**NTNU – Trondheim**
Norwegian University of
Science and Technology

TDT4258 Energy Efficient Computer Design
Laboratory Report

# Energy Efficient Computer Game

*Group 24:*

Peter Aaser
Audun Sutterud
Jonatan Lund

November 11, 2015

## Abstract

This report covers a description about writing a videogame for the EFM32GG-DK3750 prototyping board running under the uClinux operating system. In addition, to access hardware, a custom device driver was written. We implement a clone of the game "Asteroids" running on top of the operating system, interfacing with our own custom drivers. The resulting performance is measured with different optimizations and work loads, showing that even with an optimized drawing routine the board spends 8 mA to draw the graphics for asteroids, out of a total 20 mA. These findings show that efficient drawing has a large energy reduction payoff, and that spending time implementing complex algorithms to limit screen update area is worth considerable effort.

# Contents

**1 Introduction** 5

**2 Background and Theory** 6
- 2.1 Basic Terminology 6
  - 2.1.1 Operating Systems 6
  - 2.1.2 Device Drivers 6
  - 2.1.3 Kernel Modules 6
  - 2.1.4 Related Terminology 7
- 2.2 Linux 7
  - 2.2.1 The Frame Buffer 7
  - 2.2.2 Linux System Calls 7
- 2.3 Writing Device Drivers for Linux 9
  - 2.3.1 Programming in the Kernel 9
  - 2.3.2 Loading and Unloading Modules 10
  - 2.3.3 A Simple Driver 10
  - 2.3.4 Device Files 11
  - 2.3.5 File Operations on a Device File 12
  - 2.3.6 Accessing Hardware Registers 13
  - 2.3.7 Handling Interrupts 14
  - 2.3.8 Handling Concurrency 14
  - 2.3.9 Kernel Data Types 15
  - 2.3.10 Handling Errors 15
  - 2.3.11 Mechanism versus Policy 15
- 2.4 Accessing the EFM32GG-DK3750 Hardware 15
  - 2.4.1 General-Purpose Input/Output pins 15
  - 2.4.2 The TFT-LCD Display 16
- 2.5 PTXdist 17
  - 2.5.1 Project Setup 17
  - 2.5.2 Building and Flashing 18

**3 Methodology** 19
- 3.1 Development Setup 19
  - 3.1.1 PTXdist 19
  - 3.1.2 Serial Terminal Emulator 20
  - 3.1.3 Git Version Control 20

3.2 Part 1: Linux Build and Warm-Up . . . . . . . . . . . . . . . . . . . . . . . 20
3.2.1 Building uClinux . . . . . . . . . . . . . . . . . . . . . . . . . . 20
3.2.2 Communicating with uClinux . . . . . . . . . . . . . . . . . . . . 20
3.2.3 Creating a framebuffer application . . . . . . . . . . . . . . . . . 21
3.3 Part 2: The Driver . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
3.3.1 Registering a Character Device . . . . . . . . . . . . . . . . . . . 22
3.3.2 Accessing the Driver . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.3.3 Creating the Device File Automatically . . . . . . . . . . . . . . . 22
3.3.4 Reading the Gamepad Buttons . . . . . . . . . . . . . . . . . . . 22
3.3.5 Implementing the Seek Operation . . . . . . . . . . . . . . . . . . 23
3.3.6 Registering an Interrupt Handler . . . . . . . . . . . . . . . . . . 23
3.3.7 Releasing resources when not needed . . . . . . . . . . . . . . . . 24
3.3.8 Buffering Gamepad Input . . . . . . . . . . . . . . . . . . . . . . 24
3.3.9 Implementing the Input Buffer . . . . . . . . . . . . . . . . . . . 25
3.3.10 Improving Device I/O Capabilities . . . . . . . . . . . . . . . . . 27
3.3.11 Asynchronous Notification . . . . . . . . . . . . . . . . . . . . . . 28
3.4 Part 3: The Game . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.4.1 Choosing a Game Concept . . . . . . . . . . . . . . . . . . . . . . 29
3.4.2 Development . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.4.3 Program Modules . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.4.4 Optimization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
3.5 Testing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

**4 Results 32**
4.1 Energy Efficiency . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.1.1 Energy Readings . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.2 Optimizations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

**5 Conclusion 34**
5.0.1 Compendium Feedback . . . . . . . . . . . . . . . . . . . . . . . . 34

# 1 Introduction

Embedded computing systems are being deployed on an increasingly larger scale, and it is expected that this trend will continue in the forseeable future. It follows that there is a great market for developing software for embedded computing systems. In contrast to general-purpose computing, embedded computing often places strong constraints on energy consumption, performance, and cost. In order to implement these qualities in a system, software developers must make extensive use of low-level hardware functionality, and specific knowledge of the hardware components and technology used is a requirement.

TDT4258 Energy Efficient Computer Systems is a university-level course at the Norwegian University of Science and Technology. The course gives an introduction to low-level microcontroller programming with a strong focus on energy-efficiency. Through a series of lectures and three comprehensive programming exercises, the students get theoretical knowledge as well as practical experience.[8] The target platform used is the Silicon Labs EFM32GG-DK3750 development kit, and some of the tools used are the GNU Compiler Collection, the PTXdist build system from Pengutronix, and the energyAware Tools from Silicon Labs.[3]

In the third exercise of the course the students make a computer game for the prototyping board. The game should run under the uClinux operating system, and all hardware must be accessed through device drivers. The game should also make use of a special prototype gamepad, and a gamepad driver must be written to access it. An important exercise goal is that the program should be energy-efficient. To improve energy consumption and to utilize the limited hardware resources well, the program must use appropriate algorithms and programming techniques, and may also need to use special hardware functions. To learn the required programming techniques, the students may have to study relevant theory on their own.[3]

# 2 Background and Theory

This chapter is an introduction to theory and concepts needed to understand the rest of the report. It begins with a section explaining operating systems terminology and a section introducing the Linux operating systems and important system calls. Then it continues with a section on driver programming in Linux. The next section introduces some relevant hardware of the EFM32GG-DK3750 prototyping board, and the final section provides an overview of the PTXdist build system.

## 2.1 Basic Terminology

This section introduces some general terminology and concepts concerning operating systems and device drivers in general.

### 2.1.1 Operating Systems

An operating system is a special piece of software that provides two important functions in a computer:

- Managing the hardware resources.
- Providing a useful hardware abstraction layer for application programmers.

The core of the operating system is called the *kernel* and runs in a privileged software that gives the kernel complete access to all hardware resources. The code running outside the kernel is often referred to as the *user space* or the *userland*, and has only restricted access to hardware. Because of this organization, all hardware-related activities necessary to run the operating system and execute programs is performed by the kernel, and the user space programs only access hardware through the kernel by special *system calls.* [9]

### 2.1.2 Device Drivers

The kernel uses *device drivers* to more easily manage hardware devices with different characteristics. A device driver is a program that manages low-level access to a particular hardware device, providing a clean interface for the rest of the kernel.

### 2.1.3 Kernel Modules

The functionality of the kernel may sometimes need to be modified, for instance to accommodate new hardware. While this could be achieved by modifying and rebuilding the kernel, it is often a more attractive alternative to extend the functionality with *kernel* *modules*, small kernel programs that are loaded at runtime. It is common to write device drivers as kernel modules.

