Lab Exercises in TDT4258 Low-Level Programming

Computer Architecture and Design Group
Department of Computer and Information Science

2015-08-20

Contents

List of Figures

Abbreviations

CMU Clock Management Unit

DAC Digital to Analog Converter

DMA Direct Memory Access

DUT Design Under Test

GCC GNU Compiler Collection

GDB GNU Debugger

GNU GNU's Not Unix

GPIO General Purpose I/O

GUI Graphical User Interface

I/O Input/Ouput

IDE Integrated Development Environment

IOCTL Input/Output Control

LCD Liquid Crystal Display

LED Light Emitting Diode

MMU Memory Management Unit

PC Program Counter

RISC Reduced Instruction Set Computer

SP Stack Pointer

SR Status Register

TFT Thin-Film-Transistor

USB Universal Serial Bus

1. Introduction

This document provides information for the lab exercises in TDT4258 Low-Level Programming. There are three exercises which will be graded and included in the final grade in the course.

The introductory chapter will introduce you to the lab. Each of the following three chapters describe a single exercise. Background material for the exercises can be found across the chapters so that new information for the actual exercise lies in the corresponding chapter. At the end of each chapter, there is a description of the exercise and a suggested approach to solving it. We recommend that you carefully read the background theory before beginning to work on an exercise.

Not all necessary information is found in this document. It is necessary to read some official technical documents, in addition to understanding the more general theory given in the lectures.

1.1. Practical Goal: Simple Game

The practical goal of the lab exercises is to implement a small game for the EFM32 development board shown in figure 1.1. This development board has a microcontroller, a display and sound. In addition, you are given a small prototype gamepad with buttons and LEDs, shown in figure 1.2. Together, this system provides all the hardware components which are needed for making a computer game. Your task will be to program the microcontroller to control all I/O–components needed for the game and use these in a computer game you will implement yourselves.

This course has a focus on energy efficiency, and so have the exercises. It is possible to measure the power consumption of the microcontroller in realtime, and you should use this to optimize your programs for low power consumption.

1.2. Learning Outcome

These exercises will give you practical experience in developing SW for microcontrollers in an embedded system. You will get experience with various techniques and HW components found in many typical embedded systems.

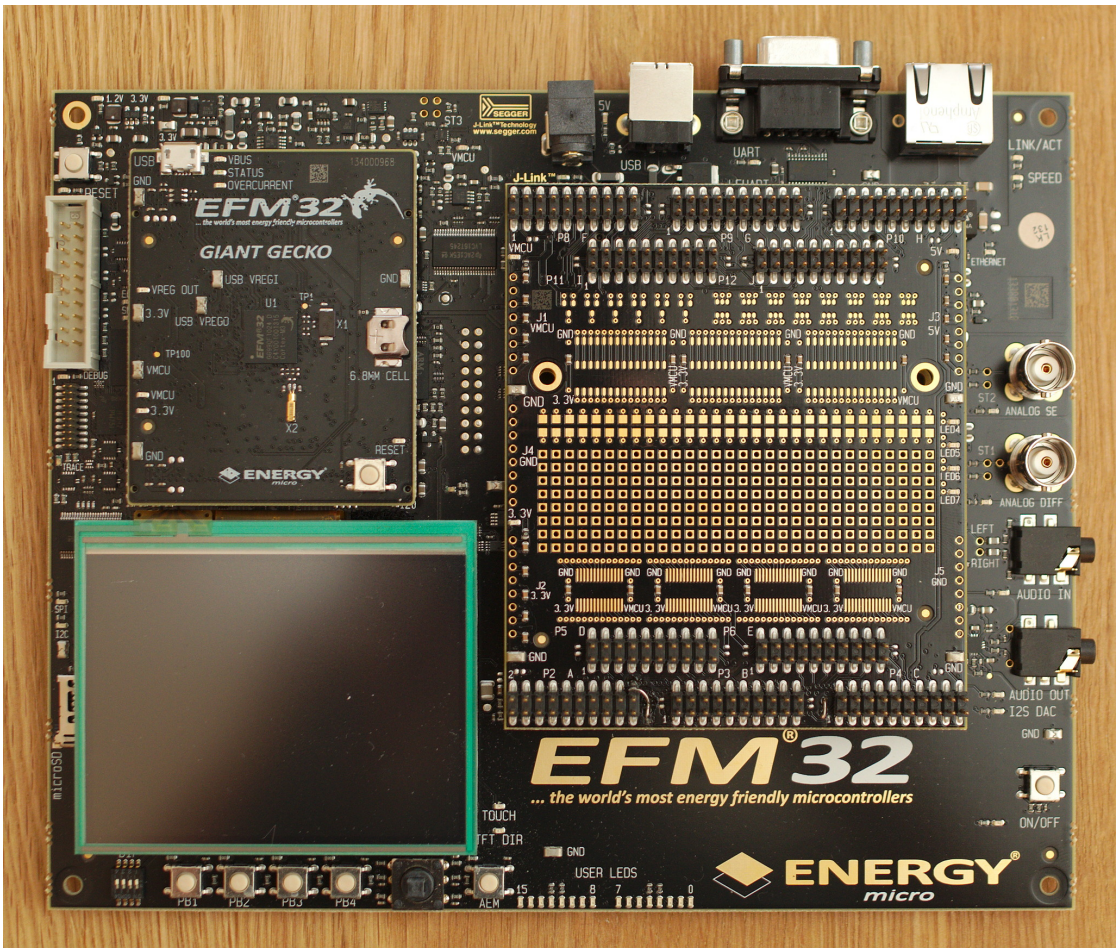

Figure 1.1.: EFM32GG DK3750

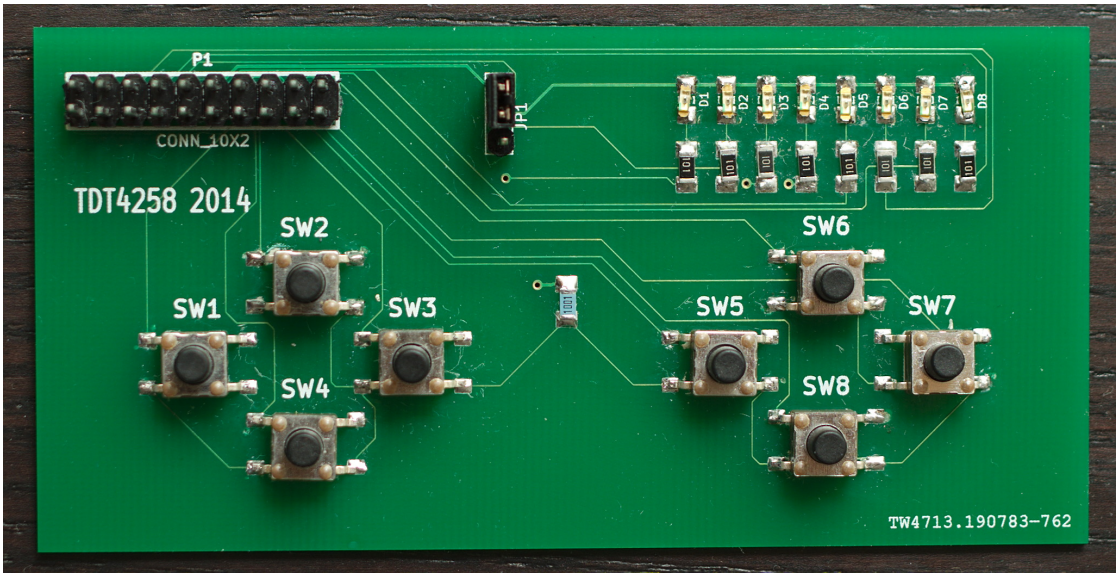

Figure 1.2.: Gamepad

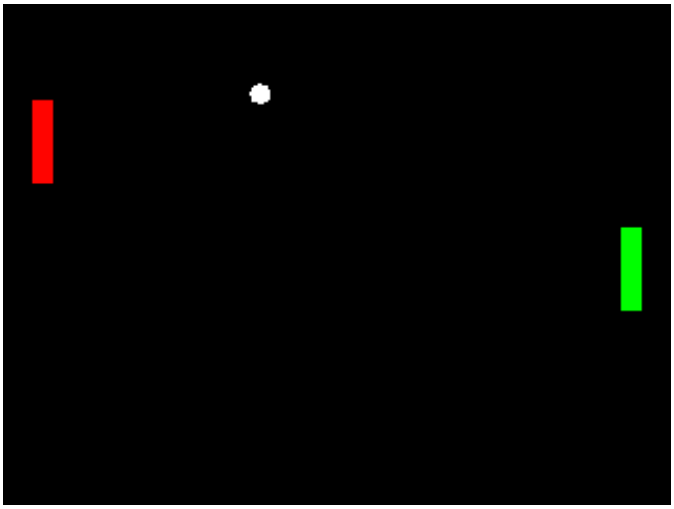

Figure 1.3.: Pong, a classic computer game

You will get experience with the following:

- Programming for an ARM-based microcontroller
  - Assembly programming
  - C-programming (with no operating system)
  - C-programming for Linux
- Programming I/O-components, both in assembly and C
  - General Purpose I/O (GPIO) (buttons and LEDs)
  - Digital to Analog Converter (DAC) for generating sound
- Use of interrupts, both in assembly and C.
- Programming Linux drivers
- Techniques for reducing power consumption

1.2.1. Exercise 1

In exercise 1, you will familiarise yourself with some of the widely used development tools from GNU. These will be used for programming in assembly code for ARM.

You will learn how to write the startup code for ARM processors. In this exercise you will control everything yourself, from the very first instruction executed by the processor. You will learn how to program GPIO. This will be used to detect which buttons are pressed on the gamepad and to turn on or off LEDs. In addition, you will learn how to handle interrupts in assembly.

1.2.2. Exercise 2

In exercise 2, you will continue to use the GNU toolchain, but this time you will program in C.

You will have no operating system available. This means that you have to do most of the job yourselves, as you did in exercise 1, but in the much more comfortable C language. You will also have the most basic startup code and C libraries available.

Here you will again program for the gamepad. In addition, you will program the sound effects which will be used in the computer game. You will have to write code that generates sounds and program the DAC, a hardware component which converts digital data to analog signals. The analog signals generated by the DAC are further sent to an amplifier and to a speaker (headphones).

1.2.3. Exercise 3

This is the last exercise in the course, which will result in a small game running on the development board. This time, you will install an embedded variant of the Linux operating system on the board and make your game as a Linux application.

The gamepad is necessary for controlling the game, so a Linux device driver must be implemented that communicates with the buttons and which is used by the game application. The game output will use the TFT display on the board, programmed through the Linux framebuffer device.

1.3. Practical Information

1.3.1. Evaluation

Exercises will be evaluated based on the delivered report, the code and a short presentation of your solution. During the presentation you will be asked some questions about the solution and the choices you made for it to show that you worked on it independently.

The number of points for the solution will depend on the following:

- In what degree the requirements for the exercise have been met
- Quality of the report
  - Clarity and readability
  - Complete yet concise
- Code quality and technical solutions
  - Well structured, clear and commented
  - The choice of technical solutions
- Power consumption
  - Chosen techniques for reducing power consumption
  - Results and discussion in the report
- Testing
  - How the tests were performed
  - Test results
- Presentation for the teaching assistant
- Solutions which stretch beyond requirements

1.3.2. Deliveries

Report, all code and all material needed for running tests must be included in the delivery.

1.3.3. Late Deliveries and Copying

Deadlines for each exercise delivery are given in the timesheet on the course It's learning page. For late deliveries, 10% of the points scored for the exercise will be taken off for each day after the deadline. The only exception is absence because of illness. In such cases, you have to provide a written document from your doctor.

As the grades given for the exercises are included in the final grade for the course, copying source code or report text is not tolerated at all. Information sharing between groups is, however, accepted and encouraged.

1.3.4. Lab and Assistance

You are free to work on the exercises in the lab whenever it is convenient for you. However, your group will be assigned to a certain time slot where a student assistant will be present. On these slots, the assigned groups will have priority both to equipment and to the assistant.

If you want to get the most from the lab assistance, you are advised to attend the designated lab hours already well prepared and acquainted with the exercise.

1.3.5. Lab Equipment and Software

In the lab you will use a PC with the Ubuntu Linux operating system. Those who are not well acquainted with Linux from before will, therefore, have to learn about Linux in order to work on the exercises. Some knowledge about the command line is essential for these exercises. Linux will not only be used on the PC but later, in exercise 3, also on the development board.

1.4. Before You Begin...

These exercises are quite open ended, making them both challenging and exciting. However, low level programming for resource constrained devices is quite different from using high level APIs on powerful desktop computers, reducing your productivity quite a lot from what you might be used to from other programming projects.

Keep these things in mind and start working early. It is better to finish one week earlier than to get stuck the last night before the deadline when there are no student assistants to help you.

2. Exercise 0

2.1. Introduction

The purpose of this exercise is to familiarize yourself with the development environment and tools that will be used during the course of the lab. *You do not have to deliver anything for this exercise as it is intended more as a tutorial. Still, it is strongly recommended to go through the practical steps described here in order to become familiar with the equipment. This will increase your productivity for the remaining exercises, especially if you have no experience with microcontroller programming.*

In the following sections, a description of the development environment and two of the software tools that you will be using for development will be presented. You will have the opportunity to program the board with a pre-built executable and observe the energy consumption in real-time.

2.2. The Development Board

Throughout this course, you will be using the EFM32GG-DK3750 development board from Silicon Labs/Energy Micro. This is a development board containing an energy-effective 32-bit ARM Cortex M3-based microcontroller, a full-color TFT LCD display and various other input/output peripherals. It also contains a power measurement subsystem, which allows the user to observe how much power the board is using at present.

Physically, DK3750 is divided into several boards:

- The mainboard, with most of the peripherals (I/O units) and onto which the other boards are connected.
- The CPU board (top left) with the EFM32GG microcontroller
- The prototyping board (right). This board provides access to the pins of the microcontroller, most importantly the GPIO pins.
- The TFT display (bottom left)

A block diagram overview of the board showing the peripherals and the connections is presented in in Figure 2.2. Technical information on the development board, including the components used and the connections can be found in [12].

