TDT4258 - Ex3 Report

Jan Alexander S. Bremnes
Magnus Kirø

April 2011

Contents

1 Abstract

This exercise introduced us to writing programs for the STK1000, that would run on top of a Linux operating system. In addition, we were required to write a device driver that would control the LEDs and buttons. This was to be written as a kernel module that should be loaded dynamically while the kernel is running. An implementation of the game Pong was written and the driver was used to control the game.

2 Introduction

This exercise is the last of three in the course TDT4258 Mikrokontroller Systemdesign and it is a continuation of the two previous exercises, described in previous reports. The exercise was split in two parts.

The exercise had the following requirements:

Part 1: Write a driver enabling the use of LEDs and buttons on the STK1000. The driver is to be implemented as a kernel module. You are free to design the driver as you want it, but it has to provide minimum support for the functionality needed for the Pong game to work

Part 2: Implement the game Pong. Use /dev/fb0 directly to write to the LCD screen. Use /dev/dsp0 to send audio to the speakers. Use your own driver t oread the buttons on the STK1000. The driver should also be used to control the LEDs. These can be used to show information about the game state, or they could just blink and flash

Translated from the exercise description in Øvingsheftet[1]

In this exercise we created a version of pong running on our STK1000 development board. We used a self made device driver to control the LEDs and buttons. To access the display and speaker, we used the display and sound driver alrady installed with the Linux kernel. We expected to use quite a lot of time to finish this excercise and, as expected, used the las few days excessively.

[1]Øvingsheftet TDT4258 Datamaskingruppa IDI, 13.januar 2011

3 Description and Methodology (describe your work step by step)

3.1 STK1000 - Setup

For the stk1000 to be functional to our purpose we have to set it up accordingly. For us this mainly means to set some jumpers connect the LEDs and buttons to GPIO. The following jumpers need to be set:

- SW1: Set to SPI0
- SW2: Set to PS2A/MMCI/USART1
- SW3: Set to SSC0/PWM[0,1]/GCLK
- SW4: Set to GPIO
- SW5: Set to LCDC
- SW6: Set to MACB0/DMAC
- JP4: Set to EXT.DAC
- JP5: Set to EXT.DAC

All the jumpers can easily be found on Figure 1. We used PIOB for both buttons and LEDs. LEDs and buttons are then connected to the same GPIO port. The cables for LEDs and buttons should then be connected as follows:

- LED – J1
- Switch – J2

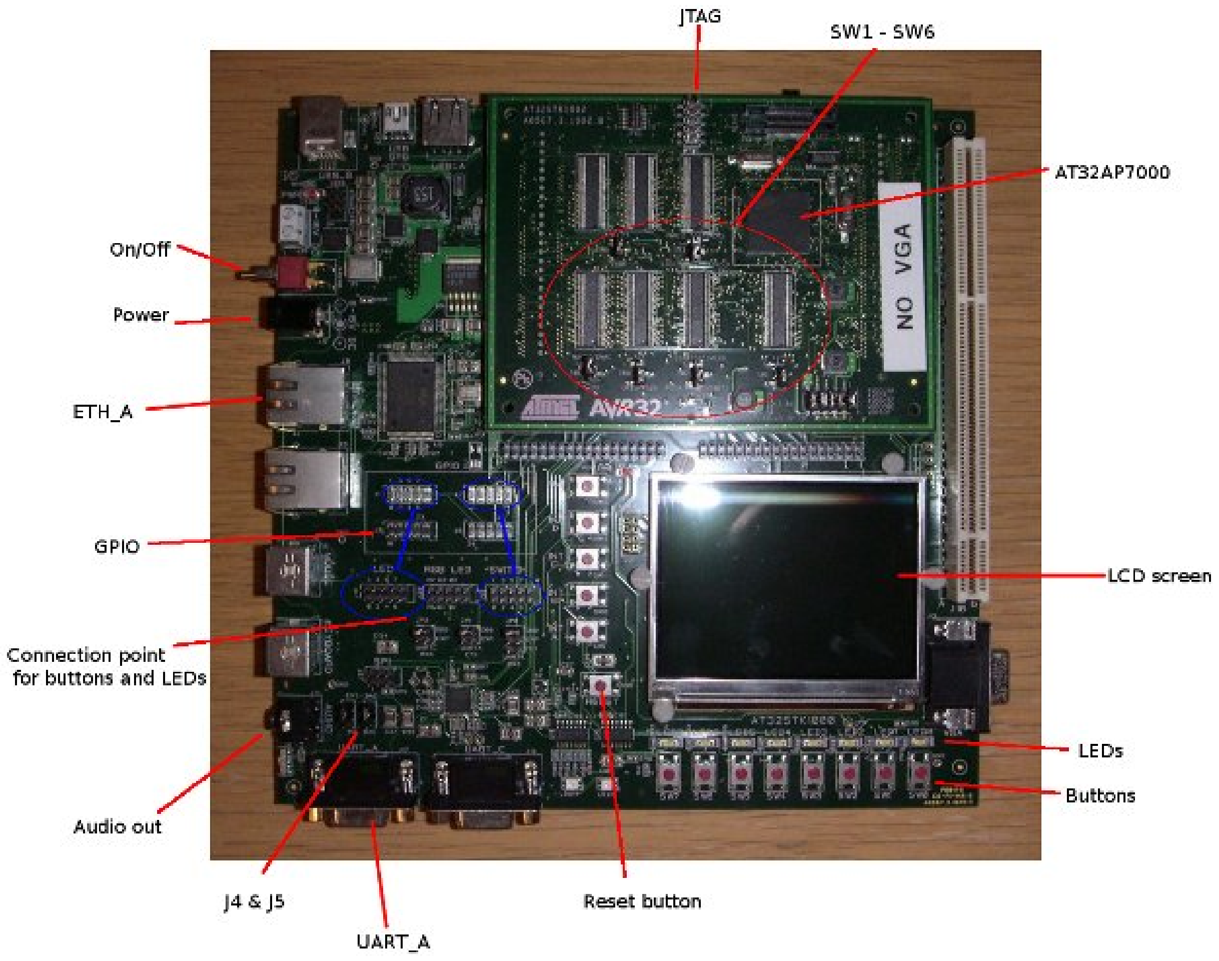

Figure 1: STK1000

In addition, the JTAGICE have to be connected both the computer and the STK1000, as it's needed to program the STK1000 with the bootloader file, which permits it to boot from the SD-card. An RS232-cabel needs to be connected between the STK1000's UART_A port, and the computers serial port. This will allow communication with the STK1000 through Minicom and similar programs. To make things easier when transferring programs to the STK1000, we also connected an ethernet cable to ETH_A.

3.2 Linux on to the STK1000

We had to set up the development card with Linux before we could insert driver modules and run programs on it. To do this we had to load the boot loader to the micro controller and boot Linux from the provided memory card. The memory card alrady had the Atmel avr32 linux installed on it. For this to work we had to do the following: The JTAGICE mkII had to be connected. First we erased the flash memory on the micro controller with the command: The stk1000 and the JTAGICE must be connected and turned on.

$$avr32programeraseu - boot.bin \tag{1}$$

Then we programed the flash memonry with the boot loader:

$$avr32programprogram - vf0,8Mbu - boot.bin \tag{2}$$

Restart the stk1000

3.3 Connecting and communicating with the STK1000

When the STK1000 is running Linux, it is possible to communicate with it via Serial Port and/or over the Internet. We chose mainly to communicate with it via Minicom, as the Telnet sessions had a tendency to crash after loading a driver.

Through the serial port:

- run ”minicom” in a terminal
- power up the stk1000 with the memory card inserted.
- Now we can control the micro controller with standard linux commands

Or through telnet:

- Turn on the stk1000
- Aquire the ip address to the stk1000 (run ifconfig)
- Run telnet with the ip address to target
- Now we can control the micro controller with standard linux commands

To transfer our program files to the STK1000, we used minicom to log into it, and then used the *wget* command to fetch the files over the Internet. We stored the files on our home folder on *folk.ntnu.no.* We wrote a shell script, *init.sh* to automate the task of downloading files, creating folders, loading the driver etc. We only have to manually input the ”major” and ”minor” number for the driver .

