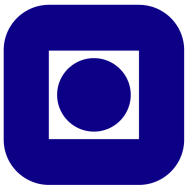

NTNU

TDT4258 - Energy Efficient Computer Systems

Exercise 3

Group 13

Mathias Ose & Øyvind Robertsen

April 27, 2014

Abstract

In this final exercise of the TDT4258 course, the group implemented a simple game on the EFM32GG development board, using the same gamepad peripheral as in the previous exercises to control the game. As in the previous exercise the game was implemented in the C language and an ARM compatible GNU toolchain was used to build and flash the development board. The difference from the previous exercise is theat for this one a Linux distro would first be flashed to the development board, and the game would be run inside this operating system. A dedicated driver was written to facilitate communication between the game and the peripheral, through the OS and the hardware GPIO.

Contents

1 Introduction

For the final exercise of the TDT4280 course, we will implement a simple game on the EFM32GG-DK3750. It will be controlled with the same GPIO peripheral as in the previous exercises. This time however, we will be running the game inside an operating system running on the development board. A custom driver will be created to handle the GPIO inputs, and the game will utilize this to trigger different events.

2 | Description & Methodology

This section describes the development process, use of tools, debugging techniques and details of the implementation. As listing entire implementations spread across several files for each development iteration would clutter the report and take away from the subject matter being discussed, we will only list code or formulas relevant to the task/section being described, not complete implementations. For details on the entire implementation, we refer to the source code attached to this report.

2.1 Development process

The development part of this exercise was, as with the previous ones, done at the computer lab in the IT-Vest building at NTNU (room ITV-458). The lab is equipped with workstations connected to the EFM32GG-DK3750 development board made by Silicon Labs. The subject staff provided us with a framework on which to base our work, more on this in section 2.2. As per usual, we put the project under version control immediately to ease collaboration. Seeing as this is a multipart exercise, collaboration aided by version control became an important aspect of the process to a larger degree than in the previous exercises.

As mentioned above, the exercise consists of several parts. The two major ones; implementing a device driver for the gamepad connected to the board and implementing a game using said driver as an input interface. We decided upon which game to implement soon after reading the exercise requirements. This proved to be beneficial in the development of the driver, seeing as we had a clear image of how we wanted to interface with the driver.

2.1.1 Devices

The devices used in this exercise are the same as in the previous exercises, the EFM32GG-DK3570 produced by Silicon Labs, and the gamepad peripheral connected to the development board via GPIO. For this exercise, a Linux distribution called uClinux is flashed to the developement board. uCLinux is Linux distribution targeting microcontrollers.

2.2 Project setup & toolchain

The framework provided by the subject staff for this exercise has grown in complexity in comparison to the handouts for the previous exercises, but in accordance with the complexity of the exercise requirements. The development in this consisted of two largely separate parts, the driver and the game. While both were implemented in C, the driver

was made as a kernel module, and the game as a regular user space application. Developing a kernel module requires a slightly different mindset than developing general user space programs. Some of the notable differences:

- The module must implement a predefined minimal set of functions in order for the kernel to know how to utilize the module.
- The module is restricted to calling only other functions defined in the kernel. For example, functions from the C standard library may no be used.
- The module must be event based. Polling for input using an infinite loop will cause the kernel to hang.
- The module must be compiled in context of the kernel version it is meant to be used in, in much the same way we have to cross-compile our C code for use on the ARM processor on the development board.

To ease development and automate some common and tedious build tasks, the PTXdist system is used in this exercise. PTXdist is a build system for Linux, giving us a way of reproducibly building our distro from source, with any desireable modifications to both the kernel and userspace.

2.2.1 PTXdist usage

Figure 5.1 in the subject compendium [1, p. 48] provides a good visual overview of the build process PTXdist facilitates. In short, PTXdist allows us to define simple, Makefile like rules dictating how each independent piece of the system is to be compiled and installed, as well as a simple way of defining how to run tests. The following is an overview of usage of the PTXdist utility.

Setup

Assuming the `ptxdist` utility and a suitable toolchain has been installed on the host system, the necessary configuration is quite straight forward. All `ptxdist` commands must be run from a directory chosen to be the the root directory of the project. In our case, this directory is the `OSELAS.BSP-EnergyMicro-Gecko`-directory in the exercise framework. First of all, we specify a platform configuration file and a project specific userland configuration. Secondly, a toolchain compatible with our target platform is specified. See listing 2.1.

Listing 2.1: Config

```
> ptxdist select <path_to_project_config>
> ptxdist platform <path_to_platform_config>
> ptxdist toolchain <path_to_toolchain_bin_dir>
```

For this project, both the platform and the project configurations where provided for us by the subject staff. The platform configuration file contains most of the platform specific architectual information PTXdist needs in order to compile and build for the device in question. (Processor architecture, processor endianness, presence of MMU, etc.) The userland configuration defines which applications will be built and how they

will be packaged. In most cases, one will use the `ptxdist menuconfig` and `ptxdist kernelconfig` to amend the configurations, as opposed to editing the files directly. The selected toolchain consists of the same tools and utilities utilized in the previous exercises (GNU CC, GNU LD, GNU AS, etc.)[6, p. 4], but in this case, a so called `arm-cortexm3-uclinuxeabi` version of the tools. This to ensure that all code is compiled and built in accordance with the ARM Cortex-M3 processor architecture and the uCLinux application binary interface.

