define which functions are the init and exit functions of the module.

### 8.2.2 Major and Minor Numbers

7.3.1 defined the device nodes that should be used in this system. All these will normally have the same major number, but they should have different minor numbers. The kernel module has to allocate the major and minor number it will use, which means that one major number has to be allocated, and one minor number for each of the nodes. To allocate major and minor numbers for a char device, the function `alloc_chrdev_region` should be used. This allocated a unused major number and a chosen number of minor numbers.

The allocated minor numbers will be used to number the device nodes. The first eight numbers (0-7) were given to the analog input channels, the next two (8-9) were given to the two analog output channels and the two last were given to digital input and digital output. A header file called `minorNumbers.h` was made, to make it easier to use these minor numbers. This file is shown below.

```
1
2  // Defines number of analog channels.
3  #define COUNT_AIN                8
4  #define COUNT_AOUT               2
5  #define COUNT_TOTAL              (COUNT_AIN + COUNT_AOUT +2)
6
7  // Minor numbers of first channel of each type.
8  #define MINOR_AIN                0
9  #define MINOR_AOUT               COUNT_AIN
10 #define MINOR_DIN                (MINOR_AOUT + COUNT_AOUT)
11 #define MINOR_DOUT               (MINOR_AOUT + COUNT_AOUT + 1)
12
13 // Macros for finding minor number for a channel.
14 #define CH_TO_MINOR_AIN(ch)      (MINOR_AIN + ch)
15 #define CH_TO_MINOR_AOUT(ch)     (MINOR_AOUT + ch)
16
17 // Macros for testing if a minor number is a spesific type.
18 #define IS_AIN(minor)            ((minor>=MINOR_AIN)&(minor<MINOR_AOUT))
19 #define IS_AOUT(minor)           ((minor>=MINOR_AOUT)&(minor<MINOR_DIN))
20 #define IS_DIN(minor)            (minor==MINOR_DIN)
21 #define IS_DOUT(minor)           (minor==MINOR_DOUT)
22
23 // Functions for converting minor numbers into channels.
24 int MinorToChAin(int minor)
25 {
26    if(IS_AIN(minor)){
27       return minor - MINOR_AIN;
28    }
29    else{
30       return -1;
31    }
32 }
33
34 int MinorToChAout(int minor)
35 {
36    if(IS_AOUT(minor)){
37       return minor - MINOR_AOUT;
38    }
39    else{
40       return -1;
41    }
42 }
```

### 8.2.3 Char Device registration

Before the char device driver can work, one or more char devices have to be registered in the kernel. This is done by making a `cdev` structure and use a function to add it into the kernel. The `cdev` structure contain a pointer to a `file_operations` structure. This structure points to different functions for the different file operations that are associated with the char device.

When an user mode program should communicate with device driver through a device node, the module has to react to different operations done to the node. The `file_operations` structure defines what functions are associated with the different file operations. Below is a list of the file operations used in this module, and what they do.

- Open
  This file operation is performed each time an user mode program opens the device node. This has to be done before any other operations can be done on the node.
- Release
  It's possible to open a file multiple times, without closing it. Each opening will increment a timer, and each closing will decrement it. This file operation is performed when the file is closed, and this counter equals zero. This means that the file is no longer open, and used memory can be released, hence the name.
- Write
  This file operation is performed when a user mode program tries to write to the node. In this device driver this file operation will be used by user mode programs to write values to outputs on the I/O-card.
- Read
  This file operation is performed when a user mode program tries to read from the node. In this device driver this file operation will be used by user mode programs to read values from inputs on the I/O-card.
- Ioctl
  Is a special file operation for device nodes. That's used to give commands to the node that aren't normal data. In this device driver this file operation will be used by user mode programs to change the properties of the I/O-card.

### 8.2.4 SPI device and driver

To use SPI, some data structures has to be defined. The SPI driver points to a probe function for the device, that are used by the module to find out if it's possible to use a SPI device. In this function the SPI device is configured and properties like the SPI frequency are set.

### 8.2.5 Changes in Linux Kernel Source Code

During development the device driver will be compiled as a module, but it still needs some minor changes in the kernel to work. This is done by adding some lines to the `arch/avr32/boards/atstk1000/atstk1002.c` file in the linux source. These lines should be added to the `spi_board_info` structure, and the new code should look like below. The new file is available on the CD-ROM.

```
38 static struct spi_board_info spi0_board_info[] __initdata = {
39         {
40                 /* QVGA display */
41                 .modalias       = "ltv350qv",
42                 .max_speed_hz   = 16000000,
43                 .chip_select    = 1,
44         },
45    {
46        .modalias     = "avr32io",
47        .max_speed_hz = 16000000,
48        .chip_select  = 2,
49    },
50 };
```

### 8.2.6 avr32io

`avr32io.h` and `avr32io.h` contain the structures and functions described earlier in this chapter. These are all defined in the header file below. The code file are in the appendix C.2.2.

```
1  #ifndef __KM_AVR32_IO_H
2  #define __KM_AVR32_IO_H
3
4  // Includes
5  #include <linux/module.h>
6  #include <linux/init.h>
7  #include <linux/errno.h>
8  #include <linux/fs.h>
9  #include <linux/cdev.h>
10 #include <linux/device.h>
11 #include <linux/spi/spi.h>
12 #include <linux/ioctl.h>
13 #include <linux/delay.h>
14 #include <asm/uaccess.h>
15 #include <asm-avr32/io.h>
16 #include <linux/types.h>
17 #include <linux/wait.h>
18 #include <linux/interrupt.h>
19
20 //Defines IOCTL numbers.
21 #define AVR32IO_IOCTL_MAGIC 'k'
22 #define AVR32IO_IOCTL_RES_AOUT _IOW(AVR32IO_IOCTL_MAGIC, 0, int)
23 #define AVR32IO_IOCTL_EN_AIN _IOW(AVR32IO_IOCTL_MAGIC, 1, int)
24
25 // Defines if driver should print out debugging information.
26 //#define VERBOSE_DEBUG
27 #ifdef VERBOSE_DEBUG
28         #define DEBUG(fmt, args...) { if(printk_ratelimit()) printk(KERN_ALERT fmt,
              ## args); }
29 #else
30         #define DEBUG(fmt, args...)
31 #endif
32
33 // Defines if buffering mode or interrupt mode will be used for ADC.
34 #define BUFFERING_MODE
35 //#define INTERRUPT_MODE
36
37 // Runs when kernel is started/module is installed. Initialize the needed data
      structures.
38 static int __init avr32io_Init(void);
39
40 // Runs when module is removed. Removes data structures.
41 static void __exit avr32io_Exit(void);
42
43 // Runs when spi driver is started. Initialize the SPI driver.
44 static int __devinit avr32io_Probe(struct spi_device * spi);
45
46 // Runs when an user-space program opens a device node. Tries to mark the SPI as
      busy, error if SPI already is busy.
47 static int avr32io_Open(struct inode *node,struct file *filp);
48
49 // Runs when the last user-space program closes a device node. mark the SPI as not
      busy.
50 static int avr32io_Release(struct inode *node,struct file *filp);
51
52 // Runs when an user-space program tries to read from a device node. Send command
      to I/O-card and return the result.
53 static ssize_t avr32io_Read(struct file *filp, char __user *userBuf, size_t len,
      loff_t *f_pos);
54
```

```
55 // Runs when an user-space program tries to write to a device node. Send command to
       I/O-card.
56 static ssize_t avr32io_Write(struct file *filp, const char __user *userBuf, size_t
       len, loff_t *f_pos);
57
58 // Runs when an user-space program tries to send an IO control command. Send the
       command to I/O-card.
59 static int avr32io_Ioctl(struct inode  *node, struct file *filp, unsigned int cmd,
       unsigned long arg);
60
61 // Struct containing data structures for the kernel module.
62 struct avr32io_device {
63    dev_t number; // Major and minor numbers.
64    struct cdev *charDevice; // Char device.
65    struct spi_device *spi; // SPI device.
66    int busy; // Device busy.
67 #ifdef INTERRUPT_MODE
68    int irqID; // Interrupt used for analog in.
69    int irqFlag; // Flag when receiving interrupt.
70 #endif
71 };
72
73 // Pointer to device used in driver.
74 static struct avr32io_device * device;
75
76 // Struct that defines the functions used for different file operations.
77 static struct file_operations fops = {
78    .owner = THIS_MODULE,
79    .read = avr32io_Read,
80    .write = avr32io_Write,
81    .open = avr32io_Open,
82    .release = avr32io_Release,
83    .ioctl = avr32io_Ioctl,
84 };
85
86 // Struct that defines the SPI driver.
87 static struct spi_driver avr32io_driver = {
88    .probe = avr32io_Probe,
89    .driver = {
90       .name = "avr32io",
91    }
92 };
93
94 DECLARE_MUTEX(spiMutex);
95
96 // Includes.
97 #include "../minorNumbers.h"
98 #include "../spiByteCodes.h"
99 #include "avr32io_SPI.h"
100 #include "avr32io_Cmd.h"
101
102 // Defines init and exit functions.
103 module_init(avr32io_Init);
104 module_exit(avr32io_Exit);
105
106 MODULE_LICENSE("GPL");
107 MODULE_AUTHOR("Oyvind␣Netland,␣NTNU,␣Trondheim,␣Norway");
108 MODULE_DESCRIPTION("Device␣driver␣for␣AVR32␣I/O-card␣developed␣by␣Oyvind␣Netland,␣
       during␣a␣master␣thesis␣on␣NTNU,␣Trondheim,␣Norway");
109
110 #endif // __KM_AVR32_IO_H
```

### 8.2.7 avr32io_Cmd

`avr32io_Cmd.h` and `avr32io_Cmd.h` contain functions that sends messages containing the commands that the user-space program has asked for. The header file is shown below, and it includes descriptions of the structures of the different SPI messages.

```
1  #ifndef _KM_AVR32IO_CMD_H
2  #define _KM_AVR32IO_CMD_H
3
4  /*
5  Sends code until ready code is received.
6  */
7  int Cmd_WaitForAM128(u8 code);
8
9  /*
10 Sends SPI message for Analog in
11
12 Number   Send to slave      Returned by slave
13 1        code               READY
14 2        DUMMY              code +1
15 Wait for interrupt in interrupt mode.
16 3        DUMMY              vauleH
17 4        valueH + 1         vauleL
18 5        valueL + 1         DUMMY
19 6        MASTER_OK          SLAVE_OK
20 */
21 int Cmd_GetAIn(u8 *toUser, int ch);
22
23 /*
24 Sends SPI message for Analog out.
25
26 Number   Received from master    Returned to master
27 1        code                   READY
28 2        valueH                 code + 1
29 3        valueL                 vauleH + 1
30 4        DUMMY                  vauleL + 1
31 5        MASTER_OK              SLAVE_OK
32 */
33 int Cmd_SetAOut(u8 *fromUser, int ch, int sch);
34
35 /*
36 Sends SPI message for Digital in
37
38 Number   Send to slave      Returned by slave
39 1        code               READY
40 2        DUMMY              code +1
41 3        DUMMY              vaule
42 4        value + 1          DUMMY
43 5        MASTER_OK          SLAVE_OK
44 */
45 int Cmd_GetDIn(u8 *toUser);
46
47 /*
48 Sends SPI message for Digital out.
49
50 Number   Received from master    Returned to master
51 1        code                   READY
52 2        mask                   code + 1
53 3        value                  mask + 1
54 4        DUMMY                  vaule + 1
55 5        MASTER_OK              SLAVE_OK
56 */
57 int Cmd_SetDOut(u8 *fromUser);
58
59 #ifdef BUFFERING_MODE
60 /*
61 Sends SPI message for enable analog in.
```

```
62
63 Number   Received from master    Returned to master
64 1        code                   READY
65 2        enable                 code + 1
66 3        DUMMY                  enable + 1
67 5        MASTER_OK              SLAVE_OK
68 */
69 int Cmd_EnableAIn(u8 en, int ch);
70 #endif
71
72 /*
73 Sends SPI message for analog out resolution.
74
75 Number   Received from master    Returned to master
76 1        code                   READY
77 2        resolution             code + 1
78 3        DUMMY                  resolution + 1
79 5        MASTER_OK              SLAVE_OK
80 */
81 int Cmd_SetAOutResolution(u8 resol, int ch);
82
83 #ifdef INTERRUPT_MODE
84 DECLARE_WAIT_QUEUE_HEAD(wq); // Makes a wait queue.
85
86 // Function that runs on interrupt.
87 irqreturn_t irqFunk(int irq, void *dev_id, struct pt_regs *regs);
88 #endif
89
90 #include "avr32io_Cmd.c"
91
92 #endif // _KM_AVR32IO_CMD_H
```

When using interrupts for the analog input command, an interrupt function has to be used. This functions runs when the kernel receives an interrupt, and it can wake up a sleeping analog input command. These functions are all very similar, and below the `Cmd_SetAOut()` function are showed as an example of the functions in the code file.

Line 72 of this function checks that the channel is a valid channel, since the function from 8.2.2 will return an negative channel number if a wrong minor number was used. The next part uses a function that synchronizes with the ATmega128 by sending the code until the ATmega128 returns the correct "ready" code.

After synchronizing, the function sends data to the ATmega128. This is done by the function call on line 88. The arrays sent with this function contain the data that should be sent, and the data it expects the ATmega128 will return. All three bytes are sent at the same time, because it's faster than to send them individually.

If the data transfer was successful, the function ends by sending a "master OK" code to the slave. If it receives a "slave OK" signal, the function returns successfully.

```
66 int Cmd_SetAOut(u8 *fromUser, int ch, int sch)
67 {
68    int ret;
69    unsigned char tx[3], expectedRx[3];
70
71    // check for valid channel.
72    if(ch < 0) return -EIO;
73
74    // Synchronize with ATMega128
75    ret = Cmd_WaitForAM128(CH_TO_CODE_AOUT(ch, sch));
76    DEBUG("code: %i count: %i\n", CH_TO_CODE_AOUT(ch, sch), ret);
77    if (ret < 0) return ret;
78
```

```
79      // Prepare to send several bytes.
80      tx[0] = fromUser[0];
81      tx[1] = fromUser[1];
82      tx[2] = DUMMY;
83      expectedRx[0] = CH_TO_CODE_AOUT(ch, sch) + 1;
84      expectedRx[1] = fromUser[0] + 1;
85      expectedRx[2] = fromUser[1] + 1;
86
87      // Send data and check returns.
88      ret = SPI_SendTableOfBytes(tx, expectedRx, 3);
89      if(ret < 0) return ret;
90
91      // Confirm command.
92      ret = SPI_MasterTransferByte(MASTER_OK);
93      DEBUG("masterOK: %i slaveOK: %i\n", MASTER_OK, ret);
94      if(ret != SLAVE_OK) return -EIO;
95
96      return 0;
97  }
```

### 8.2.8 avr32io_SPI

`avr32io_SPI.h` and `avr32io_SPI.h` contain functions for doing single or multiple SPI transfers. These functions are used by `avr32io_Cmd.c`. The header file is shown below.

```
1  #ifndef __KM_AVR32IO_SPI_H
2  #define __KM_AVR32IO_SPI_H
3
4  // Function that transfer a given number of bytes over SPI.
5  int SPI_MasterTransfer(u8 *rx, u8 *tx, int len);
6
7  // Function that transfer one byte over SPI.
8  int SPI_MasterTransferByte(u8 tx);
9
10 // Function that sends a table of data, and checks the received data against
       another table
11 int SPI_SendTableOfBytes(u8 *tx, u8 *expectedRx, int len);
12
13 #include "avr32io_SPI.c"
14
15 #endif // __KM_AVR32IO_SPI_H
```

The function `SPI_MasterTransfer()` describes how the kernel module does a SPI transfer. As seen below it used two data structures that contains the data and length of the transfer. Since these two structures has to be defined and initialized, there are some overhead work each time some data are transferred. This makes it, as mentioned in 8.2.7, faster to send three bytes in one transfer than using one transfer for each.

```
2  int SPI_MasterTransfer(u8 *rx, u8 *tx, int len)
3  {
4      // Structs for the transfer.
5      struct spi_message msg;
6      struct spi_transfer transfer;
7
8      // Makes the transferstruct.
9      memset(&transfer, 0, sizeof(transfer));
10     transfer.len = len;
11     transfer.rx_buf = rx;
12     transfer.tx_buf = tx;
13
14     // Makes the messagestruct, and includes the transfer in it.
15     spi_message_init(&msg);
16     spi_message_add_tail(&transfer, &msg);
```

```
17
18      // Execute transfer and return result.
19      return spi_sync(device->spi, &msg);
20  }
```

## 8.3 Linux Kernel Patch

Compiling the device driver as a module was a practical solution during development. This made it possible to quickly recompile and install the module when changing the code. For users that don't want to change the source code of the device drivers, it's easier to have it compiled into the kernel. This would mean that the device driver starts up during start up, and it will work until the system shuts down. To make this as easy as possible for users, a Linux kernel patch was made that includes the driver into the kernel. This patch are included on the CR-ROM and the instruction for patching and compiling the kernel are in 12.1.2.