### 2.1.4 Related Terminology

In addition to the concepts described above, several other terms are used in the context of operating systems:

**Boot Loader**
The bootloader is a small program that runs before the operating system starts, and makes the necessary preparations to load the kernel.

**Linux Root Filesystem**
In Linux, the root filesystem is the filesystem available at the top-level directory. It is denoted with a forward slash, `/`.

## 2.2 Linux

This introduces some relevant concepts of the Linux operating system, such as system calls and certain Linux devices.

### 2.2.1 The Frame Buffer

The frame buffer device is an abstraction that allows user applications to access the frame buffer of some underlying graphics hardware, and usually appears as `/dev/fb0`. The pixels displayed on the screen are stored in the frame buffer in a hardware-dependent format, and can be changed by writing to the frame buffer. Frame buffers are *memory devices* and can be mapped into process virtual memory with `mmap()` as explained in section 2.2.2.[2]

### 2.2.2 Linux System Calls

This section explains a few Linux system calls relevant in this report.

#### Reading and writing files

Several system calls in Linux are related to file operations that perform file I/O. This section will introduce some of the most commonly used. The following system calls are available from the headers `<fcntl.h>` and `<unistd.h>`. [1]

`int open(const char *pathname, int flags);`
Before any file operations can be performed on a file, a call to `open()` is needed to allocate the necessary system resources. Here `pathname` is the absolute path to the file, and `flags` specify access modes such as read-only or read-write. On success, `open()` will return a file descriptor that is used as a handle on the file in future system calls.

`ssize_t read(int fd, void *buf, size_t count);`

After opening a file, the contents of the file can be read as individual bytes with `read()`. Here `fd` is the file descriptor previously returned from `open()`, `buf` is a buffer where the file contents should be stored, and `count` is the number of bytes to read from the file. The returned value is the number of bytes that was read into `buf`. Note that `read()` gives no guarantee that the requested number of bytes has actually been read into `buf`. It is up to the application programmer to provide such guarantees by checking the return value.

`ssize_t write(int fd, const void *buf, size_t count);`

To write bytes to a file, the very similar function `write()` can be used. The parameters roles are slightly changed: `buf` is a buffer containing the contents to write to the file, `count` specifies the number of bytes to write, and the value returned is the actual number of bytes written. As expected, `fd` is the file descriptor.

`off_t lseek(int fd, off_t offset, int whence);`

Read and write operations on a file is performed at the bytes located at a specific offset from the beginning of the file, called the *file offset.* The file offset is initially zero, and is incremented with the number of bytes read or written by a file operation. To operate on bytes in a noncontiguous fashion, the offset can be changed with a call to `lseek()`. Here the meaning of `offset` depends on the parameter `whence`. Three possible options are to specify the new offset directly, relative to the current offset, or relative to the end of the file.

`int close(int fd);`

After an application is done with a file it should call `close()` to release the file's associated resources. This will invalidate the file descriptor.

#### Mapping files to memory

Accessing a file in an arbitrary manner using the system calls described in section 2.2.2 will involve many redundant calls to `lseek()`. A better alternative then is to map the file into the virtual address space of the process, allowing the file to be accessed with ordinary memory accesses. The following system calls are available from the headers `<sys/mman.h>` and `<unistd.h>`. [1]

`void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);`

The function `mmap()` can be used to map a number of `length` bytes from the file specified by the file descriptor `fd` into memory. The mapping starts `offset` bytes from the beginning of the file. Optionally, `addr` can be used to give the kernel a hint as to where the mapping should be placed in virtual memory, or it can be `NULL`. `prot` is a bit field specifying the memory protection of the mapping, and `flags` sets the visibility of updates to the mapping to other processes.

`int msync(void *addr, size_t length, int flags);`

Updates to a memory mapped file might not become visible immediately. To synchronize a file with a memory map, the application should call `msync()`.

`int munmap(void *addr, size_t length);`
: After the application is done with the memory map, it may call `unmap()` to delete the mapping and release its associated resources. This will also synchronize the file with the memory map.

## 2.3 Writing Device Drivers for Linux

This section is an introduction to device driver development for the Linux kernel. It begins by explaining some practical details regarding driver development, then it highlights some important parts of the Linux kernel and common programming constructs, and finally it provides a brief outline on driver design. The main focus is on drivers implemented as kernel modules. For an explanation on the terminology used in this section, see section 2.1. This section relies heavily on the book *Linux Device Drivers* (see [4]) as a source of information on driver programming and the Linux kernel.

### 2.3.1 Programming in the Kernel

Kernel modules are different from user space applications in several important ways, and there are some considerations to take into account when writing a module.

#### Module bugs are more serious

As a module is a part of the kernel, it runs in privileged mode with access to all hardware. A natural consequence of this is that bugs are much more serious. A module bug can easily crash the whole system, for instance by overwriting other parts of the kernel code in memory. Or, perhaps worse, a bug might introduce a security hole.

#### Modules cannot link user space code

Again, because a module runs in privileged mode, it cannot be linked against any user space code. This includes the standard C library with all its nice functions such as `malloc()`, `strcmp()` and `printf()`. Instead, the module is linked against the kernel and has access to kernel routines.

#### Module debugging is different

Debugging a module is different from using ordinary software debuggers such as the GNU Debugger. The Linux kernel must be debugged with the built-in kernel debugger.

#### Modules must be linked against the kernel source tree

Since modules are linked against the kernel, the developer must have a kernel source tree available to build it against. The process of manually downloading, configuring and building a kernel source tree involves a fair amount of work, especially for a novice module programmer.

### 2.3.2 Loading and Unloading Modules

As explained in section 2.1.3, the kernel functionality can be extended by modules loaded at runtime. The following two commands can be used to load and unload modules:

`modprobe <module>` will load and initialize a module along with all its dependencies.

`rmmod <module>` will perform cleanup and unload a module.

The modules control their own initialization and cleanup during loading with special initialization and cleanup functions. A module must notify the kernel of these functions by using the kernel macros `module_init()` and `module_exit()`, as shown in section 2.3.3.

### 2.3.3 A Simple Driver

The following example program illustrates the basic structure of a driver module:

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

static int hello_init(void)
{
  printk(KERN_ALERT "Hello,␣world.\n");
}

static void hello_exit(void)
{
  printk(KERN_ALERT "Bye␣bye␣world,␣nice␣meeting␣you.\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

When this module is loaded, the initialization function uses `printk()` to print a message to the system log and any open terminals. Similarly, a message is also printed when the module is unloaded. Notice the use of `module_init()` and `module_exit()` at the last two lines: these are kernel macros that indicates to the kernel which functions to use for initialization and cleanup. The included header files are included by almost all modules. Note that they are not system headers, but headers from the kernel source tree.

### 2.3.4 Device Files

In Unix-like operating systems, user space applications access drivers through so-called *device files* or *special files*. An application communicates with a driver through one of the driver's device files by using system calls for file operations, such as those described in section 2.2.2. Notably, this is true for all drivers, even though not all drivers provide access to storage devices. The driver provides its own implementation of the file operations it needs to support, and this code constitutes a core part of the driver.

#### The `/dev` directory

The device files are conventionally placed in the `/dev` directory. Each device file in this directory contains a *device number* that uniquely identifies the device file and the driver that owns it, composed of a *major* and a *minor* revision number. Drivers often use multiple device files to provide different services.