Building the root filesystem

With the configuration out of the way, we can build the complete root filesystem with a single command `ptxdist go`. Using the information from the userland and platform configuration files, PTXdist now compiles and builds applications selected via `ptxdist menuconfig`, preserving dependencies and places all binaries in the correct directory in the root filesystem.

Creating flashable images

The next step is creating flashable images of all necessary parts of the system. The `ptxdist images` command accomplishes this. Images for, amongst others, the bootloader, the kernel and the root filesystem are created. It is also possible to specifiy wich images one wants to create, with the syntax `ptxdist image <image_name>`. During development, we utilized the command `ptxdist image root.romfs` to build only the image for the root filesystem. Compiling and creating an image for the kernel takes a lot of time, and is not required unless a change to the kernel configuration has been made.

Flashing

We were provided with two simple ways of flashing images to the development board by the subject staff. The first one `ptxdist test flash-all`, flashes all images, bootloader, kernel and root filesystem. During development, the `ptxdist test flash-rootfs` variant was used to flash only an updated root filesystem image.

2.2.2 Other tools

- Vim as main editor
- Git for version control, Github as a centralized repository host
- LaTeX for typesetting the report

2.3 Configuring and building Linux

As a first step in this exercise, we configured PTXdist and flashed all images to the development board, as outlined in section 5.3 in the compendium [1, p. 47] and section 2.2.1 above. This part of the exercise was fairly straight forward. See summary of steps in listing 2.2

Listing 2.2: Setup summary

```
> ptxdist select configs/ptxconfig
> ptxdist platform configs/platform-energymicro-efm32gg-dk3750/
    platformconfig
> ptxdist toolchain /opt/ex3/OSELAS.Toolchain-2012.12.0/arm-cortexm3-
    uclinuxeabi/gcc-4.7.2-uclibc-0.9.33.2-binutils-2.22-kernel-3.6-
    sanitized/bin
> ptxdist images
> ptxdist test flash-all
```

Having done this, we fired up `miniterm.py` and aquainted ourselves with the system. One of the first things we tried improving was the general development workflow. We tried amending a `printk` statement in the driver, to familiarize ourselves with the minimal set of steps necessary to get new code onto the board during development. The steps in listing 2.3 outline what has to be done to get any changes in the source for a package running on the development board. This is quite tedious, so we made a project wide Makefile simplifying the process. (Listing 2.4)

Listing 2.3: Development steps

```
> ptxdist clean <name_of_package>
> ptxdist compile <name_of_package>
> ptxdist image root.romfs
> ptxdist test flash-rootfs
```

Listing 2.4: Development Makefile

```
.PHONY: init build

init:
    ptxdist images
    ptxdist test flash-all

build:
    ptxdist clean game
    ptxdist clean driver-gamepad
    ptxdist go
    ptxdist image root.romfs
    ptxdist test flash-rootfs
```

2.4 Gamepad driver

Armed with knowledge from the previous exercises and some perception of how a device driver should be structured from reading Linux Device Drivers [3] and the compendium, we knew that we wanted our driver and it's interaction with userspace applications to be as interrupt based as possible. Regardless, having never written a driver before, we decided to follow an incremental approach, starting from the basics.

2.4.1 Hello World, from the kernel

The very basics of a kernel module are fairly straight forward. Listing 2.5 gives a brief overview of the different components that are required in a kernel module.

Listing 2.5: Basic kernel module

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>

// This function is called when the module is inserted into the kernel.
static int __init template_init(void)
{
    printk("Hello World, from the kernel!\n");
    return 0;
}

// This function is called when the module is removed from the kernel.
static void __exit template_cleanup(void)
{
    printk("Goodbye cruel world!\n");
}

module_init(template_init);
module_exit(template_cleanup);

MODULE_DESCRIPTION("Small module, demo only, not very useful.");
MODULE_LICENSE("GPL");
```

This in itself is quite selfexplanatory, so we quickly moved on to adding the functions required for having allowing a userspace application interface with the driver.

2.4.2 Char driver

A device driver written as a kernel module can expose itself as a regular file to the rest of the OS, typically in the `/dev/` directory. This is called a char driver. For a simple device like the gamepad, a basic example of interfacing would be to open the char device as a file in the application, and reading from it, as one would with any other file. To be able to do this, some boilerplate registration code is necessary in the kernel module. Amongst other things, functions corresponding to the various I/O operations have to be implemented and registered. First of all, we amended the `gamepad_init` function to begin implementing a char driver. See listing 2.6.

Listing 2.6: init function for char driver

```
static int __init gamepad_init(void)
{
    printk(KERN_ALERT "Attempting to load gamepad driver module\n");

    int result;

    /* Dynamically allocate device numbers */
    result = alloc_chrdev_region(device_nr, 0, DEV_NR_COUNT, DRIVER_NAME
        );

    if (result < 0) {
        printk(KERN_ALERT "Failed to allocate device numbers\n");
        return -1;
    }