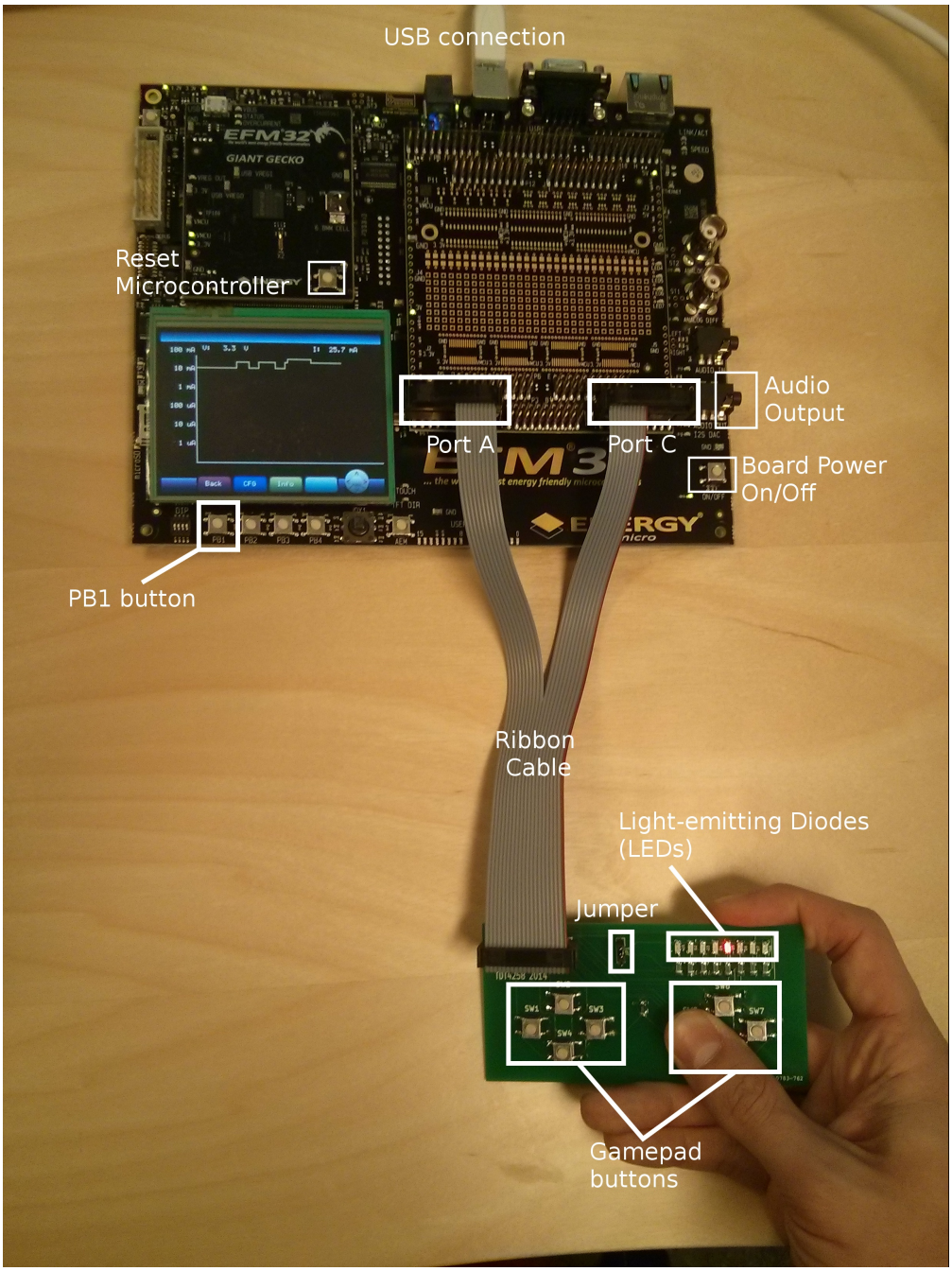

Figure 2.1.: An example of using the gamepad buttons to control the lights, while observing the energy consumption of the system. The important parts of the system for the exercises are labelled.

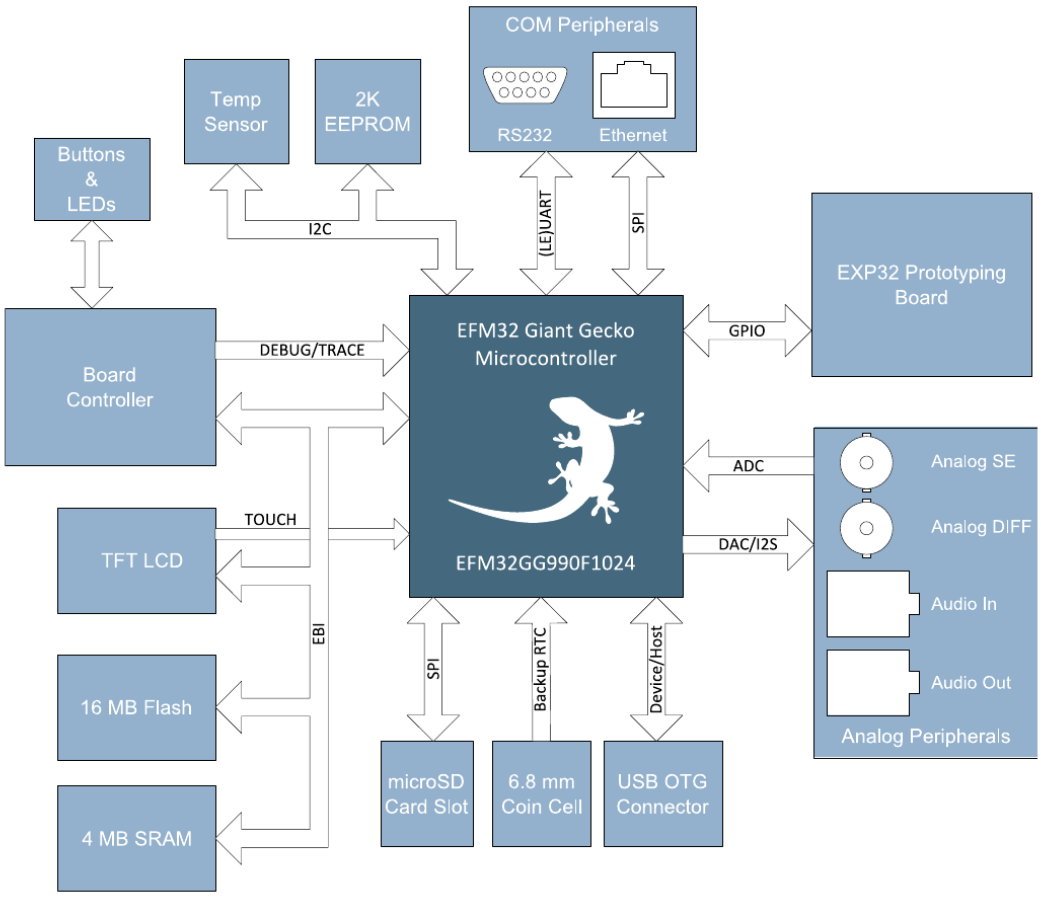

Figure 2.2.: DK3750 block diagram overview (taken from [12])

For this course, the important parts to understand is:

- EFM32GG microcontroller, which contains the ARM CPU and all I/O controllers.
- GPIO connections to the prototyping board
- DAC connection to the audio out connector

In addition to the components in the block diagram, the DK3750 contains support circuits for programming and debugging, and a system for doing energy measurements.

2.3. Practical Basics

The first step for all laboratory tasks in the course is to power up the development board. First, it must be ensured that the USB cable is connected to both the DK3750 and to one of the lab computers through a working USB port. A green LED next to the USB port should light up when the cable is connected. Afterwards, the on/off button close to the lower right-hand corner of the board should be pressed to power up the board. The board will take several seconds during the boot process, upon whose completion the LEDs labelled VMCU and 3.3V on the CPU board will light up green, and the Energy Micro logo will be displayed on the TFT LCD screen. The EFM32GG-DK3750 is then ready for development.

The following subsections will cover more basic aspects of the practical work in the laboratory.

2.3.1. Resetting and Power-Cycling

You will often want to re-execute the program on the microcontroller from start, for instance after re-programming or simply to observe the program execution once again. To do this, use the Reset button on the CPU board, instead of using the on/off button. This will only reset the microcontroller instead of power-cycling the whole board, and will be much faster.

If the development board stops responding (typical symptoms are the PB buttons not responding and ”unable to connect” messages from the Energy Micro tools) you may want to power-cycle it by using the on/off button, or removing and re-inserting the USB cable.

2.3.2. Measuring Power Consumption

One of the most interesting features about the DK3750 is that it contains a system for doing real time power measurements of the running software.

The easiest way to do this is to simply turn on the DK3750, and then press the PB1 button shown in Figure 2.1. The display will then show a real time graph of the power consumption. Note that the y-axis showing the current is with a logarithmic scale. The same display can also be shown on your PC if you start the program called eAProfiler (EnergyAware Profiler).

Note that the on-board current measurement circuitry has a current reading range of [1 uA, 50 mA] and may display incorrect readings outside this range.

2.3.3. The Gamepad

The gamepad is to beconnected to the development board through a Y-shaped ribbon cable, with the double-ends of the ribbon cable inserted into the Port A and Port C headers of the prototyping board, as can be observed in Figure 2.1. It contains eight buttons and eight LED that will be used to provide input/output functionality in the following exercises. Further details on how the gamepad is connected to the development board will be provided as part of Exercise 1.

The gamepad also contains a jumper, which can be used to exclude the gamepad LEDs from power measurements. When the jumper is connected between the lower two pins, the gamepad LEDs will be excluded from the power measurement.

2.3.4. Uploading Software to the Board

To be able to run software on the board, the internal program Flash memory of the EFM32GG must be programmed with the desired executable. A Linux tool is installed in the lab that can be used for uploading programs to this flash, called **eACommander** (EnergyAware Commander). This tool has both a graphical user interface (GUI) and a command line interface, both of which will be described briefly below.

eACommander GUI

If you start the program without any arguments, GUI mode is used. To upload a binary file to the Flash, do the following:

1. Connect the USB cable to the board
2. Start eACommander
3. Press “Connect” button to connect to the board
4. Press the Flash button in the left side panel
5. Select your binary file
6. Press the “Flash EFM32” button

eACommander Command Line

eACommander can also run from the command line. If the USB cable is connected to the board, the following is sufficient to upload a file to the flash:

```
eACommander.sh -r --address 0x00000000 -f "program.bin" -r
```

The `-r` flag resets the microcontroller. The `-f` flag takes a file name as an argument, and uploads this file to the address specified as an argument to the `--address` flag. Consequently, the example command first resets the microcontroller, uploads the file "program.bin" to address 0 in the flash, and then resets the microcontroller again. For more information, run

```
eACommander.sh --help
```

2.4. Description of the Exercise

For this exercise, you have been provided with a binary file (`gamepad-test.bin`) that is intended as a simple test to see if the gamepad hardware works correctly. Your task is to program the development kit with this binary file and test the gamepad, and to observe how the power consumption changes by turning on LEDs.

1. Connect the gamepad to the development board before powering on the development board. This is good practice for all electronic connections that do not support "hot plugging".
2. Connect the development board to the PC and power it on. Issue the `lsusb` command and see if the board is detected by the computer.
3. Program the development board with the `gamepad-test.bin` file using eACommander.
4. When the programming is finished, try pressing different combinations of buttons on the gamepad. Each button should light a different LED while pressed.
5. With the gamepad jumper in the upper position, use the eAProfiler or the onboard power monitor to observe the power consumption while turning on one or several LEDs. How much current does each lighted LED use?
6. Move the gamepad jumper to the lower position. Are you still able to observe an increase in the current while turning on LEDs?

If you discover some faulty hardware during the course of this exercise, please inform the course staff so it can be replaced.

3. Exercise 1

3.1. Introduction

In this exercise you will write a program which allows the user to use the gamepad buttons to control the row of LEDs on the gamepad. You must write your program in assembly.

This exercise requires little actual program code, most of your time will be spent on understanding the hardware and learning low level SW programming. Looking up information in the documents [10], [12], and [13] will be crucial

3.1.1. Learning Outcome

The learning outcome of this exercise is:

- General architecture of ARM Cortex-M3 microcontrollers
- EFM32GG DK3750 development board
- Understanding object files and the task of a linker
- Use of the GNU-toolchain
  - AS (assembler)
  - LD (linker)
  - Make (automatic use of assembler and linker)
  - GNU Debugger (GDB) (debugger)
- Programming in assembly for ARM Cortex-M3
- GPIO, I/O controller which control buttons and LEDs
- Interrupt handling in assembly
- Power measurements on DK3750
- Simple energy optimizations

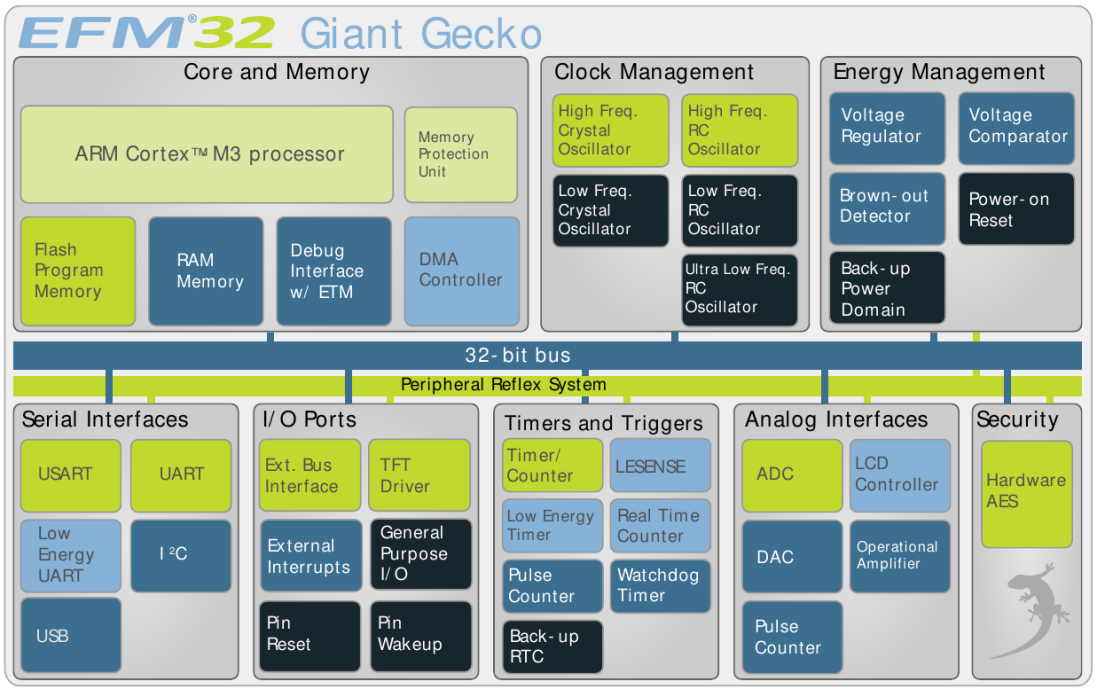

Figure 3.1.: EFM32GG overview (taken from [13])

3.2. EFM32GG Microcontroller

The EFM32GG microcontroller is an ARM based microcontroller with a focus on low power applications. A block diagram is given in figure 3.1. Documentation for this microcontroller is given in [13].

3.2.1. ARM Cortex-M3

The main component of the EFM32GG is the ARM Cortex-M3 processor. Most of the time for this exercise will be spent learning to program this processor in assembly. The Cortex-M3 is a 32 bit pipelined RISC processor that supports the ARM Thumb instruction set.

The M3 is documented in [10], where all the instructions are summarized and explained. In addition, an instruction set quick reference can be found in [1].

Exception Vectors

It is important to understand how the M3 starts its execution after reset, in order to do exercise 1.

