TDT4258 Microcontroller System Design
Assignment 3

Einar Uvsløkk & Simen Natvig

May 4, 2012

Abstract

This report describes our work with assignment 3, in TDT4258 Microcontroller System Design. The assignment was to write a Linux device driver for buttons and LED diodes on a STK1000 development board, as well as writing a computer game, in C, making use of the Linux device driver. We followed the recommendations for the assignment, found in the compendium. This resulted in ...

1 Introduction

The task for this assignment was two fold. Part 1 was to write a Linux device driver for the buttons and LED diodes on the STK1000 board. Part 2 was to make a computer game, *The Scorched Land Defense*. All program code was to be written in C, and er were encouraged to reuse the code from exercise 2. The assignment also included Linux to be run on the STK1000 board.

To compile our device driver we used the standard Linux kernel toolchain. To compile and link our game implementation we used a Linux toolchain customized for the AVR32 microcontrollers, provided by Atmel.

In the following sections we will present how we worked towards the final solution (section 2), how we tested our implementation and what test-results we got (section 3), evaluate our work for the assignment (section 4), and finally present our conclusion (section 5).

2 Description

As proposed in the assignment text [1] we started by setting up the development board and checking that the development tools was working correctly. We then proceeded with writing a simple kernel module, following the guidelines in *Linux device drivers* [3], before we started our work on the game implementation.

In the following subsections we describe the setup and configuration of the hardware we was working with (section 2.1), and the implementation of our solution (subsection 2.2).

2.1 Setup

In order to start on our implementation, we first needed to make sure the STK1000 development board was correctly set up. We first checked that the

required jumpers was set according to the assignment description. We needed to set:

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

In addition, we needed to connect the JTAGICE to the STK1000 and the PC in order to be able to program the micro controller from the PC. We also connected the STK1000 board (UART A) and the PC by an RS232-cable, to be able to communicate with the board over the COM-port, using `minicom`. In order to use the buttons and the LED-diodes we connected their connectors to the GPIO port B, pin 0-7 and pin 8-15, with one flat cable each, respectively. To be able to listen to our sound effects, while developing our implementation we also connected headphones to the audio contact.

2.2 Implementation

In this section we will describe the implementation of our device driver (section 2.2.1) and our version of *The Scorched Land Defense* (section 2.2.2).

2.2.1 Part I: The device driver

We started out by writing a simple device driver, which only used the `printk` kernel function call to print messages to the console. This was fairly easy, and we started to implement functionality towards the STK1000 development board. At this stage we needed to decide whether to write one device driver that handled both buttons and LED diodes, or to write two seperate device drivers (one for handling the buttons and one for the LED diodes). Because we only needed to know which buttons was pressed for the game, we only needed to *read* the buttons. Likewise it was sufficient, for the game, to *write* to the LED diodes, in order to turn them on or off. We therefore decided to write one device driver, handling both cases. The device driver was impemented as a character device.

Writing LED diodes

Reading buttons

2.2.2 Part II: The game

The main goal of this exercise was to create a game named "The Scorched Land Defence". The requirements for this game was vaguely stated if at all in the compendium. Therefore we took a few liberties when making the game, in an attempt to make it more challenging and fun. The premise of the game described in the exercise lectures and the compendium is for one player to control a character from one corner to the opposite corner. This is done while a cannon placed in the corner opposite from the characters starting position shoots at the character. To make matters worse for the character, each shot that misses creates a fire in the region hit. This makes it fairly simple for the cannon to block the different regions leading in to the cannon with fire, blocking the character from reaching the cannon.

Since we thought that the game described made it too easy for the cannon to win we decided early to try and make some adjustments to make the game a bit more interesting. The first thing we decided to change was to make the cannon shots into "character-seeking" missiles. When doing this we also decided that making the game turn-based was probably a good idea, both for implementation purposes and the fact that the buttons on the development board is not fit for frenetic pressing when almost hit by a dangerous objects.

Game logic The implementation of the game consists of initialization and a while loop. Initialization sets up the driver, the graphics, the sound and the game state. After this the while loop is entered and the game itself starts. While the game runs it will stay in one of the five game states for each while loop iteration. The different game states are:

1. Intro state
2. Player state
3. Cannon state
4. Player victory state
5. Cannon victory state

The Intro state starts the intro music and waits for a single button press on the 7^{th} to set the game in the Player state and sets all variables from the game to the start position.

The Player state is the one of the two main states of the game. This state waits for a button press on one of the four lower buttons. When one of these are pressed the player moves in the corresponding direction, each of the missiles take one move. When all the movement is done the missiles update their tracking of the character and collision detection between the missiles themselves and the player is done. If there is a collision at all the explosion sound is played and if the character was involved he is set to inactive which is a losing state for him. Finally the graphics are updated and the victory conditions are checked, in which case the states will be set to the corresponding victory state. If neither of the victory states are set the default next state is used which is the Cannon state.

The Cannon state is the other main game state. In the Player state there will only be one move, but in the Cannon state the targeting reticule can be moved over the whole map before deciding on where to fire a missile. After the missile is fired the map is redrawn and the game state is changed back to the Player state.

