Report
Exercise 2
TDT4258

Group 1
Jan Bremnes
Magnus Kirø

Table of Contents

Abstract

In this exercise, we got our first experiences with the C programming language. We learned how to use C to program the AVR32, and how to use an ABDAC to generate sound. Two different audio wave forms were used, the square wave and the sine wave, and we learned a little bit about audio programming.

Introduction

This exercise was a continuation of Exercise 1, which is described in a previous report, in the course TDT4258 Mikrokontroller Systemdesign.
The requirements of the exercise were the following:

"Write a program in C, which is to be run directly on the STK1000 (without any operating system), and which plays different sound effects when pressing the different buttons. Every sound effect must be sound effects you intend to use in the finished implementation of Pong. You have to make at least three different sounds (e.g. ball hits wall, ball hits paddle, player dies). In addition, you may create a start up sound to be played when the game starts.
The microcontroller has an internal ABDAC, which it is recommended that you use. It is mandatory that you use an interrupt routine to feed samples to the ABDAC"

- translated from the exercise description in Øvingsheftet [1]

This was the second step towards implementing our own version of the game Pong. The goals in this exercise was to learn C-programming, I/O-handing and processing of interrupts on the AVR32 in C and use of the microcontrollers ABDAC for generating audio.
As in the previous exercise, we used the AT32AP7000 microcontroller from Atmel, mounted on an STK1000 development board.
We began the development with coding Exercise 1 in C. Then we continued with the creation of sound. Followed by songs and sound effects. We finished of with code cleaning and writing the report.

Description and Methodology

Preparations and setup.

Before we could start the development of our sound driver we had to configure the development board correctly. This is done by setting different straps and cable connectors on the development board. We used a flat cable to connect the buttons (J25) to GPIO (J1) and the LEDs (J15) to GPIO (J3). This is needed to make the buttons and LED work, PIOB will now control the buttons, and PIOC controls the LEDs. In addition there are eight jumpers we had to put in the right positions. It is SW1 - SW6 and JP4 and JP5. This has to do with the sound handling on the board. Making sure the sound goes to the audio IO port.

List of switches and their position:(switch - position)

1. sw1 - SPI0
2. sw2 - PS2A / MMCI / USART1
3. sw3 - SSC0 / PWM[0,1] / GCLK
4. sw4 - GPIO
5. sw5 - LCDC
6. sw6 - GPIO
7. JP4 - INT.DAC
8. JP5 – INT.DAC

JP4 and JP5 decides whether the internal or external DAC should be used. Our setting chooses the internal DAC, which is what the exercise description recommends.
See the picture below to see where the different jumpers and important connectors are located on the development board. The green circles represent the connection between the buttons and GPIO, and the blue circles represent the connection between the LEDs and GPIO.