## 8.4 User Mode Driver

The user mode driver consists of functions that open, read/write and close the device nodes, and by doing this give commands to the I/O-card. The different functions and what they do are described int he header file `avr32io_driver` below, and the code file are in the appendix C.3.1.

```
1  #ifndef _AVRDRIVER_H
2  #define _AVRDRIVER_H
3
4  #include <sys/ioctl.h>
5  #include <unistd.h>
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <fcntl.h>
9  #include <math.h>
10 #include <pthread.h>
11 #include <linux/ioctl.h>
12
13 //Defines IOCTL numbers.
14 #define AVR32IO_IOCTL_MAGIC 'k'
15 #define AVR32IO_IOCTL_RES_AOUT _IOW(AVR32IO_IOCTL_MAGIC, 0, int)
16 #define AVR32IO_IOCTL_EN_AIN _IOW(AVR32IO_IOCTL_MAGIC, 1, int)
17
18 // Defines if driver should print out debugging information.
19 //#define VERBOSE_DEBUG
20 #ifdef VERBOSE_DEBUG
21         #define DEBUG(fmt, args...) { printf(fmt, ## args); }
22 #else
23         #define DEBUG(fmt, args...)
24 #endif
25
26 // Get integer result from analog in channel.
27 long avr32io_GetAnalogIn(char channel);
28
29 // Set integer value to analog out channel and subchannel.
30 short avr32io_SetAnalogOut(char channel, char subchannel, unsigned short value);
31
32 // Set double value to analog out channel and subchannel.
33 short avr32io_SetAnalogOutDouble(char channel, char subchannel, double value);
34
```

```
35 // Get digital in byte.
36 short avr32io_GetDigitalIn();
37
38 // Get single bit of digital in.
39 short avr32io_GetDigitalInBit(char bit);
40
41 // Set digital out with a mask and value.
42 short avr32io_SetDigitalOut(unsigned char mask, unsigned char value);
43
44 // Set just a single digital out bit.
45 short avr32io_SetDigitalOutBit(char bit, char value);
46
47 // Set a new analog out resolution for a channel.
48 short avr32io_SetAnalogOutResolution(char channel, char resolution);
49
50 // Enables or disables an analog in channel (will only work in BUFFERING_MODE)
51 short avr32io_EnableAnalogIn(char channel, char enable);
52
53 // Containing a double value for as fast as possible convert 0-5 double value to
       integer with the correct resolution.
54 double aoutMultiplier[2];
55
56 // Mutex used so only one thread can access the SPI at the same time.
57 pthread_mutex_t spiMutex = PTHREAD_MUTEX_INITIALIZER;
58
59 #include "avr32io_driver.c"
60
61 #endif // _AVRDRIVER_H
```

The `avr32io_SetAnalogOut()` function are shown below, and are an example on how these functions works. The function are called from a normal user mode program that has included the header file, and the new value for the analog out are sent as a parameter. The function then opens the correct device node, and writes the data to it. The data that originally was a 16-bit integer are converted into two 8-bit integers so the 8-bit ATmega128 can use it.

```
34 short avr32io_SetAnalogOut(char channel, char subchannel, unsigned short value)
35 {
36    int ret, fd;
37    char devFile[14];
38    unsigned char data[3];
39
40    // Prepare data to send.
41    data[0] = subchannel;
42    data[1] = value >> 8;
43    data[2] = value;
44
45    pthread_mutex_lock(&spiMutex);
46
47    // Open the correct device node.
48    sprintf(devFile, "/dev/avr32Aout%i", channel);
49    fd = open(devFile, O_RDWR);
50    if(fd < 0){
51       DEBUG("avr32io_user: error on open: %i\n", fd);
52       return fd;
53    }
54
55    // Write data to device node and close it.
56    ret = write(fd, data, 3);
57    close(fd);
58    if(ret < 0){
59       DEBUG("avr32io_user: error on write: %i\n", ret);
60       return ret;
61    }
62
63    pthread_mutex_unlock(&spiMutex);
```

```
64
65    return 0;
66 }
```

## 8.5 Threaded User Mode Driver

The threaded driver was implemented as a layer above the user mode driver, which means that it will use the functions of this driver. It should contain function calls for starting the analog I/O-threads and functions that sends and gets data to these threads. The header file for the threaded driver are below, while the code file is in the appendix C.3.2.

```
1  #include "periodicTask.h"
2  #include "avr32io_driver.h"
3
4  #define MASTER_TIMER_PERIOD 0.001
5
6  // Structure containing information about each of the analog input channels.
7  struct sAinChan{
8          int chanNumber;
9          double value;
10         pthread_mutex_t mutex;
11         struct sPeriodicTask samplingTask;
12 };
13 struct sAinChan ainChanList[8];
14
15 // Structure for list of command the worker thread has to execute.
16 struct sAoutCmd{
17    int chanNumber;
18         int subchanNumber;
19         double value;
20    struct sAoutCmd *next;
21 };
22 struct sAoutCmd *aoutCmdListFirst, *aoutCmdListLast;
23
24 pthread_t aoutWorker; // Worker thread for analog out.
25 pthread_mutex_t cmdListMutex = PTHREAD_MUTEX_INITIALIZER; // Mutex for command list
       .
26 pthread_mutex_t workMutex = PTHREAD_MUTEX_INITIALIZER; // Mutex for new command
       signal.
27 pthread_cond_t workCond = PTHREAD_COND_INITIALIZER; // New command signal.
28 int workCount = 0; // Number of commands in list.
29 int IO_Started = 0; // Protects the Init_IO() function from beeing run several
       times.
30
31 // Initialize IO.
32 void Init_IO();
33
34 // Initialize an analog input channel, and start a periodic thread that samples
       this channel.
35 int avr32io_InitAnalogIn(int chanNumber, double period);
36
37 // Function for analog input sampling thread.
38 void *Sampling_Ain(void *ptr);
39
40 // Function for retriving analog input value from buffer. For some reason returning
        a double didn't work with the RTW program, so the value will be returned in a
       pointer instead.
41 int avr32io_GetAnalogInThread(int chanNumber, double *data);
42
43 // Give command to the worker thread about a new analog output value.
44 int avr32io_SetAnalogOutThread(int chanNumber, int subchanNumber, double value);
45
46 // Function for the worker thread, used to execute commands with new analog output
       values.
```

```
47 void *Worker_Aout();
48
49 #include "avr32io_threads.c"
```

### 8.5.1 Threaded Analog Input

To make periodic tasks that samples from the I/O-card, this driver uses the `periodicTask` library, that allows to easily make periodic threads. These tasks samples from the I/O-card and store the analog values in variables, that are protected by a mutex. A function `avr32io_GetAnalogInThread` is used by the main thread to get the currently buffered value.

The SPI connection is also protected by a mutex, to avoid that two threads tries to use it at the same time. The device driver also has protection against this, that will return an error if the SPI connection is busy. By using a mutex here, these errors are avoided, and it's therefore a better solution.

### 8.5.2 Threaded Analog Output

The worker thread starts a forever loop and waits for a signal from the main thread. This signal is a `pthread` condition variable, and is the same type of signals that are used to control the periodic tasks in `periodicTaks`. When the worker thread receives a signal it means that there is one or more items in the command list. This list is a single linked list that contain information about new values to analog out channels. After receiving the signal, the thread starts executing the commands in the list, one at the time until the list it empty. This list has to be protected with mutexes as well, so new items won't get added to the list while the worker thread reads a command from it.

# Chapter 9

# Testing with prototype card

This chapter desribes tests performed on the prototype card and the software developed for it. The parts that are tested, are the API communication and the analog I/O.

## 9.1 Test of SPI communication

### 9.1.1 Frequency of SPI connection

In 5.5.4 the highest possible SPI frequency was about $1.9MHz$, which was about one quarter of the clock frequency of STK500. When using a clock twice as high, the ATmega128 should be able to receive SPI transfers with twice as high frequency. In `avr32io.c` the following line sets the SPI frequency to the highest that works with ATmega128 as slave.

```
spi->max_speed_hz = 2100000;
```

When using an oscilloscope to probe the SPI clock, the time used by one high and one low period was measured to $0.25\mu s$. This is the time used to transfer one bit of data. This gives a frequency of $4MHz$. This is one quarter of the frequency of the ATmega128, which is the theoretical maximum.

### 9.1.2 Time used on SPI transfers

In 9.1.1 the time used to transfer one bit of data was measured to $0.25\mu s$. The SPI transfer consists of 8 bits of data serially, which means that the total 8 bits (= 1 byte) takes $2\mu s$. To test how much time the kernel uses before and between SPI transfers, a small test system was created that sent out pairs of data transfers. By using an oscilloscope, the time between the two pairs of data was measured. The test system could send pairs of data in two different ways. Either sending two individual bytes of data, or sending two bytes of data at the time.

When sending two individual bytes, the test system used $23.5\mu s$ from the end of the first to the start of the last byte. When sending the two bytes at the same time, this time was reduced to $9.2\mu s$. First of all, this means that the AVR32 uses much more time preparing and configuring the SPI transfer than it uses on the SPI transfer itself. It also means that

to minimize the time used, it's important to send several bytes at the same time whenever possible.

This have been used in the device driver. When sending data to the I/O-card, all the data are sent at once, and the acknowledgments returned by the I/O-card was checked afterwards. When receiving data from the I/O-card this was not possible to do, since each acknowledgment consist of the previously received byte. This means that only one byte can be sent at the time. This makes the digital input command slower than the digital output command, as seen in 9.1.3, even if they transfer the same amount of data.

### 9.1.3 Time used on Commands

How many commands that could be sent during a period of time depends on how long it takes to send a command. To find out this, a test was designed to find the average time used by each command when doing 10000 equal commands. When testing the analog out command the code below was used.

```
StartStopWatch(&sw);
for(i=0;i<COUNT;i++){
        ret = avrIO_SetAnalogOut(0, 0, 2.5);
        if(ret < 0) break;
}
usec = StopStopWatch(&sw);
printf("Analog out result: %i usec: %i\n", ret, usec/COUNT);
```

Table 9.1 shows the average number of microseconds the different commands used. It shows that the commands use between $180\mu s$ and $400\mu s$. This means that only a two to five commands can be sent each millisecond.

Table 9.1: Time used for executing the different commands

| Command | Average time |
|---|---|
| Enable analog in channel | $229\mu s$ |
| Change analog out resolution | $233\mu s$ |
| Get analog in voltage continuous mode | $286\mu s$ |
| Get analog in voltage interrupt mode | $406\mu s$ |
| Set analog out voltage | $250\mu s$ |
| Get digital in | $217\mu s$ |
| Set digital out | $182\mu s$ |

### 9.1.4 Reliability

When testing the average time used by each command, the reliability of the communication was tested as well. If any of the 10000 commands returned an error, the test would abort. The test was performed several times, with almost no errors, so it's safe to say that the communication is almost free for errors. Since the transfer nearly error free, the error detection might not be necessary.

### 9.1.5 Reliability when using Timeout

The ATmega128 can use timeouts when it waits for SPI transfers, to avoid the possibility of ending in an unknown state. The problems with this is that a timeout may be triggered before the AVR32 is able to send the next SPI transfer. A test was performed to find out how often errors occurs for different timeout values. The results are in table 9.2. The results show that a high timeout has to be selected to avoid errors, but when using a timeout about $100\mu s$ the chance of error is low enough to be ignored.

Table 9.2: Chance of error with different slave SPI timeouts

| Timeout | $\mu s$ | Error % |
|---|---|---|
| 1 | 20 | 59.1% |
| 2 | 40 | 5.85% |
| 3 | 60 | 5.23% |
| 4 | 80 | 0.70% |
| 6 | 120 | 0.020% |
| 8 | 160 | 0.005% |
| 10 | 200 | 0.004% |
| 15 | 300 | 0.004% |
| 20 | 400 | 0.004% |
| 25 | 500 | 0.000% |

## 9.2 Test of Analog Output

### 9.2.1 Simulation

One reason for making a prototype card was to try out different filters for the analog out channels. To convert PWM signals to analog signals, low-pass filters were used, as explained in 5.3. As explained in 5.3.3, a filter has a $RC$ value that determine the response time. With Simulink simulations, the value $RC = 1.0 * 10^{-4}$ was found to give a response time of about $0.3ms$. This is shown in figure 9.1

### 9.2.2 Testing Filters

In 9.2.1 Simulink simulations was used to find that when testing filters, they should have a $RC$ values around $1e^{-4}$. Three different filters were tested, one higher, one lower and one approximately this value. The ripple and response time where measured with an oscilloscope for the different filters with 8, 9, 10, 11 and 12 bits resolutions. Since the response time does not change for different resolution, only the value for 8-bit was measured. Table 9.3 shows the results.

These results shows the that ripple increases for higher resolution and for lower $RC$ values. It also shows that the acutual response time was higher than the simulated response time. $RC = 1.0 * 10^{-4}$ gave a response time of $0.5ms$. A higher response time than this is not acceptable if the system should be able to have a fast changing analog output. The final card should have $RC$ values around this value.

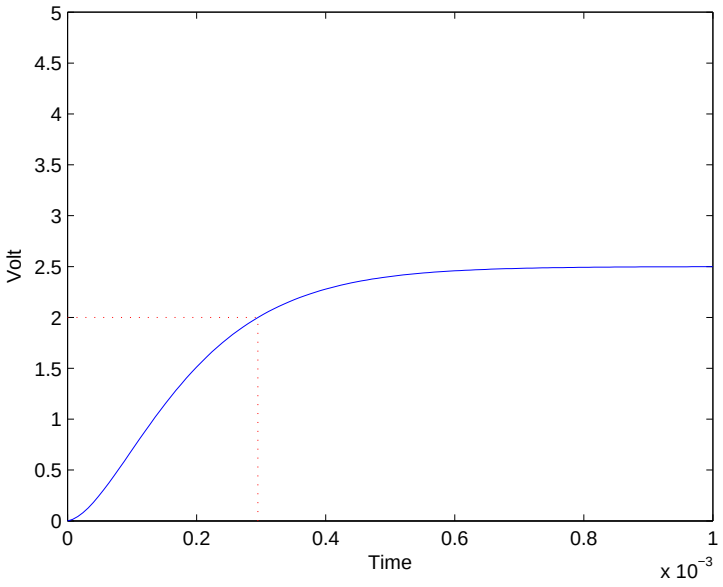


Figure 9.1: $RC = 1.0 * 10^{-4}$ gives a response time of $0.3ms$.

Table 9.3: Results of analog output filters

| $R$ | $C$ | $RC$ | Response time | Resolution | Ripple |
|---|---|---|---|---|---|
| $0.36k\Omega$ | $100nF$ | $0.36 * 10^{-4}$ | $0.2ms$ | 8-bit | $20mV$ |
| | | | | 9-bit | $75mV$ |
| | | | | 10-bit | $222mV$ |
| | | | | 11-bit | $640mV$ |
| | | | | 12-bit | $1400mV$ |
| $1k\Omega$ | $100nF$ | $1.0 * 10^{-4}$ | $0.5ms$ | 8-bit | $5mV$ |
| | | | | 9-bit | $15mV$ |
| | | | | 10-bit | $45mV$ |
| | | | | 11-bit | $140mV$ |
| | | | | 12-bit | $400mV$ |
| $3k\Omega$ | $100nF$ | $3.0 * 10^{-4}$ | $1.3ms$ | 8-bit | $< 5mV$ |
| | | | | 9-bit | $5mV$ |
| | | | | 10-bit | $10mV$ |
| | | | | 11-bit | $25mV$ |
| | | | | 12-bit | $70mV$ |

### 9.2.3 Operational Amplifiers

When connecting the op amps to the prototype card, the op amps was only able to give out voltages between $2V$ and $3V$. This is because the op amp that was used (MC1458) could not give outputs that was less than $(V^- + 2V)$ or more than $(V^+ - 2V)$. Since $V^- = GND$ and $V^+ = V_{CC}$ the output range became limited. The final version of the card should have another type of op-amps and these will need other supply voltages.

## 9.3 Test of Analog Input

### 9.3.1 Precision of Analog Input

In 5.2.1 the precision of the ADC of ATmega128 was tested. The result was not very precise. When testing the ADC of the prototype card the AVR32 used the analog input command to do the conversion. Table 9.4 shows that the analog input part of the I/O-card gives very accurate results. On the prototype card, the inductor of the LC-filter was replaced with a short circuit, bu it didn't seem like this affected the result.

Table 9.4: Results of ADC test with prototype card

| Voltage | Multimeter voltage | ADC result in voltage |
|---|---|---|
| $2.5V$ | $2.50V$ | $2.47V$ |
| $3.3V$ | $3.27V$ | $3.26V$ |
| $5V$ | $5.00V$ | $5.00V$ |

### 9.3.2 Buffering and Interrupt mode

The design and implementation of the analog input was done in two ways, and both gives correct values. In 9.1.3 the time used by the two different modes was measured, and the result was that with interrupts the command used about $120\mu s$ longer. This is just a bit more than the $104\mu s$ that an ADC conversion should take according to 7.2.2, and is as expected.

The advantage of using interrupts is that the returned result are a newly converted result. When using buffering mode, there is no way to know how old the result are. If many channels are used, the result might be almost $1ms$ old. If few channels are used, the result will be almost new. The advantage of saving almost one third of the time used for getting an analog input value means that buffering mode will be used in the rest of the thesis. Interrupt mode will be available in the code, and can be used by changing the header files of both the ATmega128 code and the device driver.

# Chapter 10

# Final version of I/O-card

The purpose of the prototype card was to make a customizable card with good debugging capabilities, while the new and final version of the card was designed to be of less physical size and easier to use. The new card also had corrected the flaws of the prototype card and introduced a few extra features.

## 10.1 Changes from Prototype

- Used SMD components when possible, which makes the card smaller.
- Removed the RS-232 connection, since this was only used for debugging. This makes the card smaller, and also reduced total cost of the card, since the MAX233 is fairly expensive. The UART signals of the ATmega128 were made available on headers, so it's possible to use them if needed.
- Added header interface for the NGW board. This is a new AVR32 development board that has a slightly different expansion header than STK1000. The headers are also smaller, and only use the necessary signals. The interrupt signal was no longer sent through the normal header, but has to use another wire between a interrupt header on the I/O card and the header on the STK1000/NGW.
- Used headers for digital I/O instead of screw clamps. The choice of screw clamps for these signals was a mistake, since they took to much space. Headers are smaller and equally easy to use. They are also commonly used for digital signals.
- Added a bridge rectifier before the voltage regulator. This means that the polarity of the external power source doesn't matter. Without a rectifier, the voltage regulator might be broken if a negative voltage is applied. This makes the card more "fool-proof".
- Added a screw clamp power supply in addition to the ordinary power jack plug. This makes it easy to connect different power sources.
- On the new card all screw clamps and headers are marked with numbers that corresponds to the number of pin on ATmega128. Analog in channel 0 is the same as the

ADC0 pin on ATmega128, and so on. Ordering these signals gives a slightly more complex routing, but it's more practical and easy to use.

- A new type of op amps were used, to correct the flaw described in 9.2.3. These op-amps are single supply op amps, that are able to give output between $GND$ and $(V^+ - 2V)$. As $V^+$ the rectified but unregulated power supply was used, since the op amp don't need regulated power supply. This means that the output range of the op amp are defined by the supply voltage of the I/O-card. The disadvantage of this solution is that the op-amps, and therefor the analog output, will not work when running on power from the STK1000 board.
- All three subchannels on the first channel (channel 0) of the analog out was designed with two jumpers each. One of these jumpers were used to bypass the filter, meaning that the output will be a PWM signal instead of a analog signal. This is useful when using components like servos. The other jumper are used to change the op-amp circuit from being a voltage follower to a non-inverting amplifier that amplifies the $0V - 5V$ signal to a $0V - 10V$ signal. To use this option, the power supply has to be high enough so the op-amps can give out $10V$.
- The PWM filters were made with resistors with value $R = 3.9k\Omega$ and capacitors with value $C = 22nF$, giving them $RC = 8.58 * 10^{-5}$. This should give a response time around half a millisecond.

## 10.2 Schematic

The whole schematic of the final version of the card are in the appendix B.2. This schematic is the same as the prototype except for the changes described in 10.1

## 10.3 Layout

The layout of the final I/O-card are shown in 10.1 on the facing page and 10.2 on the next page. Since it used SMD components whenever possible and other design changes, this card was almost half the size of the prototype. Since more components are SMD, almost all of them were placed on the top layer of the card instead of the bottom layer on the prototype. Since the I/O-signals were ordered after pin numbers, it meant that the routing of signals was more complex than for the prototype card. This made the gain of placing the ATmega128 on the bottom layer less than for the prototype card, since both solutions would be equally complex. Therefor the top layer was chosen because of cosmetic reasons.

Figure 10.3 on page 86 is an image of an almost assembeled final version of the I/O-card.