#### Allocating device numbers

Before any device files can be created for a driver, the driver needs to obtain a range of device numbers. The recommended approach here is to request dynamically allocated device numbers from the kernel with the following function:

```
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor,
        unsigned int count, char *name);
```

If successful, this will return a range of major-minor device numbers in the memory pointed to by `dev_t`. The parameter `firstminor` specifies the first minor device number, `count` is the total number of minor device numbers, and `name` is the name of the device. The device numbers are freed with:

```
void unregister_chrdev_region(dev_t first, unsigned int count);
```

#### Creating device files

After device numbers have been allocated, a device file needs to be created to make the device file visible to user space applications.

A device file can be created by the device driver itself at runtime. First the driver needs to call the function:

```
struct class* __class_create (struct module *owner, const char *name,
          struct lock_class_key *key);
```

This will return a pointer to a newly created `struct class` needed in the next function call. Here `owner` is the module that owns the device file, and `name` is the name of the class. The parameter `key` is not needed for simple use, and can be passed as `NULL`. The next function creates the device file:

```
struct device* device_create ( struct class *class, struct device *parent,
          dev_t dev, void *drvdata, const char *name, ...);
```

This will return a pointer to a `device` structure associated with the newly created device file. Here `class` is a pointer to the class returned by `__class_create()`, and `dev` is the device numbers of the module. The file name of the device file is `name`, and can be chosen freely. For simple use, `NULL` can be passed the parameters `parent` and `drvdata`, and the variable arguments can be omitted. When a driver is finished using the `class` and `device` structures, it should destroy them with `class_destroy()` and `device_destroy()` respectively.

In addition to creating device files at runtime, it is also possible to create the device file manually by issuing the command:

```
mknod /dev/<device name> c <major rev> <minor rev>
```

Here `<major rev>` and `<minor rev>` should correspond to a pair of major and minor revision numbers owned by the driver. The major revision number can be read from `/proc/devices`, and the minor revision number can usually be inferred.

### 2.3.5 File Operations on a Device File

Having explained in section 2.3.4 that drivers are accessed through file operations that they define on a device file, we shall now look closer at how those file operations are actually defined, introducing the structures `file` and `file_operations`.

#### File management in the kernel

To manage an open file, the kernel uses the structure `struct file` from the kernel header `<linux/fs.h>`. Some important fields of this structure are listed here:

`mode_t f_mode;`
: `f_mode` is a bit field that stores the access mode of the file in the two bits `FMODE_READ` and `FMODE_WRITE`.

`loff_t f_pos;`
: `f_pos` is a 64-bit integer storing the current offset into the file.

`struct dentry *f_dentry;`
: `f_dentry` points to a structure storing the directory entry of the file.

`struct file_operations *f_op;`
: `f_op` points to a structure containing file operations to be used with the file.

The important thing here is the `file_operations` structure associated with files through the pointer `f_op`. The `file_operations` structure contains pointers to the file operations of a file, and is mainly where special files differ from ordinary files: in a special file, the `f_op` pointer will point to a `file_operations` structure defined by the driver. Some of the important fields of the `struct file_operations` structure is listed below:

`int (*open) (struct inode *, struct file *);`
: Pointer to a function that is called whenever `open()` is called on the file.

`ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);`
: Pointer to a function that is called whenever `read()` is called on the file.

`ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);`
: Pointer to a function that is called whenever `write()` is called on the file.

`loff_t (*llseek) (struct file *, loff_t, int);`
: Pointer to a function that is called whenever `lseek()` is called on the file.

These correspond to the file operations described in section 2.2.2. It is by initializing the function pointers in this structure that a driver makes functionality available to user space applications.

#### The `inode` structure

The function pointer to `open()` in the `file_operations` structure identify a file using a `struct inode`. The `inode` plays a different role than the `file` structure: while the `file` structure stores information about an open file, the `inode` structure stores information about a file stored on the disk. For a device file, it will contain the following two important fields:

`dev_t i_rdev;`
: The device number.

`struct cdev *i_cdev;`
: A pointer to a structure the kernel uses to manage character devices.

### 2.3.6 Accessing Hardware Registers

Linux uses the concept of *I/O ports* to manage access to peripheral devices. To access a set of registers, a process must first allocate the corresponding I/O ports through the special kernel call `request_region()`, defined in `<linux/ioports.h>`. When done, the process should release the I/O ports with `release_region()`.

To read and write registers directly, the kernel defines the functions such as `inb()` and `outb()` to read or write a register of a specific size. The I/O port is passed as a parameter to specify the target register.

Registers can also be mapped directly into physical memory using `ioremap()`. This function will return a pointer to a memory region that corresponds directly to the hardware registers. The similar function `ioremap_nocache()` can be used to turn off caching of the memory region. To release a memory-mapped region, the function `unmap()` should be called.

Code optimizations may be an issue when writing to registers. Specifically, memory operations might be reordered by either compiler or hardware, changing the order the registers are written and causing unwanted side-effects. To impose ordering on memory operations, the programmer should call kernel functions that act as memory barriers, such as `barrier()` (compiler memory barrier) and `mb()` (hardware memory barrier).

### 2.3.7 Handling Interrupts

To communicate efficiently with hardware, the driver must make use of hardware-generated interrupts.

#### Setting up interrupts

Setting up interrupts is done in two steps. The driver requests an interrupt channel with the function `request_irq()`, specifying a handler routine that should be called to handle interrupts:

```
int request_irq(unsigned int irq,
  irqreturn_t (*handler)(int, void *, struct pt_regs *),
  unsigned long flags, const char *dev_name, void *dev_id);
```

Here `irq` is the platform-specific irq number of the interrupt channel, and `handler` is the handler routine that should handle interrupts. After a handler is registered and the driver is ready to receive interrupts, the driver should enable interrupt-generation in the hardware.[4]

#### Considerations when writing an interrupt handler

There are some special considerations when writing an interrupt handler. An interrupt handler should finish quickly. This avoids missing interrupts, but also avoids stalling the system in a handler that has disabled interrupts. In addition, because interrupt handlers do not run in the context of a process, interrupt handlers cannot transfer data to or from user space, and cannot do anything that would cause them to sleep, like locking semaphores.

An interrupt handler that needs to perform a lengthy amount of work, or some work that might require sleeping, should be split into *top and bottom halves*. The top half would finish quickly, scheduling the rest of the work to run in a *tasklet* or *workqueue* function at a safer time.[4]

#### Monitoring Interrupts

Interrupts on a system can be monitored in the file `/proc/interrupts`.

### 2.3.8 Handling Concurrency

The driver is shared between the processes running on a system, and must correctly handle the scenario that two or more processes wants to use the driver at once. To share the driver between multiple processes, it might be necessary to manage resources on a per-process basis using data structures. In addition, the resources shared between the processes needs to be sufficiently protected with synchronization primitives to avoid race conditions.

As a driver registers its facilities, other parts of the kernel might try to use those facilities immediately. The driver needs to handle this case that some parts of the kernel try to use it before it is completely initialized.

### 2.3.9 Kernel Data Types

The Linux kernel defines several data types, in addition to the standard C types. Some of these types make it easier to write portable code, like the type `u32` and friends, and some types implement useful functionality, such as the linked list implementation.