3.4 The driver

We started with compiling the supplied Linux kernel, and followed the instructions in *Linux Device Drivers*[2] on how to write a simple HelloWorld module. After writing it, and compiling it with our makefile, we downloaded it to our card, ran *insmod hello.ko* checked *lsmod* to see that it was loaded, ran *rmmod hello* to remove it, and checked *dmesg* to see that it had actually worked the way it was supposed to.

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");
static in hello_init(void)
{
printk(KERN_ALERT "Hello, world\n");
return 0;
}

static void hello_exit(void)
{
printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

The makefile we used, we got from the tutorial *AVR32743: 32-bit AVR AP7 Linux Kernel Module Application Example*[3] , and it's supplied files This is because we had some problems getting the supplied makefile to work, and the one from Atmel worked straight away. In order to compile a driver module, the makefile needs to specify the path to the compiled Linux kernel the driver is to be designed for, and needs to tell the compiler which architecture the compiled driver will run on. This is accomplished with the following lines in the makefile:

```
KDIR := ../linux-2.6.16.11-avr32-20060626
ARCH ?= avr32
CROSS_COMPILE ?= avr32-linux-
```

After this, we started to write the driver for the buttons and the leds. Here we ran into some problems, as it seemed that the the old, non-functioning driver was still somehow present in the system, so we didn't manage to get our driver to work, before we changed the name. Giving the driver a new name allowed it to be properly loaded and assigned a major and minor number. We followed the instructions in *Linux Device Drivers*, chapters 2,3 and 9. Since both the LEDs and the buttons are connected to PIOB, the driver needs access to the

[2]Jonathan Cobert, Allesandro Rubini, Greg Kroah-Hartman
http://lwn.net/Kernel/LDD3/

[3]http://www.atmel.com/dyn/products/product_docs.asp?category_id=163family_id=607subfamily_id=730part_id=3903

system registers PIOB is connected to. If this access is not granted, the driver will not be able to do anything. If access has been granted, then the driver needs methods for reading and writing to and from the specified registers. It is these two methods, read and write, that will be called by the Pong game in order to register a button press and turning on and off leds. Please see the supplied source code for the driver to see how this functionality is implemented. We tested the driver by writing random values to it, with *cat /dev/urandom > dev/drivername* causing the LEDs to flash on and off in random patterns. We also wrote a simple program to test the buttons.

At this point, we split the workload, Magnus would take a look at the sound driver and how to implement audio in our game, while Jan would start with the graphics and the game mechanics.

3.5 The LCD screen

Now that we knew that our driver was functional, we began looking at how to get the screen to work. We tested the screen by writing random values to */dev/fb0*. But getting our program so that it would write to the screen, was a bit more challenging.

Initially we thought about using bitmaps to create the graphics. But 16- and 24-bit bitmaps would not be displayed correctly, as the lcd-screen on the STK1000 uses 32-bit values for each pixel; 8 bits for Red, Green and Blue (RGB) and 8 unused bits. We tried a 32 bit bitmap, created by saving an image in 32-bit bitmap format in GIMP, but GIMP gave us only two different choices for the ordering of the bits in the image, and these would either cause the image to have a deep blue or red hue. Most likeley, the blue or red channel in the bitmap was written to the unused bytes of the pixel.

We thought about using 24-bit images and shift every pixel 8 bits to get the required format, but we quickly decided not to do this, as none of us had worked with bitmaps before and we figured it would quite possibly take us some amount of time to learn how to manipulate bitmaps with C. Instead, we decided to have simpler graphics and get a functioning game, and if we had time afterwards, we could experiment with images. We quickly figured out how to write to each seperate pixel on the screen, by mapping the screen to memory, and creating a one-dimensional char array which contains the values to be written to the screen.