## 10.4 Problems with Analog Output

The new I/O-card also had problems with the analog output. The operational amplifiers and the supply voltage for these worked better than on the prototype, but the filters for converting the PWM signals to analog voltages didn't work. They gave out lower voltages

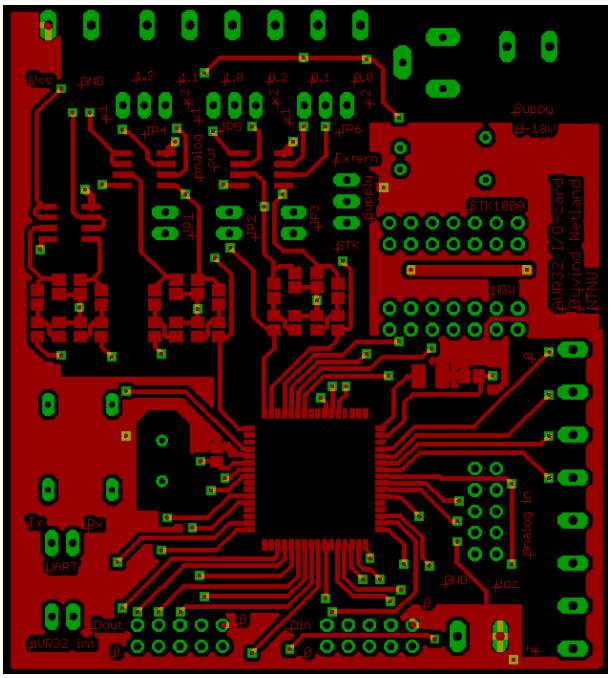


Figure 10.1: Top layer of final card. (Not to scale)

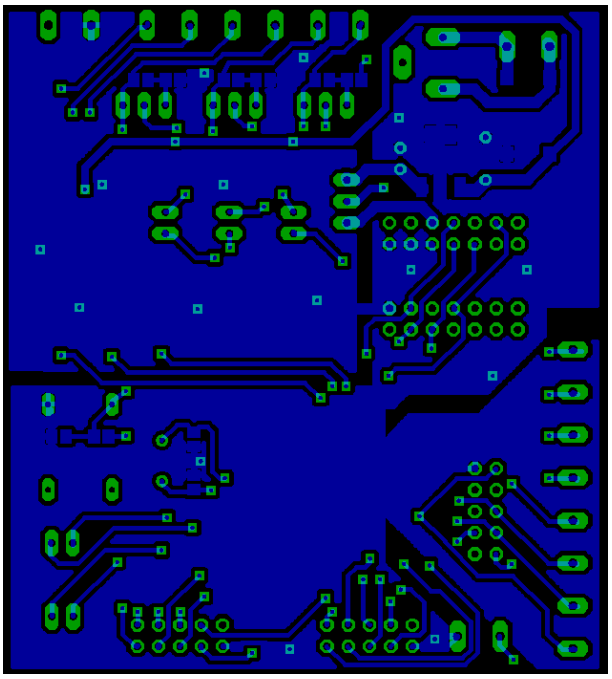

Figure 10.2: Bottom layer of final card. (Not to scale)

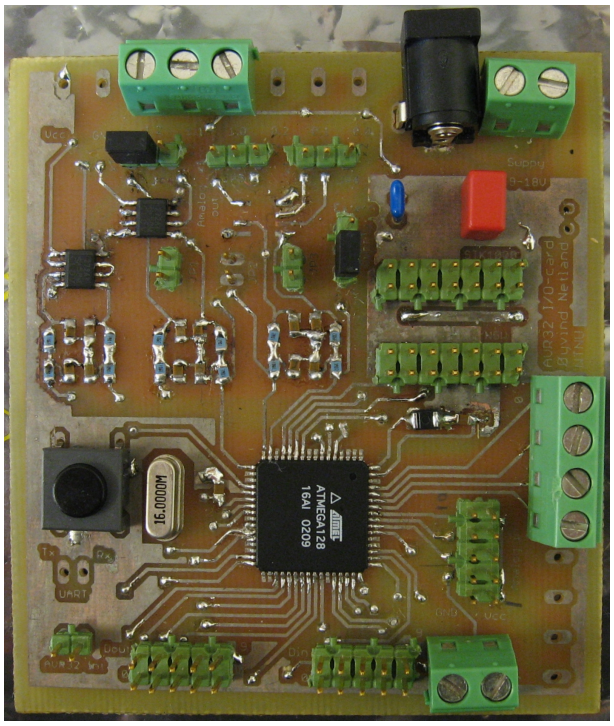


Figure 10.3: Final version of I/O-card

than they should have. The reason for this is unknown, but the filters was difficult to solder since the SMD resistors and capacitors were very small and were placed on a small surface. Inaccurate assembly may be the reason for the poor performance of the filters. Redesign of this I/O-card should give the filters more space.

# Chapter 11

# Matlab Real-Time Workshop

## 11.1 Matlab Real-Time Workshop

Matlab Simulink can be used to design and simulate mathematical models. With the help of Matlab Real-Time Workshop (RTW for short) C-code can be generated from the Simulink model. The code will (if compiled) give a program that does the same simulation as in Simulink. This can be used for rapid prototyping control systems, as described in 2.4. How to use RTW is described in [13]

### 11.1.1 Target Language Compiler (TLC)

When Matlab Real-Time workshop generates C-code from a Simulink diagram it actually does it in two steps. The first step is to make a `.rtw` file. This file contains all the information from the Simulink diagram presented in a hierarchy The TLC language is describes in detail in [12].

The TLC is the second step of the code generation, and it generates multiple source files from the information in the `.rtw` file and several `.tlc` files. The `.rtw` file contains information about the Simulink diagram, while the different `.tlc` files contain information on how the generated code should look like.

It's two different types of `.tlc` files. The system target file specify the overall structure of the generated code, and it's selected before generating code among a list of system target files supplied with RTW. It's also possible to change existing or make new system target files. For rapid prototyping the general real-time target (*grt*) is a good starting point. This makes C-code that can run on most platforms, as shown in 4.3.

Code generated with *grt*, will as in Simulink execute the simulation as fast as possible. A control system has to execute the simulation with correct time between each time step, or it wouldn't work as intended. To make this possible with *grt*, some of the code have to be modified.

Each Simulink block has its own block target file. This file contain the information TLC needs to generate code that does the same as the block would do in Simulink.

### 11.1.2 S-functions

S-functions are Simulink blocks written in a computer programming language, like Matlabs own language M, C, C++, Ada and FORTRAN. This allows users to add their own blocks in Simulink, that can have functionality that normal Simulink blocks can't provide. When using Matlab RTW as a rapid prototyping tool for a control system, S-functions are often used to control I/O.

S-functions are written as a code file for the selected computer language, but this file has to follow a specific structure. To use the S-function the code has to be compiled using the Matlab command `mex`. As for other compiled program, this file has to be compiled again each time changes have been made.

When using RTW, it's possible to use three types of S-functions, that are described below.

- Noninlined S-functions are S-functions that doesn't have a `.tlc` file, which means that it's treated equally by Simulink and RTW. This is the easiest solution, since everything that works in Simulink will work equally in RTW, and it's not necessary to write any TLC code. Unfortunately this is not very efficient, and will need additional CPU and memory usage when running a program from the generated code.

- Fully inlined S-functions have a `.tlc` (target block) file, and the behavior of the S-function in RTW is defined by this file, and not by the S-function code file used by Simulink. This means that the S-function might not do the same thing when simulating in Simulink as, when running the program made by RTW generated code. This can both be a good and a bad feature. If the block is supposed to do the same in both cases, it will be error-prone and time consuming to upgrade two files at the same time. In some applications like controlling I/O, it might be preferable that the Simulink simulation does something different than the RTW generated code.

- Wrapper S-functions are a special type of inlined S-functions, where a third code file is used. This file has functions that both the `.tlc` file and S-function code files uses. This allows the Simulink and RTW to execute the same code as in a noninlined S-function, but without the loss of performance.

## 11.2 Making a AVR32 Real-Time Target

When using Matlab RTW, a system target has to be chosen. This system target defines how code will be generated and compiled. As mentioned in 11.1.1, the GRT system target generates code that will run on most platforms, and in 4.3 GRT generated code with a few modifications, compiled and ran on AVR32.

One of the goals with this thesis is to make a system that is easy to install and use. Because of this, a new system target was created, based on the GRT. This system target was named *AVR32 Real-Time Target* and will be installed by moving a directory into `MATLAB_ROOT/rtw/c`.

### 11.2.1 avr32.tlc

This is the system target file for the new *AVR32 Real-Time Target*, and it's the file that controls how code are generated. Since the code generated by GRT will compile and run on AVR32 with a few modifications to the Makefile, this file will be very similar to `grt.tlc`. The few changes necessary are to specify a new Makefile template and change some names.

Each system target has a short text that describes itself, which is showed in the list where users can choose system targets. To change this text, and specify a new Makefile, the following lines has to be added to the top of the `avr32.tlc` file. Any similar lines (containing `SYSTLC` or `TMF`) should be removed.

```
1 %% SYSTLC: AVR32 Real-Time Target \
2 %%     TMF: avr32.tmf MAKE: make_rtw EXTMODE: ext_comm
```

In the end of the `grt.tlc` file, a suffix for the generated code directory is specified. This should be changed to another name by using the line below instead of the original line.

```
168   rtwgensettings.BuildDirSuffix = '_avr32_rtw';
```

### 11.2.2 avr32.tmf

This is the Makefile template of the system target, that was made by doing some modifications to the `grt_unix.tmf` file. The Makefile template for GRT on UNIX/Linux platforms. The first change was to replace the "grt.tlc" with "avr32.tlc" as system target file. This is done changing the original system target file definition with the line below.

```
46 SYS_TARGET_FILE = avr32.tlc
```

The next step is to change the compiler and compiler flags used by the Makefile. This is done by adding the following lines before the "C flags" section of the original template. This should be around line 185 of the original file.

```
182 #------------------------------ AVR32 Specific ------------------------------------
183
184 #AVR32 compilator
185 CC=avr32-linux-gcc
186
187 #Removes -m32
188 OPT_OPTS = -O -ffloat-store -fPIC
189
190 #Links with pthread
191 LDFLAGS = -lpthread
192
193 #enables // comments
194 ANSI_OPTS =
```

The line starting with `CC` are used to specify the AVR32 compiler, and the line with `OPT_OPTS` are used to remove the `-m32` flag causes an error, as I discovered in 4.3. This flag specifies that the target is a 32-bit processor, and is unknown for the AVR32 compiler. With these two changes, the compilation to AVR32 works, and it produces a runnable executable.

The line starting with `LDFLAGS` means that the executable will be linked with the pthread library. This is required for using pthread, that are needed for using threads, mutexes and

other pthread functionality. The last line was used to remove the `-ansi` flag. This doesn't affect the compilation, it will only make it possible to comment code with double slash `//`.

The next issue that needs to be modified is the C file containing the main function used by the generated code. This is done by replacing the `grt_main.c` with `avr32_main.c` in line 233 of the original file. The new line should then look like this:

```
244     SRCS                    += $(MODEL).$(TARGET_LANG_EXT) avr32_main.c rt_sim.c $(
            EXT_SRC) $(SOLVER)
```

Since the new AVR32 target files are in another directory than the GRT target files, the Makefile has to be changed so it compiles the correct files. This are done by adding the following lines. The lines that are equal but contain the `rtw/c/grt` instead should be removed.

```
339 %.o : $(MATLAB_ROOT)/rtw/c/avr32/%.c
340         @$(GCC_TEST_CMD) $< $(GCC_TEST_OUT)
341         $(CC) -c $(CFLAGS) $(GCC_WALL_FLAG_MAX) $<
```

### 11.2.3 avr32main.c

The Makefile template specifies that the new system target will use a C file named `avr32_main.c`. This file is also based on the GRT C file called `grt_main.c`. As mentioned in 11.1.1, GRT version of the file makes a program that execute all time-steps as fast as possible. The reason for modifying the `avr32_main.c` is to make a control system that executes the time steps of the simulation periodically, and it has to include the files necessary to use the I/O-card.

For making the program execute the time steps periodically `periodicTask` library presented and used in 4.2 was used. The I/O-card user mode driver has to be included, since they are used by the S-functions, see 11.3. For debugging and testing of the RTW generated code, the `stopWatch` library was used. This was used to calculate how much time the AVR32 used on each of the time step.

The following lines was added after the other `#include` statements of the original file. These lines contain the different files required by this new main function file. The threaded version of the user mode driver was included, since it includes the non-threaded version.

```
48 /* AVR32 includes */
49 #include "driver/avr32io_threads.h"
50 #include "driver/periodicTask.h"
51 #include "driver/stopwatch.h"
```

The following lines are global structures and variables that are needed to make the time steps periodical and to measure the workload,

```
156 /*====================*
157  * Global AVR32 Data *
158  *====================*/
159 struct sPeriodicTask matlabSimulationPeriode;
160 struct sStopWatch timerStopWatch, workStopWatch;
161 long usecWork, usecTimer;
162 int count=0;
```

The following lines are added to make the time steps periodical. These should be added twice in the file, one time for single tasking and one for multitasking. Both places where

this function should be added are marked in the original file with a comment saying that "interrupts should be enabled here".

```
227     /*************************
228      * AVR32 Timing interrupt *
229      *************************/
230     StartStopWatch(&timerStopWatch);
231     usecWork += StopStopWatch(&workStopWatch);
232     WaitPeriodicTask(&matlabSimulationPeriode);
233     usecTimer += StopStopWatch(&timerStopWatch);
234     StartStopWatch(&timerStopWatch);
235     StartStopWatch(&workStopWatch);
236     count++;
```

The `WaitPeriodicTask` function is a function that wait until the next period before returning. This function is defined in the `periodicTask` library. Two different stopwatches (two different stopwatch structures) are used to measure the time used in each period, and how much of the period the processor are working (the time it doesn't wait for next period).

The following code are in the start of the main function, and it initialize and start the timer. `rtmGetStepSize(S)` will return the time step in seconds, and this is used to set the period time. Since that function needs the variable `S` to work, these lines have to be added last in the initialization part of the main function. This means just above the big comment block which is named "Execute the model".

```
636     /*************************
637      * Initialize AVR32 Timing  *
638      *************************/
639     InitPeriodicTasks();
640     StartPeriodicTask(&matlabSimulationPeriode, (int)(rtmGetStepSize(S) * 1000));
```

The last bit of code is used to print out the results of the time measurements. This code has to be added in the main function after the simulation is finished. It's not dependent on any RTW code, so it can be added just above the `return` statement.

```
713     /* For debugging */
714     printf("Avrage periode: %i\n", usecTimer/count);
715     printf("Avrage work: %i\n", usecWork/count);
```

## 11.3 S-functions for I/O-card

11.2 described the changes needed to compile the generated code for AVR32, and make the code execute the time steps periodically. This means that systems made in Simulink can run on AVR32, but it doesn't make it able to use the I/O-card. S-functions were made to control the I/O-card. They are, as explained in 11.1.2, Simulink blocks that are coded in a programming language.

There are different types of S-functions, but here only fully inlined S-functions were used. First of all, it's important that the program are as effective as possible, since the AVR32 aren't very fast and STK1000 have a very limited amount of RAM. That meant that noninlined functions wouldn't be a good choice. Fully inlined was preferred before wrapper, since these S-functions should behave different when running simulations in Simulink and

when running a RTW generated program on STK1000. It wouldn't make sense to try and contact the I/O-card via SPI from a workstation computer.

The following S-functions has been created:

- Analog input, reads one or more analog input channel of the I/O-card and gives values between 0 and 5.
- Threaded analog input, does the same as the normal analog input, except that it uses the threads for I/O operations.
- Analog output, takes value between 0 and 5 and writes it to the used subchannels. Can also select resolution between 8-bit and 12-bit.
- Threaded analog output, does the same as the normal analog output, except that it uses the threads for I/O operations.
- Digital input, reads all digital input channels of the I/O-card and gives either low (0) or high (5) values.
- Digital output, writes values to digital outputs of the I/O-card. Limit values can be selected for each channel, and are used to decide if the digital output should be high or low.

### 11.3.1 Simulink S-function File

The Simulink S-function file is the file that defines how the S-function works in Simulink. `sfuntmpl_basic.c` is a template file for simple S-functions, and it will be used here. Since the workstation computer that runs Simulink, can't use the I/O-card, the S-functions will have Simulink S-function files that doesn't do anything during simulation. This means that the only function from `sfuntmpl_basic.c` that will be modified is the `mdlInitializeSizes()`, that defines different sizes of the S-function.

The sizes that are changed from the template is the number of input, outputs and parameters used by the S-function. These numbers are different for the different S-functions, as described in table 11.1. Notice that the input S-functions has outputs and visa verca, since a input S-function reads from the I/O-card and send the signals out of the block.

Table 11.1: Sizes of the different S-functions

| S-function | Inputs | Outputs | Parameters |
|---|---|---|---|
| Analog input | 0 | 8 | 0 |
| Threaded analog input | 0 | 8 | 8 (sampling time) |
| Analog output | 6 | 0 | 2 (resolution) |
| Threaded analog output | 6 | 0 | 2 (resolution) |
| Digital input | 0 | 8 | 0 |
| Digital output | 1 | 0 | 8 (limit value) |

Below is the `mdlInitializeSizes()` function of the analog input S-function. This is an example of what these functions look like. The whole files are on the CD-ROM

```
45 static void mdlInitializeSizes(SimStruct *S)
46 {
47     int i;
```

```
48      /* See sfuntmpl_doc.c for more details on the macros below */
49
50      ssSetNumSFcnParams(S, 0);  /* Number of expected parameters */
51      if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
52          /* Return if number of expected != number of actual parameters */
53          return;
54      }
55
56      ssSetNumContStates(S, 0);
57      ssSetNumDiscStates(S, 0);
58
59      if (!ssSetNumInputPorts(S, 0)) return;
60
61      if (!ssSetNumOutputPorts(S, 8)) return;
62      for(i=0; i<8; i++){
63          ssSetOutputPortWidth(S, i, 1);
64      }
65
66      ssSetNumSampleTimes(S, 1);
67      ssSetNumRWork(S, 0);
68      ssSetNumIWork(S, 0);
69      ssSetNumPWork(S, 0);
70      ssSetNumModes(S, 0);
71      ssSetNumNonsampledZCs(S, 0);
72
73      ssSetOptions(S, 0);
74  }
```

### 11.3.2 Target Block Files

Target block files are written in the TLC language and defines how code should be generated for the S-function. Two TLC functions are used, the `InitializeConditions` function generates code that initialize the I/O-card, and the `Outputs` function generates code that reads or writes to the I/O-card each time step. Since the TLC language is able to find out which inputs and outputs of the blocks that are connected, code will only be generated for the connected ports.

The target block files for the S-functions are in the appendix, from C.4.1 to C.4.4. Lines start start with `%` are some kinds of TLC statements, like looping through all inputs or outputs of a S-function. There are also some `%< >` tags in these files, that are TLC variables or functions. All these will be replaced with some text in the generated code.

## 11.4 Results

Several Simulink diagrams were made to test how signals could be sent through the I/O-card. With Real-Time Workshop generated code. This was done by connecting inputs and outputs of the I/O-card together, so a signal that was sent to an output in Simulink could be read by an input in Simulink.

### 11.4.1 Results of Analog Test

As mentioned in 10.4, the analog output channels of the I/O-card did not work as planned. To be able to perform this test, a back up filter was made with regular resistors, capacitors

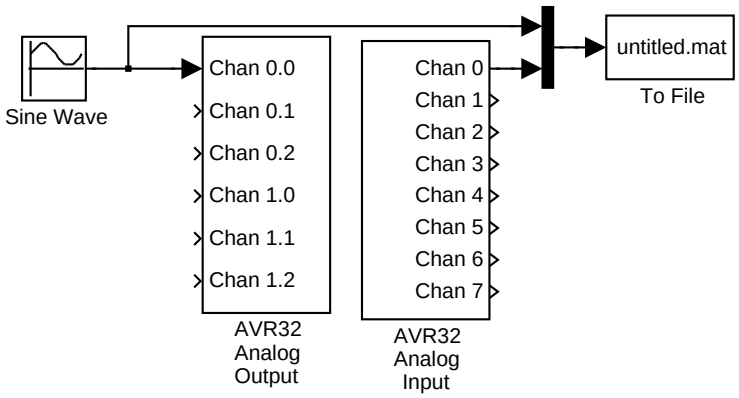


Figure 11.1: Simulink model used for testing the S-functions and the I/O-card

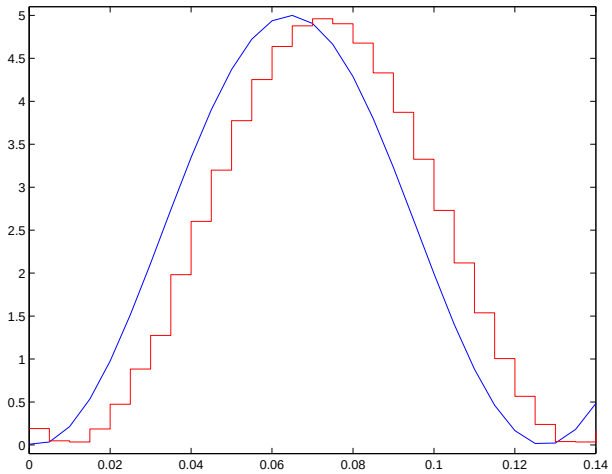


Figure 11.2: Results of sending sinus signal through analog I/O

and a breadboard[1]. This gave a much better analog value than the SMD components on the I/O-card did, which strenghten the theory that the card was assembled poorly.