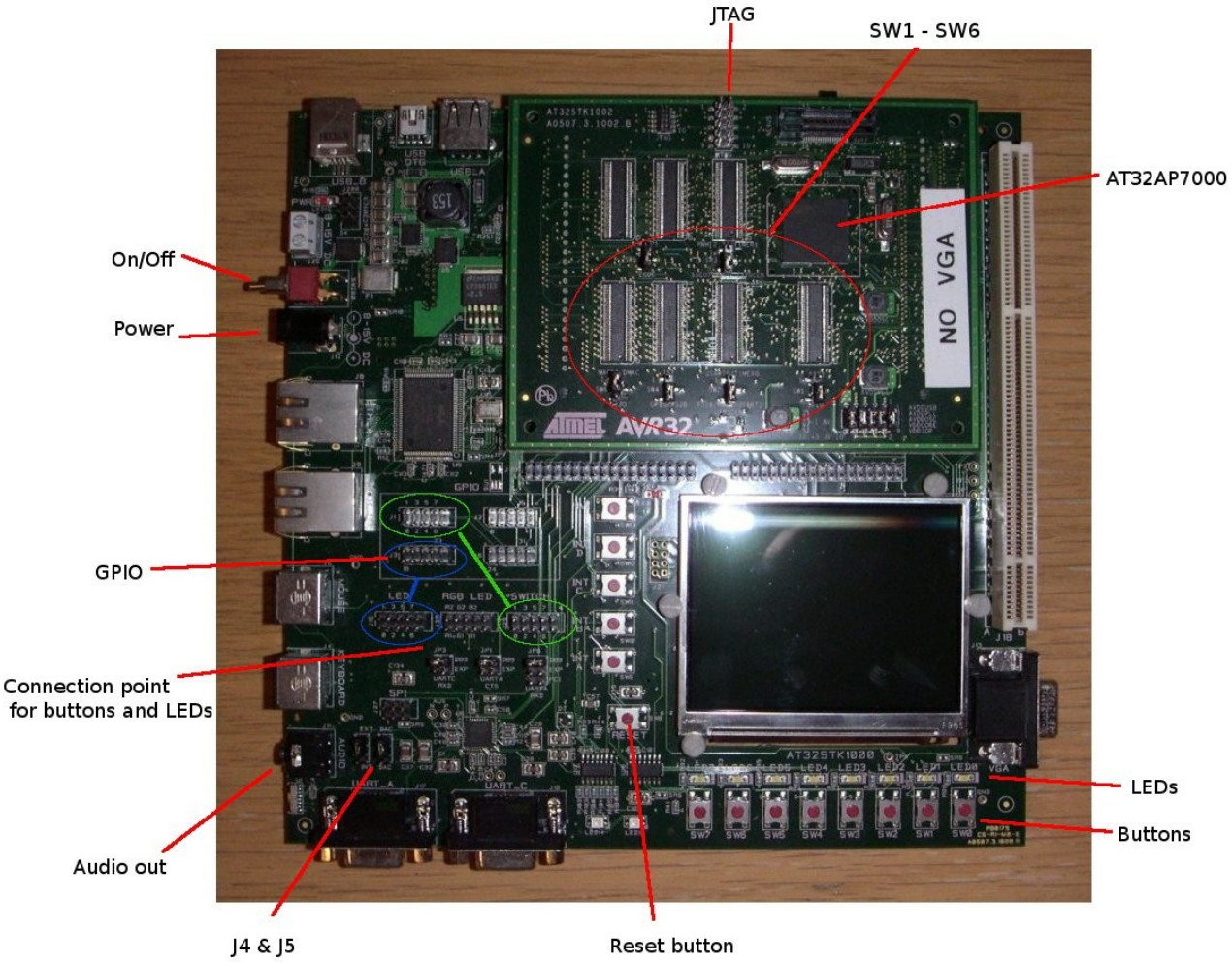


Fig. 1

As in the previous exercise, the JTAGICE was used to communicate with, and program the microcontroller.

GNU Development Tools

Programming was done in a GNU/Linux environment (Kubuntu), and we used Kate as our editor and GCC as the compiler, which is accessible from the terminal (Konsole).

The GCC is a broadly used compiler that works well with C. To compile .c files with GCC the command “gcc filename.c” can be used. This is a simple command that compiles the code to an a.out file. Compiling “by hand” every time the program code is changed, would be time consuming and tedious, so we used a makefile to handle the compiling and linking. The makefile we used, is based on the makefile from Exercise 1, we have just rewritten it enough to compile C code instead

of assembly.
To generate the look up table for the sine wave functionality, we wrote a script in Python, which produces the sinetable.h file when executed. The Python script is supplied along with the program code (generate_sine.py). The Python scripts does not need to be run for the program to work, as "sinetable.h" has been included in the source code.

The make file. header files etc.