### 2.3.10 Handling Errors

When a driver registers a facility during initialization, there is no guarantee that the kernel is able grant the driver's request. The driver might still recover from such an event to provide either full or some partial service, but in the event that the driver is not able to continue, it should unregister all facilities previously registered successfully. This may require that the driver uses some sort of scheme to track which facilities was registered successfully and which failed.

### 2.3.11 Mechanism versus Policy

In linux it is a desire to split mechanism and policy when writing a driver. This means a driver should offer some capability, but it should not decide how those capabilities should be used. For instance a gamepad driver should simply offer a memory area from which button inputs can be read, and a notification when new input is availible.

## 2.4 Accessing the EFM32GG-DK3750 Hardware

The target platform used in the exercise was an EFM32GG-DK3750, from here on referenced as EFM32GG, running the uClinux operating system. This section introduces those parts of the hardware that we needed to access directly when writing drivers. When referring to hardware registers, we shall refer to their corresponding names as specified in the EFM32GG Reference Manual, [7].

### 2.4.1 General-Purpose Input/Output pins

The General-Purpose Input/Output (GPIO) pins makes it easy to connect peripherals to the EFM32GG. The pins are organized into ports of 16 pins each, numbered from A to F. It is possible to configure each pin individually for either input or output, in addition to configuring more advanced features such as drive-strength or pull-up resistors.

#### Setting up Port C for gamepad input

To set up GPIO port C to receive input from the gamepad, some of the port C registers needed to be written:

`GPIO_PC_MODEL`
: This register is used to individually configure the mode for each of the lower pins of GPIO port C. The register can configure the pins for either input or output, as well as doing more advanced configuration. To configure the lower pins for gamepad input, the value `0x33333333` is written to the register.

`GPIO_PC_DOUT`
: This register has a function depending on what is written to `GPIO_PC_MODEL`. To configure the lower pins for gamepad input, the value `0xff` is written to the register.

#### Setting up gamepad interrupts

To set up the GPIO hardware to generate interrupts when a gamepad button is pressed or released, some of the GPIO interrupt registers needed to be written:

`GPIO_EXTIPSELL`
: This register selects which of the GPIO low pins should generate interrupts. For each pin number, the register specifies the port that should generate an interrupt on that pin. To set up the port C lower pins to generate interrupts when the gamepad buttons are pressed, the value `0x22222222` is written to the register.

`GPIO_EXTIFALL`
: This register is used to specify for each low pin whether interrupts should be generated on a transition from 1 to 0. It must be set to `0xff` to enable interrupt generation on gamepad button presses.

`GPIO_EXTIRISE`
: This register is used to specify for each low pin whether interrupts should be generated on a transition from 0 to 1. It must be set to `0xff` to enable interrupt generation on gamepad button releases.

`GPIO_IEN`
: This register enables interrupt generation individually for both low and high pins. It must be set to `0xff` to enable interrupts for each of the low pins.

`GPIO_IFC`
: This register is written to clear interrupt flags. An interrupt handler for the gamepad needs to write to this register to `0xff` to clear the interrupts.

### 2.4.2 The TFT-LCD Display

The EFM32GG has a 3.5-inch TFT-LCD display with 320x240 pixels that we will use for our game. The display can be driven by the integrated SSD2119 controller.

#### The TFT-LCD Display Driver

We will access the display through a driver written by Asbjørn Djupdal in 2013. This driver is very simple: apart from the default ioctl calls it only implements a single ioctl call to update a rectangular area of the screen. It actually implements file operations to perform several other drawing operations that would be useful for applications needing faster graphics, but these have no ioctl and therefore cannot be used.

For a description of general use of Linux framebuffers, see section 2.2.1.

## 2.5 PTXdist

PTXdist is a build system used to develop userland software for embedded platforms running Linux. It can control most parts of the build process:

- Download package sources
- Extract package sources and apply patches
- Configure packages
- Build packages
- Create filesystem images
- Flash filesystem images to target

PTXdist builds the Linux kernel used along with the userland software.

### 2.5.1 Project Setup

Setting up a PXTdist project consists of the following steps:

- Selecting a userland configuration.
- Selecting a hardware platform.
- Selecting a toolchain.

The userland configuration defines what user programs will be built for the target platform apart from the kernel, and the hardware platform defines the target platform of the build. The toolchain selected is the toolchain PTXdist should use when building sources. As PTXdist looks for toolchains in the /opt folder, selecting a toolchain is not always necessary.

### 2.5.2 Building and Flashing

After the project has been setup, the kernel and the user programs can be built with one simple command:

```
ptxdist go
```

This will download, extract and compile all needed sources, in addition to installing the sources into a file system for the target located at the host. This command may take some time to execute the first time, but will reuse the installed sources later.

The next step is to generate filesystem images for the target:

```
ptxdist images
```

This command will create several filesystems for the target platform containing kernel, bootloader, and all userland programs specified by the userland configuration.

Finally, the images can then be flashed onto the target platform using platform-specific software, such as eACommander for EFM32GG. An illustration of how the PTXdist build process might be for a computer game is shown in figure 2.1.

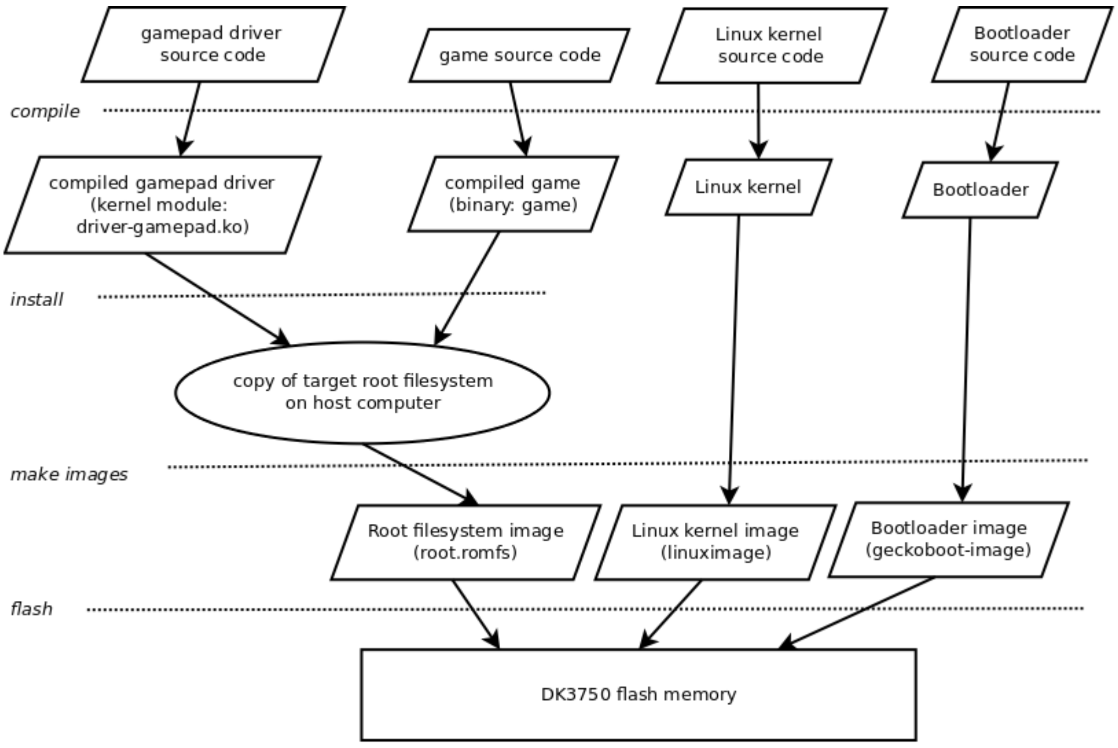
gamepad driver source code
game source code
Linux kernel source code
Bootloader source code
compile
compiled gamepad driver (kernel module: driver-gamepad.ko)
compiled game (binary: game)
Linux kernel
Bootloader
install
copy of target root filesystem on host computer
make images
Root filesystem image (root.romfs)
Linux kernel image (linuximage)
Bootloader image (geckoboot-image)
flash
DK3750 flash memory

Figure 2.1: An example overview of a PTXdist build process for a computer game.