A Simulink diagram was made, that contained a sinus source, a data logger and the blocks for analog input and output. This can be seen in figure 11.1. The time step used in this diagram was $5ms$. Figure 11.2 shows the original sinus signals together with sinus signals sent through the analog I/O, both by using normal and threaded I/O blocks.

Figure 11.2 shows that sinus signals that was sent through either the normal or threaded analog I/O blocks was delayed compared to the original signal. The normal blocks were delayed one time step of $5ms$, which makes sense, since it means that the value sent to the analog output one time step will be read by the analog input the next.

The threaded blocks have higher delays, about 3 time steps. This is because of the double buffering and that the complexity of the threaded drivers adds some overhead. This is a considerable delay, and means that threaded I/O gives a worse result than normal I/O. To get a performance gain from using threads, the implementation has to be optimized.

[1] A breadboard is a electronics prototype board where components can easily be connected without soldering.

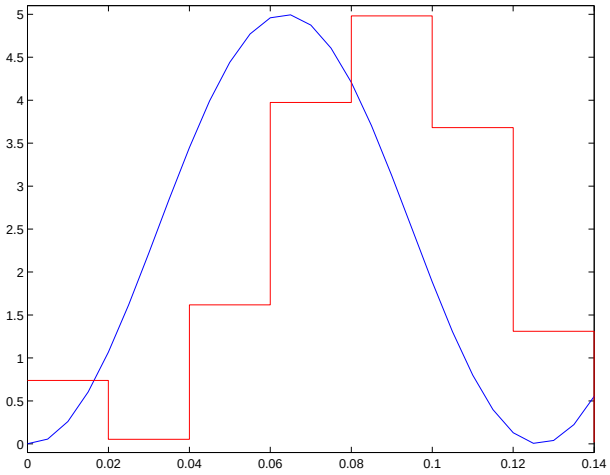


Figure 11.3: Results of sending sinus signal through analog I/O with low sampling frequency

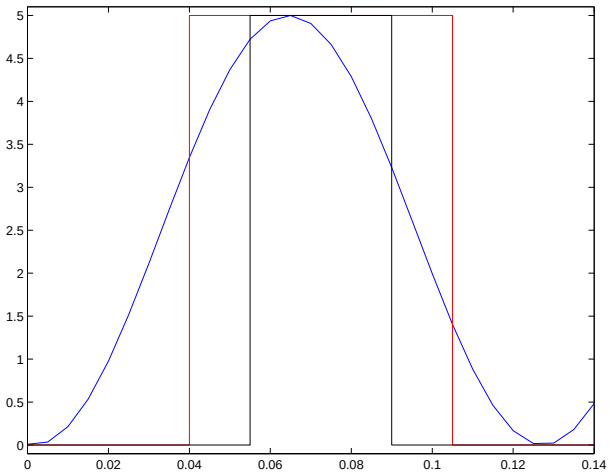


Figure 11.4: Results of sending sinus signal through digital I/O

11.3 shows what may happen if the sampling thread has a long period. Several of the Simulink time steps will read the same analog input value. This will be a problem if the input signal changes fast, but for a more stable signal it would be a good opportunity to reduce the SPI usage.

### 11.4.2 Results of Digital Test

This test was executed almost in the same way as the analog test. This means that a sinus signal was sent to one of the channels of the digital outputs, that was connected to the digital input. Since the digital output only can send $0V$ or $5V$ signals, the digital input should receive a square signal. Figure 11.4 shows how two different square signals are made in this way with a sinus signal. The two signals have different *duty-cycles*, since they have been made with different limit values in the digital output S-function.

# Chapter 12

# User manual for AVR32 I/O

In June 2007 Øyvind Netland finished a master thesis[24] about how to use the new Atmel AVR32 processor architecture as a control system. In this work a STK1000 development board was used together with an I/O-card that was developed during the thesis. This I/O-card communicates with the STK1000 through SPI, and it has 6-channel analog output, 8-channel analog input, 8-channel digital output and 8-channel digital input.

This manual describes a control system platform that consists of AVR32 Linux running on a STK1000 or NGW development board from Atmel and the AVR32 I/O-card. This platform is used together with a Linux or Windows workstation that can use Matlab Real-Time Workshop as a rapid prototyping tool. Readers of this manual should have some experience with using Linux, since not all basic Linux commands are explained. Readers should also read the documentation for AVR32 Linux on AVR freaks Wiki `http://www.avrfreaks.net/wiki` and the AVR32 Linux project `http://avr32linux.org`.

## 12.1 Installing

Before using this system platform, some tools has to be installed on the workstation. This workstation can be either a Windows or Linux workstation that has Matlab Real-Time Workshop installed. The install instructions are based on a Ubuntu Workstation, but installing on other Linux distributions should be similar.

### 12.1.1 Install AVR32 tool-chain

For Ubuntu the AVR32 tool-chain can be installed with the Apt package management system. To add the AVR32 repository to the Apt sources, the following line has to be added at the end of `/etc/apt/sources.list`.

```
deb http://www.atmel.no/beta_ware/avr32/ubuntu/dapper binary/
```

To install the tools, execute the following commands, and answer yes to all questions.

```
~$ sudo aptitude update
~$ sudo aptitude install avr32-linux-devel
```

If this approach doesn't work, or instructions for installing on other platforms, check the AVR freaks Wiki (http://www.avrfreaks.net/wiki).

### 12.1.2 Compile Linux kernel with AVR32 I/O-card support

This system uses a patched version of the Linux kernel version 2.6.20. The patches needed are the 2.6.20-avr2 patchset from `http://www.avr32linux.org` and the avr32io patch from the CD-ROM. To patch, configure and compile the kernel with AVR32 I/O-card support, Use the commands below from a directory with the Linux source tarball and the two patches.

```
~$ tar xjf linux-2.6.20.tar.bz2
~$ cd linux-2.6.20
~$ patch -p1 < ../linux-2.6.20-avr2.patch
~$ patch -p1 < ../linux-2.6.20-avr32io.patch
~$ make ARCH=avr32 CROSS_COMPILE=avr32-linux- menuconfig
~$ make ARCH=avr32 CROSS_COMPILE=avr32-linux-
```

The second last command will open a menu based configure tool for the Linux kernel. To enable AVR32 I/O, the `Atmel SPI Controller` and `AVR32 I/O-card` has to be enabled. These are both found in `device drivers/SPI support`. Another solution is to use the `linux-2.6.20-avr32io-config` file from the CD-ROM and rename it to `.config` and move it to the linux source folder. The last command will compile the kernel and make an `uImage`, which is the kernel binary file.

To be able to use the AVR32 I/O-card support, the file system of AVR32 Linux needs some device nodes for this card. These are made by running a script as root on the workstation called `mknod.sh` that are on the CD-ROM. The script has to be modified to use the correct major number and file system path. The major number are printed out during booting, or can be found when running `more /proc/devices` on AVR32 Linux. The file system path is the path to the file system of AVR32 Linux, either a SD-card or a shared folder on the workstation.

### 12.1.3 Install AVR32 support in Matlab

Matlab Real-Time Workshop can generate code that with few modifications can be compiled for AVR32, but to make it easier to use, a new system target files have been created. These are located on the CD-ROM, in the folder `matlab/avr32`. The content of this folder should be copied into `matlab_root/rtw/c/avr32` on the workstation computer, where `matlab_root` is the folder where Matlab is installed.

When copying the `avr32` directory, the subdirectory `blocks` is also copied. This directory contain the S-functions that are used to control the I/O-card. These has to be compiled for Matlab. Start up Matlab and move to the `blocks` directory. If using Linux, Matlab has to be started as root. Run the commands below to compile the S-functions.

```
~$ mex avr32io_ain.c
~$ mex avr32io_aout.c
~$ mex avr32io_din.c
~$ mex avr32io_dout.c
~$ mex avr32io_ain_thread.c
~$ mex avr32io_aout_thread.c
```

If this gives an error, the mex compiler has to be configured. The following command starts a simple interactive configuration.

```
~$  mex -s
```

To use the S-functions in the library, this directory has to be in Matlabs path variable. To add the path use the following commands in Linux, or similar if Matlab is installed somewhere else.

```
~$ addpath /usr/local/matlab/rtw/c/avr32/blocks
~$ savepath
```

To add the path when using Windows, the following commands should be used:

```
~$ addpath c:\Program Files\Matlab\rtw\c\avr32\blocks
~$ savepath
```

Now the S-functions for the I/O-card are available in the library in the `blocks` directory, and they can be used in other Simulink models.

## 12.2 AVR32 I/O-card

### 12.2.1 Hardware description

AVR32 I/O-card is an add-on card for STK1000. It provides the following I/O:

- 8-channel analog input, with a resolution of 10 bits. The analog value can be between $0V - 5V$, where $0V$ equals ground.
- 6-channel analog output, with a resolution between 8 and 12 bits. 3 of the channels has output between $0V - 5V$, where $0V$ equals ground. The 3 other channels have jumpers for selecting output type, see 12.2.4.
- 8-channel digital input.
- 8-channel digital output.

Figure 12.1 show the front side of the I/O-card, below are a description of the different parts:

1. Screw clamps for ground and $V_{CC}$, that gives easy access to these signals.
2. Jumper for selecting if analog output channels $0.0 - 0.2$ should have $5V$ or $10V$ output.
3. Screw clamps for analog output signals.
4. Connector or screw clamp for external power supply. The supply should be between $9V$ and $18V$, and the polarity doesn't matter, because of a bridge rectifier.
5. Jumper for selecting if external power supply or supply from STK1000 should be used.
6. Operational amplifiers used as voltage followers or non-inverting amplifiers.

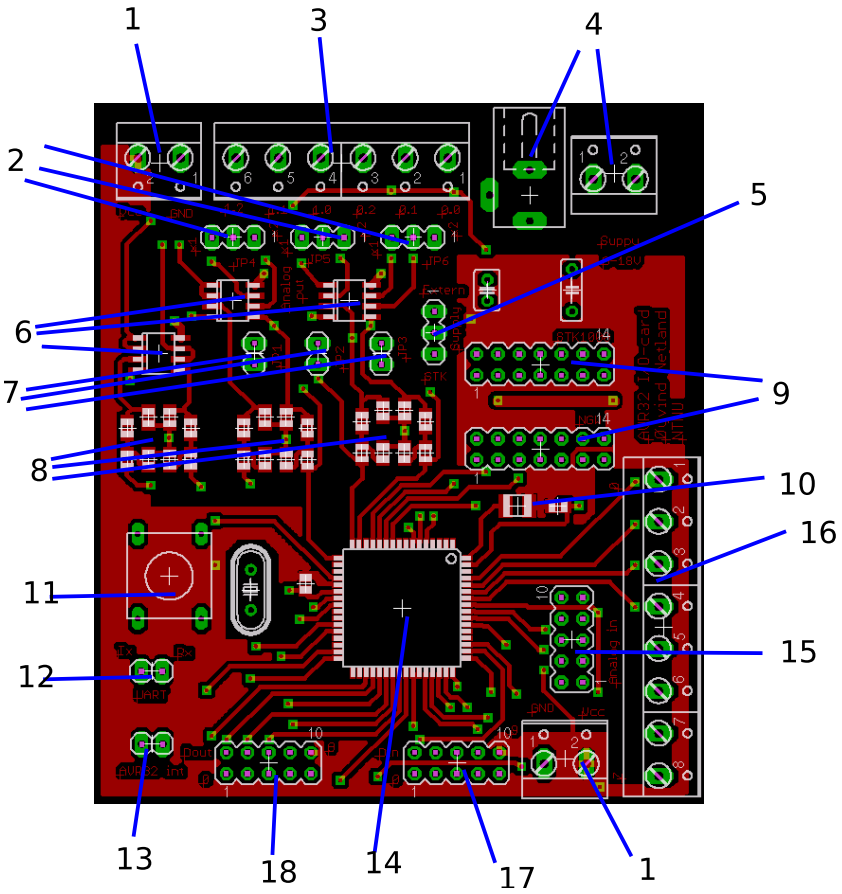


Figure 12.1: I/O-card

7. Jumpers for bypassing the filter which means that the analog output will become a PWM signal.
8. Passive filters that are used to convert a PWM signal into an analog voltage.
9. Headers for connecting STK1000 or NGW development boards.
10. LC-filter for the analog in power-supply.
11. Reset button
12. UART headers that can be used for debugging.
13. AVR32 interrupt signals, that can be connected to STK1000/NGW if interrupts are used. Not normally used, but might be useful for further development.
14. ATmega128 is the microcontroller that controls the I/O-card.
15. JTAG header for programming and debug ATmega128.
16. Screw clamps for analog input signals.
17. Header for digital input signals.
18. Header for digital output signals.

### 12.2.2 Connecting I/O-card

The I/O-card connects to either a STK1000 or NGW board through a 14 pin header cable. On STK1000 cards this header cable connects from the header marked *STK* on the I/O-card, to the general expansion header marked *J29* on STK1000. On NGW it connects from the header marked *NGW* to the general expansion header called *J5* on NGW. On both boards the header has to be connected to pins 0-13. If interrupts are used, interrupt signals from the I/O-card has to be connected to the STK1000 or NGW boards aswell.

### 12.2.3 Analog Input

The card has 8 Analog input channels with 10-bit resolution. Each of these can measure an voltage between $0V$ and $5V$ above $GND$. The I/O-card will continuously convert analog values into digital values and buffer them. Each analog input channel that are used increased the time a value is stored in the buffer before it's updated by about $110\mu s$. This means that if four channels are used, the values are stored about $440\mu s$ before they are updated.

### 12.2.4 Analog Output

The card has 2 Analog output channels with 8-bit to 12-bit resolution. Each of these channels have 3 subchannels that can have different analog values. The analog outputs are generated with PWM timers and passive filters. This means that a high resolution will give high ripple on the analog output voltage. Table 12.1 contains the maximum ripple amplitude for different resoultions.

Table 12.1: Rippe for different resolutions

| Resolution | Ripple |
|---|---|
| 8-bit | $10mV$ |
| 9-bit | $20mV$ |
| 10-bit | $60mV$ |
| 11-bit | $180mV$ |
| 12-bit | $500mV$ |

Channel 0 has jumpers that can select the output range of the analog output to be either $(0V - 5V)$ or $(0V - 10V)$. And other jumpers that can be used to bypass the filter, so the output becomes a PWM signal instead of a analog signal. These jumpers are on the card to make it possible control as many instruments as possible.

## 12.3 Rapid prototyping with Matlab Real-Time Workshop

### 12.3.1 AVR32 System Target

The AVR32 system target are used to easily configure Real-Time Workshop to generate and compile code correctly for rapid prototyping on AVR32. In addition to selecting this system target file, some Simulink preferences has to be changed. The solver has to be a fixed-step solver, since the generated code has to use a constant time step. It's also a good idea to specify the time step, since Matlab might try and use a lower period than the AVR32 is able to run. The minimum time step is $1ms$, but a complex system has to use a higher time step.

The code automatically logs the average period time, and the average time used for work each period, and this information will be displayed after completing the simulation. This makes it easy to find out if the periods are stable and if the AVR32 are able to complete each time step inside the period.

### 12.3.2 S-functions for I/O-card

Four S-functions have been made for each of the four I/O on the card, analog input, analog output, digital input and digital output.

The analog input has 8 channels, that gives a value between 0 and 5, which is an analog voltage between $0V$ and $5V$. Each channel that are connected is an active channel. This means that the generated code will read from the I/O-card, and this takes approximately $300\mu s$. If connected, a channel will be read, even if the value isn't used for anything useful.

The analog output has 2 channels with 3 subchannels each. Each channel can choose a resolution between 8-bit and 12-bit. The I/O-card will give out a voltage between $0V$ and $5V$, depending on the value of the S-function blocks input. Each channel connected to the will be written to each period, and it will use about $250\mu s$.

The digital input has 8 channels, and each channel gives out the value 0 if the channel is low and the value 5 if the channel is high. If any digital input channels are used, it will be

read each period and it takes $220\mu s$ no matter how many of the channels that are in use.

The digital output has 8 channels, and each channel can be configured with a threshold value. For all values over this threshold the digital output channel will give a voltage of $5V$, and all values under it will give a voltage of $0V$.

# Chapter 13

# Discussion

The work in this thesis has consisted of developing the hardware and software necessary for using the a AVR32 processor in a control system. The work has consisted of several smaller tasks, that had to work together. Each of these tasks are discussed separately

## 13.1 AVR32 Linux

Atmel Norway have done a good job when porting the Linux kernel and development tools to the AVR32 architecture, but when the work on this thesis began (January 2007), some of the ported software had some problems. Especially AVR32 Linux version 2.6.16 was problematic, as described in 3.2. By the end of this thesis (June 2000), both the Linux kernel version and development tools was more mature and also easy to install on Windows and on popular Linux distributions like Ubuntu and Fedora.

A small library for making periodic threads and tasks was developed, since control system needs to execute work periodically. This library used the Linux timer, which gave acceptable accuracy. The shortest period that was possible to make was $1ms$, that turned out to be more than enough because of the speed of the AVR32.

## 13.2 I/O-card

The I/O-card has an ATmega128 microcontroller that controls everything on the card. It communicates with the AVR32, sets analog and digital output and measure analog and digital input. Since this component does all this, only a few other components are needed on the card. The disadvantage of using a ATmega128 for analog I/O, is that making an analog output signal from a PWM-signal isn't the best solution and it needs many additional components like passive filters and operational amplifiers circuits. The ADC (analog input) of the ATmega128 is not the best ADC available.

The analog output part of the I/O-card didn't work as planned. The prototype card had working low-pass filters, but a wrong type of operational amplifiers was used. The final card on the other hand had a better type of operational amplifiers, but the low-pass filters didn't work as they should have. The most likely reason for this is that the filters were

poorly assembled. Design of future versions of this card should give the filters more room to make the assembly easier. Apart from the problems with the analog output, all the parts of the I/O-card worked as planned.

Two different solutions were developed for using the ADC of the ATmega128. The reason for this was that the ADC conversion takes a significant amount of time. This was solved by either interrupting the AVR32 when the ADC conversion was finished, or continuously convert analog values and store them in a buffer. Since the drivers of the I/O-card doesn't use threading the continuous mode is the best choice, specially if only some of the Analog input channels are used. The interrupt mode would be a better choice if a thread was running sending the analog input command, since then the rest of the program could use the CPU cycles wasted while waiting for the interrupt from the I/O-card.

## 13.3 I/O-card Communication and Drivers

The communication between the AVR32 and the I/O-card was successfully implemented with SPI. A protocol was developed for this communication, that the AVR32 uses to give commands to the I/O-card. This protocol has effective error detection, so both the AVR32 and the I/O-card will detect if an error occur. The error detection means that all data passing between the two processors are returned to the sender for confirmation. This uses the full duplex capability of SPI, but it still takes longer to send a command than it would without error detection.

Sending a command to the I/O-card takes between $180\mu s$ and $400\mu s$. This is a considerable amount of time, specially if the control system uses small time steps. The main reason that sending a command takes this long is that the SPI driver for AVR32. Transferring one byte takes only $2\mu s$, but when sending several bytes after each other, the AVR32 uses at least $9\mu s$ between each transfer. If two bytes are sent individually after each other, as much as $23\mu s$ was used between the transfers.

The SPI driver is obviously not very effective, since so much time is used before and between transfers. For the implementation of the I/O-card driver it meant that it was important to try and avoid transferring individual bytes, since this takes longer than sending several bytes at the time.

When testing the SPI communication, almost no commands ended with errors. This means that the error detection may not be necessary, since it's almost never used. By not using error detections, the commands would probably be executed faster, since the number of transfers would be reduced. For special situations where the time used for sending SPI messages has to be as low as possible, this may be a possibility, but normally the error detection should be used.