    /* Request access to ports
     * It is recommended to request continuous memory regions,
     * but seeing as most of the addresses we require are
     * non-continuous, we do three separate calls.
     */
    /*
    if (request_mem_region(GPIO_PC_MODEL, 1, DRIVER_NAME) == NULL ) {
        printk(KERN_ALERT "Error requesting GPIO_PC_MODEL memory region,
             already in use?\n");
        return -1;
    }
    if (request_mem_region(GPIO_PC_DOUT, 1, DRIVER_NAME) == NULL ) {
        printk(KERN_ALERT "Error requesting GPIO_PC_DOUT memory region,
            already in use?\n");
        return -1;
    }
    if (request_mem_region(GPIO_PC_DIN, 1, DRIVER_NAME) == NULL ) {
        printk(KERN_ALERT "Error requesting GPIO_PC_DIN memory region,
            already in use?\n");
        return -1;
    }


    */
    //ioremap_nocache(GPIO_PA_BASE, GPIO_IFC - GPIO_PA_BASE);

    /* Init GPIO
     * For portability, these writes should be performed with
     * a base address obtained from the ioremap_nocache
     * call above and an offset.
     * What we are doing below is possible since we're not
     * using virtual memory.
     */
    iowrite32(0x33333333, GPIO_PC_MODEL);
    iowrite32(0xFF, GPIO_PC_DOUT);
    iowrite32(0x22222222, GPIO_EXTIPSELL);

    /* add device */
    cdev_init(&gamepad_cdev, &gamepad_fops);
    gamepad_cdev.owner = THIS_MODULE;
    cdev_add(&gamepad_cdev, device_nr, DEV_NR_COUNT);
    cl = class_create(THIS_MODULE, DRIVER_NAME);
    device_create(cl, NULL, device_nr, NULL, DRIVER_NAME);
    return 0;
}
```

In short, we allocate major/minor device numbers, request access to the area in memory the GPIO registers are mapped to, configure the GPIO (much like in the previous exercises) and register our char device with the kernel. We also added the corresponding deinitialization functionality to the `gamepad_exit` function.

At the top of our kernel module we added some function prototypes, a few constants and a file operations struct, as well as the `cdev` struct corresponding to the one used in the `init` function. (Listing 2.7) With all this in place, our char driver appears as `/dev/gamepad` after `modprobe`'ing it.

Listing 2.7: Definitions

```
/* Defines */
#define DRIVER_NAME "gamepad"
#define DEV_NR_COUNT 1

/* Function prototypes */
static int __init gamepad_init(void);
static void __exit gamepad_exit(void);
static int gamepad_open(struct inode*, struct file*);
static int gamepad_release(struct inode*, struct file*);
static ssize_t gamepad_read(struct file*, char* __user, size_t,
    loff_t*);
static ssize_t gamepad_write(struct file*, char* __user, size_t,
    loff_t*);
static irqreturn_t gpio_interrupt_handler(int, void*, struct pt_regs*);

/* Static variables */
static dev_t device_nr;
struct cdev gamepad_cdev;
struct class* cl;


/* Module configs */
module_init(gamepad_init);
module_exit(gamepad_exit);
MODULE_DESCRIPTION("Device driver for the gamepad used in TDT4258");
MODULE_LICENSE("GPL");
```

While the basics are in place, our driver would crash under usage at this point, as there is no implementation for the various functions called for different I/O operations. The implementations are outlined in listing 2.8. The only one of these functions that actually does anything interesting is `gamepad_read`. The bitstring representing which buttons are pressed is read from the `GPIO_PC_DIN` register and copied to a userspace memory buffer. We chose not to manipulate the bitstring at the driver level, to avoid imposing restrictions on the applications utilizing the driver.

At this point in the implementation, we were able to test the driver in the context it was meant to be used, as an input device. The next step was to make everything work with interrupts and signals.

Listing 2.8: File operations implementation

```
static int gamepad_open(struct inode* inode, struct file* filp)
{
    printk(KERN_INFO "Gamepad driver opened\n");
    return 0;
}

/*
 * gamepad_release - function called when closing
 *
 */

static int gamepad_release(struct inode* inode, struct file* filp)
{
    printk(KERN_INFO "Gamepad driver closed\n");
    return 0;
}

/*
 * gamepad_read - reads current button status from GPIO_PC_DIN
 *
 * Copies a decimal number representing the
 * bitstring of buttons pushed to user space buffer.
 */

static ssize_t gamepad_read(struct file* filp, char* __user buff,
        size_t count, loff_t* offp)
{
    /* Read gpio button status and write to buff */
    uint32_t data = ioread32(GPIO_PC_DIN);
    copy_to_user(buff, &data, 1);
    return 1;
}

static ssize_t gamepad_write(struct file* filp, char* __user buff,
        size_t count, loff_t* offp)
{
    printk(KERN_INFO "Writing to buttons doesn't make sense.");
    return 1;
}
```

2.4.3 Interrupts and signals

While the implementation we had at this point worked perfectly well, it forced our user space application to continuously poll the driver for new inputs, effectively blocking the process from doing other things. For instance, had the I/O between the driver and the application been non-blocking, we could tell the OS to suspend the process until a signal is received, potentially conserving energy.

Working with interrupts in device drivers is well documented in Chapter 10 of Linux Device Drivers [3, p. 258]. How to implement asynchronous notification is also discussed (Chapter 6, LDD [3]). We decided to start at one end and set up interrupts and handling on the driver side first, then move on to working on notifying userspace applications of the interrupt.

Interrupt generation and handling

Having the GPIO generate interrupts would work pretty much exactly as in the previous exercises (i.e. by writing to the correct registers). We would however have to be careful not to enable interrupts to in the initialization code and make sure that a proper handler was registered prior to enabling. A basic interrupt handler in a kernel module is a simple function, see listing 2.9

Listing 2.9: Base interrupt handler