# 3 Methodology

This chapter explains the complete process that lead to our final product, and the details of the implementation. It begins by explaining the development setup we used, and then goes through the process step by step. The exercise compendium recommended to complete the exercise sequentially in three parts:

1. Part 1: Linux Build and Warm-Up
2. Part 2: The Driver
3. Part 3: The Game

We tried to follow this approach closely. This chapter dedicates an individual section to each part.

## 3.1 Development Setup

This section briefly describes technical details regarding the system setup we used on our development computers.

### 3.1.1 PTXdist

The build process of this exercise was quite involved, and for this reason it was necessary to use the PTXdist build management tool, described in section 2.5. In order to install PTXdist on our computers, two options were provided along with the exercise:

- A virtual machine with PTXdist and cross toolchain installed.
- A README file with instructions on how to build and install PTXdist and cross toolchain from sources.

Initially we used a computer with limited processing resources for development, and building PTXdist from sources was therefore the favoured alternative. By following the instructions, PTXdist and cross toolchain was installed successfully in a couple of hours.

A possible third option was to use the lab computers with PTXdist and cross toolchain already installed. This option was regarded as a fall-back solution in case of technical difficulties.

### 3.1.2 Serial Terminal Emulator

In order to access the running operating system, we connected a laptop running a serial terminal emulator to the serial port of the EFM32GG. The program used for terminal emulation was *minicom*, an open source communication program available for most Unix and Unix-like operating systems.[6] According to the compendium, the serial port was configured with the following parameters:

- Baud rate: 115200 bps
- Data bits: 8
- 1 stop bit
- No parity
- No flow control

### 3.1.3 Git Version Control

To manage different code revisions, we used the Git version control system. The free hosting service GitHub was used to host a Git repository at `https://github.com/jonatanl/TDT4258`. During exercise 3, code was developed on the branch “master”, and the report was written on branch “report3”.

## 3.2 Part 1: Linux Build and Warm-Up

When solving the exercise, we started out by doing the preliminary warm-up steps recommended by the compendium: building the uClinux operating system, and creating a simple application that used the framebuffer device `/dev/fb0`. The purpose of these steps was to familiarize with the development workflow and driver interfacing in Linux.

### 3.2.1 Building uClinux

We used PTXdist to build uClinux by executing the commands `ptxdist images` and `ptxdist test flash-all` in the top-level project directory. A large amount of sources were downloaded and compiled, and the whole process took about fifteen minutes. PTXdist and its build process is described in section 2.5. After the board was restarted, Tux appeared on the display showing that uClinux was running.

### 3.2.2 Communicating with uClinux

In order to communicate with the operating system, we configured the serial terminal emulator *minicom* as described in section 3.1.2. This gave us access to a shell where we could execute commands. The output of executing the unmodified game application is shown in listing 3.1.

```
OSELAS(R)-Gecko-2012.10.0+ / energymicro-efm32gg-dk3750-2012.10.0+
ptxdist-2013.07.1/2015-03-16T15:23:07+0100

/ # game
Hello World, Im game!
```

Listing 3.1: The original game application

To further verify that our code was actually compiling, we made a small modification to the game application by changing the `printf()` function arguments in `game.c`. After rerunning the necessary ptxdist actions, the output of the `game` command had indeed changed:

```
/ # game
Hello World, Im also game!
```

Listing 3.2: A modified game application

Notably, in order for `ptxdist compile game` to actually compile the modified sources of the game, we first had to run the command `ptxdist clean game`.

### 3.2.3 Creating a framebuffer application

Our next step was to write a simple application accessing the framebuffer device described in section 2.2.1.

#### Accessing the framebuffer as a file

Initially we used the Linux system calls `open()`, `write()`, and `close()` described in section 2.2.2 to write data to the framebuffer. The application would fill an array with `uint16_t` values corresponding to yellow pixels, and then write those values to the framebuffer. The array was large enough that half the screen was drawn yellow, making Tux disappear.

#### Accessing the framebuffer with a memory map

The compendium recommended to access the framebuffer through a memory mapping. This was achieved with the function `mmap()` described in section 2.2.2. Mapping the framebuffer to memory allowed us to access the framebuffer as an array of pixels. To demonstrate the increased freedom, the application colored every other pixel yellow, partly obscuring Tux behind a yellow haze.

## 3.3 Part 2: The Driver

Having completed the first part of the exercise, it was time to write the gamepad driver. Before continuing, the reader might consider reviewing section 2.3, as it introduces theory fundamental to this section.

Our initial aim for the driver was to implement a driver that supported interrupt-driver I/O with asynchronous notification to notify user processes of available data. This was highly recommended in the compendium as a solution with superior energy efficiency to polling. However, the road there was not quite straight, and our driver accumulated several other features along the way.

### 3.3.1 Registering a Character Device

Following steps described in the compendium and in [4], we wrote a simple driver that requested dynamically allocated device numbers and registered a character device in the kernel on initialization. On exit, the driver would remove the character device from the kernel and then deallocate the device numbers.

To pick up any errors, we checked for relevant errors by using `printk()` to print out any nonzero error codes. To verify our implementation, we checked the devices in `/proc/devices` before and after using `modprobe` and `rmmod`: the driver would appear with major device number 253 after running `modprobe`, and disappear again after calling `rmmod`.

### 3.3.2 Accessing the Driver

The next step was to access the driver through the functions in the `file_operations` structure. We added a `printk()` statement to each of the functions `open()` and `close()` that would output the name of the function, and rewrote the game application so that it would try to open and close the file `/dev/gamepad0`. Then we rebooted the operating system, loaded the new driver and created a driver device file manually with the command `mknod /dev/gamepad0 c 253 0`. unning the game application produced the expected output:

```
/ # game
Trying to open the gamepad at /dev/gamepad0 ...
[   99.810000] my_open()
Trying to close the gamepad ...
[   99.820000] my_release()
```

Listing 3.3: Accessing the driver

### 3.3.3 Creating the Device File Automatically

So that we would not have to create the device file manually with `mknod` each time testing the driver, we rewrote the driver so that it would create the device file “driver-gamepad” on initialization, as described in section 2.3.4.

### 3.3.4 Reading the Gamepad Buttons

In order to receive input from the gamepad buttons, we needed to access the GPIO port to which the gamepad was connected. For a description of the GPIO hardware on the

EFM32GG, see section 2.4.1. However, we could not access the hardware directly: the Linux operating system manages hardware access through so-called I/O ports. Drivers that want to access hardware must acquire access to that hardware through special kernel calls. A description of I/O ports and hardware access in Linux is found in section 2.3.6.

