TDT4258 Microcontrollers System Design Exercise 3

Implementing a Multimedia Game on a Microcontroller Supported by an Operating System

Vegard Edvardsen
Jean Niklas L'orange
Caroline Sæhle

May 4, 2012

Abstract

An operating system will provide an abstraction such that receiving and sending data is made easy for programmers. However, to have such an abstraction for a specific piece of hardware, one needs a device driver. This report shows how to implement such a character device driver for Linux on the STK1000 board. The driver will make it possible to read which buttons are pressed on the board as well as writing which LEDs should be on and off. The report will also explain how to use device drivers through an implementation of the game Scorched Land in C. The game will use the driver implemented to read buttons and turn on and off the LEDs, a sound driver to play different sounds, and a frame buffer device driver to print the game on to the screen on the board. The report will also include problems we believe others may make, and solutions to those problems.

Contents

1 Introduction

The objective of this task is that the group members will get a better understanding in how device drivers in Linux work, how to implement a character device driver in C for the buttons and LEDs on the STK1000 board and experience on how to use device drivers in a relatively large C program by making a game named *Scorched Land*. The task given to us in the course Microcontroller System Design (TDT4258) to learn these things was to
"*Make a driver for the use of buttons and LEDs on the STK1000. It should be implemented as a kernel module. [...] Complete the game. Use* `/dev/fb0` *directly for writing to LCD screen. Use* `/dev/dsp` *for producing the sound. Use your own driver for reading the status of the buttons on STK1000. Use also your own driver for the control of the LED diodes. These can be used, for example, to show how many lives a player has left or some other status information about the game. Or you can just blink in some nice repetitive way.*"[1]

As the specifications on how the driver exactly should work were left unspecified, we decided to let the logic be as simple as possible: Reading a char from the `/dev/stkboard` module gives information about which buttons are pushed, and writing a char to the device will set the LEDs in the configuration specified. The proper specification is written in 2.2.

The game was developed by splitting the work in three: One to work on integrating the device drivers with the GUI and the game, one to work on creating and finding sounds and graphics as well as implementing code to properly load those resouces, and one to work on the game logic. By designing an API for image, sounds and game logic, the code became very modular and it made testing and debugging each module on its own an easy task.

Apart from the requirement that the game should use the driver we made, `/dev/fb0` and `/dev/dsp`, the game logic was not specified. The game logic was thus developed from the images in the compendium[1] and what we already knew about the military strategy known as scorched earth: The game designed is a turn-based game where a soldier tries to get up to a tank, and the tank tries to block the soldier's way by firing its gun to scorch the earth or to hit the soldier. If there's no way for the soldier to get up to the tank or the tank manages to hit the soldier, the tank wins. Otherwise, if the soldier manages to get up to the tank, the soldier wins. A more detailed specification is listed in 2.3.6.

2 Description and Methodology

Roughly, the assignment consisted of three parts: *uploading Linux* to the microcontroller board, *developing a kernel driver* for the LEDs and buttons, and *developing a game* utilizing these.

Developing the game was the most time-consuming part of the assignment. This part consisted of several components that had to be developed. In addition to

the game logic and the user interface, the game needed support for rendering to the screen, playing sounds, importing images, reading hardware buttons and controlling LEDs.

2.1 Installing Linux

To get Linux to run on the microcontroller, a boot loader is needed in the Flash memory as the first stage of the booting process. To upload this boot loader, `avr32-program` was used as in the previous assignments.

The boot loader will look for a Linux kernel on the board's memory card. The memory card thus needs to contain a valid file system, as well as the root file system for the OS to boot into. This was all supplied in an image file that was ready to be written to the memory card. Writing the image file to the card was done using `dd` on the lab computer.

2.2 Developing the kernel driver

Our kernel driver follows the required steps listed in the compendium [1]. The skeleton code was extended to perform the following initialization steps:

- Allocate a device number for a character device with `alloc_chrdev_region`.
- Request exclusive access to the I/O memory areas with `request_region`.
- Set I/O registers to initialize LED output.
- Set I/O registers to initialize button input.
- Register the character device with `cdev_add`.

