NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4258 - Low-level programming

Third Lab Exercise Report

Group 15:

Dávid Danaj
Salahuddin Asjad
Valentin Lemenut

November 16, 2015

Abstract

This is a report for the third assignment in the subject TDT4258 - Low-level programming, which is about making a game for a microcontroller. We have chosen to create a simple two-player game, where both players will share the same gamepad. To make the microcontroller energy efficient, we have implemented interrupts so the system uses less power during idle state.

1 Introduction

The practical goal of the lab exercises is to implement a small game for the EFM32 development board which has a microcontroller, a display and audio effects. We also have access to a small prototype gamepad with buttons and LEDs so we can perfectly make our own simple computer game by programming the microcontroller which allows us to control the I/O-components we have.

Today, microcontrollers are widely used because they fit into almost every task. It is easy to find low-end microcontrollers for small applications and high-end microcontrollers for more complex ones. Microcontrollers have a big advantage that the design and hardware cost can be kept to the minimum level, because of the wide range of microcontrollers in the market.

Microcontrollers are usually built to do a specific repeating task which often makes only small piece of complex system. This allows engineers to create a microcontroller, which perfectly fits needs of a task from hardware as well as software point of view. Size of these systems provides possibilities to fit them nearly everywhere. Since microcontrollers are getting smaller and smaller, they are often integrated into places where it is difficult or impossible to get. They are often found isolated from other systems where their exclusive power source is a battery. In cases like that, it is extremely important to focus on minimization of power consumption, which prolongs lifespan of a device.

Furthermore, energy efficiency of our solution is an important goal of the course and this is why we have a specific component on our development board to measure the realtime power consumption of all components we have, which allows us to optimize our program to minimize the power consumption. Most microcontrollers are intended to work in systems where power consumption matters. Also in our case we want an efficient application, that uses minimum of power to keep the system going. That's why we have implemented different ways of accomplish that, which will be described later.

1.1 The game

After working with the two previous exercises, we are now familiar with most of the boards components and its functions of the EFM32GG microcontroller and we can play different sound effects according to the button we press on the gamepad. In this exercise we will have to configure and compile a complete Linux operating system for the EFM32GG, create a Linux device driver (Kernel module) to communicate towards the gamepad and create a Linux user space program (the game) that communicates with our driver and the display driver that is already made. Both the driver and the actual game are written in C language and we will program the hardware in Linux. Eventually

we will have to implement a game that use our driver. As this project focuses more on the content than the form and because of the limited resources of the microcontroller, we will have to create a simple game, but make it as fun as possible. Our idea is to create a two-players game to give ourselves more challenge where each player will use 4 buttons on the gamepad to play. The screen shows which button to press, and the first person to press the right button within 3 seconds gets a point. The first player to reach 5 points will wins the game.

2 Background and Theory

To solve the exercise we will need to use our technical knowledge with microcontrollers and C programming for Linux. Like in the previous exercises we will have to read the reference manual of the EFM32 development board to get the required knowledge about the microcontroller we are using. [2] The reference manual provides with important information about the different modules and peripherals in the EFM32 development board. The book 'Linux Device Drivers' [1] is also a really useful document to fully understand the concepts of this exercise.

2.1 Hardware Description

As mentioned in the previous section, we are using a EFM32 development board to solve the problem. The EFM32 microcontroller familiy is one of the most energy friendly solution of any other 8-, 16- and 32-bit microcontrollers available. The development board consists of a high performance 32-bit ARM Cortex M3 processor, which offers many benefits for the developers like fast interrupt handling and ultra-low power consumption. The development kit supports five different energy modes named from EM0 to EM4. These modes make it easy for us to choose dependent on the task. In case of energy mode 0 (EM0), the CPU is running and all peripherals are active. While in case of energy mode 4 (EM4), all chip functionalities are disabled, but can be turned on by GPIO pin wake-up or Power-On Reset for example.

The Development Kit is intended to be a complete platform for developing applications for the EFM32 microcontroller. The kit includes important components such as 16 MB NOR Flash memory for non-volatile storage and 4 MB SRAM to provide fast memory access to the processor. The embedded debugger allows applications to be downloaded and debugged directly. A set of useful peripherals are provided, and custom hardware can be developed on the prototyping area, where all the microcontroller's I/O pins are made available. For our purpose, we are connecting the gamepad to some of the available GPIO pins.

2.1.1 Main Peripherals Description

As seen in Figure 2.1, the development kit has a lot of peripherals available. We can shut down most of them in order to prevent excess current leakage when we don't use them.

In this exercise we are not able to use the LCD screen for monitoring the current consumption of the board. Instead we can using the eAProfiler from the development computer to measure the power consumption with a sufficient precision.