```
irqreturn_t gpio_interrupt_handler(int irq, void* dev_id, struct pt_regs
    * regs)
{
    printk(KERN_ALERT "Handling GPIO interrupt\n");
    return IRQ_HANDLED;
}
```

Having implemented a handler, we moved on to registering said handler for the `irq_lines` for both odd and even GPIO interrupts in our init function. (Table 5.2, [1, p. 57]) We also added the necessary `iowrite` calls to enable GPIO interrupts. (Listing 2.10)

Listing 2.10: Initialize interrupts

```
/* Setup for interrupts */
request_irq(GPIO_EVEN_IRQ_LINE, (irq_handler_t)gpio_interrupt_handler,
    0, DRIVER_NAME, &gamepad_cdev);
request_irq(GPIO_ODD_IRQ_LINE, (irq_handler_t)gpio_interrupt_handler, 0,
    DRIVER_NAME, &gamepad_cdev);


/* Actually enable interrupts */
iowrite32(0xFF, GPIO_EXTIFALL);
iowrite32(0x00FF, GPIO_IEN);
iowrite32(0xFF, GPIO_IFC);
```

After testing that our code so far worked, we moved on to fleshing out the interrupt handler, and notifying the userspace application. Asynchronous notification works quite similarily to interrupts. An application registers itself as a listener with the notifier, which can then send a signal to all registered listeners whenever it wishes. On the driver side, this requires registering a so called `fasync` function, which gets called when a process wishes to add itself as a listener, as well as a `async_queue`, a list of all listeners. The interrupt handler sends a signal to all listeners each time there is an interrupt and the signal gets handled on the userspace side. (Outlined in listing 2.11)

Listing 2.11: Asynchronous notification

```
struct fasync_struct* async_queue;

static struct file_operations gamepad_fops = {
    .owner = THIS_MODULE,
    .open = gamepad_open,
    .release = gamepad_release,
    .read = gamepad_read,
    .write = gamepad_write,
    .fasync = gamepad_fasync,
};

/* Interrupt handler */

irqreturn_t gpio_interrupt_handler(int irq, void* dev_id, struct pt_regs
    * regs)
{
    printk(KERN_ALERT "Handling GPIO interrupt\n");
    iowrite32(ioread32(GPIO_IF), GPIO_IFC);
    if (async_queue) {
        kill_fasync(&async_queue, SIGIO, POLL_IN);
    }
    return IRQ_HANDLED;
}

/* fasync function */

static int gamepad_fasync(int fd, struct file* filp, int mode)
{
    return fasync_helper(fd, filp, mode, &async_queue);
}
```

On the userspace side, we implemented a signal handler that reads the current button status from the driver, each time a signal is received. More on this in section 2.5.2

2.5 Developing the game

While developing the gamepad driver, we also began prototyping the game. Since we had an existing implementation of the game [2] to reference, the game mechanics were fairly straightforward to implement.

As the entire source code of the game is quite large, this section will only discuss the more (in our opinion) interesting parts of it. The entire source code is attached to the report.

2.5.1 Game board data structure

The game is based around a simple 16-slot array representing a four by four grid of numbers. Button presses causes the numbers to move around and merge, and the updated array is translated to graphics that are shown on the gameboard screen.

Since the board was only to contain values of the form $v = 2^n$, it made sense to store n instead of v. Only the case of $n = 0$ is different, as these are not to be displayed as tiles with the value 1, but rather as empty tiles.

0	0	0	0	1	0	0	2	0	0	2	3	5	7	6	5

Table 2.1: An example of how the array `uint8_t b[16]` could look.

Figure 2.1: How the screen looks when `b` is as in table 2.1.

2.5.2 Gamepad input

Upon starting the game program, the gamepad driver listening is initialized by the function `init_gamepad`.

Listing 2.12: Initializing the gamepad driver listening

```
int init_gamepad()
{
    device = fopen("/dev/gamepad", "rb");
    if (!device) {
        printf("Unable to open driver device, maybe you didn't load the
            module?\n");
        return EXIT_FAILURE;
    }
    if (signal(SIGIO, &sigio_handler) == SIG_ERR) {
        printf("An error occurred while register a signal handler.\n");
        return EXIT_FAILURE;
    }
    if (fcntl(fileno(device), F_SETOWN, getpid()) == -1) {
        printf("Error setting pid as owner.\n");
        return EXIT_FAILURE;
    }
    long oflags = fcntl(fileno(device), F_GETFL);
    if (fcntl(fileno(device), F_SETFL, oflags | FASYNC) == -1) {
        printf("Error setting FASYNC flag.\n");
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

This sets up the program so that when the driver registers an interrupt, the function `sigio_handler()` is called.

Listing 2.13: Input handler function

```
void sigio_handler(int signo)
{
    int input = map_input(fgetc(device));
    switch (input) {
        case 1:
            left();
            break;
        case 2:
            up();
            break;
        case 3:
            right();
            break;
        case 4:
            down();
            break;
        case 6:
            if (last_input == 6) {
                new_game();
            }
            break;
        case 8:
            if (last_input == 8) {
                running = false;
            }
            break;
    }
    last_input = input;
}
```

The gamepad driver is accessed in the handler with `fgetc(device)`, which returns a bit string that represents the state of each of the eight buttons. For this game, we were not interested in any button press combinations, so we used the function we defined in exercise 2 to map the input to a number representing a single button pressed.