The screen has a resolution of 320 x 240 pixels, each pixel is represented by four bytes, so that gives us a total number of 320 x 240 x 4 = 307,200 bytes. A char array of this size was therefore created, allowing us to write to a single pixel on the screen, by writing values between 0 and 255 to four consecutive elements in the array, and then writing this array to the screen. As mentioned, each pixel has four channels, RGB and one unused. By giving different values to different channeles, a multitude of colors can be generated. We decided to give our Pong game bright colors that were easy to make, so we opted for blue, yellow, green and purple. If one writes equal values to the green and blue channel, the pixel will have a yellow color, etc.

Below you can see how we wrote the blue line dividing the playing field, to the buffer

```
for(i = 0; i < SCREEN_HEIGHT; i++)
{
location = (middle * X_OFFSET) + (i * Y_OFFSET);
buffer[location + 1] = 255;
buffer[location + 2] = 0;
buffer[location + 3] = 0;
}
```

Location is the starting position of the pixel we want to write to. We use this as an index into the char array; the value at the index and the four values following it is the four bytes of the pixel. Since we want a blue line, we write nothing to the first byte (the unused one), 255 to the blue channel, and zero to the red and green channel.

The X_OFFSET is the value we have to increment with to get to the starting point of the next pixel, from left to right on the screen, which is 4. The Y_OFFSET is the value needed to move to the starting point of the next pixel, from the top of the screen and down. The Y_OFFSET is 1280, because the screen is 320 pixels wide, and we need to move a whole row down, so we need to move from pixel Z to pixel Z + 320, which is done by moving 320 x 4 bytes.

3.6 Audio

A part of the requirements for the exercise, were to use the sounds we made in Exercise 2. It was not specified how these sounds were to be implemented, and we decided to record them with Audacity, and convert them to .wav files, unsigned 8-bit PCM audio 8000hz samplerate. It would probably have been possible to use the code we wrote for the sound generation, and modify it to create a table of sample values for each sound effect, and include these tables in the Pong program. The tables would contain one period of each frequency in the sound effect, in the correct order. To play the sounds we would then iterate over the sample table, outputting each frequency period x times, where x equals the frequency. But we chose the easy way out, using .wav files.

The external dac on the STK1000 takes 8-bit sample values, have one channel, compared to two for the internal dac, and a default bitrate of 8000 hz. This bitrate can be changed with the following

```
#include <linux/soundcard.h>
int dsp_rate = 44100;
ioctl (fd_dsp, SOUND_PCM_WRITE_RATE, &dsp_rate);
```

We chose to keep the default samplerate, as you only gain an increase in sound quality from increasing the samplerate, and the sounds we were using doesn't require higher quality than what we get from 8 kHz The sounddriver in Linux is accessed by opening the file **/dev/dsp** This is done the following way in the program

```
int dev_s;

if((dev_s = open("/dev/dsp", O_RDWR)) == -1)
{
printf("[ERROR] Error opening SOUND driver /dev/dsp.\n");
exit(1);
}
```

We only want to write to the sounddriver, so it doesn't need to be read. The following function writes the selected .wav file to the sounddriver, one sample at a time

```
for(i = 0; i < soundList[currentSound].length; i++)
{
//writes a wav sample to the sound device.
write(dev_s, &soundList[currentSound].song[i], 1);
}
```

The sounddriver is closed by calling close(dev_s)

When the program starts, all the soundfiles are loaded into memory so they don't have to be decoded every time they're played. The combined file size of the wav-files is less than 20 kB, so the cost of storing them in memory is low.

We chose to use four different sound effects, so a ball hitting a wall has one, ball hitting paddle another, one for point scored, and one when the game is won. When the program closes, the wav-files are removed from memory, freeing the space they occupied.

3.7 The pong game

The pong game is a basic step by step routine in programming. Move, check, draw, repeat. Our implementation is very similar to this. We split the program in two threads; the game runs in the main thread, while the buttons are read in the spawned thread. The basic program flow is described in Figure 2. We start the program by moving into */home/pong* and then running *./mjp*. The needed drivers, game objects and game data are initiated, a new thread for reading the buttons is spawned, and the game is started.