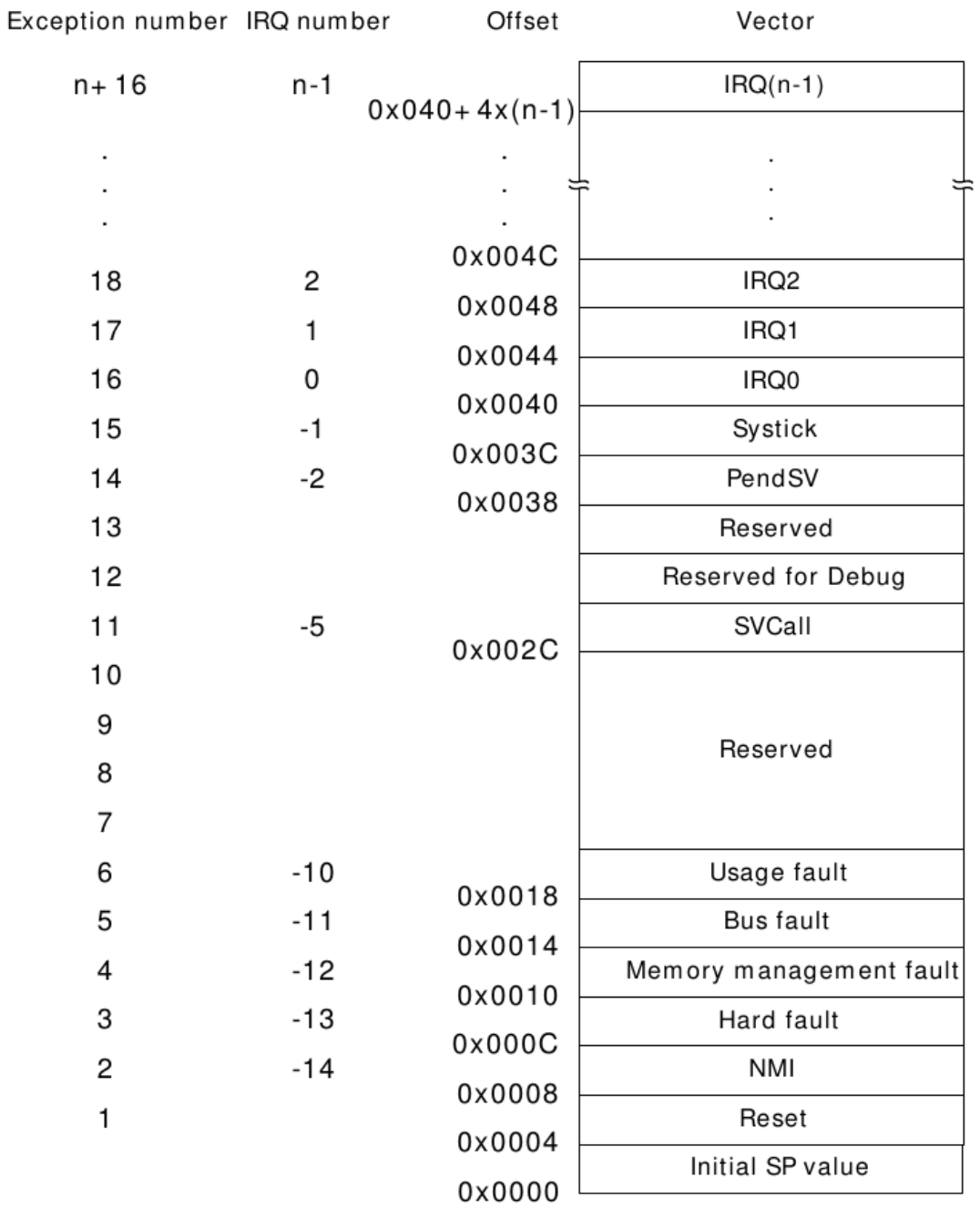

Figure 3.2.: Exception vector table (taken from [10])

```
/* Place the following from address 0 in memory */
.long   0x1000          /* Initial stack pointer */
.long   reset           /* Address of reset handler */
/* if required, other handler addresses follows here */

.thumb_func
reset:
/* program code here */
```

Figure 3.3.: Minimum Cortex-M3 assembly program

A table called the "exception vectors" is located at address 0 (which is Flash memory in our EFM32GG microcontroller). This exception vector table contains several 32 bit data words, as shown in figure 3.2, and specifies where in memory the handler routines for the different exception and interrupt types are located.

To make the most basic program possible for the M3, it is necessary to do the following:

- Create a reset handler somewhere in memory, which is simply the program you want to execute after reset
- Put the initial stack pointer value at address 0
- Put the address of your reset handler[1] at address 4

A minimum program is shown in figure 3.3. A reset will load the SP to the value 0x1000 and then start executing the reset handler.

Some exception vector entries, like the initial SP value and the reset, have semantics determined by the ARM core. However, each microcontroller may independently decide which interrupts are associated with the IRQi-entries. It is therefore necessary to consult the microcontroller reference manual to determine what exactly these entries are used for.

More information can be found in section 2.3 in the the Cortex-M3 reference manual [10]. In addition, the various interrupts of the EFM32GG is documented in section 4.3.1 in the EFM32GG reference manual [13].

3.2.2. Memory Map

Reads and writes to memory addresses from the EFM32 microcontroller initiates accesses to different units, depending on the address accessed. Parts of the address space are used for regular memory, while others are used to access peripheral units. An overview of what access to an address will actually do is given in figure 3.4.

[1] Even though the handler is word aligned, the addresses in the vector table must have bit 0 set to indicate Thumb mode. This is most commonly done automatically by the assembler and linker with the `.thumb_func` directive. See section B.5 in the appendix for more details.

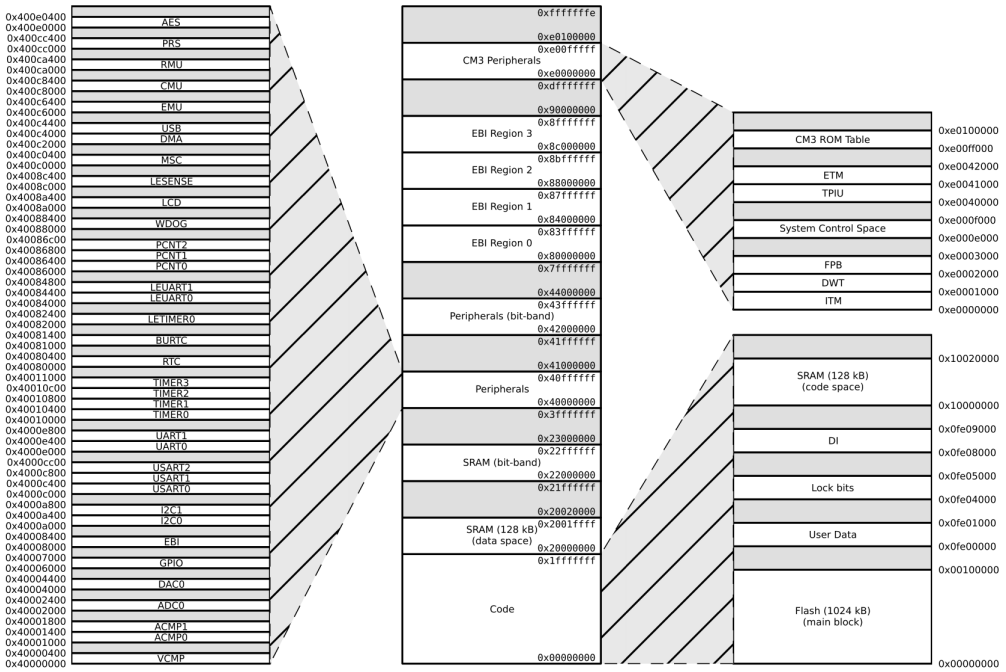

Figure 3.4.: The memory map of the microcontroller, taken from [13].

3.2.3. I/O Controllers

All the I/O controllers in the EFM32 are memory mapped, which means that you can program the controllers by reading and writing special memory locations, called the I/O registers. This can be seen in figure 3.4. I/O registers for the various I/O controllers are described in the EFM32GG reference manual [13].

CMU

To save power, only the I/O controllers that are actually used is clocked. This means that for every I/O controller you want to use, you need to specifically enable its clock. There is a special I/O controller for that: The Clock Management Unit (CMU). This is documented in section 11 in the EFM32GG reference manual [13].

The CMU is quite easy to use. Register `CMU_HFPERCLKEN0` is a 32 bit register where each bit corresponds to a specific I/O controller. To enable an I/O controller, set the corresponding bit to 1 in the CMU_HFPERCLKEN0 register. See section 11.5.8 in the EFM32GG reference manual [13] for a description of the individual bits.

Example: To enable clock for the GPIO controller, you need to set bit 13 in CMU_HFPERCLKEN0.

C:

```
#define CMU_HFPERCLKEN0  ((volatile uint32_t*)(0x400c8044))

*CMU_HFPERCLKEN0 |= (1 << 13);
```

Assembly:

```
CMU_BASE = 0x400c8000       // base address of CMU
CMU_HFPERCLKEN0 = 0x044     // offset from base
CMU_HFPERCLKEN0_GPIO = 13   // bit representing GPIO

...

// load CMU base address
ldr r1, cmu_base_addr

// load current value of HFPERCLK ENABLE
ldr r2, [r1, #CMU_HFPERCLKEN0]

// set bit for GPIO clk
mov r3, #1
lsl r3, r3, #CMU_HFPERCLKEN0_GPIO
orr r2, r2, r3

// store new value
str r2, [r1, #CMU_HFPERCLKEN0]

...

cmu_base_addr:
                .long CMU_BASE
```

3.2.4. NVIC

The Cortex-M3 contains an interrupt controller which can be used to enable or disable various interrupts. This is documented in section 4.2 of the Cortex-M3 Reference Manual [10].

Usually, only one register is necessary: `ISER0` contains one bit for each interrupt source. To enable an interrupt, set the corresponding bit to 1 in ISER0.

Example: Two different interrupt handlers handle interrupts from even-numbered GPIO-pins and interrupts from odd-numbered GPIO-pins. To enable both even and odd interrupts for GPIO, set bits 1 and 11 to one (i.e. write 0x802 to `ISER0`).

GPIO

A special gamepad has been created specifically for this course, shown in figure 3.5. The gamepad contains 8 buttons and 8 LEDs. You will control this gamepad through the

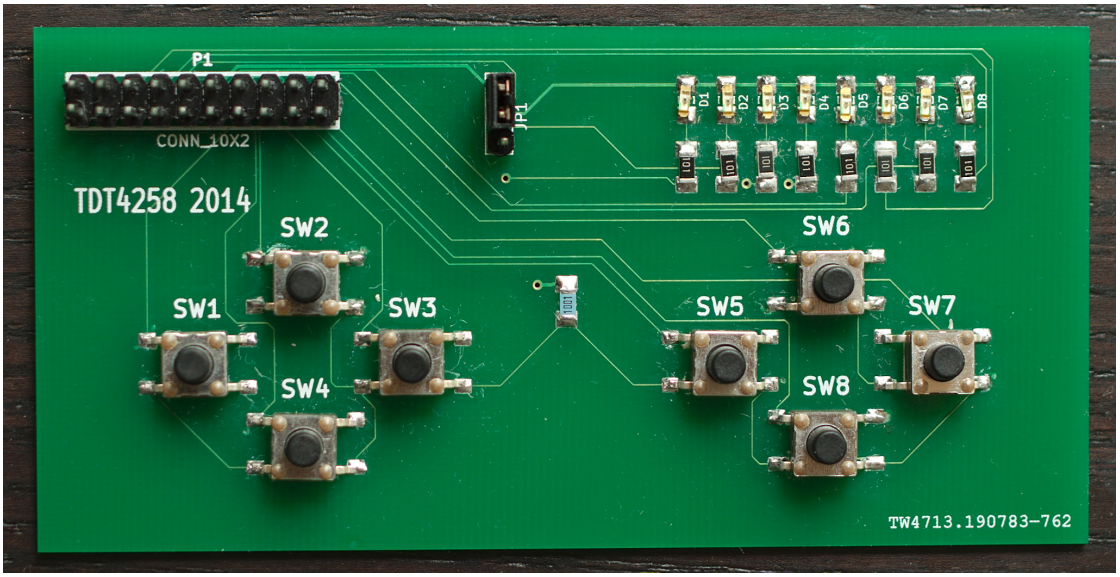

Figure 3.5.: Gamepad

GPIO controller. The gamepad also has a jumper that selects if LEDs are included in power measurements or not.

GPIO stands for "General Purpose I/O" and is a very generic I/O controller that can be used for a lot of various purposes. In short, the GPIO controller can programmatically set output pins on the microcontroller to a specific value, or it can read the value of input pins. You will read the status of buttons and turn on or off individual LEDs.

The GPIO pins of the microcontroller is exposed on the connectors on the prototyping board. This gamepad must, therefore, be connected to the prototyping board, as was shown in figure 2.1. When connected like that, figure 3.6 shows the schematic in detail. Note that no external pull-up resistors[2] are present for the buttons, the internal pull-ups of the EFM32GG must therefore be used in order to correctly read logical high when buttons are released.

Example for setting up pins 8-15 of port A for output (LEDs on the gamepad):

- Enable GPIO clock in CMU

[2] A pull-up resistor is used to keep an otherwise disconnected signal stable. Since we are connecting the GPIO-pin to a button, most of the time the button will not be pressed. When the button is not pressed, the GPIO-pin is not connected to anything, and its voltage value would be unpredictable. To avoid this, a voltage source is connected through a resistor to the GPIO-pin to make sure that the voltage level is high when the button is not pressed. When the button is pressed, the GPIO-pin will be connected to ground, and the voltage level of the pin will be read as zero. If the button connected the pin to a voltage source instead, you would use a pull-down resistor.

Since not all devices connected GPIO necessarily require this feature, the pull-up resistor must be configurable. This would typically be done by using a transistor to selectively connect pull-up circuitry to the pin.

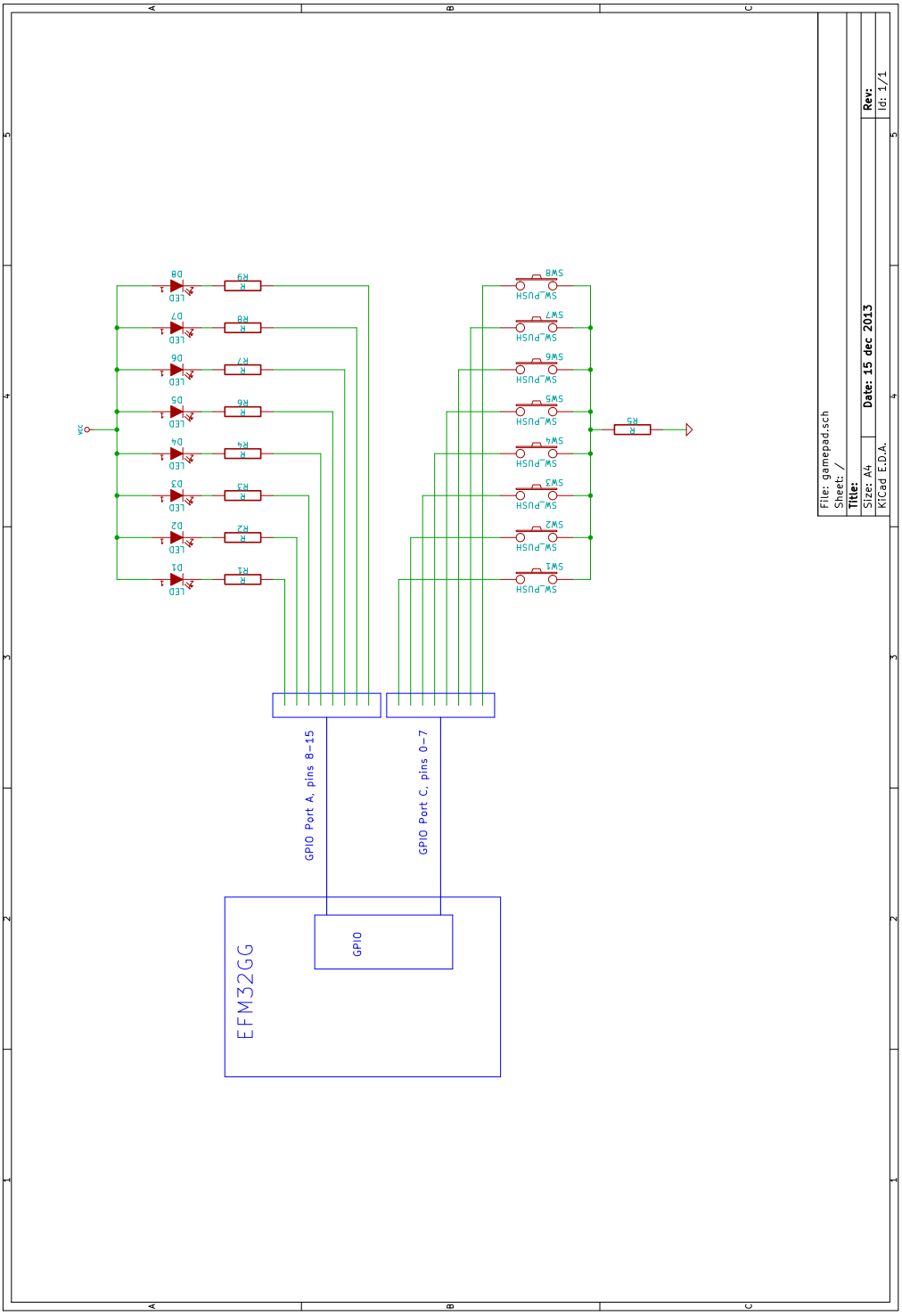

Figure 3.6.: Gamepad simplified schematics

- Set high drive strength by writing 0x2 to GPIO_PA_CTRL
- Set pins 8-15 to output by writing 0x55555555 to GPIO_PA_MODEH register
- Pins 8-15 can now be set high or low by writing to bits 8-15 of GPIO_PA_DOUT. The LEDs are active-low, as can be seen in the gamepad schematics in figure 3.6.