As the PIO C port is reserved for other uses in this exercise, we needed to use the higher bits of the PIO B port for the LEDs. However, the mapping between bit positions in this I/O register and the physical pins is not one-to-one. Thus, we needed to convert the physical pins to register bits with some extra logic. The code for this is shown in figure 1.

Upon read requests to the device, the driver returns a single byte representing the current data values on the button pins. For instance would byte value 129 (`1000 0001`) mean that buttons 7 and 0 are pushed.

Upon write requests, the driver reads a single byte and updates the eight LED pins accordingly. For example, if you write the byte value 66 (`0100 0010`) to the device, then LED 6 and 1 will be lit, while the others will be turned off if they aren't already off. Notice that there's no way to read the LEDs from the

```
28  j = i;
29  switch (i) {
30      case 3:
31      case 4:
32      case 5:
33      case 6: j += 2; break;
34      case 7: j = 22; break;
35  }
```

Figure 1: Logic in `driver/led.c` for I/O pin mapping. `i` is the requested LED, `j` is the corresponding I/O pin.

device, as this was not a specification requirement. To do that in a program, we will have to keep track of the current configuration of the LEDs ourselves.

The "protocol" described for our device driver is thus very simple – a single byte is the unit for both read and write operations. This removes the need for any buffering in the driver, as well as making the code running in kernel mode as simple as possible. The flip side is that the application code requires more logic to handle the devices.

2.3 Developing the game

As mentioned in the introduction, the game consists of several components. The game logic and the hardware interfacing (screen, sound, buttons and LEDs) were developed in parallel. Afterwards, the user interface logic connecting the two halves was developed. This has led to a code base with a clean seperation of the different parts.

Figure 2: User interface: The game intro screen.

2.3.1 Rendering to the screen

Support for rendering to the screen is implemented in `screen.c` and `screen.h`. An initialization function opens the framebuffer device at `/dev/fb0` and uses memory mapping to allow this device to be used as a regular C array. An in-memory array of the same size is allocated to function as a back buffer. This is to implement double buffering, as the image will flicker otherwise.

A custom C structure `color_t` is defined, that abstracts away the makeup of the pixels in the framebuffer array as colors with red, green and blue components. To plot a pixel, the macro `screen_put` is used (see figure 3). This is implemented as a macro for efficiency reasons.

```
74  #define screen_put(x, y, color) (screen_buffer[(y) *
        SCREEN_WIDTH + (x)] = (color))
```

Figure 3: Pixel plotting macro in `game/screen.c`.

The function `screen_show_buffer` shows the back buffer on the screen. In addition, there are several other `screen_[...]` functions that mainly deal with drawing shapes like filled rectangles and lines.

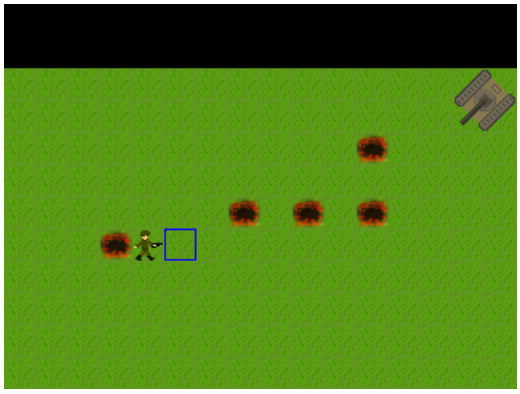

Figure 4: User interface: The soldier's turn to move.

2.3.2 Importing image files for graphics

For our graphics format, we chose uncompressed 24-bit *Truevision TGA* format (also known as TARGA format) for its ease of implementation and because it is not heavily patented. Truevision TGA is a raster graphics file format with options for 8 to 32 bits of precision per pixel, with up to 24 bits of RGB color and an additional 8 bit alpha channel, and it offers the option of lossless RLE compression. Truevision TGA files have an 18 byte header, of which the last 10 bytes are the image specification. [2]

We extract the image dimensions and format from this header to find the exact location and size of the image data, and discard the other information. We then render the raw image data directly, pixel for pixel, as can be seen in `image.c`, with one exception. We wished to make partially transparent images to make overlaying seamless (as in the case of the little soldier and the tank), but exporting 32-bit Truevision TGA images with an 8-bit alpha channel proved difficult with the available tools. Instead, we substituted a bright purple colour not used elsewhere (printer's magenta, hexadecimal `FF00FF`) for the transparency, and ignore this during rendering, as can be seen in the `image_draw` function in `image.c`.