The init_game(() function sets our game parameters so the game is ready to be played. Our pong() function is the main loop in the program. Here we move the ball and check if the score of the players. As described in the flow chart the, pong() function contains a loop. The loop is run until one player reaches five goals.

In the loop we use three functions. draw_screen(), move_ball() and score. The score function is really two functions, but they do the same thing. Wich is drawing the score of the designated player. The draw_screen() function updates the picture seen on the screen. The move_ball() method does most of the game logic. The logic as in where is the ball, and is it in a place where the other player scores a point or plays a sound.

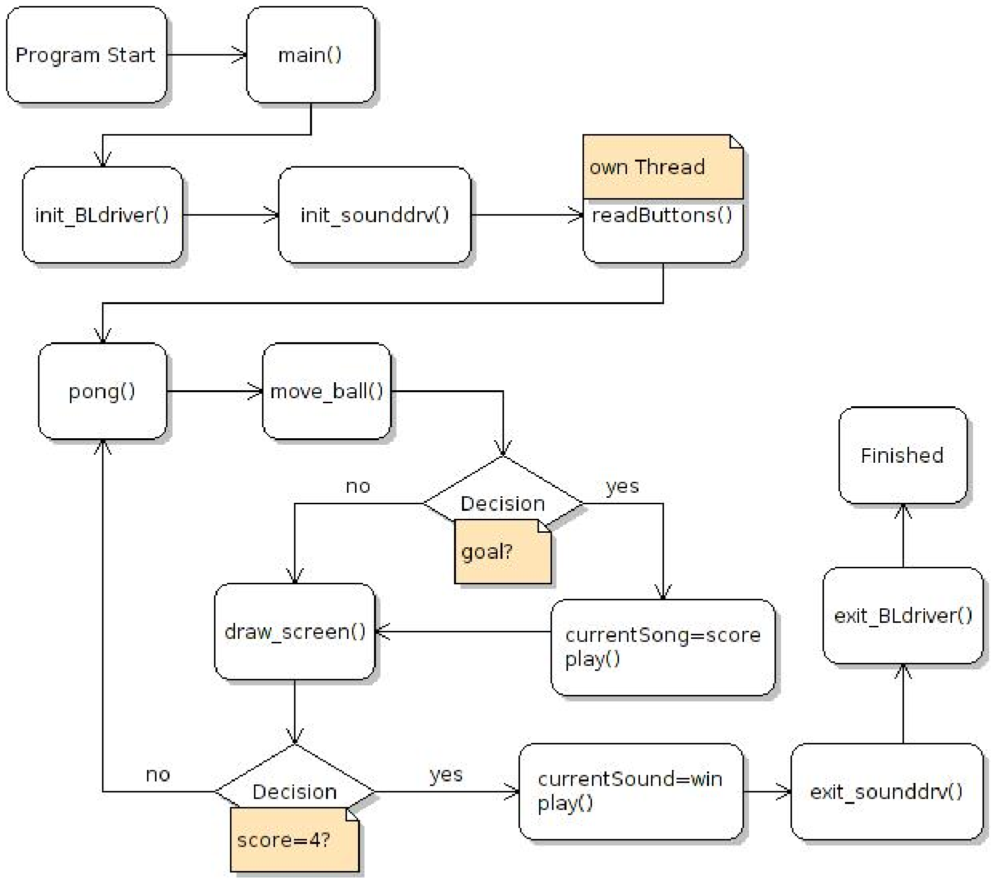

Figure 2: Flow chart

To create the graphics, we made an ”object”-struct, and the paddles and ball were declared as objects. This struct holds essential information about the objects on the screen; width, height, position and size. The objects are represented graphically by creating an array, equal in size to the object.size variable. These arrays hold the addresses of the pixels that would make up the object. When the screen is repainted, the program iterates over each of these arrays, making the necessary changes to the pixels whose addresses is read. Please see the source code to see exactly how this works.

Information about the game state, such as game speed, player scores, etc. is stored in a game_data struct. Figure 3 shows a screenshot from the game.