The Victory states shows a screen saying who has won and then waits for input from the 7^{th} button which sets the state to the Intro state. This takes you to the intro screen and makes it possible to play the game again.

Graphics To access the LCD screen on the STK1000 board, we used the *framebuffer device*. This device, `/dev/fb0`, represents the LCD screens graphic memory. To write pixels to the screen, we decided to use `mmap` to map the driver to an array in the memory. This way we could write pixels to the screen by writing values in a C-array. Listing 2 shows how this was done.

```
int fb_fd;  /* Framebuffer file descriptor */
char *fbp;  /* Pointer to the framebuffer */

fb_fd = open ("/dev/fb0", O_RDWR);
fbp = (char *) mmap (0, SCREEN_SIZE,
                     PROT_READ | PROT_WRITE,
                     MAP_SHARED, fb_fd, 0);

/* Display a black LCD screen */
memset (fbp, 0, SCREEN_SIZE);
```

Listing 1: Mapping the framebuffer device to an array in memory

We created separate arrays for each graphics item we needed, and used a for loop to copy a given array to the framebuffer array. To further improve the performance, instead of using for-loops to copy arrays slot by slot, we switched to using `memcpy`. This way we just copied the whole content of a given graphics array, to the desired location in the framebuffer array.

The STK1000 board uses a 24-bit LCD data bus, and, hence, the color coding is RGB. In order to get the pixel data from our images, we made use of *Simple DirectMedia Layer* (SDL)[2], and its image subsystem `SDL_image`. This way we was able to easily retrive the RGBA values for each pixel in our graphics, and process them separatly to suit our needs. Figure **??** illustrates how we retrived the RGBA values from a given pixel.

```
/**
 * Get the different RGBA channel values from the pixel data.
 *
 * \param pixel The pixel data.
 * \param *r Pointer to store the red value.
 * \param *g Pointer to store the green value.
 * \param *b Pointer to store the blue value.
 * \param *a Pointer to store the alpha value.
 */
void get_rgba (int pixel, int *r, int *g, int *b, int *a)
{
        *b = (pixel >>  8) & 0xFF;
```

```
        *g = (pixel >> 16) & 0xFF;
        *r = (pixel >> 24) & 0xFF;
        *a = pixel & 0xFF;
}
```

Listing 2: Retrieving RGBA values from a pixel. label

We also implemented "support" for alpha channels, which was achieved by marking all pixels with a given level of transparacy, to not be included in our internal graphics buffer structs. This enabled us to draw tile sprites on top of the game map.

Sound To enable sound effects in the game, we used the digital sampling and recording device, `/dev/dsp`. Writing to this device accesses the D/A converter to produce sound, which is very similiar to how we produced sound with the internal ABDAC on the STK1000 board, in assignment 2. This meant that we were able to reuse most of the sound related code from assignment 2, for this assignment. The only modification we did was to reduce the sample size from 16 bits to 8 bits. In addition to this we had to write a new routine that built a suitable sample buffer, that we could write to the sound device.

3 Results

For our game implementation we put down a list of expected behaviour for the different game units. Some of these behaviours are dependent on the current game state. Table 1 describes the various expected behaviour and their corrosponding game state dependencies.

4 Evaluation

5 Conclusion

One of the biggest challenges in this assignment was to get the actual toolchain set up and working.

The game could be improved in a number of ways. We could have utilized a the Simple DirectMedia Layer (SDL) library at a greater degree. This would probably make the implementation easier and more straight forward. Still our choice, not to use it more than we did, is justified by the learning outcome from handling both sound and graphics *the hard way*.

Table 1: Expected behaviour

Id	**Game State**	**Action**	**Expected behaviour**
EB.0	Intro	Button 7 is pressed	Start game.
EB.1	Player	Button 0 is pressed	Player moves to the right if, not blocked by end of map or fire.
EB.2	Player	Button 1 is pressed	Player moves upwards, if not blocked by end of map or fire.
EB.3	Player	Button 2 is pressed	Player moves downwards, if not blocked by end of map or fire.
EB.4	Player	Button 3 is pressed	Player moves to the left, if not blocked by end of map or fire.
EB.5	Cannon	Button 0 is pressed	Cannon reticule moves to the right, if not blocked by end of map.
EB.6	Cannon	Button 1 is pressed	Cannon reticule moves upwards, if not blocked by end of map.
EB.7	Cannon	Button 2 is pressed	Cannon reticule moves downwards, if not blocked by end of map.
EB.8	Cannon	Button 3 is pressed	Cannon reticule moves to the left, if not blocked by end of map.
EB.9	Cannon	Button 4 is pressed	Start countdown for missile at the given tile.
EB.10	Victory	Button 7 is pressed	Restart game, show intro screen.

References

[1] IDI CARD Group. Lab Assignments in TDT4258 Microcontroller System Design, 2011.

[2] Simple DirectMedia Layer. `http://www.libsdl.org/`.

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