TDT4258 - Exercise 2

Magnus Halvorsen Julian Lam

April 28, 2014

1 Abstract

Operating systems provide an abstraction such that interacting with hardware is made much easier for programmers. However, in order to be able to support the vast amount of hardware available, the operating systems requires device drivers. In this report we will make device drivers for Linux on a EFM32GG to add hardware support for a gamepad. We will also show how device drivers can utilize kernel timers and how they can send signals to user applications in userspace. We will also use these drivers in order to make a simple computer game, that needs to interact with hardware.

Contents

List of Figures

2 Introduction

The purpose of this assignment was to get a solid understanding of how Linux device drivers work, how they can control hardware and how user applications can interact with hardware through them. An operating system provides a nice and clean software interface to hardware for user applications. Thus hiding the complexity of hardware, and making user application development a lot easier. But in order to support the vast amount of hardware available, the operating system needs device drivers control for their specific behaviour. The operating system provides the driver with an interface of functions that the driver can support(e.g. read, write, iseek), without caring how the driver implements them. So when a user application wishes to access hardware it invokes a system call. The kernel then calls the respective device driver (which resides in kernel space), which directly interact with the hardware.

To get first hand experience with this; we were given the task of making a computer game for Linux, which should interact with a simple gamepad through a character device driver. We were provided with an EFM32GG development card, which had a simple gamepad attached(see figure 2). The assignment tasks were the following[3]:

1. Configure and build the uClinux for the board.
2. Make a driver for the buttons. It should be implemented as a kernel module. You are free to make the driver as you wish, but the minimum requirements are to support your needs for the game to work.
3. Complete the game. Use the framebuffer driver for writing to the display. Use your own driver for reading the status of the buttons.

Since the main foucs of the assigment was to learn how to make device drivers and how to make them interact with userspace, we decided to make a very simple game. In the game you control a shield which you are supposed to use to catch astroids that are falling down. The shield can be moved to the left or right by pressing button 1 and 3, respectively. See figure 1.

We ended up making three device drivers for the game: one driver for the gamepad, a driver that worked as a timer, and an attempted sound driver. The purpose of the gamepad driver was to read the status of the gamepad buttons. The timer driver was used to signal the game whenever it was supposed to update its screen (i.e. move the shield, add more astroids, move the astroid downward etc.). We also attempted making a sound driver, using the direct memory access (DMA), but we were unable to get to function properly.

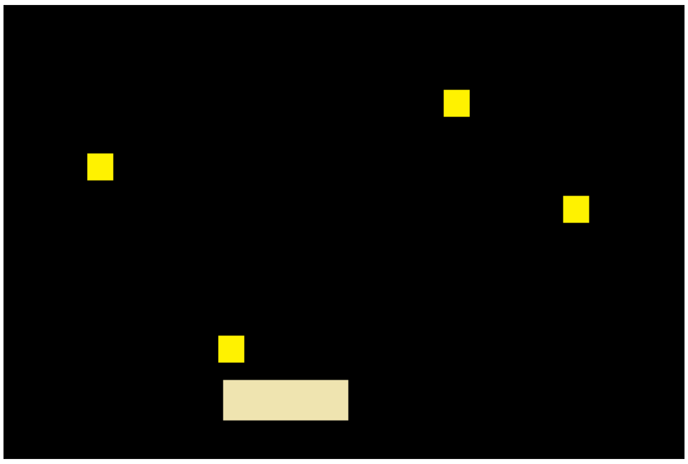

Figure 1: The game. Move the shield to the left and right to prevent the yellow astroids from reaching the buttom.