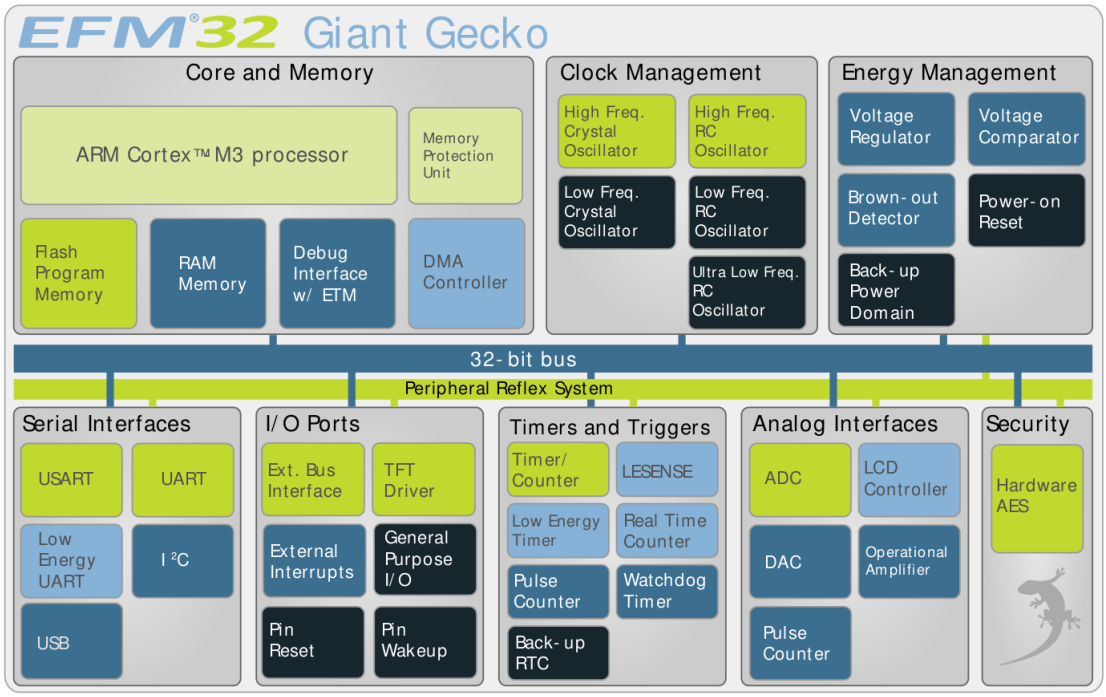


Figure 2.1: Block Diagram of the Development Kit.

To control what the microcontroller will do under runtime, we are using a gamepad. The gamepad is connected to the prototyping board on the development board.

2.2 Important concepts

2.2.1 Linux system for the EFM32GG

As embedded systems are getting more and more complex, the gap between embedded microcontrollers and full blown microprocessors is reducing. Therefore we now need more than a good toolchain to achieve all the tasks we want to. Using generic Linux distributions could be a good idea as they are open-source and always improved by a large community but the problem is that they require too much of the hardware to be installed in comparison to what we can have in the embedded system world[3].

Often embedded Linux is connected to real-time systems, because some embedded Linux distribution are made to achieve real-time computation. Real-time can be classified as soft or hard real-time. Example of soft real-time systems are banking telling machines and streaming of audio/video where the systems is not considered to have failed if the system doesn't meet the exact deadline. But in case of hard real-time systems, if the system fails to meet the deadline (even once), the system is considered to have failed because it can damage the equipment and lives, like the ABS systems in our cars. So with real-time systems, it is very important that the code is bug-free and the interrupts