Example for setting up pins 0-7 of port C for input (buttons on the gamepad):

- Enable GPIO clock in CMU
- Set pins 0-7 to input by writing 0x33333333 to GPIO_PC_MODEL
- Enable internal pull-up by writing 0xff to GPIO_PC_DOUT
- Status of pins 0-7 can now be found by reading GPIO_PC_DIN

GPIO interrupts For better energy efficiency, interrupts should be used so that the CPU can sleep when there is nothing to do.

The GPIO has the possibility to have different interrupt handlers for even and odd pins. This is rarely useful, so it is a good idea to use the same handler for both of them.

To set up interrupts for pins 0-7 of port C (gamepad buttons), do the following:

- Put the address of your interrupt handler to both address 0x44 and 0x6c in the excpetion vector table.
- Write 0x22222222 to GPIO_EXTIPSELL.
- Set interrupt on 1->0 transition by writing 0xff to GPIO_EXTIFALL.
- Set interrupt on 0->1 transition by twriting 0xff to GPIO_EXTIRISE.
- Enable interrupt generation by writing 0xff to GPIO_IEN
- You can determine the source of the interrupt by reading the GPIO_IF register. In the interrupt handler, you can clear the interrupt by writing the value of GPIO_IF to GPIO_IFC. If you forget this, the interrupt handler will (incorrectly) be called repeatedly after the first interrupt.
- Enable interrupt handling by writing 0x802 to ISER0.

3.2.5. Energy Modes

One of the main features of the EFM32GG is its focus on energy efficiency. It is possible to run the microcontroller in different energy modes, depending on how many units needs to be active.

Energy modes are controller by the Energy Management Unit (EMU), which is documented in sections 3.4 and 10 of the EFM32GG reference manual [13].

The main idea is to go to sleep when the CPU is no longer doing anything useful (e.g. waiting for button input). The CPU will wake up when it receives an interrupt, and go back to sleep after the interrupt handler returns.

Example of going to energy mode 2:

- Write 6 to SCR (System Control Register, see section 4.3.7 in the Cortex-M3 reference manual [10]). This enables deep sleep, and automatic sleep on return from interrupt handler.
- Execute instruction `wfi` to enter sleep mode.

3.3. GNU-Toolchain

GNU is a project sponsored by Free Software Foundation (FSF) with the main goal to provide a free and open operating system with all accompanying tools for software development. All Linux based machines widely use GNU software and many GNU tools are also used in other operating systems like MS Windows. This means that irrespective of the operating system, GNU tools can be downloaded from the internet and used within it.

The most popular GNU tools are software development tools; C compiler, debugger and accompanying tools. You will use them for the exercises in this course.

All tools from GNU come with detailed documentation which is available as a book, an info page and as an online document. Search the web to find them.

3.3.1. Cross Development

Cross development means program development on one platform and running it on another. It is the typical form of development for embedded systems. The programming is done on a PC and the program is run and tested on the target system.

You will do the following: Develop programs on a Linux PC and transfer binary files to DK3750 to see if they work as expected. Development tools installed by default on Linux computers cannot be used for cross development, special tools built for the ARM target must be used. These are installed on the lab machines you will use. Similar tools can be found freely on the Internet (or in the Ubuntu package repository) for those of you who want to use your own computers.

3.3.2. GNU AS

The assembler you will use is called GNU AS. Its manual [4] can be obtained by the use of the `info` command. GNU AS is used like this:

```
as -o <outputfile> <assemblyfile>
```

As you will work with cross development, you need to use a special variant of as command and the command line will be as follows:

```
arm-none-eabi-as -mcpu=cortex-m3 -g -o <outputfile> <assemblyfile>
```

Arguments and options have the following meaning:

- `-mcpu=cortex-m3`: You want the M3 instruction set
- `-g`: Make debug symbols so that program can be debugged with GDB.
- `-o <outputfile.o>`: Write the result of assembling to the file "outputfile.o". It is important to provide a file name with the extension ".o" because it is not an executable binary file but an object file. Read more about it in appendix C.
- `<assemblyfile>`: The name of the assembly file which will be compiled by the assembler.

3.3.3. GNU LD

You will use a linker which is called GNU LD. Use `info` command to read the manual [5] but, in brief, it is used like this:

```
arm-none-eabi-ld -T <linkerscript> -nostdlib <arg1> <arg2> ...  <argN>
```

The arguments are:

- `-T <linkerscript>`: Specifies which likerscript to use. More on that in the next paragraph.
- `-nostdlib`: Do not include any standard libraries, we do it all by our selves in this exercise.
- `-o <outputfilename.elf>`: write the result of the linking stage to the file "outputfilename.elf".
- One or more object files

Here is a concrete example:

```
arm-none-eabi-ld -T efm32gg.ld -nostdlib -o program.elf file.o fileb.o
```

This will link together object files "file.o", "fileb.o" and write the result to the file "program.elf".

Linker Scripts

The linker needs to know how the memory is to be used for your specific microcontroller. For this reason the linker can take the memory setup as an input file and use this when creating the final executive.

Take a look at the delivered linker script for the exercise to get a feel for what the linker script is used for.

3.3.4. objcopy

Even though the linker creates the finished ELF file, the eACommander tool expects a clean binary file without any extra metadata. This can be generated with the objcopy tool:

```
arm-none-eabi-objcopy -j .text -O binary <inputfile> <outputfile>
```

The arguments are:

- `-j .text`: We want to convert the text segment
- `-O binary`: We want a binary output
- `<inputfile>`: Elf file from the linker
- `<outputfile>`: Binary file to give to eACommander for flashing.

3.4. GNU Make

It is too cumbersome to compile a big project if all the commands are manually entered on the command line. Therefore, there are tools which automate this process. One of the most common of these build tools is GNU Make. In brief, a Makefile is set up where it is specified how to build the project (in other words, how to compile/assemble and link the files into an executable file) so that in the future it suffices to use the command "make" when a new version of the program needs to be built. Make is also wise enough to compile only object files which need to be compiled because some source files have been updated.

3.4.1. How to write a Makefile

A Makefile lies in the same folder as the source files and is always named "Makefile". A Makefile consists of "rules" which specify how to make, for example, an object file from a given source file. A rule has the following format:

```
target_file: dependencies
  command_line
```

- `target_file`: The name of the resulting file
- `dependencies`: A list of files (separated by a space caracter) on which the target file depends. This means in practice a list of the files where a target file must be remade if there is a change in one or more of these files.
- `command_line`: The command line which must be executed in order to generate the target file. NB: Command line must be in a separate line and it MUST begin with a tab. If you use space characters, it will not work, you have to use tab.

Here is an example:

```
example.o : example.c example.h
  gcc -c -o example.o example.c
```

This rule states that object file "example.o" depends on the files example.c and example.h, changes in these files implicate that example.o should be compiled anew. The command which will be executed to generate example.o is `gcc -c -o example.o example.c`.

Often, there are such rules for each object file which makes a program and a rule which represents a linking stage which depends on all the object files. Let us see one typical linking rule:

```
example : example.o
  ld example.o -o example
```

Here we see that this rule depends on example.o, a file which is itself generated by a make rule. When make tries to link the file "example", it will first check if example.o needs to be generated anew. In such a case, it will generate it before linking "example".

The first rule is the one which is executed by default. It often represents the executable program and, therefore, a linking stage. In our case, this should be an objcopy rule because the linker rule does not produce the finished binary file for us.

If you would specifically like to execute some other rule than the top rule, you can give that on the command line:

```
make example.o
```

This command will build example.o but it will not execute a link rule because example.o is not dependent on the result of the linking stage.

A common rule you are encouraged to make is a so–called "clean" rule. This is a rule which can look like this:

```
.PHONY : clean
clean :
  rm -rf *.o example
```

Typically it is placed at the bottom of the Makefile and it will not be executed unless you specifically ask for it when you run make, (`make clean`). The intention is to clean up, remove all autogenerated files so that only source files remain. .PHONY : clean is a notification for Make which says that clean is not a proper rule, when it is executed, it does not generate a new file which is named clean.

These simple rules suffice to get Makefiles which work but there are many tricks which can make a Makefile more elegant and easier to write and which make the job easier for program developers (you). We recommend that you look into the GNU Make manual [6], it is accessible as an `info` page.

3.5. GNU Debugger (GDB)

All programs contain errors and that is why it is essential that you have a possibility to debug the program. You will use a debugger named GDB. It is a rather powerful program which offers many possibilities to monitor the program execution, stop the execution and inspect the contents of registers and memory. It also gives the possibility to "single step" the program execution, which executes line by line from the original source code, so that the user can monitor what happens in detail.

As we do cross development, we must have a possibility to run the debugger on a PC while the program we are debugging is being executed on the development board. To do this, the debugger communicates through a program called the gdbserver. Gdbserver lives in the middle between the development board and GDB.

Gdbserver can be started like this:

```
JLinkGDBServer
```

GDB can be started like this:

```
arm-none-eabi-gdb <elf-programfile>
```

3.5.1. GDB in Emacs

In order to make debugging work efficiently, you can combine GDB with a text editor. Often you wish to see which line in the source code is being executed, for example by single stepping, and this presumes that GDB can communicate with the text editor.

One possibility is to run GDB through Emacs. Emacs is rather powerful, yet it does not hide away what happens on the bottom, for example, when debugging with GDB. As this course teach you the fundamentals about these tools, you should get exposed to the tools directly. This makes Emacs suited as a companion to GDB in this course.

The main difference between running GDB in Emacs instead of running it directly from the command line is that you will have the source files you are debugging in Emacs. The

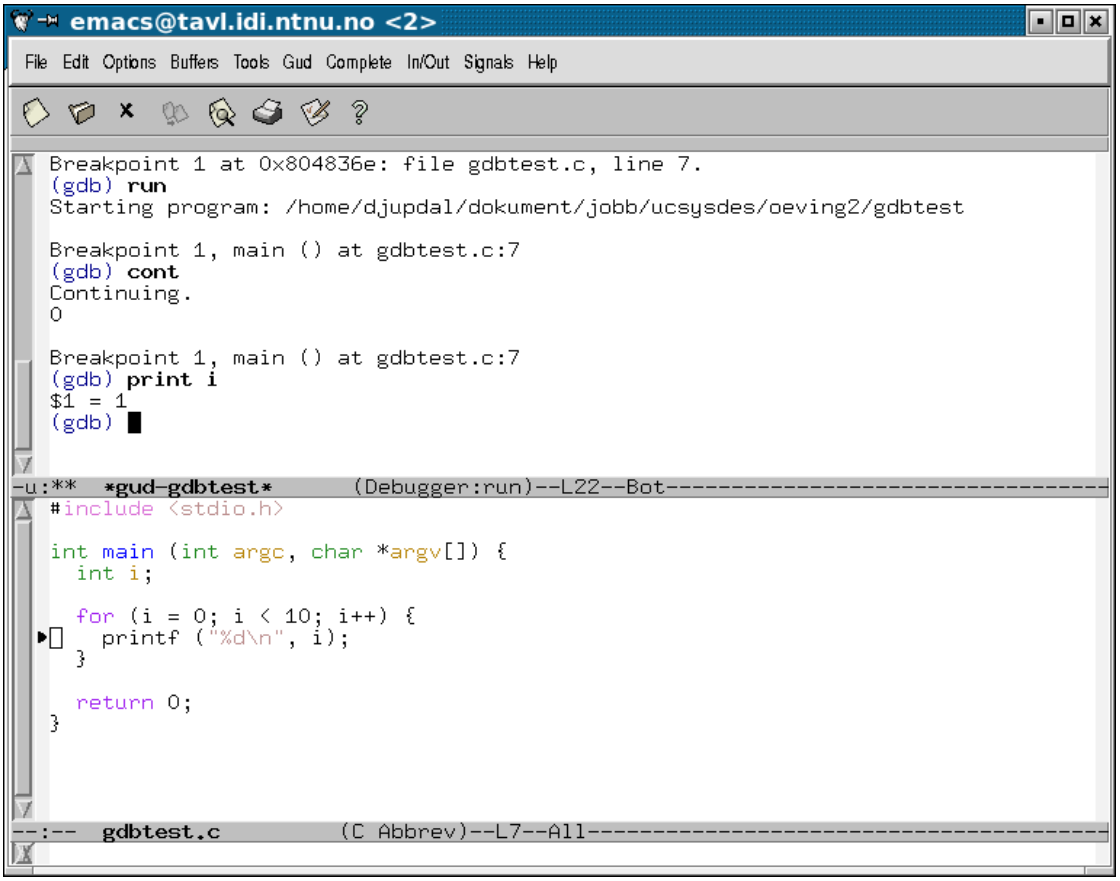

Figure 3.7.: Debugging with GDB in Emacs

way in which GDB is used is the same in any case. This is contrary to some of the IDE tools which hides what actually happens.

To start debugging with GDB, first you need to open one of the source files of your program. Thereafter you start GDB from the tools menu and specify the arguments for GDB (see the previous section). GDB commands are described in the next section.

3.5.2. GDB Commands

GDB is a large program and it can not be fully documented here. GDB manual [3] can be found as an `info` page. In addition, there is an online help system and a tutorial with FAQs on the web [14].

To connect to the development board through gdbserver, certain startup commands are needed. These are provided with the exercises as a .gdbinit file which you must place in your home folder before starting GDB.

Here is the list of the most important commands, but you are encouraged to read the manual and turn to the GDB help system to find out more about how these can be used.

- `set $reg = value`
  Set register.
- `bt`
  Print out stack trace.
- `info registers`
  Show register file.
- `x/nx address`
  Show the memory contents where `n` is a number which specifies how many words will be shown
- `help`
  Online help. It is good, use it.
- `monitor reset`
  Reset the development board
- `cont`
  Continue with running after stop.
- `si`
  Run a single instruction.
- `s`
  Run a single line of C code.

- `print <expression>`
  Evaluate expression (for example, show the contents of a variable). Example: `print varA` will print out the value of the variable "varA".
- `display <expression>`
  The same as `print` but it will print out the value of the expression every time the program execution is stopped (for example, after each `si`).
- `break <place>`
  Set a breakpoint (the line in the source code at which the program will automatically stop). It can be set more easily with the help of Emacs: Go to the source file in Emacs where you would like to place a break point and press `C-x <SPACE>` to set a breakpoint on the line where the cursor is.
- `watch <expression>`
  Set a watch point. It will stop the execution when the expression (for example, a variable or a register) changes the value.
- `info break`
  Show all breakpoints.
- `delete <nr>`
  Remove break– or watchpoint.
- `quit`
  Exit GDB.