Listing 2.14: Input decoding function

```
int map_input(int input)
{
    input = ~input;
    for ( int i = 0; i < 8; i++) {
        int match = input & (1 << i);
        if ( (1 << i) == match ) {
            return (i+1);
        }
    }
    return 0;
}
```

The main function of the game has a loop that suspends at the end of each iteration. When the driver registers an interrupt it is handled by the handler function, then the main loop updates the screen contents and suspends again.

2.5.3 Using the development board LED display

As advised in the compendium, we used memory mapping to manipulate the image to be displayed.

Listing 2.15: Framebuffer initializing function

```
int init_framebuffer()
{
    fbfd = open("/dev/fb0", O_RDWR);
    if (fbfd == -1) {
        printf("Error: unable to open framebuffer device.\n");
        return EXIT_FAILURE;
    }

    // Get screen size info
    if (ioctl(fbfd, FBIOGET_VSCREENINFO, &vinfo) == -1) {
        printf("Error: unable to get screen info.\n");
        return EXIT_FAILURE;
    }

    screensize_pixels = vinfo.xres * vinfo.yres;
    screensize_bytes = screensize_pixels * vinfo.bits_per_pixel / 8;

    fbp = (uint16_t*)mmap(NULL, screensize_bytes, PROT_READ | PROT_WRITE
        , MAP_SHARED, fbfd, 0);
    if ((int)fbp == MAP_FAILED) {
        printf("Error: failed to memorymap framebuffer.\n");
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

This maps a region of memory, identified by the pointer `fbp`, to `/dev/fb0`, the interface for manipulating the development board LED display. Each 16-bit integer in this region represents the RGB565 color of a single pixel. With this we were now able to draw shapes by iterating over the memory region and writing different values. After changing pixels, the function `ioctl(fbfd, FB_DRAW, &screen)` is called to tell the screen to refresh the pixels in the `screen` area.

One thing we noticed and were confused by for quite some time was a black square region of 5-10 pixels that would not go away. It was eventually revealed to us that this was a cursor that uCLinux placed on the screen. We were also told that this could be removed by running the command `echo 0 > /sys/class/graphics/fbcon/cursor_blink` on the board.

2.5.4 Using fonts

With a game relying as strongly on text for both user feedback and main game elements as ours, we quickly realised we would need a well structured way of defining fonts. We also wanted to be able to use different font sizes. While we could have gotten away with defining some sort of array representation of each letter of a font in a header file, we also wanted to be able to edit fonts in any modern image editor. We started investigating various simple image formats to this end.

The first problem we bumped into, was how to get the image files we needed onto the root file system and transferred to the board. The compendium offered little to no guidance on the subject, but having familiarized ourselves with the PTXdist build system, we knew we would have to amend the `rules/game.make` file. We consulted the PTXdist application note; "How to become a PTXdist Guru" [4], and found what we were looking for. Listing 2.16 shows how we accomplished this.

Listing 2.16: Including images

```
RESOURCES_DIR   := local_src/$(GAME)-$(GAME_VERSION)/res
@$(call install_copy, game, 0, 0, 0755, $(RESOURCES_DIR)/font/font_large
    .pbm, /lib/$(GAME)/res/font/font_large.pbm)
```

We first attempted to use the BMP format, and wrote quite a bit of code parsing this format, but after several hours of banging our heads against the wall, we decided to look for something simpler. The BMP format contains a little too much information about things we would never have any use for, like color depth, color planes, print resolution, etc.

We decided to use the Portable bitmap format (.pbm). This is a much simpler format than the normal bitmap format (.bmp), yet offers everything we need. Files in the .pbm format have some header data first, then a sequence of 1s and 0s that represent a simple monochrome image. For instance, the data for a very simple .pbm image is shown in listing 2.17

Listing 2.17: PBM image

```
P1
# This is an example bitmap of the letter "J"
6 10
0 0 0 0 1 0
0 0 0 0 1 0
0 0 0 0 1 0
0 0 0 0 1 0
0 0 0 0 1 0
0 0 0 0 1 0
1 0 0 0 1 0
0 1 1 1 0 0
0 0 0 0 0 0
0 0 0 0 0 0
```

We procured some pixel fonts in .bmp format online, then converted them to .pbm and included with the flash.

Now, we needed the program to read the pbm file, parse the data and store it to memory. We created a simple struct to hold the data.

Listing 2.18: Portable bitmap format data struct

```
typedef struct {
    unsigned int x;
    unsigned int y;
    bool* data;
} pbm_image_t;
```

Refer to the attached source code for the `path_to_pbm` function in the file `font.c`. It is a bit too long to include in this report, but what it does is simply read a .pbm file and parse the sequence of 1s and 0s into the boolean array `pbm_image_t->data`.

Next up was to separate the data we now had in memory into individual boolean arrays representing individual character masks. Again we created a simple struct that could hold the boolean data for the character mask.

Listing 2.19: Character mask struct

```
typedef struct {
    char name;
    bool* data;
} char_t;
```

In addition we created another struct to hold the collection of different character masks (a font).

Listing 2.20: Font struct