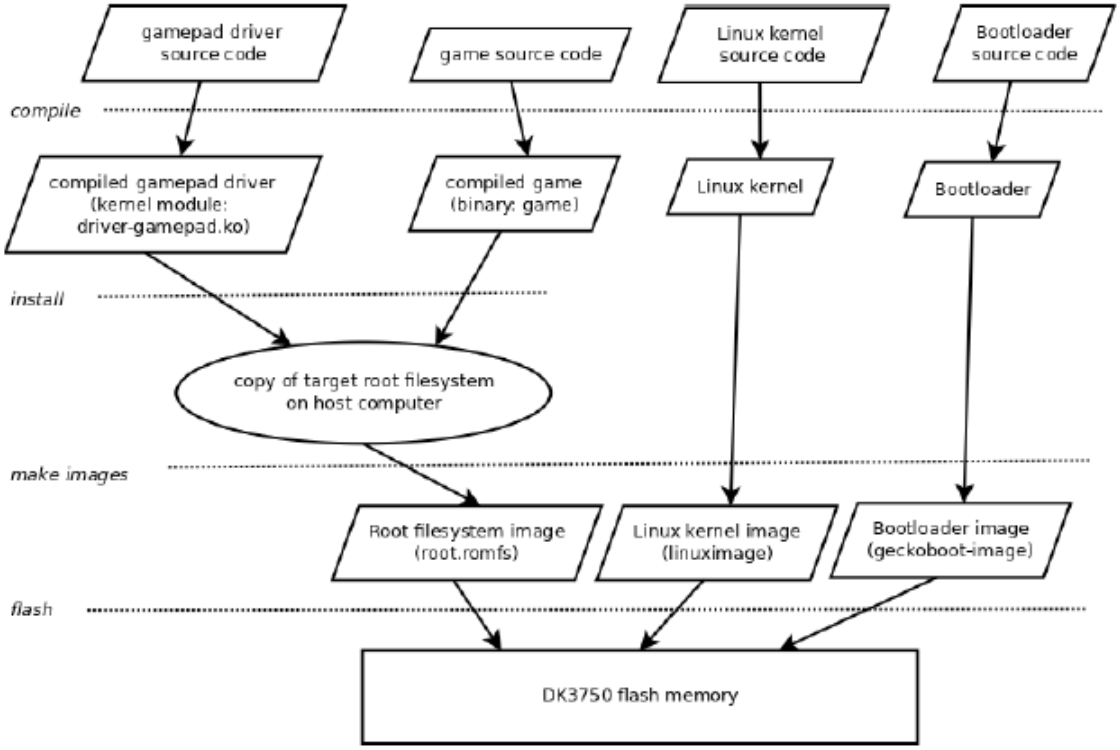


Figure 2.2: Schematic view of the build process using PTXdist

are prioritized to meet the constraints. In this assignment we are using Clinux, a popular Linux kernel for microcontrollers in embedded systems without a memory management unit. uClinux is non-deterministic, as a Linux distribution in general, but by applying hard real-time micro kernel to uClinux, it can also be used for real-time critical systems.

2.2.2 PTXdist usage

We use PTXdist which is a build system for making Linux distributions, and is especially created for embedded systems. So it compiles the kernel, module and other necessary software, and makes binary files from the compilation that can be flashed into the development board. A PTXdist project contains all information and files to populate any kind of target system with all required software components[3]:

- Specific configuration for bootloader, kernel and userland (root filesystem).
- Adapted files (or generic ones) for runtime configuration.
- Patches for all kind of components (to fix bugs or improve features).

In the following figure 2.2 we can see the PTXdist tool flow.

We can control every step of this flow with some commands. Most of them are described in pages 21-23 in [3].

Once Linux is running, the console input/output is made through serial port. It will only be used for debugging since the display of the game will be the screen of the board and the input of the game will be the gamepad. We use miniterm as the terminal to communicate with uClinux from the development computer, which uses the standard C runtime function printf to produce output on the console. Kernel modules will have to use their specific *printk* function to write in it.

2.2.3 Linux device driver (Kernel module)

In this exercise we will need to use device drivers because we cannot access hardware directly as we did in the previous exercises. The framebuffer, for the display, is already made so we can use it directly but we will have to create the one for the gamepad.

Using them

In unix systems, everything (including therefore drivers) is represented as files. When a program needs access to a driver, we have to open the file of the device driver (in the directory /dev), read or write from it and then close it. Of course, each driver has a specific meaning for the parameters we give and receive from it. There are some Linux functions to do such things. A good description of these operations files is given from page 49 to 52 in [1]. In this project, we will mainly use read, write, ioctl, mmap, open, release and fasync. To make the driver appear as a file in the /dev directory, the functions *class create* and *device create* must be called. When we exit the driver we can destroy them, as well as freeing memory and interrupts associated with them.

Fasync is used for asynchronous notification, by enabling it, this application can receive a signal whenever data becomes available and need not concern itself with polling [1], which is way more efficient. More information on it is available page 169 in [1]. We will use this concept again in the part about char device driver.