We encountered one problem: Truevision TGA format is little-endian, and Linux on AVR32 is big-endian. Thankfully, the scope of the problem was limited to the header of the TGA file, as the image data is given in individual bytes. In our case, the only multiple-byte information we need is the image height and width, both given in two bytes. We solve this by swapping the upper eight and lower eight bits through bit-shifting, as can be seen in the `image_load` function in `image.c`

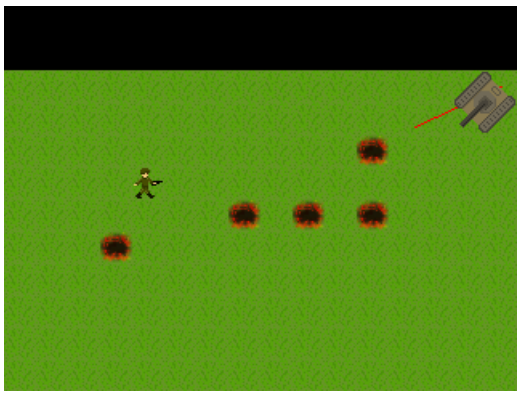

Figure 5: User interface: The tank's turn to select shooting angle.

2.3.3 Playing sound effects

We decided that a sound clip with actual music would sound better than our rendition of "Itsy Bitsy Spider" from the previous exercise, and decided on the beginning of the Walkürenritt from Richard Wagner's opera Der Ring des Nibelungen, as it sounds more dramatic. The source file for the sound clip was found in the archives at Textfiles.com, and converted to a suitable format using Audacity. Audacity was also used to generate sounds where we decided not to use sound clips.

To make playback simple, we exported the sounds in raw sample format with parameters that matched those of the `/dev/dsp` device. To play sounds, we then only need to write the samples from the sound files directly into `/dev/dsp`. This is implemented in `sound.c` and `sound.h`. As inconsistencies in sampling

rates became apparent during our test runs, we edited the sound clips to provide consistent sampling rates.

Writing samples to `/dev/dsp` is performed in a continuous loop. The write requests to the sound device will block until it's time for new samples. However, this sample processing loop has to be disconnected from the game's main loop. Thus, the sound loop runs in its own thread. This functionality is provided by the `pthreads` library.

2.3.4 Reading buttons

Reading from the hardware buttons is implemented in `button.c` and `button.h`. Initialization is simply to open the character device at `/dev/stkboard`, which is the driver implemented in the previous part of this exercise. The function `btn_read` returns a character containing the current state of all of the eight buttons. This is done by reading a character from `/dev/stkboard`.

The function `btn_is_pushed` implements further functionality to correct unwanted button behaviour. Firstly, it reads the button state *twice*, with a sleep loop inbetween. This is to alleviate the bouncing effects described in the first exercise.

Secondly, the function only returns true for a button *once* per button push. That is, once a button is pushed down and this is returned from the function, the function will no longer repeat this as long as the button is kept pushed down. This is achieved by maintaining a `btn_ignore` variable that masks out button pushes already returned.

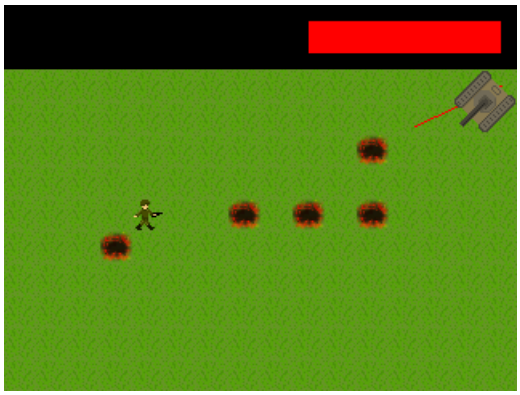

Figure 6: User interface: The tank's turn to select shooting strength.

2.3.5 Controlling LEDs

`led.c` and `led.h` implement LED control. As with the button support, the initialization step is to open the `/dev/stkboard` device. The code maintains a

global variable containing the state of all the LEDs. Updates to the LED state is written to the device driver by `led_update`. The rest of the code is re-used from the previous exercise.

