Assignment TDT4258 Assignment 3 - Group 1

Sondre Lefsaker André Philipp Håvard Wormdal Høiby

April 25, 2013

Abstract

In this assignment we had access to the Linux operating system installed on an SD card that was plugged in to the STK1000 development board. The task was to write two Linux device drivers for the lights and the buttons on the STK1000, let the drivers be recognized by Linux and being able to use these drivers through a C-program. More specifically, the task was to write a computer game called Scorched Land, that used the audio and screen on the STK1000 and the game was to be controlled by input from the buttons on the development board.

The main difference from this excercise and the other two, is that we had access to an operating system with appropriate drivers to e.g. screen and audio. This makes it possible to make rich programs that uses all the different devices that the STK1000 offers, without having to go through the trouble to write the drivers ourself.

Contents

1 Introduction

This is the techinical report for the third and final exercise in the course TDT4258. We were already fairly known with the STK1000 microcontroller after the previous exercises, including the GNU toolchain and C programming.

The new concept of this exercise was the use of device drivers for the different devices on the development board. So the main task was to understand how the Linux device drivers is built and how they work in order to create our own drivers and use the already existing drivers in a C program.

The first part of the report explains the setup on the development board and how we got the Linux operating system to work on it. It also explains how we developed the code, compiled it for the AVR32 microprocessor and ran it on the microcontroller.

The second part roughly explains how we implemented the two drivers for leds and buttons, respectively, and how they work with the rest of the system. It also explains the two frameworks that we built around the graphics device and the sound device, so that we could draw pictures of the .bmp format and play .wav sound files without any trouble. Lastly it gives a short overview over the game and how it works.

The third part shows how we solved the different problems, and what we had to go through in order to get the different devices to work with the game. We use some code examples where we see fit, to explain the implementation in more detail.

2 Setup

2.1 Board

To prepare the board for installing the drivers and the game the following must be done:

1. The jumpers and GPIO connectors must be placed properly.
2. The bootloader must be installed onto the board.
3. Linux must be installed onto a SD-card.
4. Linux must be booted with the help of uBoot and Minicom.

This section is a walkthrough of this process.

2.2 Game

To play the game it has to be compiled from the source, copied onto the board and started. This can be done from any of the lab pc's in the lab which contains the **avr32-linux-gcc** compiler.

2.2.1 Compilation

From the /game directory on the lab pc the following command will compile the game:

```
make
```

This produces the file /game/bin/game which is a executable that can be run under the linux distribution installed on the SD-card.

2.2.2 Copying the executable to the board

From the /game directory on the lab pc the following caommand copies the game executable to the board.

```
scp bin/game avr32@<IP address>:~/
```

Where *<IP address>* is the IP obtained in section ¡...¿.

2.2.3 Running the game

From the /home/avr32 directory on the board the game can now be started with:

```
./game
```

In order to play the game the user has to be root, this is obtained by running the **su** command and using password = roota.

3 Overview

An operating system is a huge and complex program that administrates the hardware of a computer (in this case the STK1000 microcontroller) and offers all the different devices to the programmer through a nice and relatively easy interface. This makes it possible for the programmer to write rich programs without having to think too much about task management, permissions to different users and how to control and handle the hardware, which is just a few important features of an operating system.

In order for an operating system, like Linux, to work with all the different hardware out there, it will need device drivers for the different devices. These drivers is a special kind of software that runs *between* the operating system and the hardware and makes it possible for the operating system to use the specific device. This could not have been done if it had not been for a common interface

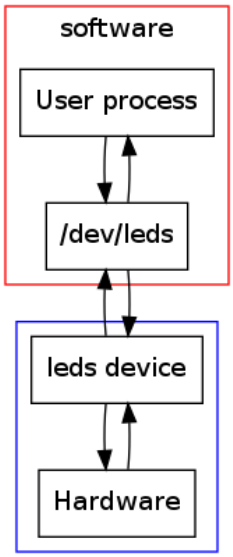


Figure 1: The location of a device driver in the Linux environment

that the different drivers could implement. Linux offers such an interface, and it is described in the book Linux Device Drivers [1].