As said before, the framebuffer device driver is already made so we can directly call it from /dev/fb0. The figure 5.2 located page 52 in the compendium [4] describes the address organisation of the framebuffer for each pixel of our screen. In each of this addresses, we can give a 2-bytes value to determine the color of the pixel by giving the intensity of the blue, green or red color as described in figure 5.3 page 54 [4]. There are two ways to write pixels on the screen, the best in this exercise will be to use mmap because we can simply write pixel as if there were in an array instead of having to use the *lseek* function and then the *write* function to access the driver. mmap() creates a new mapping in the virtual address space of the calling process. We can give a lot of arguments but the easiest is to give the size of the virtual address space we want to create, the memory protection we wish and how we want to share the updates of this space with other processes. When we use mmap, we need to tell to the driver that we have made changes in the pixels content and where are these pixels by giving a rectangular containing them inside the address space. It is done by a call to the *ioctl* function. This function is more generally used to perform various types of hardware

control via the device driver as ejecting a media, changing a baud rate or refreshing the screen in our case. This function need to know which device driver we want to use and which specific command we want to use. The numbers corresponding to the command must be unique, we can define them according to the numbers already existing in the *ioctl-number.txt*[1].

Creating them

As said before, we will have to create a device driver for the gamepad as a Kernel module. The Linux Kernel Module run in Kernel mode, meaning they have access to everything unlike the user mode. We can link Kernel module dynamically (type *modprobe* in the terminal communciating with uCLinux) when we need them using some commands in the terminal so we do not have to recompile the Kernel each time. Init and exit functions for each module should therefore be implemented. In the init function we basically define the drivers as kernel modules, allocate and memory map access to the I/O registers we use, initialize the hardware and implement useful file operations as open, close, read and write. Kernel modules can only call functions from the Kernel space. They will be run together in parallel so we have to deal with shared ressources and avoid infinte loop inside of them. There are different types of driver in Linux but we will only use the 'char device' one, described in the next paragraph.

The char device driver

We develop a character driver because this class is suitable for most simple hardware devices.[1] To be sure that we do not access hardware at the same time as another driver, we need to call the function *request_mem_region* and *ioremap* builds new page tables without allocating memory even if we do not use virtual memory in this exercise. This function accesses physical memory through virtual address space, it allows us not to worry about where the addresses are located in the physical memory which is much more convenient as programmers.

Signals Once this step is done, I/O registers can be accessed as we did in the previous exercise. We will only read the state of the gamepad button in this one. We can read them using polling but this is not really optimized and we will therefore use signal handlers. We have to parameter how we use the signals using *sigaction*. We will use signals for both gamepad buttons and timer. If we define the structure associated properly, this function will link a handler to a signal. For the buttons, we use SIGIO signal as it is an I/O signal. For the timer we use SIGALRM signal, which is the timer signal from alarm, delivered when the timer expires if this one has been set with *setitimer* function (see man page for more information). Signal handling is vulnerable to race conditions. As signals are asynchronous, another signal (even of the same type) can be delivered to the process during execution of the signal handling routine. The *sigprocmask()* call can be used to block and unblock delivery of signals. Blocked signals are not delivered to the process until unblocked. Signals that cannot be ignored (SIGKILL and SIGSTOP) cannot be blocked [6]. As we have to care about two signals in this exercise (timer and

gamepad), we will have to block one signal when we want to 'listen' only to the other in a loop waiting for this signal. A good explanation on how to deal with it is given there [5], *sigsuspend* is the equivalent of sleepmode when we work with signals.

Interrupts To register the interrupts we need to call the *request_irq* function and we have to pay attention to the IRQ numbers given in the compendium, we have to use them. The rest of the initialisation of the GPIO interrupts (enabling, clearing flags...) is done the same way as the previous exercise, except we here need to use the *writel* function to access I/O memory.

Allocating Character Device Structures One of the first things your driver will need to do when setting up a char device is to obtain one or more device numbers to work with. This is made thanks to the function *alloc chrdev region*, which gives dynamicly-allocated device numbers. We basically need two numbers : the major number identifies the driver associated with the device while the minor number is used by the kernel to determine exactly which device is being referred to [1].

Registration of File Functions and Activation of the Driver To make it possible for the user program to handle the driver as a file, the driver has to implement support for access. This is done by implementing the following four functions in our module: open, close, read and write (even if in our case we do not really need the write function as we only want to read the state of the button). The open method is provided for a driver to do any initialization in preparation for later operations. The role of the release method is the reverse of open. The read and write methods both perform a similar task, that is, copying data from and to application code. More useful information on it can be found in [1] chapter 3. The kernel uses structures of type struct cdev to represent char devices internally. Before the kernel invokes our device's operations, we must allocate and register one or more of these structures. This is achieved thanks to *cdev* functions.

Compiling them

Writing a makefile for Kernel modules can be a bit tricky when we are not used to since it differs from the typical makefile for user-space applications. A good general structure is given page 24 in [1]

2.2.4 Linux user space program

They are managed by the Kernel and therefore do not have all the access that kernel has. This is why we needed drivers in the Kernel, the user space program canot access to he hardware directly. Our user space program here will be our game. It will be implemented in C and will make use of the two drivers we have at our disposal : gamepad and screen ones. We will have a dedicated C file dealing with graphics and another more specific to run our game. We launch the game by typing the name of its main file in the terminal communicating with uCLinux. We will get in this same terminal all the text output from the game. *gettimeofday* function allows us to know the current time. It is useful to know the time elapsed between two events, the moment when the button appears and the moment when one of the player presses the correct button.