It's also possible that the device driver could be more effective. The User Mode driver opens and closes the device node each time it reads and writes from them, since the device driver has mutexes that makes it only possible that one process has it open at the time. This could possibly be made more effective.

## 13.4 Matlab Real-Time Workshop

Matlab Real-Time Workshop was used in this thesis as a rapid prototyping tool. It generates code from Simulink models, that runs on the control system. A new system target was created to allow some modifications to the way code are generated. This new system target was able to compile the code for AVR32, and make the code execute periodically. The system target can easily be installed, and are easy to use.

S-functions were developed for the different features of the I/O-card, and it gave Simulink blocks that was able to control the I/O. The S-functions that uses the normal user mode drivers worked as planned, and they were effective. A test concluded that a value sent through the I/O-card only was delayed with one time step. When using the threaded user mode driver, the results were a longer delay. This means that the threading is a less effective solutions. Optimizing the driver might help, but it's more likely that the increased overhead of the threaded driver is bigger than the effectively gain of using threads.

# Chapter 14

# Conclusion

This thesis has developed the hardware and software necessary for using the AVR32 processor architecture as a control system. This resulted in a control system platform that can easily be installed and used. The platform it specially suited for applications where low weight and power consumption is important, because both the AVR32 itself and the microcontroller used in the I/O-card are low-power devices.

An I/O-card and drivers were developed and this enabled the AVR32 to measure and control its environment with digital and analog ($0V - 5V$) signals. AVR32 communicates uses SPI to give commands to the I/O-card. This gave a reasonable fast and very reliable communication. Since SPI has no error checking, the communication protocol was designed with an effective error detection, even if the communication itself was reliable. This means that in the case of an unlikely error, the system will detect it.

The platform has be configured to use Matlab Real-Time Workshop as a rapid prototyping tool. This tool generates code from Matlab Simulink models, that makes it suited for designing and testing different controllers. With the developed S-functions, the I/O-card could be controlled with Real-Time Workshop. In this thesis standard Linux timers have been used to make the time steps periodical. This gives a minimum period of $1ms$, but this haven't been a problem since the control system aren't fast enough to use lower periods.

The platform has some performance limitation. The AVR32 processor isn't a fast processor compared to a normal computer, and it lacks a FPU. This means that floating-point operations has to be software emulated and is less effective, which limits the ability to control complex systems. If a system is to complex, the risk of breaking any real-time constraints increases. However, the platform is well suited to control less complex systems, and the workload is logged, and can be used to detect if the system is close to breaking a real-time constraint.

The user manual was made to make it easy for users to install and use the control system platform. For users, the system will function as a *black box* where they don't need to know how the system works, only that it can be controlled using Matlab Real-Time Workshop.

# Chapter 15

# Further Work

This thesis has built a control system platform that uses the AVR32 architecture, and further work would be to use this platform in a actual control system. The platform may also be developed further, to improve it by making it faster and more stable. Below is a list of ideas for further development.

- Find ways to make sending commands to the I/O-card more effectively. This can be done by making the AVR32 Linux SPI driver more efficient, drop error detection or reduce the overhead introduced by it or by optimizing the interaction between the User Mode driver and the device driver.
- Increase the number of I/O protocols that the I/O-card can use. The existing SPI communication protocol may be extended to for new commands.
- Improve the I/O protocols that are already present.

# Appendix A

# Digital appendix

The digital appendix includes the code and other files from the thesis, and have the following directories:

- code
  Contains the code developed in the thesis.
  - atmega128
    Firmware code for the ATmega128.
  - avr32
    Code used by Matlab to make the AVR32 Real-Time target. Contain drivers and blocks in subdirectories.
  - kernel_module
    Code for the device driver as a kernel module.
- eagle
  Schematics and layout of the two AVR32 I/O-card versions in eagle format.
- file_system
  Files used in the AVR32 file system.
- linux
  Patches, config files and other AVR32 Linux related files.
- report
  The report and the user manual. Subdirectory contain graphics used in the report.

# Appendix B

# Schematics

## B.1 Schematic for protoype card

Figure B.1 on the next page is the total schematics for the prototype card.

## B.2 Schematic for final card

Figure B.2 on page 116 is the total schematics for the final card.

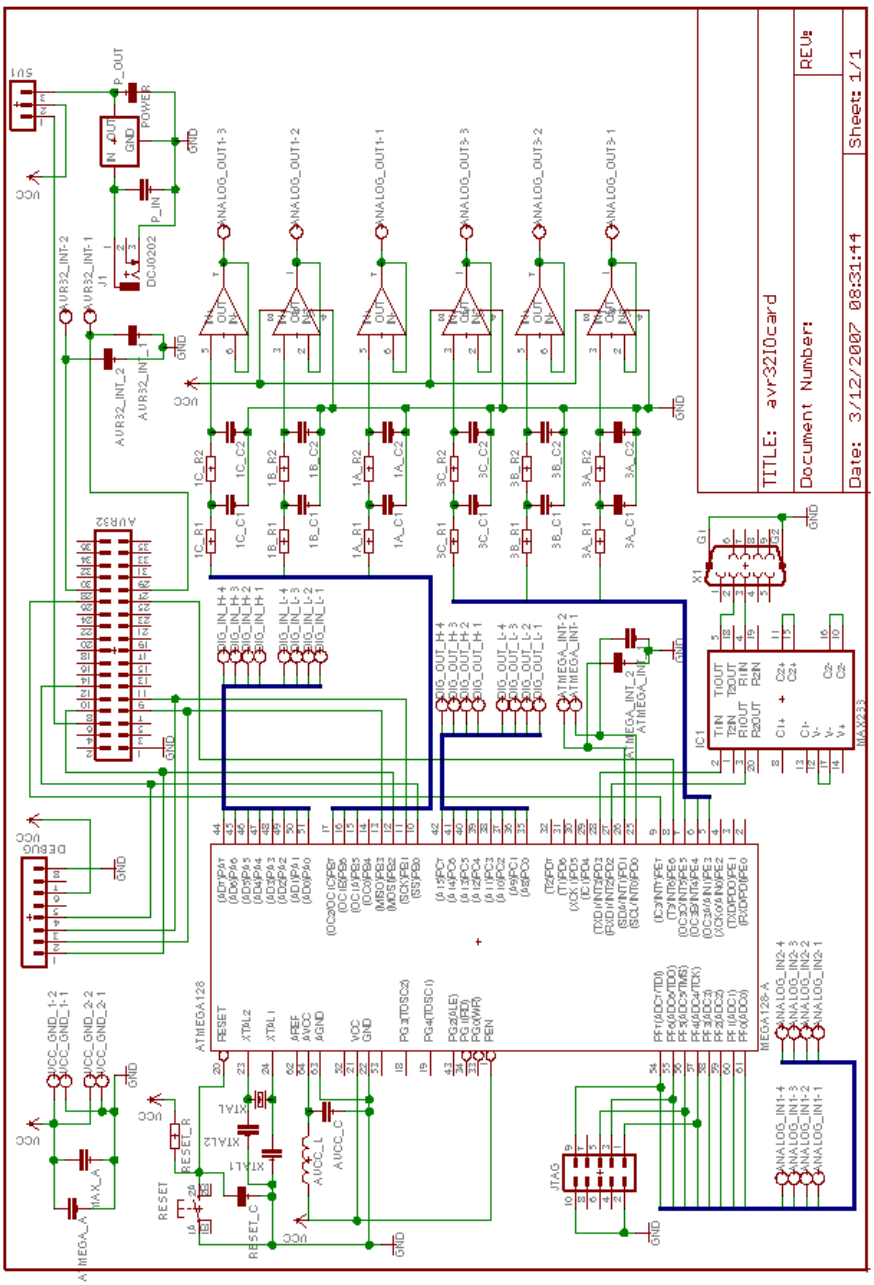


Figure B.1: Schematics for the prototype card.

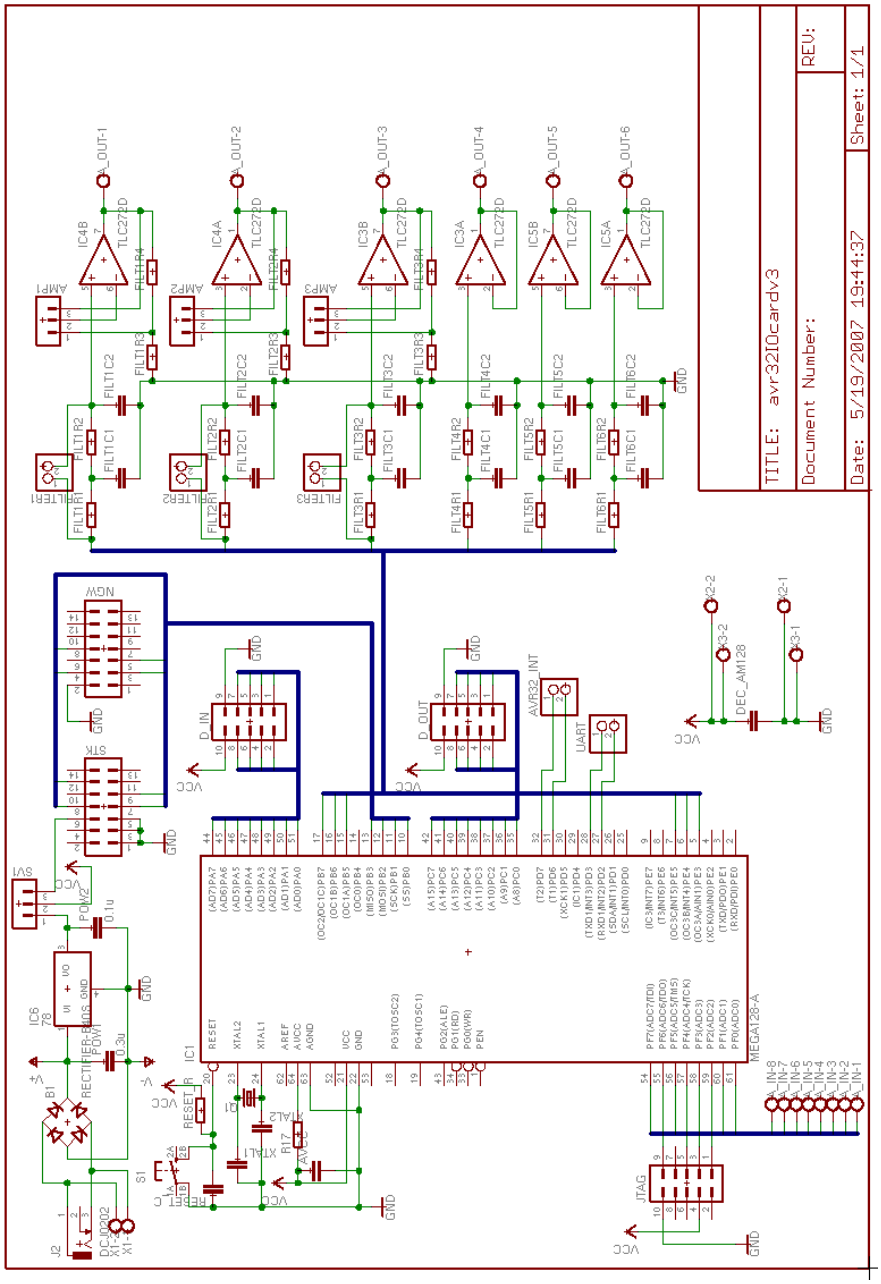


Figure B.2: Schematics for the final card.

# Appendix C

# Code

## C.1 ATmega128

### C.1.1 Makefile

```
CC=avr-gcc
OBJCOPY=avr-objcopy

CFLAGS=-g -mmcu=atmega128 -I/usr/avr/include

am128main.hex : am128main.out
	$(OBJCOPY) -j .text -O ihex am128main.out am128main.hex

am128main.out : am128main.o
	$(CC) $(CFLAGS) -o am128main.out -Wl,-Map,am128main.map am128main.o

am128main.o : am128main.c
	$(CC) $(CFLAGS) -Os -c am128main.c

clean:
	rm -f *.o *.out *.map *.hex

install:
	avrdude -P usb -pm128 -cjtag2 -Uflash:w:am128main.hex
```

### C.1.2 am128main.c

```
#include "am128main.h"

int main()
{
   // Initialize UART.
   UART_Init();

   // Initialize am128.
   IO_Init();

   // Main program loop.
   while(1){
      // Prepare to send ready signal, and wait for code from master.
      SPDR = SLAVE_READY;

      // Wait for SPI transfer.
      while(!test_bit(SPSR,SPIF)){
#ifdef BUFFERING_MODE
```

```
19          // Checks if conversion is finished if using bufering mode.
20          ADC_CheckBuffer();
21 #endif
22       }
23       // Start receiving new command.
24       IO_NewCommand(SPDR);
25    }
26 }
```

### C.1.3 am128io.c

```
1  void IO_Init()
2  {
3     // Init I/O.
4     SPI_SlaveInit();
5     PWM_Init();
6     ADC_Init();
7
8     // Direction of the digial io.
9     DDRA = 0x00;
10    DDRC = 0xff;
11 }
12
13 signed char IO_TestAck(unsigned char tx, unsigned char rx)
14 {
15    if(tx == 255){       // If the ack has overflowed.
16       if(rx != 0) return -1;
17    }
18    else{ // Normal ack.
19       if(rx != tx + 1) return -1;
20    }
21    // Ack was correct.
22    return 0;
23 }
24
25 signed char IO_NewCommand(unsigned char code)
26 {
27    // Checks the code, and start the correct command.
28    // Only test against 5 MSB, since 3LSB only contain information about channel.
29    switch(code & 0xf8){
30       case CODE_AIN:
31          IO_AnalogIn(code);
32          break;
33       case CODE_AOUT:
34          IO_AnalogOut(code);
35          break;
36       case CODE_DIN:
37          IO_DigitalIn(code);
38          break;
39       case CODE_DOUT:
40          IO_DigitalOut(code);
41          break;
42 #ifdef BUFFERING_MODE
43       case CODE_AIN_EN:
44          IO_AnalogInEnable(code);
45          break;
46 #endif
47       case CODE_AOUT_RES:
48          IO_AnalogOutRes(code);
49          break;
50       default:
51          //Prints cmd to UART if it didn't match any codes.
52          UART_Print('e', (code & 0xf8));
53          break;
54    }
55 }
56
```

```
57  signed char IO_AnalogIn(unsigned char code){
58      unsigned char rx, data[2];
59      char ret;
60
61      // 1st byte: check received DUMMY, return code + 1.
62      ret = SPI_SlaveTransferByte(code + 1, &rx, TIMEOUT);
63      if(ret < 0) return ret;
64      if(rx != DUMMY) return -1;
65
66  #ifdef INTERRUPT_MODE
67      // Starts conversion and polls until the conversion is complete.
68      ADC_StartConversion(CODE_TO_CH_AIN(code));
69      while(!ADC_CheckConversion(data));
70
71      // Send interrupt to AVR32.
72      set_bit(PORTD, 7);
73      _delay_us(2);
74      clear_bit(PORTD, 7);
75  #endif
76
77  #ifdef BUFFERING_MODE
78      // Store data that should be sent to master.
79      data[0] = dataH[CODE_TO_CH_AIN(code)];
80      data[1] = dataL[CODE_TO_CH_AIN(code)];
81  #endif
82
83      // 2nd byte: check received DUMMY, return high byte value.
84      ret = SPI_SlaveTransferByte(data[0], &rx, TIMEOUT);
85      if(ret < 0) return ret;
86      if(rx != DUMMY) return -1;
87
88      // 3rd byte: check received high byte value + 1, return low byte value.
89      ret = SPI_SlaveTransferByte(data[1], &rx, TIMEOUT);
90      if(ret < 0) return ret;
91      ret = IO_TestAck(data[0], rx);
92      if(ret < 0) return ret;
93
94      // 4th byte: check received low byte value + 1, return DUMMY.
95      ret = SPI_SlaveTransferByte(DUMMY, &rx, TIMEOUT);
96      if(ret < 0) return ret;
97      ret = IO_TestAck(data[1], rx);
98      if(ret < 0) return ret;
99
100     // 5th byte: check received MASTER_OK, return SLAVE_OK.
101     ret = SPI_SlaveTransferByte(SLAVE_OK, &rx, TIMEOUT);
102     if(ret < 0) return ret;
103     if(rx != MASTER_OK) return -1;
104
105     // Debug printing, if enabled.
106     UART_Print('c', code);
107     UART_Print('h', data[0]);
108     UART_Print('l', data[1]);
109
110     return 0;
111 }
112
113 signed char IO_AnalogOut(unsigned char code)
114 {
115     unsigned char valueH, valueL, rx;
116     char ret;
117
118     // 1st byte: store received valueH, return code + 1.
119     ret = SPI_SlaveTransferByte(code + 1, &valueH, TIMEOUT);
120     if(ret < 0) return ret;
121
122     // 2nd byte: store received valueL, return valueH + 1.
123     ret = SPI_SlaveTransferByte(valueH + 1, &valueL, TIMEOUT);
124     if(ret < 0) return ret;
```

```
125
126     // 3rd byte: check received DUMMY, return valueL + 1.
127     ret = SPI_SlaveTransferByte(valueL + 1, &rx, TIMEOUT);
128     if(ret < 0) return ret;
129     if(rx != DUMMY) return -1;
130
131     // 4th byte: check received MASTER_OK, return SLAVE_OK.
132     ret = SPI_SlaveTransferByte(SLAVE_OK, &rx, TIMEOUT);
133     if(ret < 0) return ret;
134     if(rx != MASTER_OK) return -1;
135
136     // When the message is confirmed the mask and data are applied to digital out.
137     PWM_SetOutValue(CODE_TO_CH_AOUT(code), CODE_TO_SCH_AOUT(code), valueH, valueL);
138
139     // Debug printing, if enabled.
140     UART_Print('c', code);
141     UART_Print('h', valueH);
142     UART_Print('l', valueL);
143 }
144
145 signed char IO_DigitalIn(unsigned char code)
146 {
147     unsigned char rx, value;
148     char ret;
149
150     // Only read from the port once during communication
151     value = PINA;
152
153     // 1st byte: check received DUMMY, return code + 1.
154     ret = SPI_SlaveTransferByte(code + 1, &rx, TIMEOUT);
155     if(ret < 0) return ret;
156     if(rx != DUMMY) return -1;
157
158     // 2nd byte: check received DUMMY, return value.
159     ret = SPI_SlaveTransferByte(value, &rx, TIMEOUT);
160     if(ret < 0) return ret;
161     if(rx != DUMMY) return -1;
162
163     // 3rd byte: check received value + 1, return DUMMY.
164     ret = SPI_SlaveTransferByte(DUMMY, &rx, TIMEOUT);
165     if(ret < 0) return ret;
166     ret = IO_TestAck(value, rx);
167     if(ret < 0) return ret;
168
169     // 4th byte: check received MASTER_OK, return SLAVE_OK.
170     ret = SPI_SlaveTransferByte(SLAVE_OK, &rx, TIMEOUT);
171     if(ret < 0) return ret;
172     if(rx != MASTER_OK) return -1;
173
174     // Debug printing, if enabled.
175     UART_Print('c', code);
176     UART_Print('v', value);
177
178     return 0;
179 }
180
181 signed char IO_DigitalOut(unsigned char code)
182 {
183     unsigned char mask, value, rx;
184     char ret, i;
185
186     // 1st byte: store received mask, return code + 1.
187     ret = SPI_SlaveTransferByte(code + 1, &mask, TIMEOUT);
188     if(ret < 0) return ret;
189
190     // 2nd byte: store received value, return mask + 1.
191     ret = SPI_SlaveTransferByte(mask + 1, &value, TIMEOUT);
192     if(ret < 0) return ret;
```

```
193
194     // 3rd byte: check received DUMMY, return value + 1.
195     ret = SPI_SlaveTransferByte(value + 1, &rx, TIMEOUT);
196     if(ret < 0) return ret;
197     if(rx != DUMMY) return -1;
198
199     // 4th byte: check received MASTER_OK, return SLAVE_OK.
200     ret = SPI_SlaveTransferByte(SLAVE_OK, &rx, TIMEOUT);
201     if(ret < 0) return ret;
202     if(rx != MASTER_OK) return -1;
203
204     // When the message is confirmed the mask and data are applied to digital out.
205     for(i=0;i<8;i++){
206         if(test_bit(mask, i)){
207             if(test_bit(value, i)) set_bit(PORTC, i);
208             else clear_bit(PORTC, i);
209         }
210     }
211
212     // Debug printing, if enabled.
213     UART_Print('c', code);
214     UART_Print('m', mask);
215     UART_Print('v', value);
216 }
217
218 #ifdef BUFFERING_MODE
219 signed char IO_AnalogInEnable(unsigned char code)
220 {
221     unsigned char enable, rx;
222     char ret;
223
224     // 1st byte: store received enable, return code + 1.
225     ret = SPI_SlaveTransferByte(code + 1, &enable, TIMEOUT);
226     if(ret < 0) return ret;
227
228     // 2nd byte: store received DUMMY, return enable + 1.
229     ret = SPI_SlaveTransferByte(enable + 1, &rx, TIMEOUT);
230     if(ret < 0) return ret;
231     if(rx != DUMMY) return -1;
232
233     // 3rd byte: check received MASTER_OK, return SLAVE_OK.
234     ret = SPI_SlaveTransferByte(SLAVE_OK, &rx, TIMEOUT);
235     if(ret < 0) return ret;
236     if(rx != MASTER_OK) return -1;
237
238     // When the message is confirmed the channel are enabled or disabled.
239     if(enable == 0){
240         ADC_DisableChan(CODE_TO_CH_AIN(code));
241     }
242     else{
243         ADC_EnableChan(CODE_TO_CH_AIN(code));
244     }
245
246     // Debug printing, if enabled.
247     UART_Print('c', code);
248     UART_Print('e', enable);
249 }
250 #endif
251
252 signed char IO_AnalogOutRes(unsigned char code)
253 {
254     unsigned char resolution, chEnable, rx;
255     char ret;
256
257     // 1st byte: store received resolution, return code + 1.
258     ret = SPI_SlaveTransferByte(code + 1, &resolution, TIMEOUT);
259     if(ret < 0) return ret;
260
```

```
261     // 2nd byte: check received DUMMY, return resolution + 1.
262     ret = SPI_SlaveTransferByte(resolution + 1, &rx, TIMEOUT);
263     if(ret < 0) return ret;
264     if(rx != DUMMY) return -1;
265 
266     // 3rd byte: check received MASTER_OK, return SLAVE_OK.
267     ret = SPI_SlaveTransferByte(SLAVE_OK, &rx, TIMEOUT);
268     if(ret < 0) return ret;
269     if(rx != MASTER_OK) return -1;
270 
271     // When the message is confirmed the analog out resolution of the selected
            channel will be updated.
272     PWM_SetResolution(CODE_TO_CH_AOUT(code), resolution);
273 
274     // Debug printing, if enabled.
275     UART_Print('c', code);
276     UART_Print('r', resolution);
277 }
```