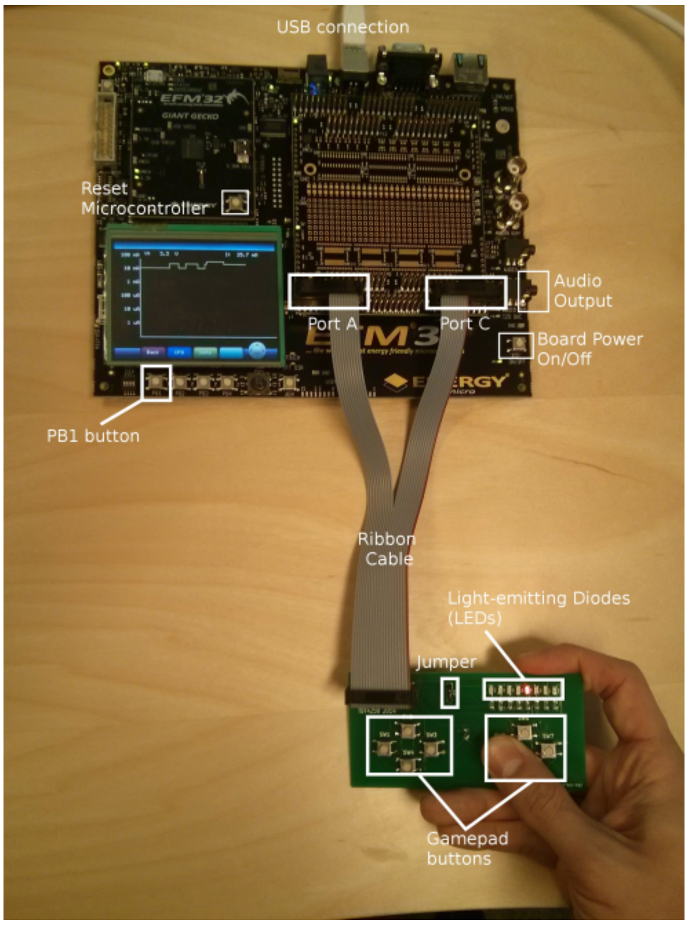

Figure 2: The EFM32GG development card and gamepad[3]

3 Description and Methodology

This section covers how the assignment was solved and different design choices. This was done in the following steps:

- Configure and build uClinux and flash it to the board.
- Create a device driver for the gamepad.
- Create a device driver to provide timer support.
- Create the game.
- Attempt to make a sound device driver.

3.1 Installing uClinux

We were provided with a skeleton code which contained the source code for uClinux and skeleton code for the driver and the game. The build tool ptxdist was used to compile the kernel, the modules and other software, package it into a binary file and flashing it to the development board. The compendium gave very clear directions on how this should be done[3].

After configuring the ptxdist project, the project was build and flash by the following commands:

```
ptxdist images

ptxdist test flash-all
```

But when we were working on the drivers, it would be too time consuming to rebuild and flash the whole project. Thus the compendium provided an alternative route for only building and flashing the respective modules. There were some problems with the provided route though, because the modules would not update. We solved it by doing the following commands when flashing the drivers/applications:

1. `ptxdist clean <packagename>`
2. `ptxdist go`
3. `ptxdist image root.romfs`
4. `ptxdist flash-rootfs`

We were able to interact with the Linux OS on the development card by using a provided program called *miniterm*, which allowed us to interact with it as a standard shell.

3.2 The Gamepad Driver

This driver should allow the userspace to read the status of the gamepad buttons. We also added interrupt support, and userspace signaling support, but later decided that this was better left to the timer driver. Thus, details on how interrupt and signaling support was implemented will be covered in the next section.

From the start of the development we wanted to make this driver as dynamic as possible, i.e. dynamically allocate all needed resources, making it more portable. The compendium provided most of the steps required for allocating the neccessary resources, making the driver interface available for the OS and visible in userspace:

- Allocate a device number for the device using `alloc_chrdev_region(...)`.
- Provide functions to the operating systems, so the device can be accessed by user applications as a file, using a `file_operations` struct and the function `cdev_init(...)`.
- Add the driver to the system using `cdev_add(...)`.
- Make the driver visible in user space using `class` struct, initalize it with `class_create(...)`, and then create the device and register it with sysfs using the fuction `device_create(...)`

In order to make sure that no other process was using the hardware, all the hardware's memory region was allocated using `request_mem_region(...)` when the driver was initalized. When the driver is removed, all the resources was released with `release_mem_region(...)`. For this driver we had to allocate the *GPIO_PC* region for reading button status, *GPIO_PA* region for output (used during testing) and some registers for interrupts.

Even though we are not using virtual memory in this assignment, we decided to program our drivers as if they were. This makes the driver more robust, as it follows the Linux standard and can thus potentially harvest all the benefits of virtual memory. We thus had to use `ioremap_nocache(...)` function to memory map the memory regions, and read/write to them using `ioread32(...)` and `iowrite32(...)`, respectively.

In order for the userspace to read the button status, the driver provides a `read(...)` function. The function contains the parameter `char __user *buff`, which is a pointer to a buffer that the calling user application can access. Thus the driver writes the first eight bits of the register *GPIO_PC_DIN* (which represents the button status) to that buffer.