The makefile assembles the file "oeving2.c" and creates "oeving2.o" which it links with "oeving2.h" and "sinetable.h" to create "oeving2.elf" which is then uploaded to the microcontroller.
Interrupt Handling.
Interrupts handling was done mainly the same way in C as in assembly. We translated our code from Exercise 1 from assembly to C to see that we were able to control the LEDs and buttons with C. Interrupt handling in C is a bit different from interrupt handling in assembly. To enable interrupts, we had to include the following header file:

```
#include <sys/interrupts.h>
```

The interrupt routine needs the following prototype:

```
__int_handler_ *int_handler (void);
```

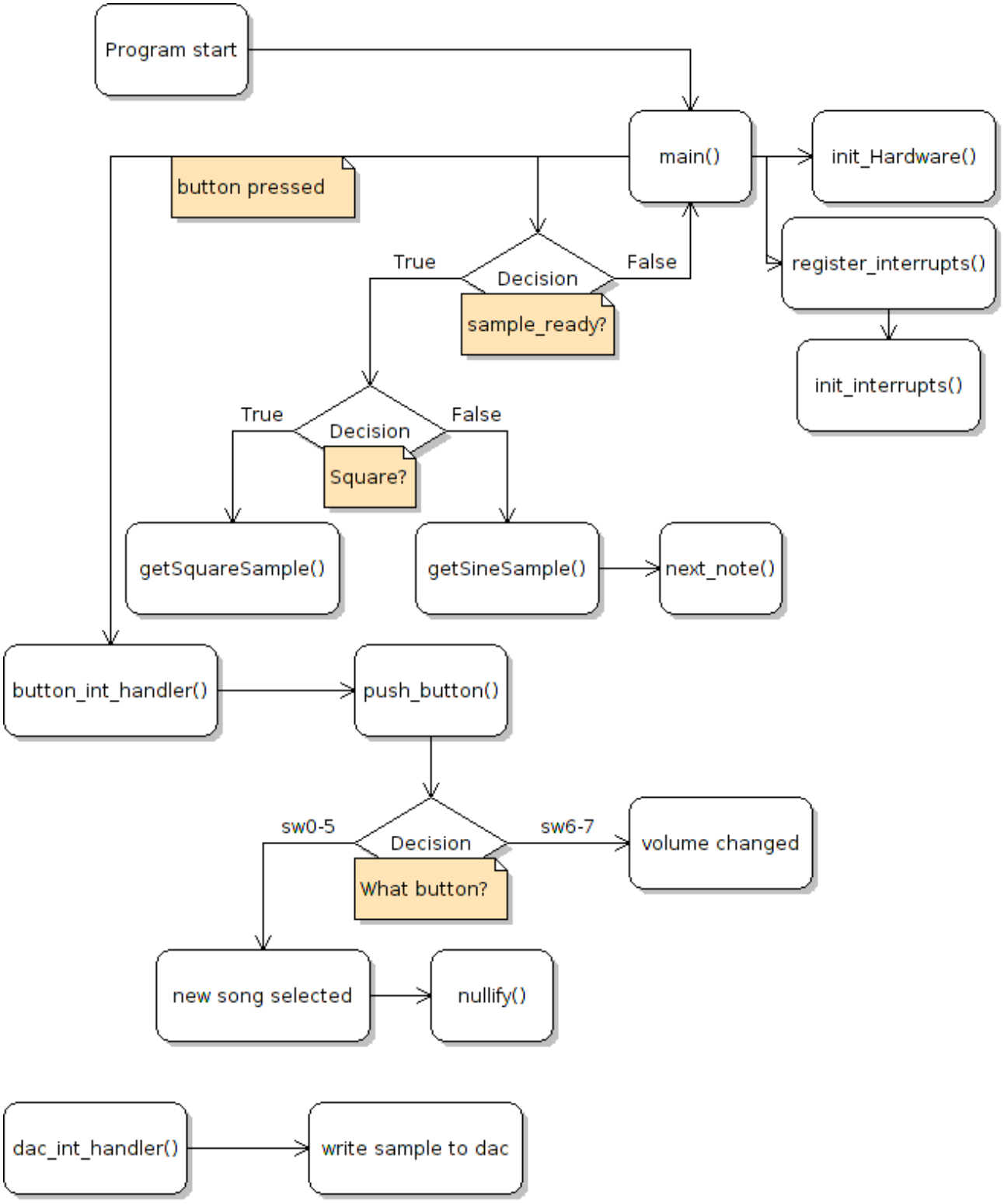
Since our program was going to generate two types of interrupts, one for the buttons and one for the ABDAC, we needed two different interrupt routines.
To initialize the interrupts, we needed to call the following functions:

```
void set_interrupts_base(void *base);
__int_handler register_interrupt(__int_handler, int int_grp, int line, int priority)
void init_interrupts(void);
```