3.5.3. Interrupts, sleep and GDB

It should be noted that *GDB, interrupts and EFM32 sleep modes do not mix well.* You may observe strange behavior from GDB if the core gets turned off as part of a low energy mode or receives interrupts while debugging. To avoid this, comment out the parts of your code that cause the microcontroller to enter sleep prior to a debugging session. Similarly, single-stepping while receiving an interrupt may cause problems. Using breakpoints inside the interrupt handler instead of single-stepping may work better, but can still cause hiccups.

3.6. Description of the Exercise

Write an assembly program which enables a user to control the LEDs in some way by pressing the buttons. One example can be to only light one of the LEDs, but allow the user to move the glowing dot right or left by pressing the corresponding buttons. Another example can be to display various patterns or even various intensities when buttons are pressed.

It is required that you write an interrupt routine (an interrupt handler) for reading the buttons. It is also a requirement that you use a Makefile for the exercise and you can debug the program with GDB. Remember that if you use a delivered Makefile instead of making your own, you need to look into it to see how it works.

Use the energy monitor to see the power requirements of your program. Analyze and discuss (or implement) improvements.

3.6.1. Recommended Approach

1. Download the support files. They can be found in a .tgz file named "ex1.tgz" which can be unpacked with the following command line:

   `tar zxvf ex1.tgz`

   You will find the following files:

   - `ex1/Makefile`: A Makefile for exercise 1
   - `ex1/ex1.s`: An assembly file which can serve as a starting point for your assembly code
   - `ex1/efm32gg.s`: An assembly file with useful constants
   - `ex1/.gdbinit`: Init file for GDB. Copy this to the home folder. Necessary for using GDB.

2. Familiarise yourself with the tools

   - Use assembler and linker to compile and link delivered files by running the `make` command
   - Upload the program to the devboard by running `make upload`
   - Try GDB by single stepping instructions, inspecting registers etc.

3. First write a variant of the program without using interrupts.
4. Then, develop the program with the use of an interrupt routine.
5. If you have time, extend your program with functionality, for example using energy modes and/or more advanced LED behaviour.

3.6.2. Tips

- Do points 3 and 4 in stages (with GDB accompanying your work):
  - Begin by enabling the GPIO clock
  - Set up the LEDs and make sure that you can turn them on and off.

- Set up the buttons and try to copy the button values directly to the LEDs
- Put the code for reading the button status in a subroutine.
- Make a main loop which manipulates the LEDs according to which buttons are pressed down.
- Put the code for reading the button status into an interrupt routine and set up interrupts.

4. Exercise 2

4.1. Introduction

In this exercise you will make sound effects. You will make different sound effects which will be played when different buttons are pressed. The code is to be written in the C programming language.

4.1.1. Learning Outcome

The learning outcome in this exercise is:

- C programming
- GPIO control in C
- Use of the microcontroller's DAC and timers for sound generation
- Interrupt handling in C

4.1.2. GNU Compiler Collection (GCC)

You will use the GNU C compiler: GCC. Use the info command to read the documentation for GCC. Here is a brief description of how GCC can be used for compiling a C file for our microcontroller:

```
arm-none-eabi-gcc -mcpu=cortex-m3 -mthumb -Wall -g -c -o <outputfile.o> <inputfile.c>
```

The arguments for this option have the following meaning:

- `-mcpu=cortex-m3 -mthumb`: Specifies our instruction set
- `-Wall`: Turn on all the warnings, this will generate warnings for those things which are allowed in a C code but considered bad practice
- `-g`: Enable the use of GDB for debugging (all symbols are included in the executable file)
- `-c`: Make an object file (since we handle linking manually)

- `-o <outputfile.o>`: Write the results into a file outputfile.o
- `<inputfile.c>`: C file to be compiled.

The GCC is a complex piece of software with many more possible command line options, but these will not be further documented here. The parameters given here should be sufficient for the scope of this exercise.

Linking

In Exercise 1, you have already learned to link object files which were generated by the assembler. The same linker is used for linking object files generated by the C compiler. However, for C programs there are some extra object files and libraries which must be linked in. This is, among others, start up code which sets a stack pointer, initializes the C runtime and which calls the main() function in your program. In addition, a standard C library is very useful to link in, as it gives you access to a wide range of pre-implemented functions.

C files can be linked with the `gcc` command, there is no need to use the `ld` command directly. If `gcc` is called as if it were `ld` (without providing C files in the list of arguments but instead giving one or more object files), it will run `ld` for your object files but with the correct arguments for linking C programs. In other words: you can perform linking as in the exercise 1 but do it with `arm-none-eabi-gcc` instead of `arm-none-eabi-ld`.

Linker command when linking your C files:

```
arm-none-eabi-gcc -mcpu=cortex-m3 -mthumb -T lib/efm32gg.ld -g -lgcc -lc -
lcs3 -lcs3unhosted -lefm32gg -Llib
```

- `-mcpu=cortex-m3 -mthumb`: Specifies our instruction set
- `-lgcc -lc -lcs3 -lcs3unhosted`: Link to various useful libraries
- `-lefm32gg`: Link to EFM32GG startup code
- `-Llib`: Add `lib` to the library search path

4.1.3. HW Access From C Code

C programs can control hardware directly. It is simply a matter of using pointers that point to the memory mapped I/O registers. A list of useful memory mapped registers is provided in efm32gg.h in the exercise support files, so you can include this file in your source code to avoid having to re-define these.

Example: Writing and reading GPIO_PA_DOUT:

```
// define register pointer as a constant
// already defined if you #include "efm32gg.h"
```

```
#define GPIO_PA_DOUT ((volatile uint32_t*)(0x4000600c))

*GPIO_PA_DOUT = 0xff00; // write 0xff to register

uint32_t x = *GPIO_PC_DOUT; // reading register PC, saving in variable x
```

4.1.4. Interrupt Handling in C

There is no standard way in which an interrupt is set up in C. You use GCC and EFM32 and therefore you must learn the method for handling interrupts in this system.

In these exercise, you must program the hardware yourself, which means you have to write the correct I/O registers to enable interrupts in the same way as you did in exercise 1. Remember that you must enable both the interrupt generation (in the peripheral), and the interrupt handling (in the NVIC).

The exception vectors are already filled in for you by the startup code. All you have to do to write a new exception handler is to write a function with the correct name. If the name is correct, the function will be used as the exception handler for the particular exception.

The following exception handlers are relevant for this exercise. If you want to use others, look at the support files for the exercise. Note the use of `__attribute__ ((interrupt))`. This is an indication to the compiler that the function should be generated for an exception handler.

```
/* handler for TIMER 1 interrupt */
void __attribute__ ((interrupt)) TIMER1_IRQHandler() {
}

/* handler for GPIO interrupts for even pins */
void __attribute__ ((interrupt)) GPIO_EVEN_IRQHandler() {
}

/* handler for GPIO interrupts for odd pins */
void __attribute__ ((interrupt)) GPIO_ODD_IRQHandler() {
}
```

4.2. Hardware Timers

EFM32GG contains several timers that can give interrupts at periodic intervals. These timers are documented in section 20 in the EFM32GG Reference Manual [13]. It is useful to remember that the timer counter registers are 16 bits, and that the core clock (which the timer clock is derived from) runs at 14 MHz by default.

To enable a timer:

1. Enable clock to the timer module by setting bit 6 in CMU_HFPERCLKEN0
2. Write the period (number of cycles between interrupts) to register TIMER1_TOP
3. Enable timer interrupt generation by writing 1 to TIMER1_IEN
4. Enable timer interrupts by setting bit 12 in register ISER0
5. Start the timer by writing 1 to TIMER1_CMD
6. Write an interrupt handler like this:

```
void __attribute__ ((interrupt)) TIMER1_IRQHandler() {
  // handler code here
}
```

7. In the handler, remember to clear the interrupt by writing 1 to TIMER1_IFC.

4.3. Sound Generator: Digital to Analog Converter (DAC)

The EFM32GG has an interal DAC (Digital to Analog Converter) which is connected to an amplifier on the DK3750 development board. This amplifier drives the Audio Out connector on the board, where you can plug in headphones.

The DAC is a piece of hardware that generates an analog signal based on digital values. The EFM32GG DAC is documented in section 29 in the EFM32GG reference manual [13].

4.3.1. Sound Wave Synthesis

The physical basis for sounds lies in the waves which are created by oscillations in some media. Therefore, the properties of sound are the properties of the wave: frequency, period, amplitude. Average hearing limits for humans are 20HZ lower and 20 kHz upper, with some individual variations.

So, how to generate a sound? There are many possibilities but one suggestion is to make a synthesiser in software. This means that you will generate sound waves synthetically. The sound is nothing else but the repetition of oscillating waves of certain frequency and amplitude. The frequency defines the tone of the sound and the amplitude defines the strength of the sound. See Figure 4.1.

To make a synthesiser, you need to decide upon the waveform you would like your synthesiser to generate. Three periods of some of the common synthesizer waveforms are shown in figure 4.2. You need to repeat this period many times, as long as you would like the tone to last.

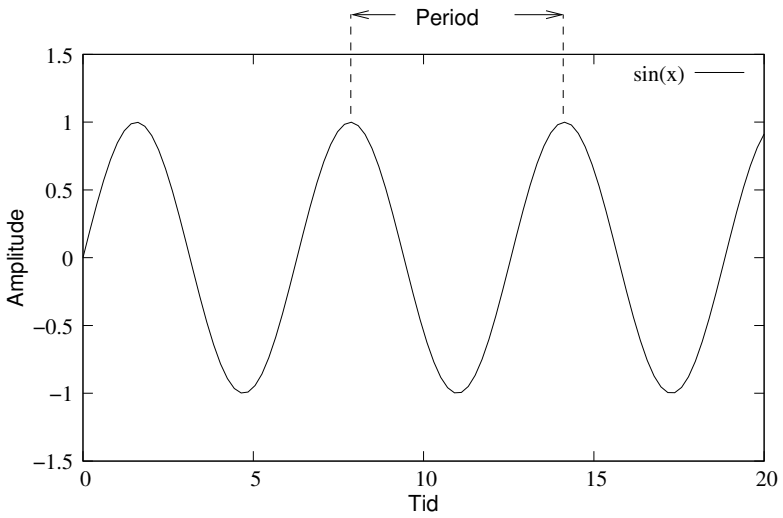

Figure 4.1.: Sound wave

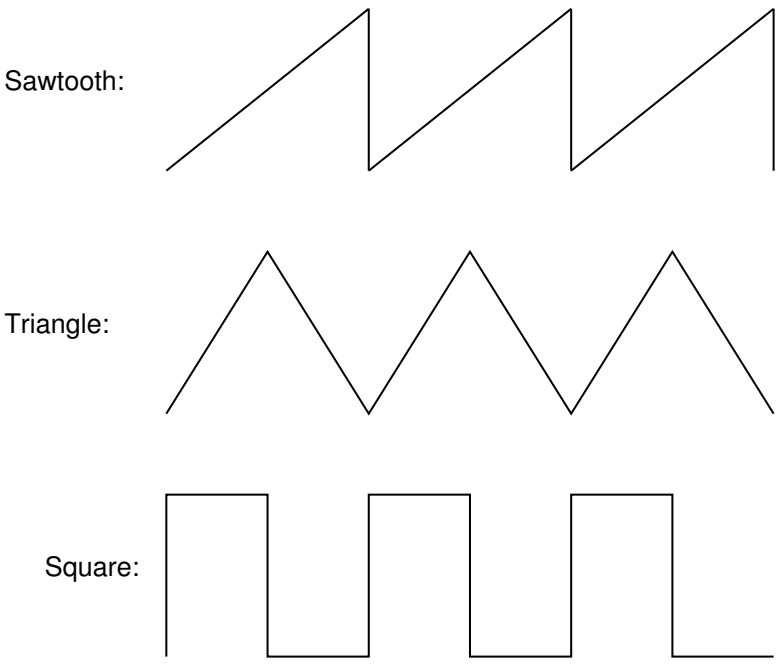

Figure 4.2.: Various sound waves

In the digital world, everything is discrete, so you need to generate the waveforms using discrete samples. It is typical to have 44100 or 48000 samples per second, but this is entirely up to your implementation. Each period must thus be divided in a certain number of samples. If you want to create a sound with frequency 441 and you decide to generate 44100 samples per second, each period in your waveform must consist of 100 samples.

Each sample must be written in a continuous stream to a data register in the DAC. This is easiest done by setting up a timer interrupt that gives an interrupt every time you need to push a new sample to the DAC.

The simplest way of using the DAC is to use the continuous mode with a high sampling frequency (which is decided by the prescaled DAC clock). To use the DAC, do the following:

- Enable the DAC clock by setting bit 17 in CMU_HFPERCLKEN0.
- Make sure the DAC clock is prescaled correctly by writing 0x50010 to DAC0_CTRL. This gives a frequency of $\frac{14}{32}$ MHz = 437.5 KHz in continuous mode, and enables the DAC output to the amplifier.
- Enable the left and right DAC channels by writing 1 to registers DAC0_CH0CTRL and DAC0_CH1CTRL.
- Push a continous stream of samples to the DAC data registers (DAC0_CH0DATA and DAC0_CH1DATA), for example from a timer interrupt.

Remember that this is a 12 bit DAC, all samples must therefore be scaled (or generated) to 12 bits. Try adding a constant positive offset if the sounds you generate are too silent (or a negative offset if too loud).

4.4. Description of the Exercise

Write a C program that runs directly on the development board (without support of an operating system) and which plays different sound effects when different buttons are pressed. Each generated sound effect will be a sound effect you may use in the final game. You have to make at least three different sound effects (for example, cannon shot, target hit, player win etc.). Make a start up melody which can be played when the game begins.

4.4.1. Recommended Approach

1. Download the support files for the exercise 2 (“ex2.tgz”) and unpack them in the same way as in exercise 1. The following files can be found:
   - `ex2/Makefile`: An example Makefile

- `ex2/ex2.c`: C file containing example main function
- `ex2/{gpio/dac/timer}.c`: C files for HW setup code
- `ex2/interrupt_handler.c`: C file for interrupt handers
- `ex2/efm32gg.h`: Header file with useful I/O registers
- `ex2/lib/efm32gg.ld`: Linker script
- `ex2/lib/libefm32gg.a`: Library containing startup code

2. Make sure that you can use the tools correctly:
   - Test whether the Makefile works and everything compiles and behaves as it should.
   - Upload the executable file to the development board (you can use `make upload`).
   - Try debugging in GDB by single stepping C lines, inspecting the variables etc.
3. Examine the provided skeleton code to ensure that you understand its structure, and the required functionality for the assignment. An example of hardware access in C code is provided in `gpio.c`.
4. Begin programming for the exercise:
   - The GPIO interrupts are probably the simplest to work with. You can try to implement in C what you did in Exercise 1 as a warm-up.
   - Verify that you can set up the timer and handle the periodic interrupts. This can be done by incrementing a value at every timer interrupt, then lighting the LEDs based on this value.
   - Verify that the DAC works correctly by setting it up and sending it some random data. You should hear some noise in the headphones. Alternatively, you can use the sine generation mode described in Section 29.3.5 of [13] for simple DAC testing.
   - Once you have the timer interrupts and the DAC working, it should be straightforward to combine them for generating sound. You can encode more complex melodies in your code in the form of sample arrays or generator functions.