3.3 The Timer Driver

The main reason for the timer driver was to provide an easy way to calculate when the game should update the frame. This was initially needed to make the astroids move downwards on specfic itervals. Using the code provided by [5] we were provided an interrupt timer that we could specify in miliseconds. Since the frame had to be updated several times a second, we decided that we thus did not need the gamepad driver to let the game know a button had been pushed. Instead we read the button status every time the timer signaled the game, for two reasons:

- Easier programming. Do not have to decide which driver that sent the signal to the game.

- Smoother gameplay. Since the gamepad is read several times a second, you can move the shield by simply keep pressing a button. Instead of having to stutter moving by creating an interrupt. And since the interrupts from the timer is needed for moving the astroid, checking the gamepad, while we are at it, does not hurt the energy efficiency.

The interaction between the game, gamepad driver and timer driver and be seen in figure 3.

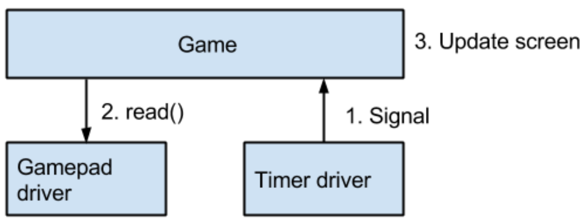

Figure 3: The interaction between the drivers and the game.

The timer followed the same steppes as the gamepad driver for initalization, resource allocation, providing functions to userspace and making itself visbile to userspace. But in addition it had to provide signal support, in order to signal the game whenever the timer was finished. *Linux Device Drivers*[4] provided the neccessary information to enable device-user application-signaling.

This was done by providing a `fasync(...)` function to userspace, which a user application could use to register itself as a listener to signals from the driver. All the listeners to the driver is stored in a `fasync_struct` struct, and are added by calling the `fasync_helper(...)` function. In order to signal the user application the function `kill_fasync(...)` was called in the timer callback function (i.e. whenever the timer was finished). The user application was removed from the listener list by calling `fasync_helper(...)` upon close, with -1 as *file_descriptor*.

But in order for the user application to add itself as a listener, it had to execute the following code:

```
signal(SIGIO, &signal_handler);
fcntl(fd, F_SETOWN, getpid());
oflags_dg = fcntl(fd_dg, F_GETFL);
fcntl(fd, F_SETFL, oflags_dg | FASYNC);
```

Where *fd* is the file descriptor and *signal_handler* is the function that will be called when a signal from the driver occurs.

The timer driver also provided a `write(...)` function, where the user could specify the number of miliseconds the timer should run for, before signaling the user application. It also provides a `read(...)` function, that returns 1 if the timer is running.

3.4 The Game

The game itself is very simple. It uses the *framebuffer*, located at `/dev/fb0`, to render the screen. For easier development and code the framebuffer was memory mapped, which allowed us treat it as a reqular C array. It also allowed to decide which part of the screen should be rerendered, using the `fb_copyarea` struct and the framebuffer's `ioctl(...)` function. Thereby saving enegy, since we do not have to rerender the whole screen when only a small part needs to be updated.

As mentioned before the game uses the gamepad driver to read the status of the gamepad buttons, and the timer driver to recognize when it needs to update the screen (see figure 1). Low update intervals gives good performance, but comes with a energy cost. The more you update the screen, the smoother it is, but uses a lot more energy. This will be discussed in more detail in the result section.

The game is interrupt-driven, and sleeps whenever it does not have to handle a signal from the timer driver. Thus it uses as little energy as possible.

As the game implementation was really simple, and a rather uimportant part of the excerise, we will not go into any more detail on the implementation. Interested readers can read the provided code.

3.5 The Sound Driver

We wanted to have sound to our game, so we tried to implement a sound driver. On our first attempt we tried by doing it the same way as in the previous assignment, where we made the CPU write directly to the *Digital to Analog Converter* (DAC). It worked, but the sound quality was *really* bad, due to performance issues. Thus we tried implementing it using *Direct Memory Access* (DMA), but it did not work. This section will cover our attempted solution.

We initalized the driver in the same way as the previous drivers, so what remained was setting up the interaction between the DMA, DAC, TIMER3 and the driver. The interaction is illustrated in figure 4.