In our implementation, the function `setup_gpio()` would be called during driver initialization. This function would request access to the GPIO registers, map them to memory, and set up the GPIO pins on port C for input. A similar function `teardown_gpio` would release the resources on driver exit. On a read, the driver would simply copy the value of `GPIO_PC_DIN` to the user buffer.

To verify our implementation, we rewrote the game application so that it would sit in a tight loop, reading the device and printing new results received to output. The output is shown in listing 3.4.

```
/ # game
Trying to open the gamepad at /dev/driver-gamepad ...
[   12.960000] gamepad_open()
Waiting for button input ... (Press Button 8 to exit)
GPIO_PC_DIN: 1
GPIO_PC_DIN: 0
GPIO_PC_DIN: 64
GPIO_PC_DIN: 0
GPIO_PC_DIN: 128
Trying to close the gamepad ...
[   20.360000] gamepad_release()
```

Listing 3.4: Reading the gamepad buttons

### 3.3.5 Implementing the Seek Operation

At this point, we implemented the `llseek` file operation. The `llseek` file operation is the back-end implementation of the `lseek` system call, described in section 2.2.2. The `read` operation was also rewritten to update the file offset on read and return `EOF` on any offset beyond the start of the file. To read the new device without having to call `lseek` manually each time, the `pread()` system way used.

At this point the utility of `llseek` was not very much. However, it was reasoned that this was a good way to provide access multiple registers.

### 3.3.6 Registering an Interrupt Handler

Currently, the game used polling to discover button presses, a solution with poor energy-efficiency. A much better alternative was *interrupt-driven I/O*. The first step was to register an interrupt handler, as described in section 2.3.7. Then we needed to set up the hardware to receive interrupts, as described in section 2.4.1.

The function `setup_gpio()` was extended with setup code that registered the interrupt handler. It would allocate memory for registers, map the memory into process virtual memory, and set up the registers to generate interrupts on both button press and button release. In addition, before enabling interrupts, it would register a simple interrupt handler. This handler would simply print an interrupt count using `pr_debug()`, clear the interrupts, and then return. The output is shown in listing 3.5.

```
/ # modprobe driver-gamepad
[   10.370000] Initializing the module:
[   10.370000] Setting up GPIO ...
[   10.380000] INTERRUPT: 1
[   10.390000] OK: No errors setting up GPIO.
[   10.400000] allocating device numbers ...
[   10.400000] setting up character device ...
[   10.410000] creating device file...
[   10.430000] OK: No errors during module initialization.
/ # [   17.120000] INTERRUPT: 2
[   18.730000] INTERRUPT: 3
[   19.440000] INTERRUPT: 4
[   20.060000] INTERRUPT: 5
[   20.660000] INTERRUPT: 6
[   21.080000] INTERRUPT: 7
```

Listing 3.5: A simple interrupt handler

Several more steps were needed to fully implement interrupt-driven I/O. However, we felt that other issues were more pressing and chose to implement interrupt-driven I/O at a later time.

### 3.3.7 Releasing resources when not needed

When a driver needs exclusive access to some resources, it should only claim those resources when absolutely necessary. This way other drivers can access those resources in the meantime and the system becomes more flexible.

Our driver needed exclusive access to two regions of GPIO memory, namely the port C registers and the interrupt registers. To make the driver more flexible, we rewrote it so that it initialized the GPIO hardware when the device was opened for the first time. Similarly, the driver would release the GPIO resources only when a release was called on the last open device instance. The driver needed to keep a count on the number of open device instances, stored in the field `open_count` of the `gamepad_device` structure.

### 3.3.8 Buffering Gamepad Input

The gamepad device now worked as a simple hardware interface providing access to a single register. However, we discovered a small problem with this approach. User applications that did not read the device immediately after button input was available would risk that some register states would be overwritten by new ones, thus they might lose valuable information about the exact sequence of button presses. This was especially true when the user pressed two gamepad buttons in rapid succession.

The solution we came up with was to store register states within the driver in a cyclic buffer; processes reading the device would then be able to read the complete sequence of button presses. For more than just four or five button presses, the user (or users) should be unable to generate input at a very high rate. Thus it should be possible to avoid most overflow even with a small buffer size. In the event that overflow should still occur, the buffer would operate on the FIFO principle, overwriting the oldest values first.

With the buffering technique, seeking no longer made sense and support was removed from the device by initializing the field `llseek` in the `file_operations` structure with the helper function `no_llseek`.

### 3.3.9 Implementing the Input Buffer

A first version of the buffer was quickly implemented, and the interrupt handler was rewritten so that it would fill the buffer with values. The buffer was initialized and destroyed together with the GPIO hardware in the functions `gamepad_open()` and `gamepad_release()`. The only remaining step was to rewrite the `gamepad_read()` function so that it would take its values from the buffer instead of directly reading them from the register.

#### Sharing the device among multiple processes

Before we could implement the `read()` function, we needed to make a decision regarding the driver design. The previous driver could be shared among multiple processes: they would simply read the same register, accessing it using individual file offsets. But in the first input buffer implementation the readers shared a global pointer to the oldest unread element, and multiple readers would compete for the gamepad input, an unfortunate situation.

As described in [4], one solution would be to make the gamepad device *single-open*, only allowing a single open device file instance at any time. Any call to `open()` other than the first would fail with the error code `EBUSY`.

Another solution described in [4] was to create a private copy of the device on each call to `open()`. We opted for this approach, as we felt it was both more correct and interesting. To solve the problem with input being consumed, an individual read pointer was associated with each open file instance, stored in the field `private_data` of the `file` structure (see section 2.3.5).

#### Concurrency and safe access

The previous implementation of `read()` was in its simplicity safe against race conditions: there was simply no way a serious race condition could occur. With the input buffer, however, the input handler would need to increment a pointer to the newest element, a pointer that might be read at exactly the same time by a process in the `read()` function. To avoid problems, the `semaphore` structure was used to achieve mutual exclusion between all regions of code accessing the write pointer.

## Work scheduling

Using semaphores to protect the shared buffer data introduced a new problem: an input handler is not allowed to sleep, but the handler needed to sleep to acquire the buffer semaphore. While this could have been solved by using a spinlock instead, we felt this was a bad idea as lengthy reads might potentially stall the whole system.

To solve the problem we used the technique *work scheduling*. The interrupt handler routine was divided into two halves as described in [4]. The *top half* would do a minimal amount of work, then schedule the remaining work to run on the *bottom half* as a different process. The work routine would insert the new value into the user buffer, and was scheduled on the global work queue. The kernel would run it later at some safe time, and it would be allowed to sleep to access the semaphores. For all its glory, this solution had its own little defects, as was discovered afterwards.