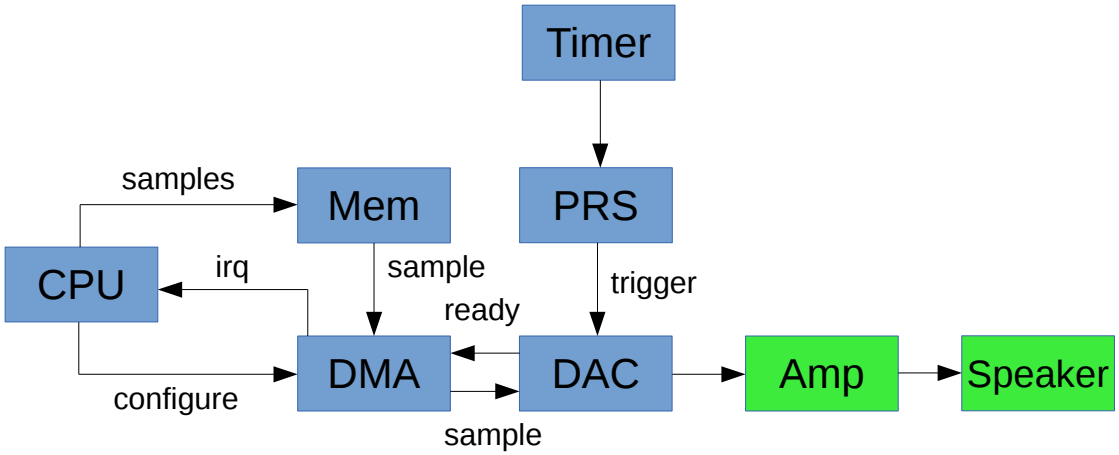

Figure 4.3.: Overview of DMA-DAC system

4.4.2. Tips

- *Be careful with your ears*: You can easily generate dangerously loud sounds. When testing your code, do not put headphones over your ears before you are sure the loudness is acceptable.
- *Code organization:* The skeleton code provided in the support material consists of several different files. Try to organize your solution so that code with similar responsibilities is contained in a its own file. This makes the code easier to understand, navigate and reuse.
- *Energy efficiency*: Remember to consider energy efficiency. You can generate sound even with busy-waiting in a loop and writing to the DAC afterwards, but this will be very energy-inefficient.

4.5. Advanced: Using DMA for Feeding the DAC

While using the DMA generally results in higher energy efficiency, this is an advanced technique and not required for the exercise.

From a data movement perspective, sound synthesis is simply copying data from a source buffer (the samples to be played) into a destination buffer (the DAC data registers) at certain intervals. Modern microcontrollers often contain a Direct Memory Access (DMA) unit that can copy data without CPU intervention. In this scenario, the CPU sets up a *DMA Channel* with a *DMA Descriptor*, which describes the source and destination addresses and the size of the data to be moved. The DMA will then perform a data copy every time this channel is triggered, while the CPU is sleeping or performing other

computations. However, implementing this will require a lot of reading of the EFM32GG Reference Manual [13]. You may also want to read the application note on DMA [11].

EFM32GG has a system for signaling between units without requiring CPU involvment. This is called Peripheral Reflex System (PRS) and is central for using DMA for DAC. Read about PRS in section 13 in [13].

An outline of the necessary steps for using DMA is shown in figure 4.3. Briefly, what is needed is the following:

- Enable the PRS and timer clocks with CMU_HFPERCLKEN0.
- Enable the DMA clock. This is *not* done with CMU_HFPERCLKEN0, but with CMU_HFCORECLKEN0. See section 11 [13].
- Setup the PRS system such that one of the timers trigger a PRS channel.
- Setup the timer with correct period (same as before). No need for timer interrupts now, this is just to create the PRS trigger.
- Setup the DAC to trigger on the PRS channel.
- Setup the DMA control block for Ping Pong mode (section 8.4.3 in [13]). Note that the control block must be 512 byte aligned in memory.
- Setup the DMA to send a new sample to the DAC every time the DAC is ready (SOURCESEL and SIGSEL in DMA_CH0_CTRL).
- Enable DMA interrupts and use the DMA interrupt handler to generate new blocks of data for the DMA to transfer.

5. Exercise 3

5.1. Introduction

In the last exercise in the course, you will make a computer game. The game will not access HW directly, but instead go through drivers in the Linux kernel. One of these drivers, the gamepad driver, you have to make yourself. You may choose which game to implement yourself, but remember the limited resources of this platform. Only simple games are realistic.

Possible examples:

- Pong
- Asteroids
- Breakout
- Any other classic game from the 80's...

All your program code will be written in C (you are encouraged to reuse the code from exercise 2) and you will use Linux as the operating system.

There are several parts in this exercise. First, you will compile a Linux installation for the development board. Then you must write a kernel driver for the buttons. The last part is to actually implement a game that uses your driver.

5.1.1. Learning Outcome

The learning outcome of this exercise is:

- C programming for Linux
- Programming Linux device drivers:
  - Compiling the Linux kernel
  - Programming hardware in Linux
  - How to make your own device drivers in Linux
- Interacting with hardware through Linux device drivers

5.2. Terminology

- **Bootloader**: A small program that executes right after reset and makes the necessary preparations for the operating system to start.
- **Kernel**: The core of the operating system, a program that manages all other running programs and provides them with a way of accessing system resources.
- **Device Driver**: A program that provides a way of managing and accessing a particular piece of hardware, serving as an abstraction layer for other programs.
- **Kernel Module**: A type of small program that extends the functionality of the kernel, often loadable after the system has started, which offers more flexibility. Device drivers may be provided in the form of kernel modules.
- **Root filesystem (rootfs)**: The filesystem at the root directory of a Unix/Linux system (denoted /), made available when the operating system has booted. It may include useful files such as configuration files, utility executables, kernel modules and other software.
- **(Linux) Distribution or Distro**: A complete Linux package including the kernel and all the non-kernel software.
- **Toolchain**: A set of utilities for translating source code into executables, often including assembler, compiler, linker, debugger and basic libraries.

5.3. Overview of ptxdist and build system

For this exercise, you will be developing software not on a bare-bones environment, but for a variant of the Linux operating system called uClinux. The most important difference with respect to the development workflow is the build system. The complete Linux distribution is built using a build system called `ptxdist` [9]. ptxdist will compile the kernel, modules and all other software, and package it into binary files that can be flashed to the development board. This is a common development workflow for embedded operating systems. Figure 5.1 provides a schematic overview of the process.

Although the necessary skeleton code has been provided for you in the exercise support files, you must first familiarize yourself with the basics of the `ptxdist` build system to achieve the goals of this exercise. Table 5.1 provides a summary of useful commands, which are are further explained in the following subsections.

It is important to remember that all commands must be run from the OSELAS.BSP-EnergyMicro-Gecko directory delivered in the exercise support files.

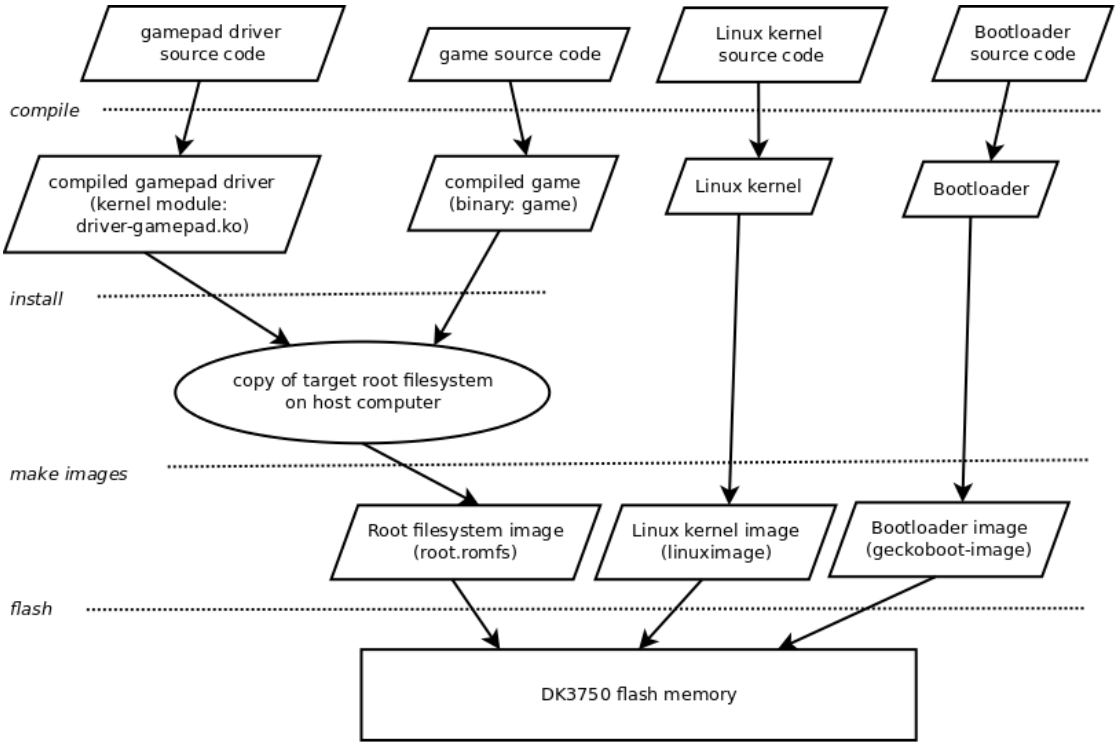

Figure 5.1.: Schematic overview of the build process for Exercise 3. The `ptxdist go` command provides a shortcut for the compile & install steps for all selected packages.

Command	Description
`ptxdist kernelconfig`	Configure the kernel
`ptxdist menuconfig`	Select packages to install
`ptxdist clean <package>`	Clean specified package
`ptxdist compile <package>`	Compile specified package
`ptxdist targetinstall <package>`	Re-install package to root filesystem
`ptxdist go`	Compile and install all desired packages
`ptxdist image root.romfs`	Make root filesystem image
`ptxdist images`	Make all images
`ptxdist test flash-romfs`	Flash root filesystem image onto the board
`ptxdist test flash-all`	Flash all images onto the board

Table 5.1.: Summary of useful `ptxdist` commands

5.3.1. Setting up the project

To get started, extract the contents of the exercise 3 support files archive into a folder, and issue the following commands to set up the ptxdist project:

1. `cd OSELAS.BSP-EnergyMicro-Gecko`
2. `ptxdist select configs/ptxconfig`
3. `ptxdist platform configs/platform-energymicro-efm32gg-dk3750/platformconfig` [1]
4. `ptxdist toolchain <path-to-toolchain-bin-directory>`

<path-to-toolchain-bin-directory> depends on where the toolchain is installed, for example under `/opt/ex3/` on lab machines. Read the README file in the top directory for the exact value, or try to find it yourself. These operations tell ptxdist where to find the necessary configuration files and the toolchain in order to build the exercise.

A word on performance: Remember that home directories on lab computers are located on network drives [2]. This may cause the build process to go slowly when, for instance, extracting compressed archives with many small files in it (e.g the Linux kernel). To speed up the process, you may extract the support files to a folder under `/tmp` (which is mapped to the local hard drive) and do all the compilation operations from there, but keep in mind that this directory is intended for *temporary* usage and its contents may disappear. So remember to copy all the files back into your home directory before logging off.

5.3.2. Configuring the kernel and selecting packages

The default configuration supplied in the exercise support files is enough to get you started and you don't have to modify it. However it is still useful to know which options for kernel configuration and additional software packages.

Configure your Linux kernel with the following command:

```
ptxdist kernelconfig
```

This brings you into the Linux kernel configuration tool.

Configure your Linux distro with the following command:

```
ptxdist menuconfig
```

Note that not everything can be compiled for the EFM32GG, the possibilities are actually quite limited. You can also see that the packages called "game" and "driver-gamepad" (which you will be developing) have already been enabled. If you would like to create a new package (e.g for the sound driver), you can issue the following command(s):

[1] You may get a "toolchain not found" error at this stage, this is safe to ignore.

[2] You may also get a ptxdist warning about "not on local disk", this can be disabled with ptxdist setup -> Developer Options -> disable local disk check

- `ptxdist newpackage src-linux-driver` (new skeleton driver)
- `ptxdist newpackage src-make-prog` (new user application)

This will create a new skeleton package in the `local_src` folder. After this, you must use `ptxdist menuconfig` to enable the installation of the new package.

5.3.3. Building and Flashing

After configuring, use the following command to build everything:

```
ptxdist images
```

Flash the DK3750 board with the following command:

```
ptxdist test flash-all
```

This will take some time, as all the image files including the kernel, the bootloader and the root filesystem will be transferred to the board. While developing the driver and the game you do not need to re-flash all the images, only the root filesystem. This will speed the flashing process, and can be done by:

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

5.3.4. Running Linux

After you have flashed the DK3750 with your linux distribution, it will start automatically after a reset just like in exercises 1 and 2.

Note that the display must be in EFM mode for Linux to boot. Press the AEM button (not PB1) on the board to change mode. You should see a picture of Tux (the penguin mascot of Linux) on the LCD when the booting is completed.

5.3.5. Communicating with Linux

You will need a serial terminal emulator to communicate with uClinux running on the board, since uClinux will send all text output through the serial port. You can use

miniterm.py[3] installed on the lab machines towards this end. You can launch it as follows:

```
miniterm.py -b 115200 -p /dev/ttyS0
```

The following serial port parameters are applicable if you need to set them up for other terminal programs:

- baud rate: 115200 bps
- 8 databits
- 1 stop bit
- No parity
- No flow control

When Linux boots, you will get a command line terminal through miniterm. You will notice that the number of programs installed are very limited. You can try to run the skeleton code for the game by simply executing the command `game`:

```
/ # game
Hello World, I'm game!
```

Note that this program uses the standard C runtime function `printf` to produce output on the console. You can also load the skeleton gamepad driver with the following command:

```
/ # modprobe driver-gamepad
[   12.860000] Hello World, here is your module speaking
```

5.4. Using Device Drivers

In the previous exercises, your program was initializing and communicating with hardware directly. This will be different in this exercise; you will be communicating with hardware via device drivers. The purpose of this section is to introduce you to the usage of device drivers by interfacing an already-made driver; the framebuffer (which controls the LCD screen).

You are strongly advised not to use much time on this (e.g by drawing complex graphics or animations) for now, as the main part of the exercise is developing your own driver for the gamepad. Once the driver is finished, you can spend as much time as you wish on implementing your game.

[3] Several other alternatives for installation on your own machine are minicom, cutecom and gtkterm

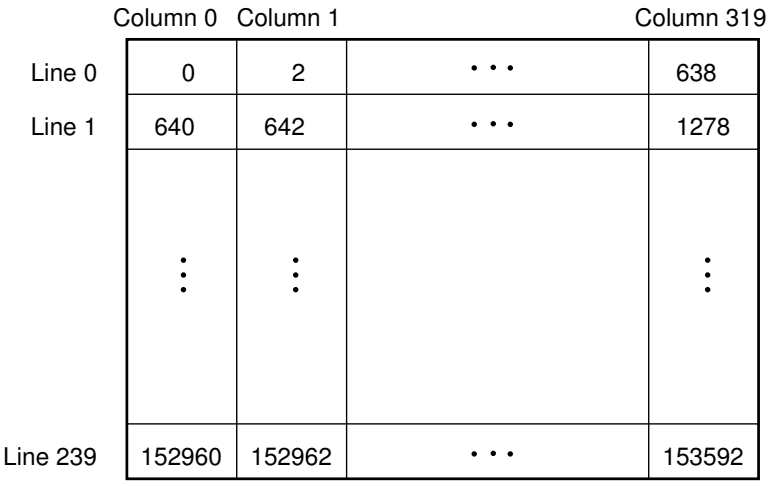

Figure 5.2.: Organisation of framebuffer. Every square corresponds to one pixel and the address of this pixel (relative to the first pixel) is written in the square.

5.4.1. Accessing Drivers from User Space Programs

In Unix (and Linux) nearly everything is represented as a file. This holds for drivers as well. Each driver has a corresponding file in the directory `/dev`. In order to use the driver, you need to open the driver file as if it was a common file, and the access to the driver's functionality is provided by the usage of the common functions for file I/O.

Common user programs can call the following Linux functions in order to get access to the files:

- open(): Open file
- close(): Close file
- read(): Read from file
- write(): Write to file
- lseek(): Search in file (change the position in the file)
- ioctl(): Give special commands (e.g. to the driver)

These functions are documented as man pages. Check for yourself how they are used by reading man pages (for example `man 2 open`).