3 Methodology

It was recommended in this exercise to use some parts of the code of the previous exercise like the gamepad or the sound driver. In this section we will for the most part focus on the things that were not implemented in the previous exercises.

We basically had to follow the recommended steps described in the compendium[4], to set up the project, configure the kernel, select the packages, build the distribution and flash it in the microcontroller. Once those steps were accomplished, we could boot uClinux on our microcontroller.

3.1 Development of the Linux kernel driver (Kernel module)

To create the kernel driver, we used the provided support files for this exercise and extended to insert the module into kernel space during load and clean up the module from the kernel space, when the module stops being used. These two functions are named __init gamepad_init and __exit gamepad_exit respectively which are called by the kernel.

If we go a bit into details, __init gamepad_init is implemented to:

- Request I/O memory when the kernel module is being loaded. By using request_mem_region, it tells the kernel that the driver will use the amount of range of I/O addresses. This prevents other drivers to call the same region.
- Dynamically allocate a device number of a character device by using alloc_chrdev_region. This will request a device number, and it will find an available starting major number.
- Set up the GPIO handler, so the buttons on the gamepad are registered when they are pressed. We couldn't use interrupts the way they were done in the previous exercise, because they now have to be allocated and initialized by the kernel. This time we implemented a function request_irq() to register the GPIO interrupt handler and enable the interrupt.

While __exit gamepad_exit is implemented to:

- Deallocate the reserved memory and unregister the GPIO interrupts when the module stops being used.

We also implemented a set of functions to perform file operations such as open, close, read and write. These functions gives the user program access to handle the driver as a file.

To make the userspace application notice that an interrupt has taken place, we had to implement an asynchronous notifier function, which basically works quite similar as interrupts. The driver requires registering a fasynq function, which is called when a process wants to be included as a listener. The interrupt handler will then send a signal to the listeners for each time there is an interrupt. The userspace application will then be able to handle the signal by a signal handler that will read the status of the buttons for each time it receives a signal from the driver.

3.2 Development of the game

As mentioned earlier, the game is based around a two-player game. The screen tells which button to press, and the first player to press the correct button will get a point. Figure 3.1a shows how the start screen looks like before the first round is started. At the top we can see the score for both players, separated by the side of the gamepad buttons the players are using. Right underneath we can see four circles which illustrates the four buttons on the gamepad the players are playing with. Figure 3.1b shows how it looks like after three rounds of playing. The score is 2-1 and the button to press on that point of time is the top button. In the middle of the four buttons, a timer is shown, for how long it took to press the last button. Figure 3.1c shows how the screen look like a player has reached five points. The timer has turned green color, and both the timer and the score for the player blinks until a button has been pressed to start a new game.