### C.1.4 am128spiSlave.c

```
1  void SPI_SlaveInit(void)
2  {
3     // Enables SPI and sets the MISO signal as an output.
4     SPCR = (1<<SPE);
5     DDRB = (1<<3);
6  
7     // Timer prescaler 256 (16us).
8  #ifdef TIMEOUT_ENABLE
9     TCCR0 =  (1 << CS02) | (1 << CS01);
10 #endif
11 }
12 
13 signed char SPI_SlaveTransferByte(unsigned char tx, unsigned char *rx, unsigned
       char usec)
14 {
15 #ifdef TIMEOUT_ENABLE
16    // Starts timer.
17    TCNT0 = 0;
18 #endif
19 
20    // Prepare data to send and wait for SPI transfer from master.
21    SPDR = tx;
22    while(!(SPSR & (1<<SPIF))){
23 #ifdef TIMEOUT_ENABLE
24       // Checks for timeout.
25       if(TCNT0 > usec) return -1;
26 #endif
27    }
28 
29    // Returns data.
30    *(rx) = SPDR;
31    return 0;
32 }
```

### C.1.5 am128adc.c

```
1  
2  void ADC_Init()
3  {
4     // Direction of adc channels.
5     DDRF = 0;
6  
7  #ifdef INTERRUPT_MODE
8     // Direction of interrupt.
9     set_bit(DDRD, 7);
```

```
10      clear_bit(PORTD, 7);
11  #endif
12
13      // Initialize ADC.
14      ADMUX = (1<<REFS0);
15      ADCSRA = (1<<ADEN) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0);
16  }
17
18  #ifdef BUFFERING_MODE
19  void ADC_EnableChan(char channel)
20  {
21      char i;
22      struct chanList *temp, *temp2;
23
24      if(chCount == 0){
25          // Make table element and start converting.
26          channels = (channeltab + channel);
27          channels->next = channels;
28          channels->prev = channels;
29          channels->channel = channel;
30          ADC_NewConversion();
31      }
32      else{
33          // Search through channels and check if the channel is already enabled.
34          temp = channels;
35          for(i=0; i<chCount; i++){
36              // If channel already is enabled the function ends.
37              if(temp->channel == channel) return;
38              temp = temp->next;
39          }
40
41          // Make new channel.
42          temp = (channeltab + channel);
43          temp->channel = channel;
44
45          // Inserts new channel in list.
46          temp->next = channels->next;
47          temp->prev = channels;
48          channels->next = temp;
49          channels->next->prev = temp;
50      }
51
52      // Increase number of channels;
53      chCount++;
54  }
55
56  void ADC_DisableChan(char channel)
57  {
58      char i, enabled;
59      struct chanList *temp;
60
61      temp = channels;
62      enabled = 0;
63
64      // Search through channels and check if the channel is enabled.
65      for(i=0; i<chCount; i++){
66          if(temp->channel == channel){
67              enabled = 1;
68              break;
69          }
70          temp = temp->next;
71      }
72      // If channel aren't enabled the function ends.
73      if(enabled == 0) return;
74
75      // Update pointers inside the list.
76      temp->prev->next = temp->next;
77      temp->next->prev = temp->prev;
```

```
78      chCount--;
79
80      if(chCount == 0) ADC_StopConversion();
81  }
82
83  void ADC_CheckBuffer()
84  {
85      if(test_bit(ADCSRA, ADIF)){
86          // Storing new data.
87          dataL[channels->channel] = ADCL;
88          dataH[channels->channel] = ADCH;
89
90          // Start new conversion.
91          ADC_NewConversion();
92      }
93  }
94
95  void ADC_NewConversion()
96  {
97      channels = channels->next;
98
99      ADMUX = ADMUX & 0xe0;
100     ADMUX = ADMUX | channels->channel;
101
102     //Start conversion.
103     set_bit(ADCSRA, ADSC);
104 }
105
106 void ADC_StopConversion()
107 {
108     clear_bit(ADCSRA, ADSC);
109 }
110 #endif
111
112 #ifdef INTERRUPT_MODE
113 void ADC_StartConversion(char channel)
114 {
115     ADMUX = ADMUX & 0xe0;
116     ADMUX = ADMUX | channel;
117
118     //Start conversion.
119     set_bit(ADCSRA, ADSC);
120 }
121
122 signed char ADC_CheckConversion(unsigned char *data)
123 {
124     if(test_bit(ADCSRA, ADIF)){
125         // Clear status bit.
126         set_bit(ADCSRA, ADIF);
127
128         // Storing new data.
129         data[1] = ADCL;
130         data[0] = ADCH;
131
132         // Finished.
133         return 1;
134     }
135     // Not finished.
136     return 0;
137 }
138 #endif
```

### C.1.6 am128pwm.c

```
1   void PWM_Init()
2   {
3       // Set the PWM ports as output
```

```
4     set_bit(DDRE, 3);
5     set_bit(DDRE, 4);
6     set_bit(DDRE, 5);
7     set_bit(DDRB, 5);
8     set_bit(DDRB, 6);
9     set_bit(DDRB, 7);
10
11    // Set the control registers for PWM to fast PWM, with no prescaler and clear
          output on compare match (non-inverting mode).
12    TCCR1A =(1<<COM1A1) | (1<<COM1B1) | (1<<COM1C1) | (1<<WGM11);
13    TCCR1B = (1<<WGM13) | (1<<WGM12) | (1<<CS10);
14
15    TCCR3A =(1<<COM3A1) | (1<<COM3B1) | (1<<COM3C1) | (1<<WGM31);
16    TCCR3B = (1<<WGM33) | (1<<WGM32) | (1<<CS30);
17
18    // Set both PWM to have 8-bit resolution.
19    PWM_SetResolution(0,8);
20    PWM_SetResolution(1,8);
21
22    // Set all PWM-outputs to zero.
23    PWM_SetOutValue(0, 1, 0x00, 0x00);
24    PWM_SetOutValue(0, 2, 0x00, 0x00);
25    PWM_SetOutValue(0, 3, 0x00, 0x00);
26    PWM_SetOutValue(1, 1, 0x00, 0x00);
27    PWM_SetOutValue(1, 2, 0x00, 0x00);
28    PWM_SetOutValue(1, 3, 0x00, 0x00);
29 }
30
31 void PWM_SetOutValue(char ch, char sch, char valueH, char valueL)
32 {
33 // Checks which PWM/timer that should be set. Either 0 or 1.
34 switch(ch){
35    case 0:
36       // Checks what channel of the PWM/timer. A=1, B=2 and C=3.
37       switch(sch){
38          case 0:
39             OCR1AH = valueH;
40             OCR1AL = valueL;
41             break;
42          case 1:
43             OCR1BH = valueH;
44             OCR1BL = valueL;
45             break;
46          case 2:
47             OCR1CH = valueH;
48             OCR1CL = valueL;
49             break;
50       }
51       break;
52    case 1:
53       // Checks what channel of the PWM/timer. A=1, B=2 and C=3.
54       switch(sch){
55          case 0:
56             OCR3AH = valueH;
57             OCR3AL = valueL;
58             break;
59          case 1:
60             OCR3BH = valueH;
61             OCR3BL = valueL;
62             break;
63          case 2:
64             OCR3CH = valueH;
65             OCR3CL = valueL;
66             break;
67          }
68       break;
69    }
70 }
```

```
71
72 void PWM_SetResolution(char ch, char resolution)
73 {
74    char resH, resL, i;
75    resH = 0x00;
76    resL = 0x00;
77
78    // Finds values for the ICRn registers that gives the requested resolution.
79    if(resolution <= 8){ // For resolution equal or less than 8-bits.
80       for(i=0; i<resolution; i++){
81          set_bit(resL, i);
82       }
83    }
84    else{ // For resolution over 8-bits.
85       resL = 0xff;
86       for(i=0; i<resolution-8; i++){
87          set_bit(resH, i);
88       }
89    }
90
91    // Writes the values to the right ICRn register.
92    switch(ch){
93       case 0:
94          ICR1H = resH;
95          ICR1L = resL;
96          break;
97       case 1:
98          ICR3H = resH;
99          ICR3L = resL;
100         break;
101   }
102
103   // Sets output on all subchannels to zero.
104   PWM_SetOutValue(ch, 1, 0x00, 0x00);
105   PWM_SetOutValue(ch, 2, 0x00, 0x00);
106   PWM_SetOutValue(ch, 3, 0x00, 0x00);
107 }
```

### C.1.7 am128uart.c

```
1  void UART_Init()
2  {
3  #ifdef RS232_DEBUG
4     // Set direction registers.
5     set_bit(DDRD, 3);
6     clear_bit(DDRD, 2);
7
8     // BAUD rate = 19.2k.
9     UBRR1H = 0;
10    UBRR1L = 51;
11
12    // Enable both transmit and receive.
13    UCSR1B = (1<<RXEN1) | (1<<TXEN1);
14
15    // Use 8 bit data, no parity and 2 stop bits.
16    UCSR1C = (1<<UCSZ11) | (1<<UCSZ10) | (1<<USBS1);
17 #endif
18 }
19
20 void UART_TxByte(char data)
21 {
22 #ifdef RS232_DEBUG
23    // Wait until last message is sent, then send this one.
24    loop_bit_is_set(UCSR1A, UDRE1);
25    UDR1=data;
26 #endif
27 }
```

```
28
29 void UART_TxSpace()
30 {
31 #ifdef RS232_DEBUG
32    UART_TxByte(32);
33 #endif
34 }
35
36 void UART_TxNewLine()
37 {
38 #ifdef RS232_DEBUG
39    UART_TxByte('\n');
40    UART_TxByte('\r');
41 #endif
42 }
43
44
45 void UART_TxDecimal(unsigned char value)
46 {
47 #ifdef RS232_DEBUG
48    char tx[3], i;
49
50    //tx[0]
51    tx[0] = '0';
52    for(i=0;i<3;i++){
53       if(value > 99){
54          tx[0] = tx[0] + 1;
55          value = value - 100;
56       }
57    }
58
59    //tx[1]
60    tx[1] = '0';
61    for(i=0;i<10;i++){
62       if(value > 9){
63          tx[1] = tx[1] + 1;
64          value = value - 10;
65       }
66    }
67
68    //tx[2]
69    tx[2] = '0';
70    for(i=0;i<10;i++){
71       if(value > i){
72          tx[2] = tx[2] + 1;
73       }
74    }
75
76    //Print
77    for(i=0;i<3;i++){
78       UART_TxByte(tx[i]);
79    }
80 #endif
81 }
82
83 void UART_Print(char symbol, unsigned char value)
84 {
85 #ifdef RS232_DEBUG
86    UART_TxByte(symbol);
87    UART_TxByte(':');
88    UART_TxSpace();
89    UART_TxDecimal(value);
90    UART_TxNewLine();
91 #endif
92 }
```

## C.2 Kernel module

### C.2.1 Makefile

```
# Include the AVR32 header files
CFLAGS += -I/usr/local/avr32-linux/include
CFLAGS += -D__AVR32_AP7000__

MAKE                    := make ARCH=avr32 CROSS_COMPILE=avr32-linux-
KERNELDIR       := /home/oyvindne/diplom/source/linux
PWD                     := $(shell pwd)

obj-m := avr32io.o

default:
        $(MAKE) -C $(KERNELDIR) M=$(PWD) modules

clean:
        rm -f *.o *.ko *.mod.* .*cmd core
        rm -rf .tmp_versions

install:
        cp avr32io.ko /home/oyvindne/diplom/fs/avr32io.ko
```

### C.2.2 avr32ioc

```
#include "avr32io.h"

static int __init avr32io_Init(void)
{
   int ret;

   if(!(device = kzalloc(sizeof(struct avr32io_device), GFP_KERNEL)))
   {
      printk(KERN_ALERT "avr32io_Init: kzalloc failed\n");
      return -ENOMEM;
   }

   // Major/minor number allocation
   ret = alloc_chrdev_region(&device->number, 0, COUNT_TOTAL, "avr32io");
   if(ret < 0){
      printk(KERN_ALERT "avr32io_Init: alloc_chrdev_region failed %i\n", ret);
      goto errRegion;
   }

   // Adding cdev.
   device->charDevice = cdev_alloc();
   device->charDevice->ops = &fops;
   device->charDevice->owner = THIS_MODULE;
   ret = cdev_add(device->charDevice, device->number, COUNT_TOTAL);
   if(ret < 0){
      printk(KERN_ALERT "avr32io_Init: charDevice failed %i\n", ret);
      goto errCharDevice;
   }

   // Registration of the driver with the linux device model
   ret = spi_register_driver(&avr32io_driver);
   if(ret < 0){
      printk(KERN_ALERT "avr32io_Init: spi_register_driver failed %i\n", ret);
      goto errRegDriver;
   }

#ifdef INTERRUPT_MODE
        // Registration of IRQ
   ret = request_irq(EIM_IRQ_BASE, irqFunk, IRQF_TRIGGER_RISING, "avr32io_interrupt
       ", NULL);
```

```
40      if(ret < 0){
41          DEBUG("avrIO_Init: request_irq failed %i\n", ret);
42          goto errRegIRQ;
43      }
44      device->irqID = ret;
45  #endif
46
47      printk(KERN_INFO "avr32io_Init: sucessfull major: %i\n", MAJOR(device->number));
48
49      //Return sucessfull
50      return 0;
51
52      //Error, roll back resourses.
53  #ifdef INTERRUPT_MODE
54      errRegIRQ:
55          spi_unregister_driver(&avr32io_driver);
56  #endif
57      errRegDriver:
58          cdev_del(device->charDevice);
59      errCharDevice:
60          unregister_chrdev_region(device->number, 4);
61      errRegion:
62          kfree(device);
63      return ret;
64  }
65
66  static void __exit avr32io_Exit(void)
67  {
68  #ifdef INTERRUPT_MODE
69      // Free the IRQ.
70      free_irq(EIM_IRQ_BASE,NULL);
71  #endif
72
73      // Unregister the driver from the linux device model
74      spi_unregister_driver(&avr32io_driver);
75
76      // Unregister the character device from the kernel
77      cdev_del(device->charDevice);
78
79      // Unregister major/minor numbers allocated
80      unregister_chrdev_region(device->number, COUNT_TOTAL);
81
82      // Free memory used by the device struct.
83      kfree(device);
84
85      printk(KERN_INFO "avr32io_Exit: module exit\n");
86  }
87
88  static int __devinit avr32io_Probe(struct spi_device * spi)
89  {
90      int ret;
91
92      // Initialize the spi struct
93      device->spi = spi;
94      spi->mode = SPI_MODE_0;
95      spi->bits_per_word = 0x08; // 8-bits per transfer.
96      spi->max_speed_hz = 2100000; //highest value that works is 2100000 @ 16MHz
97      spi->chip_select = 0x02; // Uses slave number 2.
98
99      // Setup the SPI driver and channel
100     ret = spi_setup(spi);
101     if(ret < 0){
102         printk(KERN_ALERT "avr32io_Probe: spi_setup failed %d\n", ret);
103         return ret;
104     }
105
106     dev_set_drvdata(&spi->dev, device);
107     return 0;
```

```
108 }
109
110 static int avr32io_Open(struct inode *node,struct file *filp)
111 {
112    int minor, ret;
113
114    minor = iminor(filp->f_dentry->d_inode);
115    DEBUG("avr32io_Open: minor %i\n", minor);
116
117    // Take mutex.
118    ret = down_interruptible(&spiMutex);
119    if(ret < 0) return -ERESTARTSYS;
120
121    // Check if SPI is used.
122    if(device->busy == 0){
123       // SPI busy, return error.
124       ret = -EBUSY;
125    }
126    else{
127       // SPI not busy, mark as busy and return success.
128       device->busy == 1;
129       ret = 0;
130    }
131
132    // Returns the mutex and return if the SPI was busy or not.
133    up(&spiMutex);
134    return ret;
135 }
136
137 static int avr32io_Release(struct inode *node,struct file *filp)
138 {
139    DEBUG("avr32io_Release\n");
140
141    // Take mutex.
142    ret = down_interruptible(&spiMutex);
143    if(ret < 0) return -ERESTARTSYS;
144
145    // Mark the SPI as not busy.
146    device->busy == 0;
147
148    // Returns the mutex and returns success.
149    up(&spiMutex);
150    return 0;
151 }
152
153 static ssize_t avr32io_Read(struct file *filp, char __user *userBuf, size_t len,
       loff_t *f_pos)
154 {
155    int ret;
156    u8 toUser[len];
157    int minor = iminor(filp->f_dentry->d_inode);
158
159    DEBUG("avr32io_Read: minor %i isAin: %i isDin: %i\n", minor, IS_AIN(minor),
         IS_DIN(minor));
160
161    if(IS_AIN(minor)){ //Analog In
162       if(len != 2) return -EPERM;
163       ret = Cmd_GetAIn(toUser, MinorToChAin(minor));
164       if(ret < 0)return ret;
165       copy_to_user(userBuf, toUser, len);
166       return len;
167 }
168
169    if(IS_DIN(minor)){ //Digital In
170       if(len != 1) return -EPERM;
171       ret = Cmd_GetDIn(toUser);
172       if(ret < 0) return ret;
173       copy_to_user(userBuf, toUser, len);
```

```
174         return len;
175     }
176 
177     //If no output matches the minor number the write operation fails.
178     return -EPERM;
179 }
180 
181 static ssize_t avr32io_Write(struct file *filp, const char __user *userBuf, size_t
        len, loff_t *f_pos)
182 {
183     u8 fromUser[len];
184     int minor = iminor(filp->f_dentry->d_inode);
185 
186     copy_from_user(&fromUser, userBuf, len);
187 
188     DEBUG("avr32io_Write: minor %i isAout: %i isDout: %i\n", minor, IS_AOUT(minor),
            IS_DOUT(minor));
189 
190     if(IS_AOUT(minor)){ //Analog Out
191         if(len != 3) return -EPERM;
192         if(Cmd_SetAOut(fromUser + 1, MinorToChAout(minor), fromUser[0]) >= 0){
193             return len;
194         }
195     }
196 
197     if(IS_DOUT(minor)){ //Digital Out
198         if(len != 2) return -EPERM;
199         if(Cmd_SetDOut(fromUser) >= 0){
200             return len;
201         }
202     }
203 
204     //If no output matches the minor number the write operation fails.
205     return -EPERM;
206 }
207 
208 static int avr32io_Ioctl(struct inode   *node, struct file *filp, unsigned int cmd,
        unsigned long arg)
209 {
210     int minor = iminor(filp->f_dentry->d_inode);
211 
212     DEBUG("avr32io_Ioctl: ioctl minor: %i cmd: %i isAout: %i isAin: %i\n", minor,
            cmd, IS_AOUT(minor), IS_AIN(minor));
213 
214     switch(cmd){
215         case AVR32IO_IOCTL_RES_AOUT: // Change resolution on a analog channel.
216             if(IS_AOUT(minor)){
217                 return Cmd_SetAOutResolution((u8)(arg), MinorToChAout(minor));
218             }
219 #ifdef BUFFERING_MODE
220         case AVR32IO_IOCTL_EN_AIN: // Enables or disables analog in.
221             if(IS_AIN(minor)){
222                 return Cmd_EnableAIn((u8)(arg), MinorToChAin(minor));
223             }
224 #endif
225     }
226 
227     // If cmd is not a valid command.
228     return -ENOIOCTLCMD;
229 }
```