## Testing the input buffer implementation

Having implemented shared access, safe access, and work scheduling, we reused the test code in the game application to test the input buffer implementation. Three important conditions were tested: single reader, multiple readers, and rapid input. With single and multiple readers everything worked as expected. The output from the multiple readers test is shown listing 3.6.

```
game: Waiting for button input ... (Press Button 8 to exit)
game: Waiting for button input ... (Press Button 8 to exit)
game: Waiting for button input ... (Press Button 8 to exit)
[   21.050000] gamepad-class gamepad: (interrupt on irq = 17)
[   21.060000] gamepad-class gamepad: (buffer tasklet executed)
game: got input: 4
game: got input: 4
game: got input: 4
[   21.660000] gamepad-class gamepad: (interrupt on irq = 17)
[   21.680000] gamepad-class gamepad: (buffer tasklet executed)
game: got input: 0
game: got input: 0
game: got input: 0
[   26.930000] gamepad-class gamepad: (interrupt on irq = 18)
[   26.940000] gamepad-class gamepad: (buffer tasklet executed)
game: got input: 128
game: got input: 128
game: got input: 128
game: Trying to close the gamepad ...
game: Trying to close the gamepad ...
game: Trying to close the gamepad ...
```

Listing 3.6: Buffered read with three readers. The program output was sorted.

A bug was discovered when testing with rapid input. If the user pressed buttons in rapid succession, the work routine did not always finish in time to service the interrupt handler on the second interrupt. The consequence of this was that valuable information about the sequence of button presses was lost, which was what the input buffer was supposed to prevent in the first place! This is shown in the output of listing 3.7, where two register states are missed (in both cases the register value was 1).

```
game: Waiting for button input ... (Press Button 8 to exit)
[  260.680000] gamepad-class gamepad: (interrupt on irq = 17)
[  260.690000] gamepad-class gamepad: (buffer tasklet executed)
game: got input: 1
[  260.770000] gamepad-class gamepad: (interrupt on irq = 17)
[  260.780000] gamepad-class gamepad: (buffer tasklet executed)
[  260.790000] gamepad-class gamepad: (interrupt on irq = 17)
[  260.790000] gamepad-class gamepad: (interrupt missed!)
game: got input: 0
[  260.880000] gamepad-class gamepad: (interrupt on irq = 17)
[  260.890000] gamepad-class gamepad: (buffer tasklet executed)
[  260.900000] gamepad-class gamepad: (interrupt on irq = 17)
[  260.900000] gamepad-class gamepad: (interrupt missed!)
game: got input: 0
[  260.980000] gamepad-class gamepad: (interrupt on irq = 17)
[  260.990000] gamepad-class gamepad: (buffer tasklet executed)
game: got input: 0
```

Listing 3.7: Buffered read with rapid input. The program output was sorted.

### 3.3.10 Improving Device I/O Capabilities

After implementing the input buffer, a number of improvements was made to the drivers I/O capabilities. These mostly consisted of simple changes implementing some standard driver functionality expected by many user applications.

#### Blocking I/O

Currently, the `read` operation only performed asynchronous I/O, regardless of the value of the `O_NONBLOCK` flag set in the file. This was changed by adding a `wait_queue_head_t` structure to the input buffer. The `gamepad_read` function was rewritten so that a process reading the device would now block on this queue until input was available.

#### Nonblocking I/O

Having implemented blocking I/O, we reimplemented nonblocking I/O. This was done simply by checking the flag `O_NONBLOCK` in the read method and returning the error code `-EAGAIN` if no data was available.

### The poll file operation

Any decent driver supporting nonblocking I/O should also support the file operation `poll`. Without `poll`, the user applications accessing the driver's nonblocking I/O must either resort to busy waiting, or they must implement their own polling scheme, not a satisfying situation. The function `gamepad_poll()` specified the input buffer read queue introduced with blocking I/O as poll wait queue. The bit mask returned would contain the bits `POLLIN` and `POLLRDNORM` to indicate that data was available for a normal read. The input buffer semaphore was used for synchronization.

## 3.3.11 Asynchronous Notification

Finally we were in a position to complete the interrupt-driven I/O operation with asynchronous notification, our goal all along. The last remaining step was to implement asynchronous notification.

On the driver-side, the modifications were small. First we needed to implement the `fasync` file operation used to notify the driver that a process had enabled asynchronous notification on the device file. To keep track of all the asynchronous readers, we wrote the simple function `gamepad_fasync` that would simply call the helper function `fasync_helper()`. To notify the asynchronous readers that data was available, we rewrote the work tasklet to call the function `kill_fasync()` and signal all readers on the `fasync_struct` structure that data was available each time after inserting a new value into the input buffer.

To test the driver, we needed to do a few modifications to the game application. We added the necessary code to set up asynchronous signalling, and wrote a signal handler routine that would read once from the device and print the results. The resulting output is shown in listing 3.8.

```
game: Waiting for button input ... (Press Button 8 to exit)
[   52.120000] gamepad-class gamepad: (interrupt on irq = 18)
[   52.130000] gamepad-class gamepad: (tasklet started)
game: Signal received!
game: got 1 bytes of input: 32
[   52.640000] gamepad-class gamepad: (interrupt on irq = 18)
[   52.650000] gamepad-class gamepad: (tasklet started)
game: Signal received!
game: got 1 bytes of input: 0
[   53.310000] gamepad-class gamepad: (interrupt on irq = 18)
[   53.320000] gamepad-class gamepad: (tasklet started)
game: Signal received!
game: got 1 bytes of input: 128
game: Trying to close the gamepad ...
```

Listing 3.8: Reading with asynchronous signalling.

## 3.4 Part 3: The Game

With a stable and functional driver we were ready to start developing the game.

### 3.4.1 Choosing a Game Concept

Before starting development, we needed a game concept. To save the work of developing a game concept from scratch, we decided to develop a clone of an existing game. There were two main candidates, Asteroids and Space Invaders, of which we choose Asteroids.

Asteroids is a 1979 video game released by Atari in which the player maneuvers a small spaceship in space amidst asteroids and hostile ufos. The objective of the game is to use the spaceship's frontal cannon to eliminate all the asteroids, upon which the player will advance to the next level. At its release, the game was acclaimed for its use of vector graphics.

### 3.4.2 Development

To develop our game we used an incremental approach, adding features whenever we had debugged our current features, and refactoring when nescessary. Our first versions of the game never outputted anything to the buffer at all, using only the serial connection to write status messages. We then implemented the basic draw system allowing us to draw a simple trianlge depicting the player spaceship. We then added features such as asteroids, ship movement, destroyable asteroids (destruction by button press) and firing projectiles. In the end we did not implement collision, feeling that we had spent enough time on the game and that adding additional features was out of the scope of the exercise.

### 3.4.3 Program Modules

To be able to manage a growing code base, we divided our program into five modules, each with a distinct area of responsibility:

**Game**
: The application starting point. It initializes modules on startup, and shuts them down on exit. It also contains the game loop and does game updates. See `game.h` and `game.c`.

**Logic**
: The game logic. It is responsible for manipulating the internal state of the game. It moves the spaceship, asteroids, bullets and ufos, in addition to computing collisions. See `logic.h` and `logic.c`.
The logic module itself is also divided into the following submodules:

  **Asteroids**
  : A module for containing methods related to asteroids, such as spawning, despawning and handling movement of asteroids.

**Spaceship**
A module containing methods to handle the player ship, tracking its orientation, speed and a handler for some input actions.

**Projectiles**
A module containing methods to keep track of projectiles, working in a similar way to the asteroid module.

**Input**
The input processing module. It is responsible for interfacing with the gamepad driver, and provides a clean interface for the rest of the game modules. See `input.h` and `input.c`.