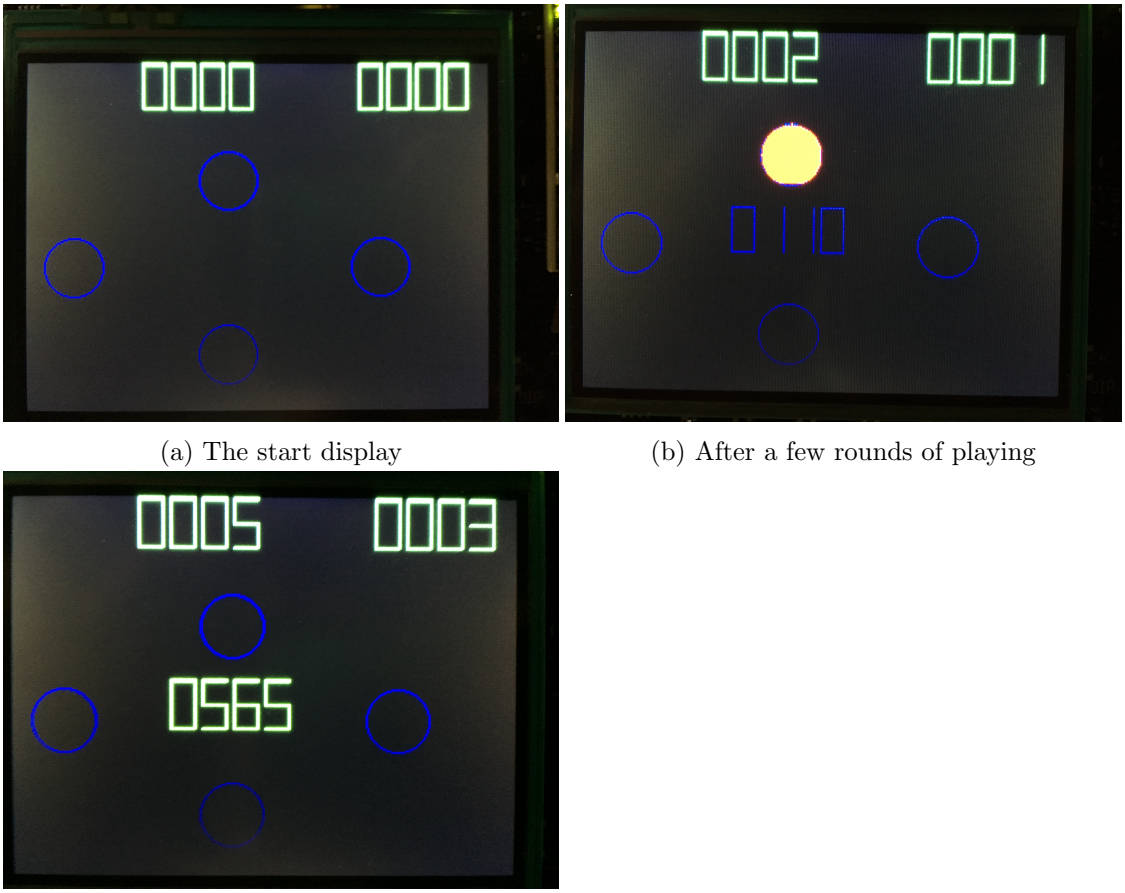
To create the functionalities for the game, we have separated it into two source files. The main file, game.c, does the actual handling towards the driver and controls the game itself. The second file, graphic.c, holds the functions to draw the graphics on the display either it is the score, timer or the circles.
During the startup of the game, first the display is initialized by the function init_display, which, as shown in listing 1, uses map pages of memory (mmap) to establish a mapping of the screen. Basically here we set the the size of the memory mapped, give read and write access and makes the memory shared for other processes.

```
screen = mmap(0, WIDTH*HEIGHT*2, PROT_READ|PROT_WRITE, MAP_SHARED, fb, 0);
```

Listing 1: mmap function in init_display

After finishing the display initilization, the intialization of the driver for the gamepad will start. The last step of the initialization part is to set up the signal actions for signal handler (towards the gamepad driver) and the timer handler.

When the initialization is done, it starts by calculating a random button the players have to press. The button chosen by the random generator is then turned into red color, and the screen is redrawn. The system will then start the timer and wait for a button to be pressed to compare it against the random generated button. This is simply done by the waitForButton() function which is waiting for a signal to be received. The actual data that is received is handled by the signal handler shown in listing 5.



(a) The start display

(b) After a few rounds of playing

(c) Finally a winner

Figure 3.1: The LCD screen of the development board

Note that we are using timer for two different purposes in this game. First of all the timer by the function waitForTimer() is waiting for a signal of when time has expired. The second timer is used to count the time consumed to press the right button.

By now we know the address of the button that is pressed, and we can now compare the buffer against the defined buttons for each player. The player who pressed the right button will at this point get its score increased by one. We update the score on the screen and start the next round of the game.

Figure 3.2 shows an activity diagram of the most important functions to make it easier to understand the game structure from initialization to game over.

```
void signal_handler(int signum){
    int bytes_read;
    printf("Received interrupt: %d\n", signum);
    bytes_read = read(file_gamepad_driver, &buf, 2);
    printf("Received: 0x%X\n", buf);
    printf("Bytes_read: %d\n", bytes_read);
    button_pressed = 1;
}
```

Listing 2: Receive the button address from the gamepad driver

```
int playerButtonPressed(int button){
    if (button == LEFT){
        if (buf == P1_LEFT)
            return P1;
        if (buf == P2_LEFT)
            return P2;
    }
    if (button == TOP){
        if (buf == P1_TOP)
            return P1;
        if (buf == P2_TOP)
            return P2;
    }
    ...
    return 0;
}
```

Listing 3: Decide which player pressed the right button

3.2.1 Drawing the graphic

To draw the content on the screen we are using the functions that are inside the graphic.c file. For instance we have functions to clear each element from the screen, we have the functions to draw the numbers and the functions to draw circles and so on. Since we couldn't find any good library to print out numbers to the display, we desided to create our own functions that prints out the numbers in 7-segment display format.