When this was done, and the button interrupts was working, we moved on to getting the ABDAC to work.
We followed the recommended procedure in Øvingshefte [1], and after setting up the clock, we were able to produce noise by feeding the DAC with random values. We chose OSC1 as our clock, which has a frequency of 12MHz[1] and since interrupts are generated every 256 cycles, using

OSC1 gives us a bitrate of 46.875 kHz. The DAC should therefore be given a new sample 46 875 times per second.

Program Flow Chart



The Buttons

The buttons are connected to PIOB, and shares this register with the ABDAC.
Because of the number of operations that the interrupt routine for the buttons perform, we found it unnecessary to implement a debounce function, instead we just read the ISR register in PIOB twice; once at the beginning of the interrupt routine, and once right before the routine returns. The eight buttons are numbered SW0 through SW7, and when pressed, buttons SW0 - SW5 will play a sound on the microcontroller. Buttons SW7 turns up the volume, and SW6 turns it down. This is done by increasing or decreasing the amplitude every time one of the buttons are pressed. The default volume level, is an amplitude of 4000, and it is adjusted in steps of 4000. Since the ABDAC needs 16 bit values, a signed short, we put the maximum volume at 32000, and the minimum is 0, thereby giving us eight different volume levels, nine if mute is counted.
The buttons are activated on lines 243-246 in “oeving2.c”, in the same way as in Exercise 1; by enabling pull-up, enabling interrupts etc.
To read the button state, we have to do it a bit differently than in Exercise 1, becasue the ABDAC also uses PIOB, which the buttons are connected to. In order to correctly read the button that has been pressed, by reading the PDSR register in PIOB, we have to mask the value that only the lower two bytes are read. This is done by taking the value in the PDSR register, and perform an AND operation with 0xFF, and then compare the result with the value of the different buttons.

The LEDs

The LEDs are connected to PIOC and does not serve any other function than displaying the level of the volume. Each LED correspond to one level of volume, so that with the default volume, only the rightmost LED is lit, and as the volume is turned up, the LEDs are turned on so that all LEDs are lit when the volume is at maximum.

The Sound Driver

The sound driver was the hardest part of the exercise. It took us some time to figure out how to set up the clock for the DAC, and get it to output clean sound instead of random noise. We started out with generating sound by using a square wave. This is the simplest of all wave forms, with the amplitude just switching between positive and negative. We also included a crude sine wave function, to get softer sounds, as the sound produced by square waves are quite harsh in comparison. The sine wave functionality is far from perfect, and we consider it a work in progress.

The Sounds

The program enables the microcontroller to play six different sounds, by pressing buttons SW0 through SW5. There are two different “bouncing” sounds; one for when the ball hits a paddle (SW0), and one for when the ball hits a wall (SW1). In addition, there’s a sound for when a player scores a point (SW2), one for loosing the game (SW4) and one for winning the game (SW3). Lastly, there’s a start up sound that will be played when the game starts (SW5). The start up sound is our interpretation of the first part of the theme “The Throne Room” from the Star Wars movies. Anyone listening to it, must please excuse our lack of musical talent. The sounds made when pressing buttons SW0 - SW3, are generated using a square wave, while the others a generated using a sine wave.