```
typedef struct {
    unsigned int char_w;
    unsigned int char_h;
    char_t* chars;
} font_t;
```

The function `pbm_to_font` (also in `font.c`) iterates over the `pbm_image_t->data` array and creates `char_t` objects for the characters `' '` through `'z'` (ASCII ordering) and stores these in a single font. After this the memory holding the `pbm_image_t` data is freed.

2.5.5 Putting everything together

Now, with easy control of every single individual pixel and multiple fonts loaded into memory, we were ready to put everything together to make our game more or less equivalent to the web game.

The first step towards putting text on the screen was to put multiple character masks together into a single mask representing a string. This mask of multiple characters we called a glyph, and we created a function that would create a glyph from a string of arbitrary length and a font.

Listing 2.21: Glyph creation function

```
bool* create_glyph(char* str, int len, font_t* font)
{
    int glyph_w = len * (font->char_w);
    bool* glyph = (bool*)malloc(glyph_w*(font->char_h)*sizeof(bool));

    for (int i = 0; i < len; i++) {
        bool* chardata = font->chars[str[i]-' '].data;
        for (int y = 0; y < (font->char_h); y++) {
            for (int x = 0; x < (font->char_w); x++) {
                int glyph_i = glyph_w * y + i*(font->char_w) + x;
                int char_i = (font->char_w)*y + x;
                glyph[glyph_i] = chardata[char_i];
            }
        }
    }

    return glyph;
}
```

With this we could now draw the game board from the values in the array `b` (section 2.5.1).

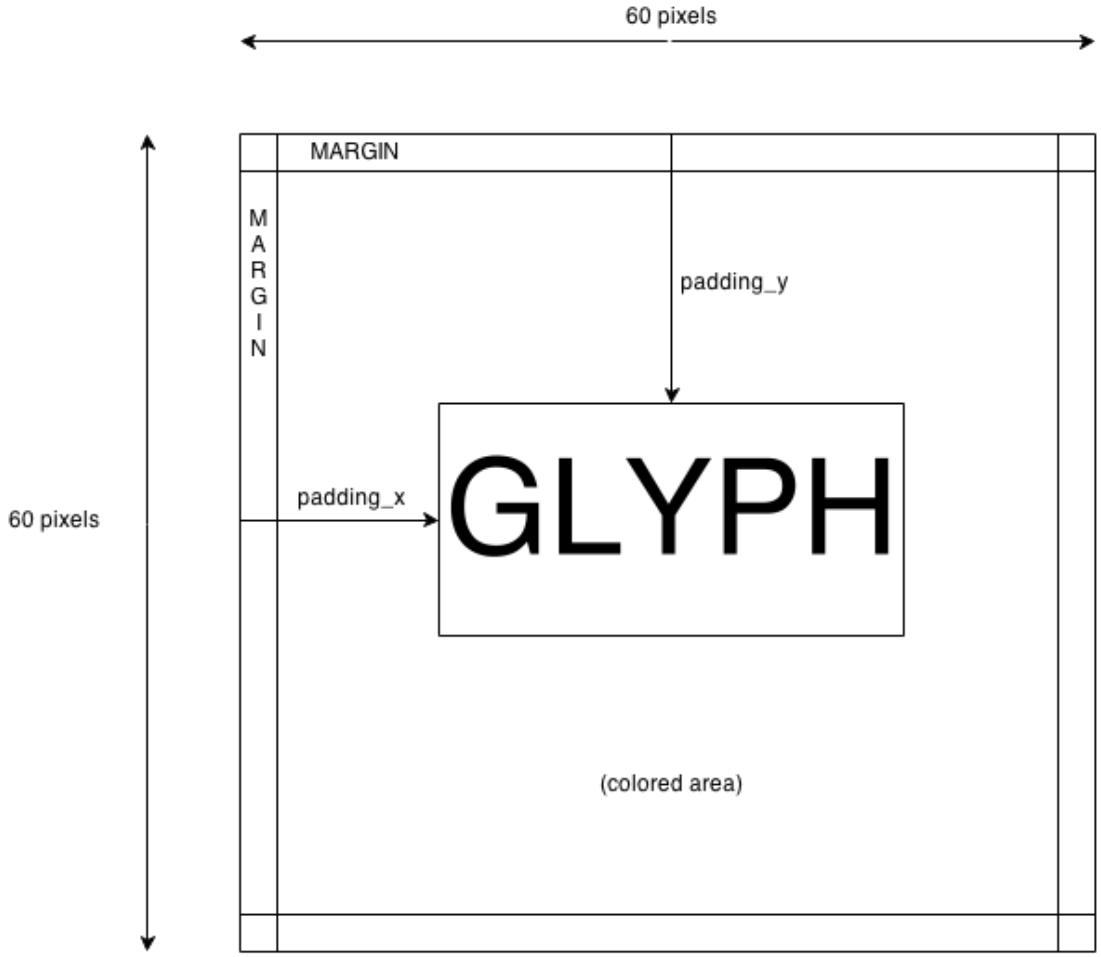

Figure 2.2: Our tile box model.

From the position in the array we calculated the 60x60 pixel area of the framebuffer corresponding, and in this we drew a square colored according to the value in the array.

In this square area we also center-positioned a glyph created from the string represenation of the tile value, and drew the pixels from this glyph in black "on top" of the colored tile.

Listing 2.22: Tile drawing function