Just to take an example, in listing 5, we can see how our functions to draw the number 5 in seven-segment format looks like. The first line that is drawn is the top horizontal line in a 5 number. We can see that the value for both y0 and y1 stays the same. But from the point x to x+SEGMENT_LENGTH, it is drawn a line. The same applies to the other lines. All the five lines combined, results to a number five on the screen.

When we want to draw a circle into the screen, we use the function called draw_circle().

```
void draw_five (int x, int y) {
    draw_line (x, y, x+SEGMENT_LENGTH, y); //x0, y0, x1, y1
    draw_line (x+SEGMENT_LENGTH, y, x+SEGMENT_LENGTH, y-SEGMENT_LENGTH);
    draw_line (x+SEGMENT_LENGTH, y-SEGMENT_LENGTH, x, y-SEGMENT_LENGTH);
    draw_line (x, y-SEGMENT_LENGTH, x, y-2*SEGMENT_LENGTH);
    draw_line (x, y-2*SEGMENT_LENGTH, x+SEGMENT_LENGTH, y-2*SEGMENT_LENGTH);
}
```

Listing 4: Draw the number 5 in seven-segment format

This is of course done in a different way than drawing numbers. We start by drawing from the bottom left point of the circle and drawing upwards to the top right side of the circle.

```
void draw_circle(int xm, int ym, int r)
{
   update_rect(xm-r, ym-r, xm+r, ym+r);
   int x = -r, y = 0, err = 2-2*r;
   do {
      plot(xm-x, ym+y);                          /*   I. Quadrant +x +y */
      plot(xm-y, ym-x);                          /*  II. Quadrant -x +y */
      plot(xm+x, ym-y);                          /* III. Quadrant -x -y */
      plot(xm+y, ym+x);                          /*  IV. Quadrant +x -y */
      r = err;
      if (r <= y) err += ++y*2+1;                           /* e_xy+e_y < 0 */
      if (r > x || err > y)                /* e_xy+e_x > 0 or no 2nd y-step */
         err += ++x*2+1;                                   /* -> x-step now */
   } while (x < 0);

}
```

Listing 5: Draw the number 5 in seven-segment format

3.3 Testing:

We did some tests to check if the microcontroller is working as we want. We have provided some of the scenarios and the observations below.

Scenario 1: Press several buttons at the same time:
Expected result: During gameplay, only the first button pressed will count.
Observed result: The same as expected.

Scenario 2: Keep the buttons pressed for several rounds:
Expected result: If the player keep pressing one or several buttons, the next round will not detect the pressed buttons.
Observed result: The same as expected. The players will have to release buttons and press again, to make a signal to the game. No cheating!

Scenario 3: During redrawing before a new round, press a random button and check if it the player get a point for that (if the button is the one to press of course):
Expected result: The player will not get a point if the button is pressed too early. That is because the actual redrawing is done before the game starts to detect a signal from the buttons.
Observed result: After a decent amount of attempts, we can safely conclude that the player will not get a point if the button is pressed too early.