Square Wave

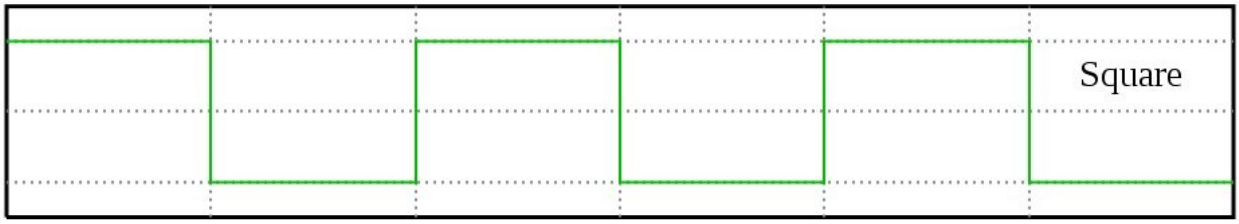


Fig 2 - illustration source: Wikipedia [2]

The square wave is the simplest of all wave forms.
For the square sound waves we have declared wave length and duration. The duration says how long the sound should last.The wave length says how long one period should be while the amplitude says how strong the sound should be. The longer the wave length the darker the created tone is. As sound is dependent on change, the clue in creating a sound wave is to change the wave from positive to negative. We do this by multiplying our sample with -1 every half wave length. This creates our square sound wave as in the figure above.
Our actual implementation of this sound wave use counters and check whether or not it should negate the amplitude that is set. Or rather we have a square_wave variable we multiply with -1 every half wave length. It change between 1 and -1, and then we set the sample to square_wave multiplied with the amplitude.
This sample is created int our “getSquareSample()” method. The created sample is played by the dac.

Examining the source code, one might notice that the square wave function is rather limited in that it allows a sound generated by the function to have only two different frequencies, which are dependant on the wave length value. This is a rather big blunder from our side, and when we realized the mistake, we didn't have time to correct it. Therefore the square wave part of our program, should only be used to generate short "bouncing" sounds, while more complex sounds should be generated with the sine wave.

Sine Wave

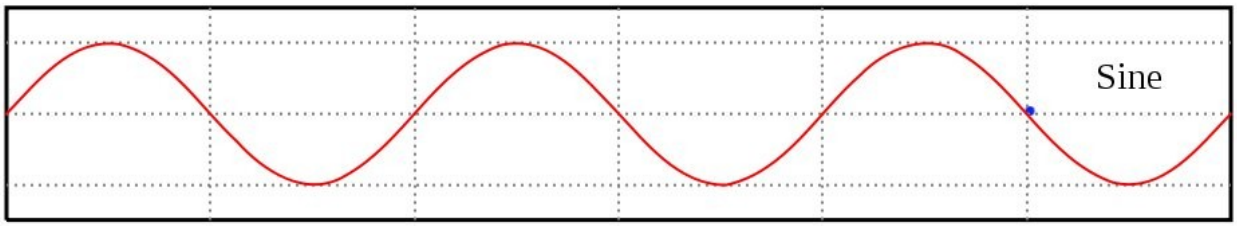


Fig 3 - illustration source: Wikipedia [2]

To generate the sine wave, we used a look up table generated with the supplied Python script. The table contains 128 values for a sine wave, and we used linear interpolation to calculate the values not in the table. A look up table with 128 values, allows us to calculate the exact value of the sine wave to within less than 1% error [3], which we consider more than accurate enough for this exercise.

We spent quite a great amount of time, trying to get the interpolation of the sine curve values to work. We couldn't see whether the problem was our logic or some other programing bug and we got quite frustrated. The interpolated values took too much time to calculate for our program to be able to feed the DAC with samples at a high enough rate. A friend of ours, pointed out to us that perhaps the microcontroller did not have an FPU, which he of course was entirely correct in assuming. Switching from floating point arithmetic to fixed point did the trick and we were able to produce the sine wave at acceptable speed.

As mentioned, the DAC outputs a new sample 46 875 times pr. second. Since our sine table contained 128 values, we had a base frequency of approximately 366Hz (46875 / 128 = 366.21). A sample rate of 46.875 kHz does not allow us to produce the frequencies needed to accurately represent the musical notes [4]. There is always a few Hz too many or too few, but we didn't take the time to figure out how to get a more accurate representation, as that could potentially be very time consuming, and is beyond the scope of this exercise.

To calculate how many sine values are needed for each frequency, we used the following formula:

no_samples = bitrate / frequency

With 128 values in the sine table, we would then have to output every n'th value, where

n = 128 / no_samples

For example, a frequency of 440Hz, would give us the following:

no_samples = 46875 / 440 = 106.534...
n = 128 / no_samples = 1.201...