### C.2.3 avr32io_Cmd.c

```
1 
2 int Cmd_WaitForAM128(u8 code)
3 {
4     int count = 0;
```

```
5     u8 ready;
6
7     // Loops until it receives the correct code from ATMega128.
8     while(ready != SLAVE_READY){
9        SPI_MasterTransfer(&ready, &code, 1);
10       if(count++ > 100) return -EIO;
11    }
12    return count;
13 }
14
15 int Cmd_GetAIn(u8 *toUser, int ch)
16 {
17    int ret;
18
19    // check for valid channel.
20    if(ch < 0) return -EIO;
21
22    // Synchronize with ATMega128
23    ret = Cmd_WaitForAM128(CH_TO_CODE_AIN(ch));
24    DEBUG("code: %i count: %i\n", CH_TO_CODE_AIN(ch), ret);
25    if (ret < 0) return ret;
26
27    // Send DUMMY, receive code ack.
28    ret = SPI_MasterTransferByte(DUMMY);
29    DEBUG("dummy: %i code+1: %i\n", DUMMY, ret);
30    if(ret != CH_TO_CODE_AIN(ch) + 1)  return -EIO;
31
32 #ifdef INTERRUPT_MODE
33    // Wait for interrupt.
34    device->irqFlag = 0;
35    DEBUG("wait\n");
36    ret = wait_event_interruptible_timeout(wq, device->irqFlag != 0, 10000);
37    if(ret < 0) return -ERESTARTSYS;
38    DEBUG("interrupt\n");
39 #endif
40
41    // Send DUMMY, receive data0.
42    ret = SPI_MasterTransferByte(DUMMY);
43    DEBUG("dummy: %i data0: %i\n", DUMMY, ret);
44    if(ret < 0)  return ret;
45    toUser[0] = (u8)ret;
46
47    // Send data0 ack, receive data1.
48    ret = SPI_MasterTransferByte((u8)(toUser[0] + 1));
49    DEBUG("data0+1: %i data1: %i\n", ((u8)(toUser[0]) + 1), ret);
50    if(ret < 0)  return ret;
51    toUser[1] = (u8)ret;
52
53    // Send data1 ack, receive DUMMY.
54    ret = SPI_MasterTransferByte((u8)(toUser[1] + 1));
55    DEBUG("data0+1: %i dummy: %i\n", ((u8)(toUser[1]) + 1), ret);
56    if(ret != DUMMY)  return -EIO;
57
58    // Confirm command.
59    ret = SPI_MasterTransferByte(MASTER_OK);
60    DEBUG("masterOK: %i slaveOK: %i\n", MASTER_OK, ret);
61    if(ret != SLAVE_OK) return -EIO;
62
63    return 0;
64 }
65
66 int Cmd_SetAOut(u8 *fromUser, int ch, int sch)
67 {
68    int ret;
69    unsigned char tx[3], expectedRx[3];
70
71    // check for valid channel.
72    if(ch < 0) return -EIO;
```

```
73
74      // Synchronize with ATMega128
75      ret = Cmd_WaitForAM128(CH_TO_CODE_AOUT(ch, sch));
76      DEBUG("code: %i count: %i\n", CH_TO_CODE_AOUT(ch, sch), ret);
77      if (ret < 0) return ret;
78
79      // Prepare to send several bytes.
80      tx[0] = fromUser[0];
81      tx[1] = fromUser[1];
82      tx[2] = DUMMY;
83      expectedRx[0] = CH_TO_CODE_AOUT(ch, sch) + 1;
84      expectedRx[1] = fromUser[0] + 1;
85      expectedRx[2] = fromUser[1] + 1;
86
87      // Send data and check returns.
88      ret = SPI_SendTableOfBytes(tx, expectedRx, 3);
89      if(ret < 0) return ret;
90
91      // Confirm command.
92      ret = SPI_MasterTransferByte(MASTER_OK);
93      DEBUG("masterOK: %i slaveOK: %i\n", MASTER_OK, ret);
94      if(ret != SLAVE_OK) return -EIO;
95
96      return 0;
97  }
98
99  int Cmd_GetDIn(u8 *toUser)
100 {
101     int ret;
102
103     // Synchronize with ATMega128
104     ret = Cmd_WaitForAM128(CODE_DIN);
105     DEBUG("1: code: %i count: %i\n", CODE_DIN, ret);
106     if (ret < 0) return ret;
107
108     // Send DUMMY, receive code ack.
109     ret = SPI_MasterTransferByte(DUMMY);
110     DEBUG("2: dummy: %i code+1: %i\n", DUMMY, ret);
111     if(ret != CODE_DIN + 1)  return -EIO;
112
113     // Send DUMMY, receive data0.
114     ret = SPI_MasterTransferByte(DUMMY);
115     DEBUG("3: dummy: %i data0: %i\n", DUMMY, ret);
116     if(ret < 0)  return ret;
117     toUser[0] = (u8)ret;
118
119     // Send data0 ack, receive DUMMY.
120     ret = SPI_MasterTransferByte(toUser[0] + 1);
121     DEBUG("4: data0+1: %i dummy: %i\n", ((u8)(toUser[0]) + 1), ret);
122     if(ret != DUMMY)  return -EIO;
123
124     // Confirm command.
125     ret = SPI_MasterTransferByte(MASTER_OK);
126     DEBUG("masterOK: %i slaveOK: %i\n", MASTER_OK, ret);
127     if(ret != SLAVE_OK) return -EIO;
128
129     return 0;
130 }
131
132 int Cmd_SetDOut(u8 *fromUser)
133 {
134     int ret;
135     unsigned char tx[3], expectedRx[3];
136
137     // Synchronize with ATMega128
138     ret = Cmd_WaitForAM128(CODE_DOUT);
139     DEBUG("code: %i count: %i\n", CODE_DOUT, ret);
140     if (ret < 0) return ret;
```

```
141
142     // Prepare to send several bytes.
143     tx[0] = fromUser[0];
144     tx[1] = fromUser[1];
145     tx[2] = DUMMY;
146     expectedRx[0] = CODE_DOUT + 1;
147     expectedRx[1] = fromUser[0] + 1;
148     expectedRx[2] = fromUser[1] + 1;
149
150     // Send data and check returns.
151     ret = SPI_SendTableOfBytes(tx, expectedRx, 3);
152     if(ret < 0) return ret;
153
154     // Confirm command.
155     ret = SPI_MasterTransferByte(MASTER_OK);
156     DEBUG("masterOK: %i slaveOK: %i\n", MASTER_OK, ret);
157     if(ret != SLAVE_OK) return -EIO;
158
159     return 0;
160 }
161
162 #ifdef BUFFERING_MODE
163 int Cmd_EnableAIn(u8 enable, int ch)
164 {
165     int ret;
166     unsigned char tx[2], expectedRx[2];
167
168     // check for valid channel.
169     if(ch < 0) return -EIO;
170
171     // Synchronize with ATMega128
172     ret = Cmd_WaitForAM128(CH_TO_CODE_AIN_EN(ch));
173     DEBUG("code: %i count: %i\n", CH_TO_CODE_AIN_EN(ch), ret);
174     if (ret < 0) return ret;
175
176     // Prepare to send several bytes.
177     tx[0] = enable;
178     tx[1] = DUMMY;
179     expectedRx[0] = CH_TO_CODE_AIN_EN(ch) + 1;
180     expectedRx[1] = enable + 1;
181
182     // Send data and check returns.
183     ret = SPI_SendTableOfBytes(tx, expectedRx, 2);
184     if(ret < 0) return ret;
185
186     // Confirm command.
187     ret = SPI_MasterTransferByte(MASTER_OK);
188     DEBUG("masterOK: %i slaveOK: %i\n", MASTER_OK, ret);
189     if(ret != SLAVE_OK) return -EIO;
190
191     return 0;
192 }
193 #endif
194
195 int Cmd_SetAOutResolution(u8 resol, int ch)
196 {
197     int ret;
198     unsigned char tx[2], expectedRx[2];
199
200     // Synchronize with ATMega128
201     ret = Cmd_WaitForAM128(CH_TO_CODE_AOUT_RES(ch));
202     DEBUG("code: %i count: %i\n", CH_TO_CODE_AOUT_RES(ch), ret);
203     if (ret < 0) return ret;
204
205     // Prepare to send several bytes.
206     tx[0] = resol;
207     tx[1] = DUMMY;
208     expectedRx[0] = CH_TO_CODE_AOUT_RES(ch) + 1;
```

```
209     expectedRx[1] = resol + 1;
210
211     // Send data and check returns.
212     ret = SPI_SendTableOfBytes(tx, expectedRx, 2);
213     if(ret < 0) return ret;
214
215     // Confirm command.
216     ret = SPI_MasterTransferByte(MASTER_OK);
217     DEBUG("masterOK: %i slaveOK: %i\n", MASTER_OK, ret);
218     if(ret != SLAVE_OK) return -EIO;
219
220     return 0;
221 }
222
223 #ifdef INTERRUPT_MODE
224 irqreturn_t irqFunk(int irq, void *dev_id, struct pt_regs *regs)
225 {
226     // Wake up queue waiting for interrupt.
227     DEBUG("received interrupt\n");
228     device->irqFlag = 1;
229     wake_up_interruptible(&wq);
230     return IRQ_HANDLED;
231 }
232 #endif
```

### C.2.4 avr32io_SPI.c

```
1
2  int SPI_MasterTransfer(u8 *rx, u8 *tx, int len)
3  {
4      // Structs for the transfer.
5      struct spi_message msg;
6      struct spi_transfer transfer;
7
8      // Makes the transferstruct.
9      memset(&transfer, 0, sizeof(transfer));
10     transfer.len = len;
11     transfer.rx_buf = rx;
12     transfer.tx_buf = tx;
13
14     // Makes the messagestruct, and includes the transfer in it.
15     spi_message_init(&msg);
16     spi_message_add_tail(&transfer, &msg);
17
18     // Execute transfer and return result.
19     return spi_sync(device->spi, &msg);
20 }
21
22 int SPI_MasterTransferByte(u8 tx)
23 {
24     int ret;
25     u8 rx;
26
27     // Execute transfer of one byte.
28     ret = SPI_MasterTransfer(&rx, &tx, 1);
29     if (ret < 0) return ret;
30     return rx;
31 }
32
33 int SPI_SendTableOfBytes(u8 *tx, u8 *expectedRx, int len)
34 {
35     int i, ret;
36     u8 rx[len];
37
38     // Execute transfer of the given number of bytes.
39     ret = SPI_MasterTransfer(rx, tx, len);
40     if(ret < 0) return ret;
```

```
41
42     // Checks that all received data was the expected data.
43     for(i = 0; i < len; i++){
44         DEBUG("tx%i: %i expectedRx%i: %i rx%i: %i\n", i, tx[i], i, expectedRx[i], i,
               rx[i]);
45         if(rx[i] != expectedRx[i]) return -EIO;
46     }
47
48     return 0;
49 }
```

## C.3 User Mode

### C.3.1 avr32io_driver.c

```
1
2  long avr32io_GetAnalogIn(char channel)
3  {
4      int fd, ret;
5      char devFile[13];
6      unsigned char data[2];
7      unsigned short result;
8
9      pthread_mutex_lock(&spiMutex);
10
11     // Open the correct device node.
12     sprintf(devFile, "/dev/avr32Ain%i", channel);
13     fd = open(devFile, O_RDWR);
14     if(fd < 0){
15         DEBUG("avr32io_user: error on open: %i\n", fd);
16         return fd;
17     }
18
19     // Read data from device node and close it.
20     ret = read(fd, data, 2);
21     close(fd);
22     if(ret < 0){
23         DEBUG("avr32io_user: error on read: %i\n", ret);
24         return ret;
25     }
26
27     pthread_mutex_unlock(&spiMutex);
28
29     // Convert result and return.
30     result = (data[0] << 8) + data[1];
31     return (long)(result);
32 }
33
34 short avr32io_SetAnalogOut(char channel, char subchannel, unsigned short value)
35 {
36     int ret, fd;
37     char devFile[14];
38     unsigned char data[3];
39
40     // Prepare data to send.
41     data[0] = subchannel;
42     data[1] = value >> 8;
43     data[2] = value;
44
45     pthread_mutex_lock(&spiMutex);
46
47     // Open the correct device node.
48     sprintf(devFile, "/dev/avr32Aout%i", channel);
49     fd = open(devFile, O_RDWR);
50     if(fd < 0){
51         DEBUG("avr32io_user: error on open: %i\n", fd);
```

```
52          return fd;
53       }
54
55       // Write data to device node and close it.
56       ret = write(fd, data, 3);
57       close(fd);
58       if(ret < 0){
59          DEBUG("avr32io_user: error on write: %i\n", ret);
60          return ret;
61       }
62
63       pthread_mutex_unlock(&spiMutex);
64
65       return 0;
66    }
67
68    short avr32io_SetAnalogOutDouble(char channel, char subchannel, double value)
69    {
70       // Converting the double value to integer and call the analog out function.
71       return avr32io_SetAnalogOut(channel, subchannel, (int)(aoutMultiplier[channel] *
             value));
72    }
73
74    short avr32io_GetDigitalIn()
75    {
76       int fd, ret;
77       unsigned char data;
78
79       pthread_mutex_lock(&spiMutex);
80
81       // Open the correct device node.
82       fd = open("/dev/avr32Din", O_RDWR);
83       if(fd < 0){
84          DEBUG("avr32io_user: error on open: %i\n", fd);
85          return fd;
86       }
87
88       // Read data from device node and close it.
89       ret = read(fd, &data, 1);
90       close(fd);
91       if(ret < 0){
92          DEBUG("avr32io_user: error on read: %i\n", ret);
93          return ret;
94       }
95
96       pthread_mutex_unlock(&spiMutex);
97
98       // Return result.
99       return (short)(data);
100   }
101
102   short avr32io_GetDigitalInBit(char bit)
103   {
104      short data;
105      unsigned char ucData;
106
107      // Get digital in data.
108      data = avr32io_GetDigitalIn();
109      if(data < 0){
110         return data;
111      }
112
113      // Check the bit, and return the result.
114      ucData = (unsigned char)(data);
115      if((ucData & (1 << bit)) > 0){
116         return 1;
117      }
118      else{
```

```
119         return 0;
120     }
121 }
122 
123 short avr32io_SetDigitalOut(unsigned char mask, unsigned char value)
124 {
125     int fd, ret;
126     unsigned char data[2];
127 
128     //Prepare data to send.
129     data[0] = mask;
130     data[1] = value;
131 
132     pthread_mutex_lock(&spiMutex);
133 
134     // Open the correct device node.
135     fd = open("/dev/avr32Dout", O_RDWR);
136     if(fd < 0){
137         DEBUG("avr32io_user: error on open: %i\n", fd);
138         return fd;
139     }
140 
141     // Write data to the device node and close it.
142     ret = write(fd, data, 2);
143     close(fd);
144     if(ret < 0){
145         DEBUG("avr32io_user: error on write: %i\n", ret);
146         return ret;
147     }
148 
149     pthread_mutex_unlock(&spiMutex);
150 
151     return 0;
152 }
153 
154 short avr32io_SetDigitalOutBit(char bit, char value)
155 {
156     unsigned char mask, newValue;
157 
158     // Prepare data for new function call.
159     mask = 1 << bit;
160     newValue = 0;
161     if (value > 0){
162         newValue = 1 << bit;
163     }
164 
165     // Write data and return.
166     return avr32io_SetDigitalOut(mask, newValue);
167 }
168 
169 short avr32io_SetAnalogOutResolution(char channel, char resolution)
170 {
171     int fd, ret;
172     char devFile[14];
173 
174     aoutMultiplier[channel] = (double)((pow(2, resolution) - 1) / 5);
175 
176     pthread_mutex_lock(&spiMutex);
177 
178     // Open the correct device node.
179     sprintf(devFile, "/dev/avr32Aout%i", channel);
180     fd = open(devFile, O_RDWR);
181     if(fd < 0){
182         DEBUG("avr32io_user: error on open: %i\n", fd);
183         return fd;
184     }
185 
186     // Perform IOCTL command on the device node and close it.
```

```
187     ret = ioctl(fd, AVR32IO_IOCTL_RES_AOUT, resolution);
188     close(fd);
189     if(ret < 0){
190        DEBUG("avr32io_user: error on write: %i\n", ret);
191        return ret;
192     }
193
194     pthread_mutex_unlock(&spiMutex);
195
196     return 0;
197  }
198
199  short avr32io_EnableAnalogIn(char channel, char enable)
200  {
201     int fd, ret;
202     char devFile[14];
203
204     pthread_mutex_lock(&spiMutex);
205
206     // Open the correct device node.
207     sprintf(devFile, "/dev/avr32Ain%i", channel);
208     fd = open(devFile, O_RDWR);
209     if(fd < 0){
210        DEBUG("avr32io_user: error on open: %i\n", fd);
211        return fd;
212     }
213
214     // Perform IOCTL command on the device node and close it.
215     ret = ioctl(fd, AVR32IO_IOCTL_EN_AIN, enable);
216     close(fd);
217     if(ret < 0){
218        DEBUG("avr32io_user: error on write: %i\n", ret);
219        return ret;
220     }
221
222     pthread_mutex_unlock(&spiMutex);
223
224     return 0;
225  }
```