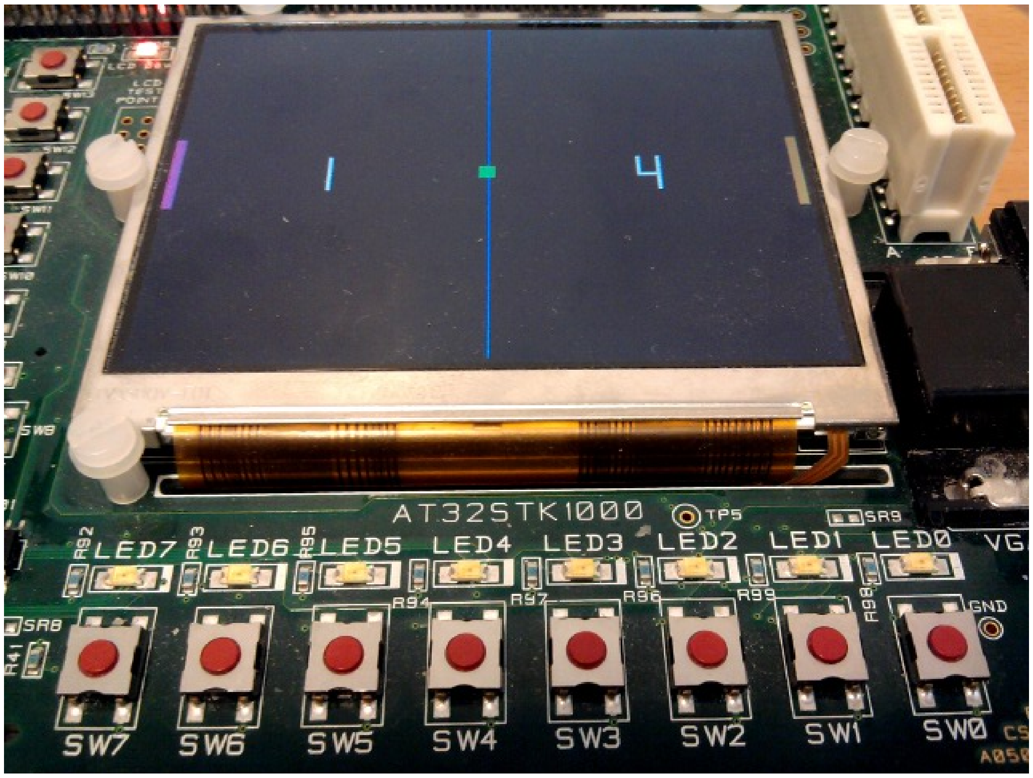

Figure 3: Pong!

There are six objects drawn on the screen; two paddles, a ball, two score counters and a dividing line. Player 1 is the left paddle, player 2 is on the right. Player 1 has one point, player 2 four points. SW7 and SW6 will move player 1 up and down, and SW1 and SW0 will move player 2. If none of the players have scored a point within a set time limit, the speed of the ball increases. The speed is reset when a point is scored. The leds will light up from right to left, indicating what the current speed level is.

4 Results and Tests

4.1 Results

We managed to write a device driver for Linux, that would let us read and write to the leds and buttons through a C program. This driver allows us to control the paddles in the Pong game, and light the leds according to different game states. We also managed to create something similar to Pong.

It is playable and bears a resemblance to Pong, but it has a few bugs that require us to call it a work in progress. The main issue is that the paddles flicker when they move downwards, this is caused by the way we repaint the screen. When a paddle is moved, it's previous location is stored in a array of identical size to the the one containing the paddle. When the pixel addresses stored in this duplicate array, is written to the buffer, the pixels get the value 0, which means the pixel will show as black. When the screen is written from the top to the bottom, the previous location of the paddle will be blacked out before the new one is written.

This does not happen when moving the paddles upwards, because then the new location is written before the old one is blacked out. A possible solution would be instead of making a complete blank copy of the paddle, we could disable only the amount of pixels that the paddle moved. Regrettably, we did not have time to attempt this possible solution.