3.1 Writing a driver

There are several steps that is necessary to go through in order to successfully write, install and use a Linux devie driver. This overview will briefly explain only the process of creating the driver for the leds, because it is pretty much the same as for the buttons.

3.1.1 Making it a kernel module

A Linux device driver is a kernel module, i.e. it runs in the kernel and not in user space. This means that it has other privileges and limitations than user space programs. Firstly, it has to implement certain functions in order to be called and executed by the kernel; a **module init-function** and a **module exit-function**. The init function is called when the device driver is mounted in the system, and does a couple of important things:

- Allocates a major and a minor number that every kernel module needs to have, in order for the operating system to uniquely identify and use it.
- Runs through the process of registering and requesting the memory region that the driver will use throughout execution.
- Sets up and registers a character device structure for the device, this device structure is registered together with the major number and the different file operations that can be exectued on the device.

The init procedure also have to handle any errors during the initialization process, e.g. errors while registering the character device structure or while requesting a memory region.

The device driver also have to specify a license for distribution of the driver, this is not so relevant for this execise, but a GPL license is a safe choice because of its few restrictions.

3.1.2 Compiling the kernel module

The kernel module is written as a normal C program, but there are a few abvious differences. Because it runs in the kernel space, a driver does not have access to the C standard library and all its functions. There is no main function in a device driver, it is accessed only through a predefined and included interface. This interface can change depending on the different versions of the Linux operating system, and thus gcc needs to know about the Linux version that the driver is to be compiled for.

When this is supplied in the Makefile, the kernel module can be compiled from source with a couple of extra flags attached:

- make ARCH=avr32 CROSS_COMPILE=avr32-linux-

The result will be a kernel module (with extension .ko) that can be mounted to the operating system.

3.1.3 Installing the kernel module

After all the previous described steps are finished, the kernel module can be mounted and be used by any other program on the system.

To insert the driver the **insmod** command is called with the kernel module as parameter:

- insmod leds.ko

This will insert the module into the kernel, and the module_init function will be called.

Next step is to make a device file out if it, calling the **mknod** command with the name of the driver, the type (character device, block device etc.) and major and minor number will create the file representing the device.

- mknod /dev/leds c 245 0

The file **/dev/leds** now repesents the eight leds on the GPIO, and the different leds will now light by writing a byte of data to the file, depening on the value of the byte. Figure 1 illustrates the driver mounted in the system.

Finally, to remove the device, the **rmmod** command can be called with the device name as paramater.

- rmmod leds

This will make the kernel call the module_exit function, and the previously registered and allocated memory and character device regions will be released and unregistered.

3.2 Graphics

3.2.1 Main graphics

The graphics of the game is contained within the /graphics folder. Here the **Screen** object is responsible for communication with the framebuffer through the **/dev/fb0** file. The **Canvas** object holds an array of **Shape** objects. Its has a method called **CanvasPaint** where it renders the screen in an 320x240 **Pixel** array contained within the **Screen**. When the whole array has been updated the **Canvas** object ask the **Screen** object to copy the contents of its internal buffer to the framebuffer. This causes the LED screen on the device to be updated.

All graphical objects are inherits their main structure of the **Shape** object. The most importaint part is the **paint** method which **Canvas** uses to render all the objects on the internal buffer.