2.3.6 Implementing the game logic

The game logic itself is implemented in `scorched_land.c` and the API created with documentation is implemented in `scorched_land.h`. By separating game logic from the STK1000-specific implementation, one can implement the game on another platform without the need to reimplement the game logic. This made it possible to make a simple terminal version of the game for Linux to debug the game logic, without the need of a GUI on the STK1000 board.

As there were no specifications on how the game logic should work, we decided to make a game logic specification before implementing the game. We ended up with the following specifications on how the game should work:

- The game is grid-based where the soldier uses 1x1 space, and begins initially in the lower left corner.
- The tank uses 2x2 space and is placed in the top right corner.
- The soldier is able to move in the directions north, west, up and down if and only if the tile the soldier wants to move to is within the grid size and the tile is not scorched land.
- The tank is stationary, and can shoot a gun which will scorch the land. The player gives an angle and a strength to the shot, and the game will calculate where it hits, if it hits the grid. It is possible to overshoot the grid.
- The soldier wins if he gets to a tile which is occupied by the tank.
- The tank wins if he hits the soldier, or if the hit makes the tank impossible to reach from the soldier's location.

We decided to omit details such as grid size and whether the game is turn based or real-time in the specifications, and instead make it possible to change these choices at compile-time.

After we had decided on specification details, we implemented the API in the `scorched_land.h` file, with explanation on what parameters each function takes, what the function does, and what you can expect from the return parameter. When we finished the API, we started implementing the game in the `scorched_land.c` file.

The game is initialized by running the `game_init` function. This will reset all variables related to the game in case these values have not been previously initialized, or if one has already played a game and wants to start over. To

let the tank shoot a bullet, one calls the function `game_shoot_bullet` with two integers: One denoting the direction in degrees the bullet is supposed to go, the second denoting how much power to use - which determines how far the bullet goes.

```
42      double radians = ( direction ) * ( M_PI / 180 ),
43             x_multiplier = cos ( radians ),
44             y_multiplier = sin ( radians ),
45             scaled_strength = strength *
    GAME_STRENGTH_SCALE;
46
47      int    x = (int) ( x_multiplier * scaled_strength ),
48             y = (int) ( y_multiplier * scaled_strength );
49      /* Flip the x-position. */
50      x = GAME_WIDTH - 1 - x;
```

Figure 7: Logic in `game/scorched_land.c` to calculate which tile to hit.

Calculating where the bullet lands is done by the code snippet shown in figure 7. We first convert the degrees to radians. We then find out how far on the x- and y-axis the shot goes by calculating the sine and cosine of the radians we just calculated. This is in turn multiplied by a previously calculated factor `GAME_STRENGTH_SCALE` and the strength given as parameter, and then rounded down to an integer. Finally, we flip the x-position to let the shot go from the correct corner.

`GAME_STRENGTH_SCALE` is calculated by taking the longest total distance a shot can go divided by `GAME_MAX_STRENGTH`, the maximal power one can shoot with. The longest total distance a shot can go is $\sqrt{W^2 + H^2}$ where W =`GAME_WIDTH` and H = `GAME_HEIGHT` length. By multiplying by this factor, we ensure that the tank is able to hit any tile on the game regardless of maximal strength and game dimensions.

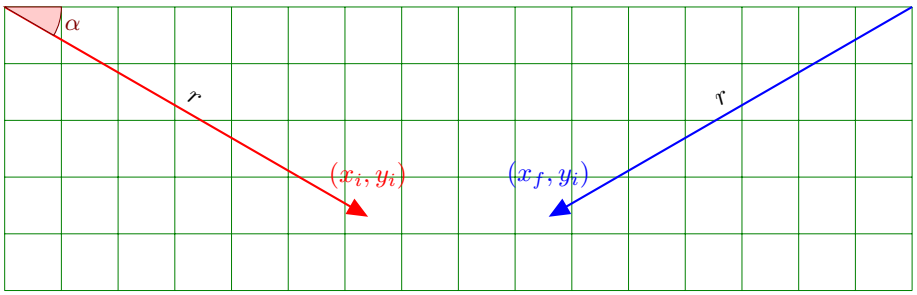

Figure 8: Calculating the landing position of a gun shot. The red arrow represents the calculated values before flipping the x-position, and the blue arrow represents the calculated values afterwards.