Another bug, is that sometimes the paddle won't move until a point has been scored, or it is first moved in the opposite direction. We have not been able to figure out if this is due to an error in our program, or hardware problems. We have tested the game on several STK1000s in the lab, and some of them hangs more than others, so it is possibly caused by a hardware problem, but we can't say for certain.

On a different note, the buttons work, usually, and the LEDs work, and we have a functioning Pong game, with a few bugs that can be annoying, but it is playable. The paddles move, and the ball bounces around when hitting the paddles and walls, speed increases when no points are scored, points are kept track of and the sound effects work as they're supposed to.

4.2 Tests

Some of the following has already been mentioned elsewhere in the report, but we repeat them here.

To test if we were able to write a driver module, and load and unload it while the kernel is running, we wrote a simple HelloWorld.ko module. This module was programmed to print KERN_INFO messages when loaded and unloaded. We executed the command *dmesg* after unloading it to check if the messages were written.

We wrote a test driver for the buttons and LEDs driver (BLdriver), without any read and write methods, to see if we were successful in allocating PIOB.

We implemented write- and read methods int the driver, and tested that we could write to it with cat /dev/urandom > /dev/BLdriver, which caused the LEDs to flash in a random pattern

We tested the screen with cat /dev/urandom > /dev/fb0, which resulted in the screen being filled with random noise and colors. Then we wrote a simple program to write zero to all pixels, causing a black screen. We filled the screen with random values again and checked to see if we could clear it with the program

We extended the program to write to a single pixel on the screen, when we managed this, we tried to draw rectangles and attempted to get them to move around on the screen. When this was done, we could start creating paddles and write methods for getting them to move up and down. A ball was added and given movement.

When we had a ball moving around, we added functions for it to bounce when it hit a wall or paddle, and be reset when it hit the right or left wall. Since the driver had not been integrated with the pong game yet, to test that the ball would bounce of the paddles, we enlarged the paddles height so that they covered the whole sides of the screen

Sounds were added and the driver integrated with the game, so that we now could test it by playing the game

5 Evaluation of assignment

As a whole the assignment was very educational. It taught us very much about microcontrollers and low level programming. Now we have also gained some experience with C, and have grown to like the language. The assignment had many different components to it, which made it a diverse task. On the other hand, that made it a very time consuming assignment.

We managed to complete the programming part of the assignment within the given time, but in order to complete the report we had to deliver a bit late. Unfortunately for us, this assignment collided with a four week joint-project across the other courses we attend, and we had many late nights because of that. But even so, it has been a very fun, exciting and eduational exercise, as well as a very interesting course which we will not hesitate to recommend to others.

6 Future work

A lot of work remains before our Pong game is as good as we would want it to be. We have already mentioned some of the bugs that need to be ironed out, and there's probably some that we haven't discovered yet. All graphics in the game, has been drawn ”by hand”, and better graphics can be achieved by using bitmap images. The ease of using wav-files for audio, allows for easy implementation of other, better sound effects.

A start-up screen can be added, perhaps with instructions and choice of theme for the game. The game play can also be improved by choosing different difficulties, for example more than one ball, increased speed, power ups etc. We could also add functionality for the game to restart by pressing a button after a game is won, instead of having to restart the entire game. None of this, however, was required by the exercise, and one could go on and on, adding feature after feature, so we choose to stop here.

7 Conclusion

The exercise was enjoyable and challenging. It was a lot of work to get it done, and we almost managed to meet the deadline. What we have ended up with, is working Pong-like game, but still a work in progress. We could have used a lot more time on it, if more time had been available. Everything concidered we are satisfied with what we accomplished.

Our pong is playable, the bugs that are there can be annoying, but doesn't ruin the gameplay. Sound effects are played, the buttons and LEDs are used and we have made our first device driver for Linux. All in all a very educational experience. We made a very basic implementation of the world famous game Pong, and had great fun while doing it.

8 Acknowledgements

Student Assistants:

- Ole Henrik Jahre
- Vegard Sjonfjell
- Stian Fredrikstad