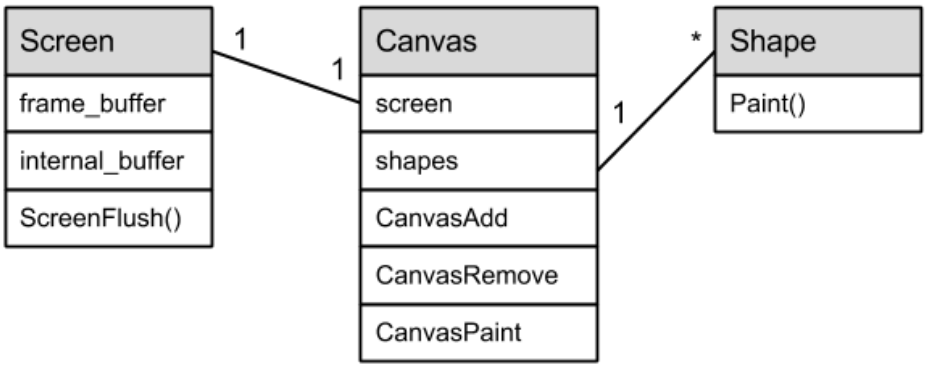


Figure 2: UML for the main part of graphics

3.2.2 Images

For all images the 24 bit version of the **BMP** (Bitmap file format) was implemented. This is done by the object **Bitmap** is hidden behind the **Image** object so that it is easy to add more file formats. The reason that bitmap was chosen is that the 24 bit version of it corresponds almost directly to the layout of the framebuffer, so the implementation was straight forward.

3.2.3 Graphical objects

Both objects for drawing lines and rectangles are where used only in the testing of the graphics module and are not used in the game. They serve as examples on how to develop more graphical objects.

3.3 Sound

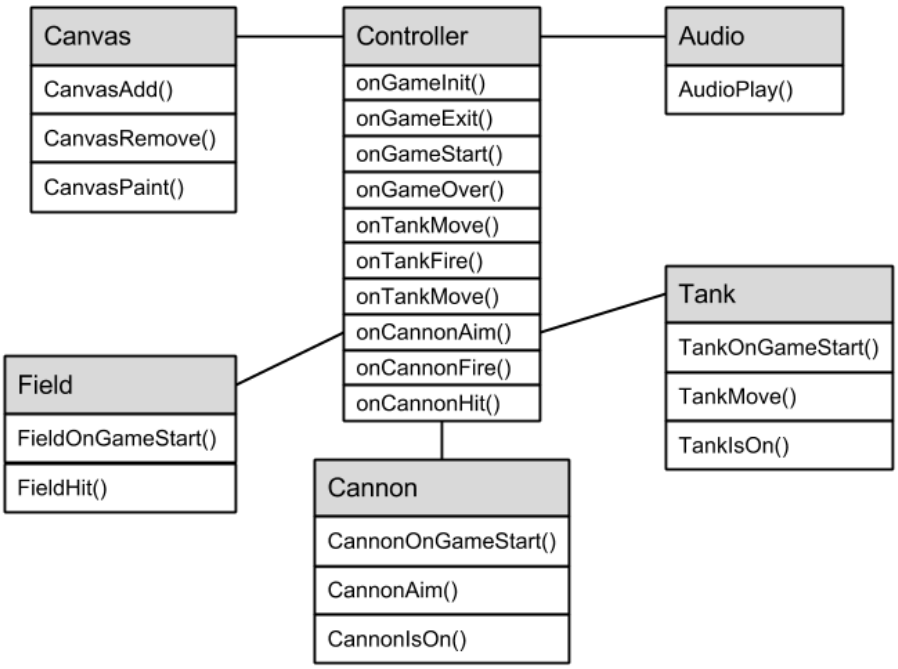
In the folder /sounds the object **Audio** takes care of playing sounds. It uses ordinary file operations to write 8 bit samples from a **wave** file to the file **/dev/dsp**.

3.4 Game

3.4.1 How to play

3.4.2 Design

The /game folder contains the objects used in the game. The most importaint are **Controller** which handles the communication between the modules in the game. The **Cannon**, **Tank** and **Field** are models for the objects in the game. The **Button** and **Led** objects are wrapper object around the char drivers, to make communication easier.