The function `game_move_player` will move the soldier with the specified direction you want the soldier to go to, if possible. The function will check whether

the position is ok to move to, and if so, will move the soldier from its current position to its next. Finally, the functions returns a reasonable response which gives the GUI the option to properly give error messages if desired.

After the initial implementation, a terminal version of the game was created to test the functionality and check that everything worked as intended. The terminal version can be compiled and run by running the command `make test && test/sc_land` in the `game` folder, or through `make && ./sc_land` in the `game/test` folder.

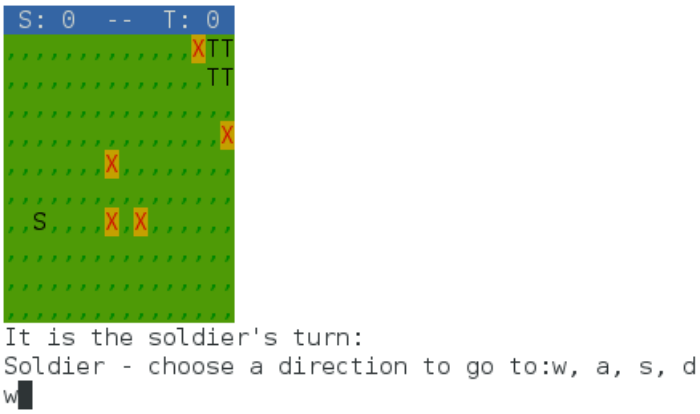

Figure 9: The terminal implementation (`sc_land`) of the game.

By implementing a terminal version, we could test the implementation much faster and debug with the "GNU Debugger" available, which provides much better debugging support than the STK1000 board does. From the implementaion, we found and solved bugs in regards to correctly placing scorched land tiles from a bullet shot. We also found out that we forgot to flip the x-position, and implemented the code needed to flip it.

2.3.7 Implementing the user interface

With all the low level support functions in place, as well as the game logic itself, a user interface tying all this together was needed. The user interface is implemented in the files `ui.c` and `ui.h`.

The user interface is a state machine that changes between the following states:

1. Intro screen.
2. Gameplay, soldier's turn to move.
3. Gameplay, tank's turn to select shooting angle.
4. Gameplay, tank's turn to select shooting strength.
5. Game over screen.

The user interface runs in an infinite loop, rendering to the screen, reading input, controlling the sound and updating the game state.

Impressions of the user interface are shown in figures 2, 4, 5, 6 and 10 throughout the report.

Figure 10: User interface: The game over screen.

3 Results and tests

This assignment did not lend itself well to simple testing, due to the complexity of the finished product. Instead, we had to resort to visual and auditory testing to check that image rendering and sound rendering were accurate, as well as component testing of individual code components.

3.1 Buttons and LEDs

As the button and LED implementation from assignments 1 and 2 was more or less complete, we tested it first, as it only needed minor modifications to it to work for this assingment. We tested LED functionality by using the following commands: `echo -ne '\x55' > /dev/stkboard` and
`echo -ne '\xaa' > /dev/stkboard`, expecting the even-numbered and odd-numbered LEDs to light up, respectively as we did this. This test was successful. We then tested button reading functionality by using `hd /dev/stkboard` and reading the results in the terminal as we pressed buttons. We expected to find the hexadecimal representation of the bitmask of the buttons pressed. This test was successful.

3.2 Graphics

The graphics part of the assignment did not have any formal testing structure, but rather consisted of successive experiments in making it work. We first tried

filling the whole screen with one color, with success. After this, we experimented with drawing lines and boxes on the screen, subsequently implementing movement. Both experiments were successful. Visual testing showed that the rendering worked, but the screen flickered badly. Double-buffering solved the flickering problem, and everything rendered as desired. After this, we experimented with rendering image files and simply checking to see if they rendered correctly on the screen. First, things didn't go so well, and the image did not render correctly. As it turned out, the C compiler added another byte of space in the `tga_header_t` struct, because it wants `short` values to be placed on 2- or 4-byte boundaries. However, the structure we read from disk did not have these boundaries, and the values we read from the structures were off by one byte. Once this was rectified, the image rendered almost correctly. Our intro image has a grey background and red-orange text, but during our test, we found the colors were reversed, and the title text was blue instead. As it turns out, the color values are in the order of *blue, green, red*, not *red, green, blue* as expected. Once the red and the blue switched place, the images rendered correctly. Finally, we implemented the bouncing soldier, and changed the parameters until we found it behaved as desired.