A driver must implement support for these operations if the user program should have the possibility to use the functions for driver access. This is described in section 5.5.2.

Bits:	15–11	10–5	0–4
Contents:	Red	Green	Blue

Figure 5.3.: Organisation of each pixel in the framebuffer

5.4.2. Framebuffer Device

To get access to the display on the development board you have to program for the "Framebuffer device". This is a device (`/dev/fb0`) which represents the graphic memory. As a small demonstration, you can do the following from the shell to draw a small blue line on the upper left part of the screen:

```
echo "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ" > /dev/fb0
```

From your own application, you can use it by opening the driver `/dev/fb0` as a file. It is possible to ask the driver about the type of display it is connected to (for example, the size of the screen), but you don't have to do that in this exercise. You can just assume the following for the screen:

- Screen size: 320x240
- Number of bits per pixel: 16 (5 for red and blue, 6 for green)

The screen is organised as shown in figure 5.2. Each pixel in the framebuffer is organised as shown in the Figure 5.3. To write to one pixel on the screen, you can search the right byte by using the function lseek() followed by the function write() to write the right value.

An easier (and quicker) way to do this is to map the driver to an array in the memory. Then you will be able to write pixels by writing directly to a usual C array. This memory mapping can be done by the function mmap(). This is something we strongly recommend, check the man page: `man 2 mmap`.

If you use mmap, one peculiarity of this system is that it needs to know about updates to the framebuffer before changes are visible on the display. This is done with the following code:

```
#include <linux/fb.h>
// setup which part of the framebuffer that is to be refreshed
// for performance reasons, use as small rectangle as possible
struct fb_copyarea rect;

rect.dx = x;
rect.dy = y;
rect.width = width;
rect.height = height;

// command driver to update display
ioctl(fbfd, 0x4680, &rect);
```

5.5. Writing Device Drivers

As previously mentioned, you will have to write a device driver in order to be able to use the gamepad in your application. This device driver will be made in the form of a kernel module. The following subsections will first describe the more general concept of kernel modules, then continue with device drivers.

This compendium gives a brief introduction to how the kernel modules and drivers can be made, but does not provide all the necessary details. To complete Exercise 3, you will have to read parts of the book "Linux Device Drivers (LDD)" [2] which can be found in Akademika or downloaded for free from the net.

Keep in mind that LDD is not updated for info on the latest Linux kernels. There may, therefore, be some slight deviation from what the book says in this exercise. The compendium will describe the parts that are different, but LDD will still give a very nice general overview of kernel modules and drivers.

5.5.1. Kernel Modules

Linux runs in two different modes, as the majority of modern operating systems: user mode and kernel mode. User programs run in *user mode* and they have limited permissions. The Linux kernel runs in *kernel mode* and it has access to everything. Drivers must typically run in kernel mode and therefore they must be programmed as part of the kernel. Fortunately, the Linux kernel supports loadable modules so that it is possible to make a kernel module which can be dynamically linked to the kernel when the module is loaded, instead of having to recompile the entire kernel.

Kernel modules have extension ".ko". They can be loaded by the use of the following command:

```
modprobe <modulename>
```

To remove a module which is running as a part of the kernel (must not be actively in use):

```
rmmod <modulename>
```

To get the list of all the loaded modules:

```
lsmod
```

A kernel module is a small program which runs in kernel mode. Therefore, kernel modules have to follow certain rules, they can't have the same behaviour as user space programs. Here is the list of limitations:

- The kernel module must implement a strictly defined interface (i.e. a set of functions) so that the kernel knows exactly how the module should be used

- A kernel module cannot call other functions than those which are defined in the Linux kernel itself. That means that none of the functions from the C standard library can be used.
- Kernel modules have to be programmed with parallelism in mind – a module must always function correctly even if different processes are trying to use it at the same time. If this is not possible, the module must deny access from more than one process at a time.
- Finally, all kernel modules are event based. They can't have loops which are running to eternity because that would cause the kernel to hang. Instead, they have only functions which are called every now and then when other programmes need access to the module.

Consult Chapter 2 of LDD for more information on kernel modules.

Interface Between Kernel and Modules

A kernel module must implement two functions which are called by the kernel:

- **static int** __init my_init (**void**): This is the function which is called when the module is loaded, you should do everything necessary for setup of your module here.
- **static void** __exit my_exit (**void**): This function is called when the module stops being used. Here you will deallocate everything you have allocated in the init-function.

You can name these functions as you wish but the kernel needs a piece of information about the module in order to be able to use it and, among other things, it needs to know the names of the init- and exit functions. This is achieved by calling the following macros at one or another place in the module's source file (typically at the bottom of it):

```
module_init (my_init); /* specifies which function will be used as init */
module_exit (my_exit); /* the same, but for exit function */
MODULE_LICENSE ("GPL"); /* specifies the license for the code */
```

Printing

Kernel modules cannot call other functions other than those which are defined in the kernel. Therefore, you cannot use printf() as in common C programs. Rather, there is a corresponding function in the kernel. Here is an example of its use:

```
printk(KERN_INFO "Variable value %d\n", i)
```

It prints out the value of variable i. KERN_INFO means that this message is only an info message. Mark that there is no comma after KERN_INFO.

5.5.2. Device Drivers

In Linux (and other Unix systems) there are several types of drivers. In this exercise, we shall only work with a type called "char device".

Drivers must be accessed as if they were files on a hard disk so there must exist a way to associate a file name with a driver. Linux does this by automatically creating files in the `/dev` directory that represents the drivers.

The following is the minimum that has to be done in order to make a char device driver:

1. Make the driver as a kernel module
2. Allocate and memory map access to the I/O hardware registers which will be used
3. Initialize hardware
4. Allocate character device structures
5. Implement a set of functions which perform file operations (open/close/read/write) on the driver and register these in the system
6. Activate the driver

All these things can be done in the module init function (see section 5.5.1). This is the old way of implementing drivers and can be used in this exercise. However, modern linux drivers should be structured slightly differently, something which is discussed in appendix E.

We shall briefly see how each of these points is performed, but you will get a better and more thorough explanation by reading chapters 2, 3 and 9 in "Linux Device Drivers" [2].

Asking for Access to I/O Ports

The driver should not just use the hardware without asking for access first. This is to prevent several drivers from accessing the same hardware. Instead, you should ask for access with the function call request_mem_region() which is described in the section "I/O-Port Allocation" of the Chapter 9 in "Linux Device Drivers".

In addition to allocating the memory region, it must generally be mapped into the virtual memory space. In this exercise you are not using virtual memory, but it is still a good idea to follow standard Linux practice. Memory map the allocated I/O region with the function ioremap_nocache().

Initializing and Using the Hardware

ARM has memory mapped I/O. Therefore, special functions are not needed for writing or reading I/O registers. After the I/O registers are allocated and memory mapped, they can be read and written just like in exercise 2.

Clocks Clocks are handled by the operating system and should not be enabled by writing registers directly. For this exercise, all relevant clocks are already turned on.

Gamepad You can use GPIO to read gamepad buttons, just like in exercises 1 and 2. You should *not* use the gamepad LEDs in this exercise, except the LEDs controlled by PA12, PA13 and PA14. These should be safe to use, but ensure that you do not modify the rest of the Port A direction or data registers. The other PA pins are used by the operating system.

Input Handling There are at least three different ways of implementing input handling from the gamepad, which have different degrees of technical difficulty and efficiency:

1. Naive solution (without interrupts): User application polls the driver (through the devnode). Each time the driver is polled, it reads the button GPIO registers and returns them to the user application.

2. ”Half-solution” with interrupts: Driver handles the button interrupt and copies the GPIO register values into the devnode-mapped memory. The user application polls the driver and reads the data from the devnode.

3. ”Full solution” with interrupts: User application registers ”signal handler”. Driver handles the hardware button interrupt, copies data into devnode-mapped memory and generates a signal. This causes the user application’s signal handler to be invoked, which then reads the devnode-mapped memory. You can read about asynchronous signals in Chapter 6 of LDD.

If you decide to use interrupts, they can *not* be initialized like in exercise 2. Instead, they must be allocated and initialized by the kernel. For this, you use the request_irq() function. This function registers your interrupt handler (given as an argument) and enables the interrupt. Read about this in chapter 10 in “Linux Device Drivers”. Note that you must still program the registers in the I/O controllers (for example the GPIO_IEN register) just like in exercise 2, it is only the general IRQ handling that is handled by the kernel.

The IRQ numbers you pass to request_irq() are not the same as the ones specified in the EFM32GG manual. Instead, use the IRQ numbers from table 5.2.

IRQ source	IRQ number	Platform IRQ index	Platform mem index
GPIO Even	17	0	0
GPIO Odd	18	1	0
Timer 3	19	2	1
DMA	20	3	2
DAC	21	4	3

Table 5.2.: Platform device data for TDT4258

Timers There are kernel timers you can use [7]. You can also use one of the hardware timers. As some of these are used by the operating system, you should limit yourself to using only timer 3.

Allocating Character Device Structures

You need to initialize certain structures and make certain functions available for the OS in order to provide user space with access to the driver.

Every char device needs a unique device number that identifies the device. This number is split into a major and minor number. Allocation of the device number is performed by the function call alloc_chrdev_region().

This is explained in the section "Allocating and Freeing Device Numbers" in Chapter 3 of the "Linux Device Drivers" [2].

Registration of File Functions and Activation of the Driver

To make it possible for the user program to handle the driver as a file, the driver has to implement support for access. This is done by implementing the following four functions in your module:

```
/* user program opens the driver */
static int my_open (struct inode *inode, struct file *filp);

/* user program closes the driver */
static int my_release (struct inode *inode, struct file *filp);

/* user program reads from the driver */
static ssize_t my_read (struct file *filp, char __user *buff,
             size_t count, loff_t *offp);

/* user program writes to the driver */
static ssize_t my_write (struct file *filp, const char __user *buff,
              size_t count, loff_t *offp);
```

You find a description of how these functions are used in chapter 3 of "Linux Device Drivers". To register these functions so that the kernel knows how they are invoked, the following structure must be created:

```
static struct file_operations my_fops = {
  .owner = THIS_MODULE,
  .read = my_read,
  .write = my_write,
  .open = my_open,
  .release = my_release
};
```

In addition, a cdev structure must be allocated and initialized, like this:

```
struct cdev my_cdev;

...

cdev_init(&my_cdev, &my_fops);
```

The cdev structure is now initialized with a pointer to the file_operations structure. All that remains now is to pass the cdev structure to the kernel with the function cdev_add().

All this is explained in chapter 3 of "Linux Device Drivers".

5.5.3. Making the Driver Visible to User Space

As explained earlier, user space programs communicate with the driver by opening a special file in the `/dev` directory that represents the driver. The user space program can then read and write data to the driver using normal file I/O functions, and the driver will get (or put) the data through the functions mentioned in section 5.5.2.

To make the driver appear as a file in the `/dev` directory, the functions class_create() and device_create() must be called:

```
struct class *cl;
dev_t devno;

...

cl = class_create(THIS_MODULE, "my class name");
device_create(cl, NULL, devno, NULL, "my class name");
```

"devno" is the device number you got from alloc_chrdev_region() and "my class name" is the name of your device class (you can choose, but normally the same as the name of your device).

This is not documented in "Linux Device Drivers", but documentation can be found by googling or by looking in kernel header files.

5.5.4. Sound Support

There is no Linux sound driver for this development board. To get sound you will have to include support for it yourself in your own driver. Sound support is *not* a requirement for the exercise, but ambitious students could attempt this.

Performance on DK3750 is an issue. If you use a timer to feed the DAC with data, you should use a low sample rate to avoid using too much CPU. Even then you might have to limit yourself to very short sound effects where you turn off the timer interrupt when there is silence (i.e. most of the time).

A much better solution is to use DMA, as explained in exercise 2, but this is also much more technically challenging.

5.6. Description of the Exercise

The exercise consists of several parts:

5.6.1. Part 1: Linux Build and Warm-Up

Configure and build the uClinux for the board as described in Section 5.3. Familiarize yourself with the development workflow and driver interfacing, by building a simple application that uses the framebuffer driver (accessed via `/dev/fb0`) to draw a simple shape on the screen, but avoid using too much time on this.

5.6.2. Part 2: The Driver

Make a driver for the buttons. It should be implemented as a kernel module. You are free to make the driver as you wish, but the minimum requirements are to support your needs for the game to work.

5.6.3. Part 3: The Game

Complete the game. Use the framebuffer driver for writing to the display. Use your own driver for reading the status of the buttons.

5.6.4. Recommended Steps

- As for all other exercises, it is recommended that you do the work in stages. For the kernel module, a convenient procedure could be:

  - Compile the module template
  - Check that module works
  - Make it into a char device
  - Create a user program that accesses the module
  - Test whether the device works by writing and reading it. Use printk() in the module to verify that your user program actually accesses it correctly.
  - Implement support for buttons by setting up HW similarly as in exercise 2.
- For the game, the following procedure is convenient:
  - Open your own driver and make sure that you can read the buttons from user space
  - Open /dev/fb0 and make sure you can use it
  - Implement the game.
- Try to get the complete system (driver and userspace) to work together before adding features
- Do not spend too much time on fancy game features, these are not important for the grade. If you have time left, try to implement sound support or other driver enhancements which are relevant to this course.

5.6.5. Suggestions for Improvements

If time permits, some of the following can be considered:

- Experiment with power efficiency:
  - Try changing kernel configuration for reducing power (hint: tickless idle)
  - Try to minimize screen updates in the game
  - The game should sleep as much as possible
- Implement sound support in your driver to get sound effects.

5.6.6. Delivery

If you haven't made changes to kernel files/configuration/framework, delivering the `local_src` folder is enough for both the driver and the app. If you've added or changed code or configuration in the kernel, then the `local_src` folder won't be enough. An option is to remove all of the files inside the `src` folder and then deliver the entire

top-level folder with everything in it. Running `ptxdist clean` also helps get rid of cruft/intermediate files.

A. Sources of Documentation

This document does not provide enough information about everything you need. In the sections with background information you will be noted where you can find more information about the corresponding topic but there are some general information sources in linux which you need to know about.

A.1. Man Pages

Luckily, all commands in Unix–based operating systems are documented with so–called "man pages". You can read them by invoking the command `man` in shell:

```
man <name_of_the_command_you_would_like_to_know_about>
```

Try this now! Read the manual for the `man`–command itself:

```
man man
```

Man pages are divided in various sections. The most important are the first three sections:

- Section 1: User commands (those which can be called from the command line)
- Section 2: Description of system calls
- Section 3: Description of functions in C–libraries

To specify that you would like to read a man page for `printf`, you can provide the number of the section it belongs to.

`man 1 printf` will open a man page for the `printf` command.

`man 3 printf` will open a man page for the C–function `printf()`.

If you do not specify the section number, the first man page with this name is shown. In the example with `printf`, a command `man printf` will open a printf page which lies in section 1.

A.2. Info Pages

Some bigger documents and manuals are available as "info pages". This is a help system which supports hypertext documents, and therefore easier to navigate for larger documents. GNU tools are typically documented as info pages.

Info pages can be opened with info command:

```
info [name_of_the_document_you_would_like_to_read]
```

These info pages are often also available on the web, google and you will find. This can often be easier than using the info command which has a somewhat strange user interface.