```
void draw_tile(int pos, int val)
{
    int number = pow(2, val);

    int screen_offset_x = (60 * pos) % 240;
    int screen_offset_y = 60 * (pos / 4) - 1;

    int len = 0;
    if (val > 0) {
        int temp = number;
        while(temp) {
            temp = temp / 10;
            len++;
        }
    } else {
        len = 1;
    }

    char str[len];
    sprintf(str, "%d", number);

    int padding_y = (60 - (font->char_h)) / 2;
    int padding_x = (60 - len*(font->char_w)) / 2;

    bool* glyph = create_glyph(str, len, font);

    for (int y = MARGIN; y < 60 - MARGIN; y++) {
        for (int x = MARGIN; x < 60 - MARGIN; x++) {
            int screen_index = vinfo.xres*(y + screen_offset_y) + x +
                screen_offset_x;

            bool g = glyph[(y-padding_y)*len*(font->char_w) + (x-
                padding_x)];
            bool b = padding_y < y && y < 60 - padding_y && padding_x <
                x && x < 60 - padding_x;
            if (val != 0 && g && b) {
                fbp[screen_index] = Black; // glyph color
            } else {
                fbp[screen_index] = colors[val]; // tile bg color
            }
        }
    }
    free(glyph);
}
```

3 Results

3.1 Program

The program we implemented is a clone of the (at the time of writing) very popular web game 2048[2].

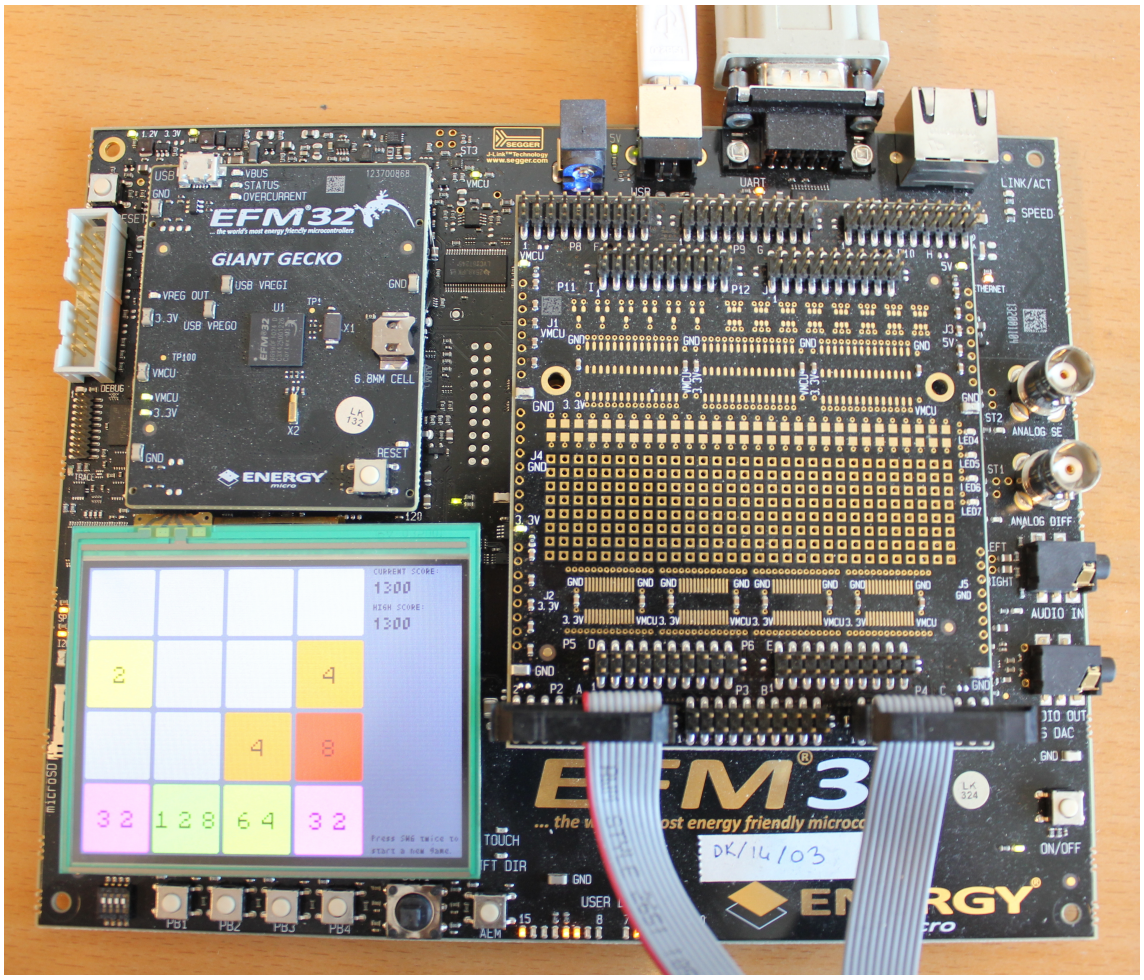

Figure 3.1: The game running on the development board.

3.1.1 Game rules

The game board consists of 4x4 slots. On game start two random slots will be filled with numbered tiles. The player must use the directional buttons (SW1 through SW4) to move the tiles. After each valid move, a new tile will be placed in a random empty

tile. When two tiles of equal value collide they merge into a single tile with value equal to the sum of its' parts. The game is "won" when the player is able to merge two 1024 tiles into a 2048 tile, although the player may continue the game and try to achieve as high a score as possible. Game over happens when the board is filled and no move is possible.

Score

When two tiles are merged, the resulting value of the merge is added to the current score.

3.1.2 Testing the program

All tests require possession of an EFM32GG-DK3750, a purpose built gamepad connected to the board on GPIO ports A and C and a computer capable of compiling for ARM based platforms and flashing software to the development board.

Setup

1. Navigate to the directory `OSELAS.BSP-EnergyMicro-Gecko`.
2. `> make init`
   Make will complain about an error 255, but everything is working as intended.
3. `> make build`
   Again, ignore error 255 when the build finishes.
4. While make build is running you may connect to the terminal in the OS now running on the development board.
   `> miniterm.py -b 115200 -p /dev/ttyS0`
   When `make build` is finished the board should reset and you should be prompted with the terminal input for the device.
   If you do not get a prompt to write in, try pressing the reset button on the development board.
5. In the device terminal you can load the gamepad driver with
   `> modprobe driver-gamepad`
6. After loading the driver the game can be started with
   `> game`
   The game is controlled with the gamepad buttons.
   Pressing SW8 twice exits the game and returns to the device terminal.

Playing the game

1. Use the directional buttons SW1 through SW4 and play the game according to the game rules as described in section 3.1.1.
2. When merging tiles, the score displayed on the right of the game board should increase.
3. When the current score increases beyond the high score, the high score should increase to equal the current score as well.
4. When game over happens the tiles should turn black and display "GAME OVER".

Restart game

1. At any point a new game can be started by pressing SW6 twice.
2. The game board should reset to contain only two tiles.
3. The current score should reset to 0.
4. The high score should not reset.

3.2 Energy efficiency

Using the eAProfiler tool, we took some readings of the running amperage while the development board was in different states.

State	Average amperage