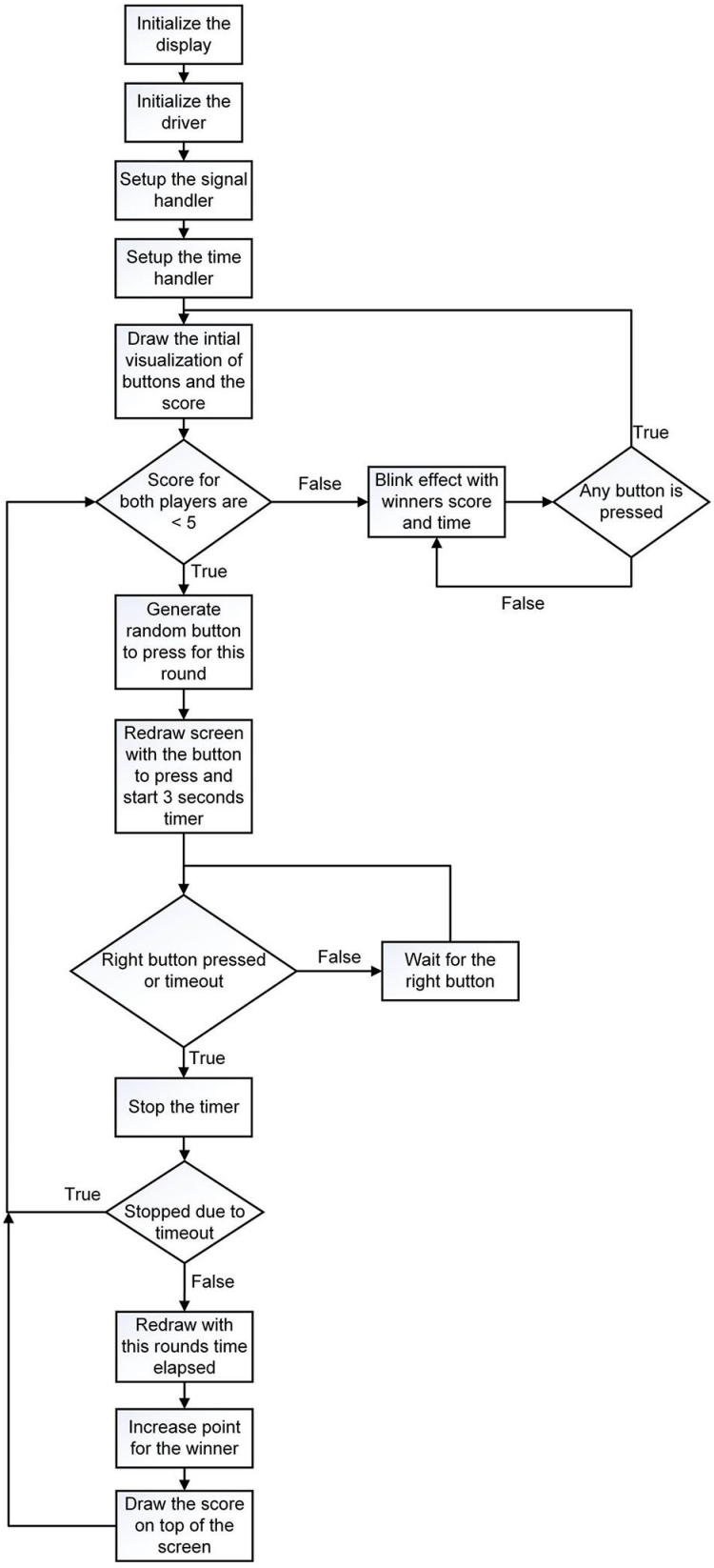


Figure 3.2: Activity diagram of the game playing

4 Results

Using the eAProfiler tool provided in the VirtualBox, we measured the power consumption while the development board was in different states. The results are shown in Table 1. The results are provided in from different measurement types. When we have measured a state for a longer time, we have used the average power consumption. But when we have measured a state for a short period of time, the amperage for the top of the spike is listed in the table.

Table 1

Description	Power consumption (mA)	Measure type
OS in idle state, driver is not loaded	10.42	Average
During driver loading into OS	29.53	Average
OS in idle state, driver is loaded	10.42	Average
During the game start up	32.29	Average
Game in idle state	10.44	Average
Draw a random button to press for the upcoming round	37.32	At most
Press the right button during a playing round	38.24	At most
Press a wrong button during a playing round	29.3	At most
When timeout after a round	30.08	At most

As we can see from the table, after the driver is loaded, the baseline power consumption is not increased. Also after the game is started up but is in idle state, the power consumption stays almost the same. As expected, when the button is pressed and while drawing something to the screen, the power consumption increases. As most the power consumption spikes to 38.24mA, which is when the right button is pressed during a round. This is the part where both redrawing and some calculations are done, so we expected this state to be the one with highest power consumption.

In our case, an eventually high power consumption wouldn't have so much effect, since we always have additional power source. But in cases where the microcontroller is running from external power sources, these results can be quite important to consider.

5 Conclusion

This exercise was the last of this project, the most challenging as well. We configured and built a Linux distro especially designed for embedded systems, we created a driver for the gamepad as a kernel module in addition to the framebuffer driver that we had at our disposal in order to create an original (and as fun as possible) game that can be controlled with the gamepad and displayed on the screen of our devkit. We used the concept of timing to enhance the basis of our game. The keys to achieve our goal were to carefully look for and find complementary information through books and website as the one we found in the compendium was not exhaustive. This research helped us to have a more thorough understanding of the complex concepts we needed in this project. Reading datasheet was not critical this time unlike the previous exercise. As usual, writing the technical report using the template in Latex, also was a useful experience, helping us to fully understand the concepts as we explained them.

5.1 Evaluation of the Assignment

This exercise was well made, and really challenging, the difficulty level and the time we needed to complete the assignment was a bit high for us.

Bibliography

[1] *Linux Device Drivers, Third Edition.* O'Reilly Media, 2005.

[2] *EFM32GG Reference Manual.* Silicon Labs, 2013.

[3] *How to become a PTXdist Guru.* Pengutronix, 2014.

[4] *Lab Exercises in TDT4258 Low-Level Programming.* NTNU, 2015.

[5] GNU.org. Using sigsuspend.

[6] Wikipedia. Unix signal.