4 Solution

4.1 Drivers

The led driver and the button driver has a lot of similar code, mainly the initialization process and the termination process of the driver. They are both character devices that registers a memory region on the PIOB and registers itself as a character device, with major numbers and file operations like read and write. The following sections describe the implementation of the led driver, but it is the same for the button driver.

4.1.1 Module initialization

The leds and buttons driver have the init functions **leds_init** and **buttons_init**, respectively. They are registered, and thus picked up by the kernel, by calling the function **module_init** passing the init-fuction as a parameter.

```
module_init ( leds_init );
```

Both the drivers have a file_operations structure, defined in the C file. All it does is map different file operations, implemented in the driver, to the ones defined in the structure. This file_operations stucture will later on be registered together with the actual device.

```
static struct file_operations led_fops = {
    .owner = THIS_MODULE,
    .open = open_leds,
    .write = write_leds,
    .read = read_leds,
    .release = release_leds
};
```

The first thing that happens inside the leds_init function is allocation of a character device region and assigning a mojor number to the device.

```
static int leds_init ( void ) {
    //...
    result = alloc_chrdev_region ( &dev, leds_minor,
        NUM_DEVICES, "leds" );
    leds_major = MAJOR ( dev );
    //...
```

If alloc_chrdev_region returns a negative number there is a problem getting a major number to the device and the init method returns with an error. If result is positive we build a dev_t data item, representing the device based on the minor and major number, and requests memory region that we are going to use on the GPIO at the PIOB address. If all goes well we initialize the *per* and, *oer* registers at piob (*per* and *puer* for buttons), in order to be able to read and write from PIOB.

```
dev = MKDEV ( leds_major, leds_minor );
result = (int) request_region ( AVR32_PIOB_ADDRESS, mem_quantum, "
    leds" );
// Initialize the leds
volatile avr32_pio_t *piob = &AVR32_PIOB;
piob->per |= 0xff;
piob->oer |= 0xff;
```

Finally, the leds_init function allocates and sets up the character device structure *char_dev*. It adds all the led file operations defined in the struct *led_fops* to *char_dev*, and it registers the device with the dev_t *dev* data item.

```
// Set up char_dev structure for the device
struct cdev *char_dev = cdev_alloc ();
cdev_init ( char_dev, &led_fops );
char_dev->owner = THIS_MODULE;
char_dev->ops = &led_fops;
int error = cdev_add (char_dev, dev, 1);
```

If there is an error, i.e. *error* is a positive number, it will be logged to the kernel. Otherwise, the device driver initialization should have completed successfully.

4.1.2 Module termination

The drivers also have an exit-function, **leds_exit** and **buttons_exit** that is registered the same way as the init function. This function is called when the kernel module is removed from the operating system.

```
module_exit ( leds_exit );
```

The termination procedure of the kernel module is much smaller than the initialization procedure. It basically clears out the output register on the device, releases the peviously requested memory region and unregisteres the previously registered character device region for the device.

```
static void leds_exit ( void ) {
    int mem_quantum = sizeof( avr32_pio_t );
    volatile avr32_pio_t *piob = &AVR32_PIOB;
    piob->codr = 0xff;
    release_region ( AVR32_PIOB_ADDRESS, mem_quantum );
    unregister_chrdev_region ( dev, NUM_DEVICES );
}
```

4.2 Led Driver

The led driver and the button driver has a lot of similar code, mainly the initialization process and the termination process of the driver. They are both character devices that registers a memory region on the PIOB and registers itself

4.3 Button Driver

4.4 Graphics

4.5 Sound

4.6 Game

5 Test Report

6 Discussion

7 Conclusion

8 References

1. Linux Device Drivers
   http://shop.oreilly.com/product/9780596005900.do

2. Character Device *http://en.wikipedia.org/wiki/Character_device#Character_devices*

3. TDT4258 - Compendium
   http://www.idi.ntnu.no/emner/tdt4258/_media/kompendium.pdf

4. AVR32 Architecture Document
   http://www.idi.ntnu.no/emner/tdt4258/_media/doc32000.pdf