OS idling, driver not loaded	10.25 mA
OS idling, driver loaded	10.25 mA
Game idling	10.25 mA
Button pressed in game	32.95 mA

Table 3.1: Amperage readings

As we can see, loading the driver does not cause the baseline power consumption to increase. The same goes for the game while nothing is happening, we see the power consumption stays in the same mA range within two decimal points. As expected, when a button is pressed, power consumption spikes. The spike lasts for about 252.5ms from it starts to rise until it has fallen completely, and while in the high state the amperage averages 32.95mA.

3.3 Discussion

3.3.1 Energy efficiency

The main advantage of our game compared to many other games, is that the program can be suspended between user input.

When updating the screen contents it would be possible to select only specific regions of the screen to be updated, which would save some energy. However in our case we opted to simply update the entire screen, justifying it with that the updating only happens after user input.

We looked into the possibility of using tickless idling to conserve even more energy, seeing as nothing in our game relies on timing. However, this was not prioritized and we were not able to get this to work before the deadline.

3.3.2 Driver

Given better time and fewer interruptions in the form of festive holidays, we could (read: should) have implemented our driver as a platform driver, ensuring portability.

In retrospect, we have also realised that it's unecessary to enable GPIO interrupts when the driver is initialized, since, at that point, there are no userspace applications available for handling the data. We could instead have enabled interrupts when the first application opens the driver and registers as a signal listener, and disabled interrupts when the last application has closed the driver.

3.3.3 Game

All core game features are implemented, so there's really not much to discuss here. One could argue that we could have spent more time on polish, animations, etc.

4 | Evaluation of assigment

Definently challenging, but that is to be expected for the final exercise. Getting to decide for ourselves which game to implement was very nice.

Some aspects of our implementation required lots of researching on our own to pull of, as it was not covered by the compendium.

Suggestions for things to add to the compendium:

- Disabling the cursor that uCLinux displays on the screen.
- Including other files with the game flash
- Some pointers about how to implement font support, like how the .pbm format is much easier to parse than .bmp.
- A better explanation of what PTXdist actually is/what problem(s) it attempts to solve.

5 Conclusion

The learning goals outlined in the exercise description were achieved. We successfully implemented the same game as the web version we were copying. We learned a lot about Linux device drivers, which we had no prior experience with, as well as a lot of aspects of memory management we had never had to consider before.

The progression from the first to this final exercise has been a very good learning experience. We can look back at the source code for the GPIO input handler from the first exercise and recognize that it is esentially the same thing we are doing in the device driver in the final exercise, just in another layer.

References

[1] Computer Architecture and Design Group. Lab exercises in tdt4258 energy efficient computer systems. Technical report, Department of Computer and Information Science, NTNU, 2014.

[2] Gabriel Cirulli. 2048. `http://gabrielecirulli.github.io/2048/`, April 2014.

[3] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. Linux Device Drivers. O'Reilly Media Inc., 2005.

[4] Pengutronix e.K. How to become a ptxdist guru. Technical report, 2012.

[5] Silicon Labs. Efm32gg reference manual, October 2013.

[6] Mathias Ose and Øyvind Robertsen. Exercise 2. Technical report, Department of Computer and Information Science, NTNU, 2014.

[7] Mathias Ose, Øyvind Robertsen, and Jørn-Egil Jensen. Exercise 1 - buttons and leds. Technical report, Department of Computer and Information Science, NTNU, 2014.