3.3 Sound

Seeing as how we'd already implemented sound in the previous assignment, implementing it this time was a fairly simple job. We did not start the sound module until after we were told about the ALSA muting problem, so implementing sound was fairly uneventful. To check whether the sounds performed correctly, they were played back on the STK1000 board, as well as on a laptop computer to compare. It soon became apparent that the files were being played back roughly twice as fast as they should. A quick check of the sampling rate of the files and the sound settings in `/dev/dsp` showed that the sound settings were double rate compared to the sound files. Once this was fixed, the sound performed correctly.

3.4 Game logic

Seeing as how testing the game logic on the board required near-perfect functionality from the STK1000 board, we decided to make a simple terminal version of the game, using the same game logic, as described in 2.3.6. Additionally, terminal testing allowed for making changes in quick succession without being hampered by testing the logic on the STK1000 board. Using this tool, we performed successive tests and improvements. All tests were performed by doing a normal run of the test program and comparing it to the logic rules laid out in 2.3.6. We started out by making sure the shooting function worked as it should. Our first test failed. The shots appeared to originate from the wrong corner in our first test run of the game. We rectified this by flipping the calculated x position. Once this was done, we ran a second test. It too, failed. Although the first problem was resolved, we discovered that it was impossible to hit the

rightmost tiles in the grid. After some searching, we discovered that we flipped the x position before we cast the floating point number to an integer, leading to it rounding the wrong way, excluding a whole column of tiles. We fixed this by casting to an integer first, and then flipping the x position. At this point, the shooting function performed as it should, and we ran a third test. It failed, because the tank would be declared winner after only one shot, even though the proper winning conditions were not met. A short investigation revealed that the depth-first search that tests for whether there is a path between the soldier and the tank (no path means that the tank has won) had a simple logical error, where `TRUE` and `FALSE` had been inverted. We fixed this, and tested again. Our fourth test was successful, as the game logic performed as expected and according to the rules laid out.

3.5 Complete test

Finally, once the individual components performed as they should, we tested the game as a whole on the STK1000 board. The game performed well, with an exception of a minor bouncing problem with the buttons, as outlined in our first assignment. Once debouncing was implemented, the game performed exactly to specification.

4 Evaluation of the assignment

This assignment was a good introduction on how to work with Linux, as well as how to implement and use kernel drivers on a microcontroller. The assignment gave us the opportunity to learn more about both how graphics and sound are rendered correctly, and how many components need to work together to make a final product.

As this is our third assignment, being more familiar with the hardware and certain aspects of the software allowed us to work on a larger project than earlier.

The assignment provided not only technical knowledge and insight, but also experience with relatively large C projects where one has to design an interface for others to use and plan functionality of the code prior to its implentation. As both are highly relevant for students studying complex computing systems, we found the assignment very rewarding.

5 Conclusion

In this assignment, we started out by implementing a kernel driver for the buttons and LEDs on the STK1000 board, before we implemented a game through

usage of the previously implemented device, a sound device and a frame buffer device.

Most of the issues in this assignment were related to hardware limitations and assumptions about specifications which we later found out were wrong. For instance, we had to tune the sampling rate on the files as well as in the sound driver settings so that the sounds played at correct speed, and we had to bit-shift because the endianness of the TGA-files and the frame buffer were different.

In the end, all of the difficulties we encountered were overcome. The resulting game is quite playable, and the overall design of the game components and the interaction between the game and the kernel driver are well-defined and tidy.

References

[1] NTNU, *Lab Assignments in TDT4258 Microcontroller System Design*, Computer Architecture and Design Group, Department of Computer and Information Science, Norwegian University of Science and Technology, 2011.

[2] Wikipedia, *Truevision TGA*, http://en.wikipedia.org/wiki/Truevision_TGA, accessed 2012-05-04.