This means that to play a sound with frequency 440Hz, every 1.201'th value in the sine table has to be fed to the DAC, for a total of 106.534 samples for each period of the wave. But since we cannot get 106.534 samples, only 106, the frequency won't be correct. Also, since the sine table does not contain a value for 1.201, 2.402 etc, we had to do a linear interpolation to calculate the correct sine value. To start with, we used floating point values, as mentioned. As this was too slow, we substituted the floating point value with the fixed point representation, multiplied by 1 000, to get some degree of accuracy. This was done as follows:

```
sample = sinetable[a]*1000+((sinetable[a+1]*1000-sinetable[a]*1000)*delta/1000);
```

Where delta is, in our example, 0.201*1000 = 210. The calculated sample is then multiplied with the amplitude before it is ready to be fed to the DAC.

```
sample = (sample/1000)*(amp/100);
```

The calculation of sine values is done in the "getSineSample()"-method in the source code. When calculating sine samples, we divide the amplitude by 100, because we use the same amplitude for both the square wave and the sine wave, and the square wave values are either 1 or -1, while the sine wave values range from 99 to -99.

Results and Tests

We had some problems with getting our buttons to work properly, because in the beginning we just read the pdsr-register and compared it with the values we had gotten for the buttons in Exercise 1. We failed to realize that since the DAC uses part of the PIOB-register, we have no way of knowing what the contents of the pdsr-register will be. So we had buttons that randomly stopped working, button presses that did not register etc. Our stud.ass pointed it out to us that since the DAC uses PIOB, we had to mask all but the lower two bytes of the PDSR-register to be able to read the buttons correctly.

To check that the sine wave was correct and gave us the expected frequencies, we used a guitar tuner app for Android telephones; gStrings from coherter.org [5]

Future Work

While we consider the goals of the exercise to be met, a lot of work can still be done, as we have only made a very rudimentary program for audio generation. Instead of having to alter the source code every time to add new sounds, we can write a parser that reads songs from a text file, and generates a .h file with the sound encoded in a format understood by the program, and then link this file with the rest of the program during compilation.

We can look into the possibility of calculating a new frequency table for the musical notes, based on the frequencies our program is capable of producing. It is also possible to add functionality for playing chords, instead of single notes, and adding amplitude envelopes to emulate the sound of different musical instruments. We can also try to implement frequency modulation, to make it possible to have more than one note played at a time [6].

Evaluation of assignment

The exercise has been very interesting, frustrating and educational.

There were many times when we couldn't understand why things weren't working the way we thought, and often we blamed the hardware, but it always turned out to be an error of our doing.

This exercise was quite a bit more challenging than the previous one, with us running into a lot more difficulties, but the rewards were greater.

Conclusion

We managed to meet all the requirements of the exercise, which was to get the microcontroller to produce at least three different sounds.

It has given us some experience with the use of a the C programming language, which neither of us had any experience with before. We have learned quite a lot about writing software that communicates directly with the hardware, instead of through an OS, and have given us some insight into audio programming, which looks like a very interesting, but complicated and extensive field of study. We wasted many hours trying everything we could think of to get our sine wave to work, until we were made aware of the fact that the microcontroller lacks a FPU. Lesson learned, and we won't mistake a microcontroller for just being a downsized multipurpose computer in the future.

Acknowledgements (optional)

Student Assistants:

- Ole Henrik Jahre
- Vegard Sjonfjell
- Stian Fredrikstad

Hallvard Norheim Bø – for making us aware of the lack of a FPU

References

1. Øvingsheftet
2. http://en.wikipedia.org/wiki/File:Waveforms.svg
3. http://www.dattalo.com/technical/theory/sinewave.html
4. http://www.phy.mtu.edu/~suits/notefreqs.html
5. http://www.cohortor.org/gstrings/Documentation.html
6. http://www.academy.cba.mit.edu/classes/MIT/863.10/people/brian.mayton/06.html#synthesis