**Draw**
The graphics module. It is responsible for updating the frame buffer, and for rendering various objects of the game model. See `draw.h` and `draw.c`.

**Debug**
Supplies some debugging tools, most notably the macros `game_debug()` and `game_error()`. The debug statements are enabled by defining the macro `DEBUG` before including the module header. See `debug.h` and `debug.c`.

### 3.4.4 Optimization

The board itself, has very limited performance. There is not a seperate GPU on the board. In other words, all the graphic calculations has to be done in the CPU. As a result we implemented some rendering algorithms ourself.

Another performance heavy part of the game was the game logic. In order to reduce expensive data operations we made sure to always use references to our data structures. We allocated some data to the data segment at compile time, and made sure that heap data was only allocated once and referred to in other parts of the program.

```c
...
static struct asteroid my_asteroids[MAX_AMOUNT_ASTEROIDS];
...
```

Listing 3.9: Allocating asteroid data on data segment.

```c
void update_asteroids(){
  struct asteroid* current_asteroid;

  for(int i = 0; i < game->n_asteroids; i++){
    current_asteroid = game->active_asteroids[i];
    current_asteroid->x_pos += current_asteroid->x_speed;
    current_asteroid->y_pos += current_asteroid->y_speed;
    wrap_coordinate(&current_asteroid->x_pos, &current_asteroid->y_pos);
    ...
  }
}
```

Listing 3.10: Updating the asteroids using their references.

This way we could make sure that no expensive allocations or data moves were made in the middle of the game, thus increasing performance.

The board is not capable of refreshing the whole screen each frame and at the same time have a reasonable FPS. To solve this problem, we implemented an algorithm to only update the moving parts of the screen. It is not necessary to update 320 x 240 pixels, when only parts of the screen move each frame. The only moving parts on the screen are: spaceship, astroids and bullets. These items are considerably smaller together compared to the total screen size.

Floating point operations is not supported in hardware in the ARM Cortex-M3 CPU. This makes floating point operations quite expensive. In our game we avoided floating point operations altogether, except when we calculated spaceship rotation.

#### Specialized algorithms we implemented in our game

- Bresenham algorithm (line rasterization) [10]
- The method of seperating axes (collision detection) [5]

## 3.5 Testing

In order to get accurate readouts we made sure the board that was warmed up to reduce randomness. We did several tests in succession with different versions of the game to gauge the impact of our optimizations. We ran tests with drawing the entire screen every update, different amounts of objects on screen and with tickless idle enabled in ptxdist kernelconfig. To collect the statistics we used eAProfiler on a laptop connected to the board.

# 4 Results

During our development we focused on performance, and this focus manifested into our results. With performance in mind during development, we were able to achieve an acceptable FPS, even though we created a real time game.

## 4.1 Energy Efficiency

We chose to make a real time game, which requires frequent screen updates. Therefore we needed to utilize the complete performance of the board. Because of our requirements, there were limited sleep time between each frame.

### 4.1.1 Energy Readings

When the board is idle and only the OS is running, the energy consumption is about 10 mA. Even when our game was running the energy consumption did not increase drastically. Energy consumption readings can be seen in table 4.1.

| Idle | Spaceship | Spaceship + Astroids | Spaceship + Astroids + Projectiles |
|---|---|---|---|
| 10 mA | 12 mA | 18-19 mA | 20 mA |

Table 4.1: Energy consumption with different amount of elements to calculate and draw

From the table we can see that the energy consumption depends on number of elements to calculate and draw on the screen. As mentioned earlier the board is put to sleep between each frame, and the sleep time is calculated dynamically. If there is more work to be done, the sleep time necessarily has to be shorter.

We also made an interesting discovery while measuring energy consumption. When we turned off screen update and only did the calculations, the energy consumption was about 12 mA. The screen updating alone requires about 8 mA alone.

#### Tickless idle

As suggested in the exercise we tried tickless idle. This had no effect while our program was running normally, but when we turned off screen drawing this lowered energy consumption with about 1 mA.

## 4.2 Optimizations

During development we used strategicly placed debug messages. However these messages wrote to `stdout` and then flushed the stream, and appeared to use a significant amount of resources. With debug messages we were able to achieve about 3-5 FPS, and after removing debug messages we achieved above 20 FPS.

# 5 Conclusion

In this exercise we have compiled an OS and implemented the necessary drivers to interact with the board via a gamepad.

On these foundations we have developed a fairly complex game running in real time with a lot of calculations, taxing the recourses of the board heavily. Even with the relatively large demands of our game, and the added overhead of running an operating system, we have shown that hitting 30 frames per second with a fairly complex vector based game is possible.

We have investigated the energy requirements of the game, showing how only drawing the changed part of the screen is a necessity to reach an acceptable level of performance. We have also shown that the power requirements double when several objects are on screen, giving a reasonable measure of how costly drawing an object is. We have also shown that a majority of the energy consumption for a game comes from updating the frame buffer, the payoff for writing complex algorithms to deduce necessary update rectangles is fairly large. Lastly we observed that running tickless idle made no real difference for the game when drawing, however, when not drawing, spending ore time sleeping, it reduced the energy consumption by  1 mA. For a game with more idle time the payoff would probably be more substantial.

### 5.0.1 Compendium Feedback

This section contains feedback and suggested improvements for the compendium.

#### Mention `ptxdist clean` in section 5.3.3

As mentioned in the compendium in section **5.3.3. Building and Flashing**, a single package can be flashed with the following four commands:

1. `ptxdist compile <packagename>, e.g game or driver-gamepad`
2. `ptxdist targetinstall <packagename>`
3. `ptxdist image root.romfs`
4. `ptxdist test flash-rootfs`

However, in order for the modified sources to be compiled, we also needed run one of the commands `ptxdist clean game` and `ptxdist drop game compile`. Maybe it would be useful to mention this in this section.

# Bibliography

[1] *The Linux man-pages project.* Available at `https://www.kernel.org/doc/man-pages/`.

[2] The Linux Kernel Documentation, Documentation/fb/framebuffer.txt, may 2001. Available at `https://www.kernel.org/doc/Documentation/fb/framebuffer.txt`.

[3] NTNU Computer Architecture and Design Group. Lab exercises in TDT4258 energy efficient computer systems, 2014. Available at `innsida.ntnu.no/sso/?target=itslearning`.

[4] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers.* O'Reilly Media, third edition, 2005. Available at `http://lwn.net/Kernel/LDD3/`.

[5] David Eberly. *Method of seperating axes*, 2008. Available at `http://www.geometrictools.com/Documentation/MethodOfSeparatingAxes.pdf`.

[6] Michael K. Johnson and Jukka Lahtinen. *Minicom man page*, july 2009.

[7] Silicon Labs. *EFM32GG Reference Manual*, 2014. Available at `http://www.silabs.com/Support%20Documents/TechnicalDocs/EFM32GG-RM.pdf`.

[8] Norwegian University of science and Technology. TDT4258 - Energy Efficient Computer Systems (course description), 2014. Available at `http://www.ntnu.edu/studies/courses/TDT4258/2014`.

[9] Andrew S. Tanembaum. *Modern Operating Systems.* Prentice Hall, fourth edition, 2009.

[10] T. Theoharis, G. Papaioannou, N. Platis, and N. M. Patrikalakis. *Graphics & Visualization*, 2008.