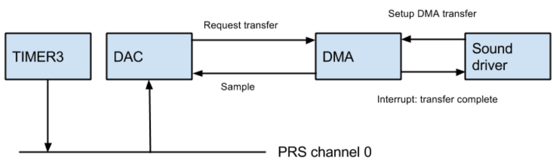

Figure 4: The interactions between the DAC, DMA, TIMER3 and CPU.

It works as follows:

- TIMER3 underflow is set to PRS channel 0 input.
- DAC channel 0 and 1 is set convert data on PRS channel 0 input.
- The DMA is set up for basic transfer. It triggers on DAC conversion complete, and arbitrate after after each transfer. Thus allowing the DAC to control the speed of the data transfer. When the DMA has transfered all of the provided samples, it sends an interrupt to the sound driver - which should provide it with more samples to transfer.
- The sound driver has access to an array of sound samples it should provide the DMA with. This is done by updating the DMA descriptor, which contains the source end address, destination end address and control bits. The control bits specifies the size of each transfer, how the source and destination addresses should be incremented, and the number of transfers (up to 1024) etc. Thus when the DMA has transfered 1024 samples, the sound driver has to provide it with the address of the next 1024 samples, until all the samples have been transfered.

But for unknown reasons this did not work, and after hours of debugging we were forced to give up.

4 Results

This section covers the testing of the program, the resulting functionality of the program and its power consumption.

4.1 Result

4.1.1 Functionality

The result of this assigment was a simple computer game where you are supposed to stop raining astroids from hitting the earth by moving a shield back and fourth. The game uses two working character device drivers to access a gamepad and kernel timers. Both timers have signal support to user applications, making it possible to make the game interrupt driven - reducing CPU usage and energy.

We tried implementing sound for the game, but we were unfortunately unable to do so.

4.1.2 Power consumption

The following current measurement where made under the described conditions:

- The Linux kernel by itself runs at 10.3 mA.
- Running the game with high performance (screen updating every 5 ms) with polling, cause it to use 22 mA.
- When it is interrupt driven and sleeps when not handling interrupts, it runs at 18 mA.
- If we rerender the whole screen, instead of only the parts that are neccessary, it uses 31 mA, and runs really slow.
- If we increase the screen update interval to 2000 ms, the game runs at 10.5 mA. But this causes the performance to be unbearable.
- Setting the screen update interval to 50 ms, the game runs at 12 mA, and the game performance is tolerable.

From these numbers we can see that the major design choices have been power efficient, especially making the game interrupt driven and only updating the neccessary parts of the screen.

We can also see that we have to make a choice between smooth and high performance gameplay, and energy efficiency.

4.2 Testing

The testing of the program was either done manually (pressing buttons and observe the result) or with GDB. The general method was testing manually to see if the program did what we intended. If this failed, and there were no obvious reasons for this, we debugged with GDB by reading the values of critical registers. Power consumption was tested by reading the energy usage of the board, using the built-in energy meter.

5 Conclusion

The assignment tasks were:

1. Configure and build the uClinux for the board.
2. Make a driver for the buttons. It should be implemented as a kernel module. You are free to make the driver as you wish, but the minimum requirements are to support your needs for the game to work.
3. Complete the game. Use the framebuffer driver for writing to the display. Use your own driver for reading the status of the buttons.

We can conclude that all of the tasks have been completed. In addition we made two more drivers in order to provide a better implementation. Two of the drivers also have support for signaling the user application. All of the drivers allocates resources dynamically and supports virtual memory, making them quite robust.

We were unfortunantly unable to implement the sound driver, but we still learned a lot by attempting to make it work.

The game may be a bit simple, and we wish we had more time to work on it, but it still illustrates how a user application can interact with device drivers. More importantly, it illustrates how an user application can interact with *our* device drivers, and that they work.

All in all, it has been an interesting and highly educating assignment.

References

[1] EFM32GG Reference Manual
http://cdn.energymicro.com/dl/devices/pdf/d0053_efm32gg_reference_manual.pdf.

[2] Cortex-M3 Reference Manual
http://www.silabs.com/Support%20Documents/TechnicalDocs/EFM32-Cortex-M3-RM.pdf

[3] TDT4258 - Compendium
Provided on itslearning

[4] Linux Device Drivers
http://lwn.net/Kernel/LDD3/

[5] M Tim Jones. Kernel apis, part 3: Timers and lists in the 2.6 kernel. *http://www.ibm.com/developerworks/library/l-timers-list/*, 2010.