### C.3.2 avr32io_threads.c

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <math.h>
4  void Init_IO(){
5
6     if(!IO_Started){
7
8     // Initialize periodic tasks.
9     InitPeriodicTasks();
10
11    // Starts analog output worker thread.
12    aoutCmdListFirst = 0;
13    aoutCmdListLast = 0;
14    workCount = 0;
15    pthread_create(&aoutWorker, NULL, Worker_Aout, NULL);
16
17    // Marks that this function have been run.
18    IO_Started = 1;
19    }
20 }
21
22 int avr32io_InitAnalogIn(int chanNumber, double period)
23 {
24    // Enables selected channel.
25         struct sAinChan *chan;
```

```
26          chan = ainChanList + chanNumber;
27          avr32io_EnableAnalogIn(chanNumber, 1);
28
29      // Start periodic thread for sampling the analog input channel.
30          pthread_mutex_init(&(chan->mutex), NULL);
31          StartPeriodicThread(&(chan->samplingTask), (int)(period/MASTER_TIMER_PERIOD
                ), Sampling_Ain, chan);
32  }
33
34  void *Sampling_Ain(void *ptr)
35  {
36          int count = 0;
37          struct sAinChan *chan;
38          chan = ptr;
39          chan->value = 0;
40
41      // Forever loop for sampling analog input.
42          while(1){
43          // Wait for periode.
44                  WaitPeriodicTask(&(chan->samplingTask));
45
46          // Lock mutex, update value and unlock mutex.
47                  pthread_mutex_lock(&(chan->mutex));
48                  chan->value = (float)(avr32io_GetAnalogIn(chan->chanNumber) *
                        0.004888);
49                  pthread_mutex_unlock(&(chan->mutex));
50          }
51  }
52
53  int avr32io_GetAnalogInThread(int chanNumber, double *data)
54  {
55          double result;
56          struct sAinChan *chan;
57          chan = ainChanList + chanNumber;
58
59      // Lock mutex, save value and unlock mutex.
60          pthread_mutex_lock(&(chan->mutex));
61          result = chan->value;
62          pthread_mutex_unlock(&(chan->mutex));
63
64          // For some reason returning a double didn't work with RTW
65          *data = result;
66          return 0;
67  }
68
69  int avr32io_SetAnalogOutThread(int chanNumber, int subchanNumber, double value)
70  {
71      // Makes new list item.
72      struct sAoutCmd *temp;
73      temp = malloc(sizeof(struct sAoutCmd));
74      temp->chanNumber = chanNumber;
75      temp->subchanNumber = subchanNumber;
76      temp->value = value;
77      temp->next = NULL;
78
79      // Lock mutex, add item to the list and unlock mutex.
80      pthread_mutex_lock(&cmdListMutex);
81      if(aoutCmdListFirst == NULL){
82          // List empty.
83          aoutCmdListFirst = temp;
84      }
85      else{
86          // List not empty.
87          aoutCmdListLast->next = temp;
88      }
89      aoutCmdListLast = temp;
90      workCount++;
91      pthread_mutex_unlock(&cmdListMutex);
```

```
92
93      // Signaling the worker thread.
94      pthread_cond_signal(&workCond);
95          return 0;
96  }
97
98
99  void *Worker_Aout()
100 {
101     struct sAoutCmd *temp;
102
103     // Forever loop.
104         while(1){
105                 // Wait for work.
106                 pthread_mutex_lock(&workMutex);
107                 pthread_cond_wait(&workCond, &workMutex);
108                 pthread_mutex_unlock(&workMutex);
109
110        // Executes commands until list is empty.
111                 while(workCount>0){
112          // Locks mutex, grab first list item and unlock mutex.
113                         pthread_mutex_lock(&cmdListMutex);
114                         temp = aoutCmdListFirst;
115                         aoutCmdListFirst = temp->next;
116                         workCount--;
117                         pthread_mutex_unlock(&cmdListMutex);
118
119          // Send value to I/O-card and free the list items memory.
120                         avr32io_SetAnalogOut(temp->chanNumber, temp->subchanNumber,
                                (int)(aoutMultiplier[temp->chanNumber] * temp->value))
                                ;
121                         free(temp);
122                 }
123         }
124 }
```

### C.3.3 periodictask.h

```
1   #ifndef DEF_PTASK
2   #define DEF_PTASK
3
4   // includes
5   #include <signal.h>
6   #include <pthread.h>
7   #include <sys/time.h>
8
9   // Struct that represents one periodic task.
10  struct sPeriodicTask{
11          struct sPeriodicTask *next; // Pointer to the next periodc task.
12          int periode; // The periode of the task.
13          pthread_t thread; // The thread thats running the task.
14          pthread_cond_t cond; // The condition that controls the task.
15          pthread_mutex_t mutex; // The mutex protecting the condition.
16  };
17
18  // Struct for the timer.
19  struct itimerval rttimer;
20
21  // Struct for the root task.
22  struct sPeriodicTask *rootTask;
23
24  // This function initialize the periodic task system.
25  int InitPeriodicTasks();
26
27  //Function that runs every millisecond, signaling all the periodic tasks.
28  void fTimer();
29
```

```
30 // This function starts a new periodic task. Started tasks can't be stopped.
31 int StartPeriodicTask(struct sPeriodicTask *pTask, int periode);
32
33 // This function starts a new periodic thread. Started threads can't be stopped.
34 int StartPeriodicThread(struct sPeriodicTask *pTask, int periode, void *(*function)
       (void *), void* data);
35
36 // Used inside the periodic task/thread to sleep until next periode.
37 int WaitPeriodicTask(struct sPeriodicTask *pTask);
38
39 #include "periodicTask.c"
40
41 #endif
```

### C.3.4 periodictask.c

```
1  int InitPeriodicTasks(){
2     static int initialized;
3
4     if(initialized > 0) return -1;
5     // Initialize the timer.
6     signal(SIGALRM,fTimer);
7     rttimer.it_value.tv_sec = 0;
8     rttimer.it_value.tv_usec = 1;
9     rttimer.it_interval.tv_sec = 0;
10    rttimer.it_interval.tv_usec = 1000;
11    setitimer(ITIMER_REAL, &rttimer, NULL);
12
13    // Initialize a empty taks table.
14    rootTask = NULL;
15    initialized = 1;
16
17    // Return zero.
18    return 0;
19 }
20
21 void fTimer(){
22    // Variables
23    static long count;
24    struct sPeriodicTask *task;
25
26    // Increase the counter.
27    count++;
28
29    // Walks through the tasks and signal the ones that need to run.
30    task = rootTask;
31
32    while(task != NULL){
33       if((count % (task->periode)) == 0){
34          // Signal a task.
35          pthread_cond_signal(&(task->cond));
36       }
37       // Next task.
38       task = task->next;
39    }
40 }
41
42 int StartPeriodicTask(struct sPeriodicTask *pTask, int periode)
43 {
44    // Variables
45    struct sPeriodicTask *newTask, *previousTask;
46
47    // The new task will be pointing to nothing.
48    pTask->next = NULL;
49    if(rootTask == NULL){
50       // If rootTask doesn't point at anything, this task is the first task.
51       rootTask = pTask;
```

```
52      }
53      else{
54          // If rootTask point at something, the first task that doesnt point at a next
                task has to be found.
55          previousTask = rootTask;
56          newTask = rootTask->next;
57          while(newTask != NULL){
58              // Will quit when a empty next pointer is found.
59              previousTask = newTask;
60              newTask = newTask->next;
61          }
62
63          // The empty newTask will now point to pTask, and the former newest task will
                point to the new.
64          newTask = pTask;
65          previousTask->next = newTask;
66      }
67      // Initialize the content of the new task.
68      pTask->periode = periode;
69      pthread_cond_init(&(pTask->cond), NULL);
70      pthread_mutex_init(&(pTask->mutex), NULL);
71      return 0;
72  }
73
74  int StartPeriodicThread(struct sPeriodicTask *pTask, int periode, void *(*
        __start_routine) (void *), void* data)
75  {
76      int ret;
77
78      // Start a periodic task and a thread.
79      ret = StartPeriodicTask(pTask, periode);
80      pthread_create(&(pTask->thread), NULL, (*__start_routine), data);
81      return ret;
82  }
83
84
85  int WaitPeriodicTask(struct sPeriodicTask *pTask)
86  {
87      // Wait for the signal to the given task.
88      pthread_mutex_lock(&(pTask->mutex));
89      pthread_cond_wait(&(pTask->cond), &(pTask->mutex));
90      pthread_mutex_unlock(&(pTask->mutex));
91      return 0;
92  }
```

### C.3.5 stopwatch.h

```
1   #ifndef DEF_SWATCH
2   #define DEF_SWATCH
3
4   // includes
5   #include <sys/time.h>
6   #include <stdio.h>
7
8   // Constants
9   #define SWATCH_IDLE 0
10  #define SWATCH_STARTED 1
11  #define SWATCH_STOPED 2
12
13  // Struct that represent a stopwatch.
14  struct sStopWatch{
15      int state; // The state of the stopwatch.
16      int usec; // Result of the stopwatch, might not be necesary.
17      struct timeval start, stop; // Structs for keeping the time.
18  };
19
20  // This function starts the stopwatch.
```

```
21 int StartStopWatch(struct sStopWatch *sWatch);
22
23 // This function stops the stopwatch and returns the result.
24 int StopStopWatch(struct sStopWatch *sWatch);
25
26 #include "stopwatch.c"
27 #endif
```

### C.3.6 stopwatch.c

```
1  int StartStopWatch(struct sStopWatch *sWatch)
2  {
3     // Returns an error if the watch have already been started.
4     if(sWatch->state == SWATCH_STARTED) return -1;
5
6     // Starts the watch, notice that gettimeofday is the last thing to happen. This
          is because that will be most accurate.
7     sWatch->state = SWATCH_STARTED;
8     sWatch->usec = -1;
9     gettimeofday(&(sWatch->start), NULL);
10    return 0;
11 }
12
13 int StopStopWatch(struct sStopWatch *sWatch)
14 {
15    // Returns an error if the watch havent been started.
16    if(sWatch->state != SWATCH_STARTED) return -1;
17
18    // Stops the watch and calculate the result, and return this. Notice that
          gettimeofday is the first thing to happen. This is because that will be most
           accurate.
19    gettimeofday(&(sWatch->stop), NULL);
20    sWatch->state = SWATCH_STOPED;
21    sWatch->usec = sWatch->stop.tv_sec * 1000000 + sWatch->stop.tv_usec - sWatch->
          start.tv_sec * 1000000 - sWatch->start.tv_usec;
22    return sWatch->usec;
23 }
```

## C.4 Matlab S-functions

### C.4.1 Analog input TLC

```
1  %implements "avr32io_adc" "C"
2
3  %function InitializeConditions(block, system) Output
4    /* %<Type> Block: %<Name> */
5
6  %% Enables all analog input channels when system starts up.
7
8     %foreach opIdx = NumDataOutputPorts
9        %if LibBlockOutputSignalConnected(opIdx)
10          avr32io_EnableAnalogIn(%<opIdx>, 1);
11       %endif
12    %endforeach
13
14 %endfunction
15
16 %function Outputs(block, system) Output
17    /* %<Type> Block: %<Name> */
18
19 %% Do command for getting analog input value.
20
21    %foreach opIdx = NumDataOutputPorts
22       %if LibBlockOutputSignalConnected(opIdx)
```

```
23          %<LibBlockOutputSignal(opIdx, "", "", 0)> = avr32io_GetAnalogIn(%<opIdx>)
                * 0.0048828;
24        %endif
25     %endforeach
26
27 %endfunction
```

### C.4.2 Analog output TLC

```
1  %implements "avr32io_dac" "C"
2
3
4  %function InitializeConditions(block, system) Output
5    /* %<Type> Block: %<Name> */
6
7  %% Set resolution for both of the analog out channels.
8
9     avr32io_SetAnalogOutResolution(0, (int)%<LibBlockParameter(P1, "", "", 0)> + 7);
10    avr32io_SetAnalogOutResolution(1, (int)%<LibBlockParameter(P2, "", "", 0)> + 7);
11
12 %endfunction
13
14 %function Outputs(block, system) Output
15    /* %<Type> Block: %<Name> */
16
17 %%Loops through all ports, and set analog out functions for the ports that are
       connected.
18
19     static int count[6];
20     int ret;
21    %foreach opIdx = NumDataInputPorts
22       %if LibBlockInputSignalConnected(opIdx)
23          %if opIdx<3
24             ret = avr32io_SetAnalogOutDouble(0, %<opIdx>, (double)(%<
                   LibBlockInputSignal(opIdx, "", "", 0)>));
25          %else
26             ret = avr32io_SetAnalogOutDouble(1, %<opIdx> - 3, (double)(%<
                   LibBlockInputSignal(opIdx, "", "", 0)>));
27          %endif
28             if(ret < 0){
29                count[%<opIdx>]++;
30                printf("%<opIdx>␣count:␣%i␣data:␣%f\n", count[%<opIdx>], %<
                      LibBlockInputSignal(opIdx, "", "", 0)>);
31             }
32                %endif
33         %endforeach
34 %endfunction
```

### C.4.3 Digital input TLC

```
1  %implements "avr32io_din" "C"
2
3  %function Outputs(block, system) Output
4     /* %<Type> Block: %<Name> */
5
6  %% Reads digital inout values from I/O-card.
7
8     unsigned char din = avr32io_GetDigitalIn();
9
10    %foreach opIdx = NumDataOutputPorts
11       %if LibBlockOutputSignalConnected(opIdx)
12          if(din & (1<<%<opIdx>)){
13             %<LibBlockOutputSignal(opIdx, "", "", 0)> = 5;
14          }
15          else{
16             %<LibBlockOutputSignal(opIdx, "", "", 0)> = 0;
```

```
17          }
18        %endif
19    %endforeach
20
21 %endfunction
```

### C.4.4 Digital output TLC

```
1  %implements "avr32io_dout" "C"
2
3  %function Outputs(block, system) Output
4     /* %<Type> Block: %<Name> */
5
6  %% Checking all channels and make a mask and data byte that are sendt as a command
       to the I/O-card.
7
8     unsigned char dout = 0;
9     unsigned char mask = 0;
10
11    %if LibBlockInputSignalConnected(0)
12       mask = mask | (1<<0);
13       if((double)(%<LibBlockInputSignal(0, "", "", 0)>) >= (double)(%<
             LibBlockParameter(P1, "", "", 0)>)){
14          dout = dout | (1<<0);
15       }
16    %endif
17    %if LibBlockInputSignalConnected(1)
18       mask = mask | (1<<1);
19       if((double)(%<LibBlockInputSignal(1, "", "", 0)>) >= (double)(%<
             LibBlockParameter(P2, "", "", 0)>)){
20          dout = dout | (1<<1);
21       }
22    %endif
23    %if LibBlockInputSignalConnected(2)
24       mask = mask | (1<<2);
25       if((double)(%<LibBlockInputSignal(2, "", "", 0)>) >= (double)(%<
             LibBlockParameter(P3, "", "", 0)>)){
26          dout = dout | (1<<2);
27       }
28    %endif
29    %if LibBlockInputSignalConnected(3)
30       mask = mask | (1<<3);
31       if((double)(%<LibBlockInputSignal(3, "", "", 0)>) >= (double)(%<
             LibBlockParameter(P4, "", "", 0)>)){
32          dout = dout | (1<<3);
33       }
34    %endif
35    %if LibBlockInputSignalConnected(4)
36       mask = mask | (1<<4);
37       if((double)(%<LibBlockInputSignal(4, "", "", 0)>) >= (double)(%<
             LibBlockParameter(P5, "", "", 0)>)){
38          dout = dout | (1<<4);
39       }
40    %endif
41    %if LibBlockInputSignalConnected(5)
42       mask = mask | (1<<5);
43       if((double)(%<LibBlockInputSignal(5, "", "", 0)>) >= (double)(%<
             LibBlockParameter(P6, "", "", 0)>)){
44          dout = dout | (1<<5);
45       }
46    %endif
47    %if LibBlockInputSignalConnected(6)
48       mask = mask | (1<<6);
49       if((double)(%<LibBlockInputSignal(6, "", "", 0)>) >= (double)(%<
             LibBlockParameter(P7, "", "", 0)>)){
50          dout = dout | (1<<6);
51       }
```

```
52      %endif
53      %if LibBlockInputSignalConnected(7)
54          mask = mask | (1<<7);
55          if((double)(%<LibBlockInputSignal(7, "", "", 0)>) >= (double)(%<
                LibBlockParameter(P8, "", "", 0)>)){
56              dout = dout | (1<<7);
57          }
58      %endif
59
60      avr32io_SetDigitalOut(mask, dout);
61
62  %endfunction
```

# Bibliography


[1] *ATmega128 datasheet.*

[2] *AVR32 32-bit MCU/DSP Overview.* Atmel Corporation. http://www.atmel.com/products/AVR32.

[3] *AVR32 Linux Wiki.* AVR32 Linux Wiki contributors. http://www.avr32linux.org.

[4] *Eagle Layout Editor.* CadSoft. http://www.cadsoft.de/.

[5] *MAXIM 233 datasheet.*

[6] *MC1458 datasheet.*

[7] *MC7805 datasheet.*

[8] *STK1000.* Atmel Corporation. http://www.atmel.com/dyn/products/tools_card.asp?tool_id=3918.

[9] *STK500 User guide.*

[10] *Ubuntu.* Canonical Ltd. http://www.ubuntu.com.

[11] *AVR32 Architecture Manual.* Atmel Corporation, Feb 2006.

[12] *Real-Time Workshop Target Language Compiler.* The Mathworks, Mar 2007.

[13] *Real-Time Workshop Target User's Guide.* The Mathworks, Mar 2007.

[14] Wikipedia contributors. *Control System.* Wikipedia, The Free Encyclopedia., May 2007. http://en.wikipedia.org/w/index.php?title=Control_system&oldid=131753209.

[15] Wikipedia contributors. *Linux Kernel.* Wikipedia, The Free Encyclopedia., Jun 2007. http://en.wikipedia.org/w/index.php?title=Linux_kernel&oldid=137027739.

[16] Wikipedia contributors. *Porting.* Wikipedia, The Free Encyclopedia., Jun 2007. http://en.wikipedia.org/w/index.php?title=Porting&oldid=136428272.

[17] Wikipedia contributors. *Porting.* Wikipedia, The Free Encyclopedia., May 2007. http://en.wikipedia.org/w/index.php?title=APT&oldid=131003365.

[18] Wikipedia contributors. *Real-Time Computing.* Wikipedia, The Free Encyclopedia., Jun 2007. http://en.wikipedia.org/w/index.php?title=Real-time_computing&oldid=135997153.

[19] Wikipedia contributors. *Ring.* Wikipedia, The Free Encyclopedia., May 2007. http://en.wikipedia.org/w/index.php?title=Ring_%28computer_security%29&oldid=131608375.

[20] Marco Cesati Daniel P. Bovet. *Understanding the Linux Kernel*. O'Reilly, 3rd edition, Nov 2005.

[21] Greg Kroah-Hartman Jonathan Corbet, Alessandro Rubini. *Linux Device Drivers*. O'Reilly, 3rd edition, February 2005.

[22] Lindergren. Åpent marked for Linux Embedded. *Datarespons - Interrupt*, 2006. `http://www.datarespons.com/templates/interrupt.aspx?id=1826`.

[23] GNU Project. *The Free Software Definition*. GNU Project, February 2007. `http://www.gnu.org/philosophy/free-sw.html`.

[24] Øyvind Netland. Rapid prototyping with matlab real-time workhsop on an embedded control system with atmel avr32 processor. Master's thesis, NTNU, 2007.