A.3. Other

If you wonder how a command works and you can't find either man or info pages, you can try to ask the command itself about how it works. It depends on the command how you can achieve that but typically one of the following arguments is given:

- `-?`
- `-h`
- `--help`
- `-help`

For example:

```
ls --help
```

B. Assembly

Assembly programming is the lowest level in which a machine is programmed. We can imagine writing binary files in a machine language but that is never done. Instead, an assembler is used which takes a description of the program instructions and "assembles" them into a binary file. This must not be mixed with compiling which takes a high level language and translates it into an executable binary file. Assembly language has an almost direct mapping from the text in the assembly file to the final binary file in machine language. Therefore, each processor has its own unique assembly but, for example, C code will be (mostly) the same for all processor types. You will learn ARM Thumb assembly. Even if it is unique for ARM processors, the principles will be the same for all processors.

B.1. Instructions

An assembly file is a list of all the instructions which the program contains. Each instruction has the following form:

```
<mnemonic> <arguments>
```

Here, "mnemonic" is the name of the instruction which will be executed and arguments are a list of arguments separated by comma. The length of the list of arguments depends on the concrete instruction. Each argument is typically a register. Here is an example:

```
mov r1, r2
```

This instruction copies register 2 to register 1. The convention in ARM thumb assembly is that the first argument is the destination where the result of the instruction will be placed. In the example above, it is register 1.

All instructions are described in the Cortex-M3 reference manual [10].

B.2. Numbers

There is often a need to specify a number in different numeral systems. To specify a number in a given numeral system, set the following in front of the number:

- Binary number: `0b`

- Octale number: `0`
- Decimal number: No prefix (default)
- Hexadecimal number: `0x`

Example: If you would like to write a hexadecimal number `5b`, you need to do it like this: `0x5b`.

B.3. Comments

Comments make assembly code easier to read. Writing comments is different from one assembly to another but in the assembly you are going to use comments are written in the same way as in C. For example:

```
/* this is a comment */
```

B.4. Symbols

In order to avoid hard coding all addresses and values, you can use symbols. A symbol is the name you give to either an address or a constant.

B.4.1. Setting Symbols Explicitly

You can set a symbol value explicitly. It is done like this:

```
SYMBOLNAME = value
```

A concrete example in which a symbol "RETURNCODE" is introduced for the value 0x13 (hexadesimal 13):

```
RETURNCODE = 0x13
```

B.4.2. Labels

Labels are an important type of symbols. They can be placed at any line in the code and the value of the label will be equal to the address of the instruction which follows it. This is very useful for all kinds of jumps because you don't need to keep track of all absolute addresses. Here is an example of a loop which counts down the register 1 until it is equal 0:

```
loop:
  subs r1, r1, #1
  bne loop
```

The `sub` instruction subtracts the counter register, and the `subs` variant updates the status register. The `bne` instruction causes the execution to leave the loop if the previous instruction gave the result 0. The `bne` instruction takes one argument: the address to which the processor needs to jump. We write a label so as to avoid to write an address to which the program needs to jump and use this label as an argument for the instruction. Assembly will then take care of computing which address will be used when the code is assembled to a binary machine code.

B.5. Pseudoinstructions

There is a special type of instructions which are not proper instructions. They will usually not translated to machine code in a binary file but, instead, they represent commands for the assembly itself. Such instructions are named pseudoinstructions. Pseudoinstructions are also known as directives.

Here is a list of useful pseudoinstructions:

- `.syntax unified`: Specifies which syntax to use. You should always put this at the top of your ARM thumb assembly file.
- `.include "filename"`: Includes the file "filename". It is useful if you want to include a list of constants (explicitly set symbols) which you would like included in more than one source file. Then, they can be put in their own source file which is included by all the others. It corresponds to a header file in C.
- `.word`: Specifies a word of data, useful if you want to specify a constant placed in a specific location in memory
- `.text`: Specifies that the code that follows will be placed in the text segment. Read more about segments in section C.1.
- `.data`: Specifies that the data which follow will be placed in the data segment. Read more about it in section C.1.
- `.globl symbol`: Specifies that a symbol "symbol" will be a global symbol, i.e. that it will be possible to refer to it from other object files. Read more about global symbols in section C about object files and linking.
- `.thumb_func`: Specifies that the following label is a thumb function

C. Object Files, Libraries and Linking

Some older assemblers make an executable file directly. GNU AS is a more advanced assembler which makes an "object file" instead which must be converted to an executable file by the "linker". The same concept is used by a C compiler which also compiles the code to an object file and not to an executable file.

One advantage is that for a given program, there can be several source files which must be assembled. Instead of assembling everything in one go, the process is split up. The linker will gather all different object files to an executable file. Often there are big parts of the program which are hardly ever changed and it is therefore not desirable to assemble everything every time because of a change in a small part of the program.

Object files are binary files which contain machine code but where the symbols are still not set to certain value. The labels can not be computed before all object files are set together in a linking step because until then it is not known in which area of the address space the source code will be placed. Additionally, it is desirable to have the possibility to reference global symbols in other source files (marked with a `.globl` directive in the assembly code) and these are not known before the details of the whole program are known in the linking stage.

A convenient thing with the use of the linking stage is that more languages can be used for program development. The linker does not care if an object file was made by assembler, C compiler or Pascal compiler as long as it is in a correct format.

In addition, most programmers use one or more "libraries". A library is an already compiled code which is made by someone else in order to be used by various programmers. Libraries are combined with your program in the linking stage.

C.1. ELF and Segments

As mentioned, the result from the linker is an executable file. In a GNU/Linux world this file is in the ELF format. In addition to the program code in binary format, it contains some extra information.

An elf file has several different segments, which specifies different parts of the program. The most important segments from the perspective of an assembly programmer are the text segments and data segments. The text segment contains the program code, while the

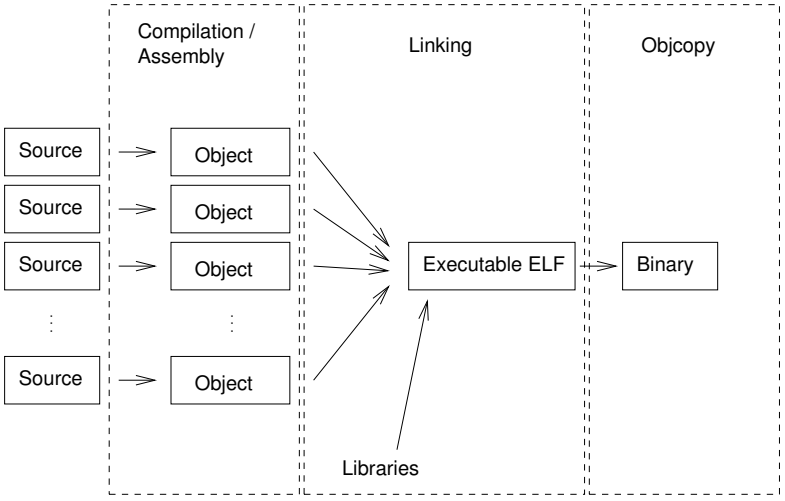

Figure C.1.: Link process, from source files via object files to an executable program. The last step (objcopy) is typically only relevant for certain embedded systems, like the DK3750

data segment contains variables used by the program code. In addition, the programmer is free to create new segments if necessary.

C.1.1. Important about Assembly Programming

When programming in assembly, you need to specify in which segments your code will reside. This is done by directives `.section text` and `.section data`. All program code will come after `.section text` directive and all (writable) variables will come after `.section data` directive.

[todo]

D. C-Programming

C programming is probably new for some of you taking this course. The good news is that Java has borrowed most of its syntax from C so there is not so much you need to learn from scratch. The entire C language can not be described at this place so we just point to some of the differences with Java. We recommend that you buy and read the book "The C Programming Language" [8]. It is a classic which will be useful, not only in this course.

D.1. Java and C: Similarities and Differences

As said, Java and C have very similar syntax. The ways in which, for example, for–loops are written are the same. Here is the list of particularities of C which a Java programmer has to learn. You should look at this as a list of what you should look up in and read about in a proper book about C. Our explanations are too brief to provide a good basis for learning C without additional help.

D.1.1. Compiling

C code is compiled to object files, as assemly code was assembled to object files by assembler in the previous exercise (see section C). Compiled source files need to be linked together into an executable file in the same way as in exercise 1.

D.1.2. Object Orientation

C is not object oriented. Therefore, there are no methods which can be connected to an object, instead there are just functions which are not members of any class. It is often wise to think in an object oriented way when programming, for example by relating a source file to an object in Java, but there is nothing in the language which enforces object orientation.

D.1.3. Structs

Instead of objects, C has structs. **Structs** are collections of variables and it can be viewed as an object whose all variables are "public" and which has no methods.

An example of the use of structs:

```
struct teststruct {  /* struct declaration */
  int a;
  int b;
};

int main (int argc, char *argv[]) {
  struct teststruct t; /* declares variable t to be of a type teststruct */
  t.a = 5;             /* exercise of variable i in struct */
  int c = t.b;         /* reading of a variable from struct */
}
```

D.1.4. Prototyping

In C, all the functions should have a "prototype". It is a declaration of the function which tells what type of arguments the function takes and what type of value it returns, but without specifying how the function is implemented. All the code which will use a function should include a header file with the function's prototype. With that provided, a function can be implemented in some other source file.

Example:

```
int test (void);  /* prototype of the function test */

int test (void) { /* implementation of the function test */
  /* code */
}
```

D.1.5. void

A function in C which does not take any arguments must explicitly declare that. Here is a prototype of one such function:

```
void test (void);
```

This function takes no arguments and returns no value.

D.1.6. main()

The function which is called when the program starts is called main() and its prototype should be like this:

```
int main (int argc, char *argv[]);
```

D.1.7. Modifiers

As in Java, C has a number of key words which tell something about a variable and which are called "modifiers". The following modifiers are important:

- `const`: It will be impossible for the program code to change the value of the variable.
- `static`: Two different meanings, depending on where it is used. If used for a local variable in a function: the variable will keep its value between different function calls. If used for a global variable: the variable will not be visible for the linker, it will be visible only for the source file in which it is defined.
- `extern`: The variable is a global variable which is defined in some other source file (and therefore it will placed in some other object file). The linker will make it possible to use the variable anyway.
- `volatile`: The variable will be kept in RAM. If a variable is not declared "volatile", the compiler will be able to perform optimisations and possibly avoid accessing RAM. Variables that represent I/O locations and variables that are shared between, for example, an interrupt routine and other functions must, therefore, be set to "volatile" to make sure that they work as expected.

D.1.8. Pointers

A pointer is a variable which contains a memory address. It often refers to another variable and, therefore, it has a type which says what type of variable it points to.

Example:

```
int a;    /* declare variable a */
int *b;   /* declare a pointer to an int */

b = &a;   /* set pointer b to point to a variable a */
          /* Actually: &a means "address of a" */

*b = 5;   /* set the value of the variable to which b points to 5 */
          /* Actually: *b = 5 means "write 5 to the memory address to which b points */

          /* a will now have the value of 5 */
```

In case the pointer points to a struct, a somewhat different syntax is used for accessing the members of the struct (mark the notation `->`):

```
struct teststruct t;  /* declare a teststruct variable */
struct teststruct *p; /* declare a pointer to a teststruct variable */

p = &t;    /* set p to point to t */
p->a = 5; /* the same as t.a = 5 */
```

There are also general pointers which don't have any types associated to themselves. These are declared as void pointers:

```
int a;          /* declare variable a */
void *b;        /* declare void pointer */

b = (void*)&a;          /* set pointer b to point to variable a */

*(int*)b = 5; /* set the value of the variable to which b points to 5 */
              /* when void pointers are used, explicit cast must be invoked */
              /* as it is here (cast to an int pointer) */

              /* a will now have value 5 */
```

D.1.9. Macros

C has support for macros. Typically, they are used for constant definitions in a header file.

For example:

```
#define RETURN_CODE 13  /* RETURN_CODE can now be used instead of 13
                           later in the code */
```

D.1.10. Header Files

In C there is a difference between two types of source files: C files and header files. All program code should be placed in a C file (file suffix ".c"). All definitions (constants, structs, prototypes) should be in a header file (file suffix ".h"). A header file is included like this:

```
#include <stdio.h>  /* include a system header file */
#include "test.h"   /* include a header file which is local for the project */
```

D.1.11. stdlib

C has a standard library with a set of functions which are called from all programs. Contrary to Java API, this is a rather small and limited library. The reasoning behind is that a C programmer should turn directly to an operating system when more functionality is needed than that which is found in the standard library. Therefore, C programs are typically specific with respect to an operating system and not so easy to port to another operating system.

In order to be able to use a function from the library, a corresponding system header file has to be included in the source code. This is because the compiler needs to know that

the code for the actual function exists at some other place so that the function can be used.

Tips: All C functions have a Unix man page, so when you would like to learn about some specific C function, you can open its documentation by writing `man 3 <function_name>`.

Note also that in embedded systems, the C standard library might not be available, or only partially available.

D.2. Code Organisation and Conventions

It is common practice to organise functions and global variables in different C files and compile them separately. Organise the files so that the functions which belong together are placed in the same source file, in about the same way you would organise the code into different classes in Java.

The functions and variables which need to be global because they will be used in a different C file must be declared in a header file which is included by all C files which need to use them. In other words: global functions must have their prototypes in a header file which is included in all C files which use these functions. Global variables must be declared as "extern" in a header file which is included in all source files where they are used.

A typical program can then have a certain number of C files and some header files which are included by the C files. In the header files, all function prototypes are gathered, together with "#includes", "#defines" and "externs". But, remember: do not write function code or variable assignments in a header file, that will only cause problems and is considered bad style.

E. Linux Platform Drivers

The driver framework presented in exercise 3 is enough to make a working Linux driver. It is, however, not the recommended way of writing a driver. You are encouraged to convert your driver into a "platform driver". This is not documented in "Linux Device Drivers". The following section will briefly discuss this, and more information can be found in the official Linux kernel documentation.

E.1. Connecting the Driver with the Platform

In older Linux drivers, the drivers do all its initialization work in the module init function. Typically, they just assume that the HW is available at some address.

Newer Linux drivers do not assume such things. Instead, the driver registers itself in the kernel as a platform driver, telling the kernel what kind of HW (platform device) it can handle. Then it waits until the kernel decides to activate the driver. The kernel will only activate a driver if a corresponding platform device exists. When activated, all information about HW addresses, interrupt numbers etc. can be found by querying the platform device.

One advantage of the new platform driver model is that drivers can be made more generic, not having to hardcode any addresses or irq numbers.

The module init function is now only used to register the platform driver, no other initialization is done there. The initialization code is instead moved to the platform driver probe function, which is called by the kernel if the corresponding HW is available and unused.

Implement the following functions:

```
static int my_probe(struct platform_device *dev) {
    // ...
}

static int my_remove(struct platform_device *dev) {
    // ...
}

static const struct of_device_id my_of_match[] = {
        { .compatible = "tdt4258", },
        { },
```

```
};
MODULE_DEVICE_TABLE(of, my_of_match);

static struct platform_driver my_driver = {
        .probe  = my_probe,
        .remove = my_remove,
        .driver = {
                .name   = "my",
                .owner  = THIS_MODULE,
                .of_match_table = my_of_match,
        },
};
```

This sets up that your driver can handle the platform device "tdt4258", which is a special platform device set up for this course and represents the GPIO and sound. In addition, it specifies the functions that are called when the driver is activated or disactivated: "my_probe" and "my_remove". The module init function can now register the driver with this function:

```
platform_driver_register (&my_driver)
```

After this, the my_probe() function will be called if there is a matching platform device ("tdt4258") in the platform.

To sum up the initialization sequence of a typical platform driver:

- User does a `modprobe driver` to load the kernel module into memory
- The module init function will be called immediately
- The module init function will register its driver details by calling the `platform_driver_register()` function.
- If there is a matching platform device in the system, the kernel will call the driver probe function
- The driver probe function will then request information from the kernel about I/O register base address, irq numbers etc., and then initialize everything.
- After the probe function is done, the driver is ready for use.

E.2. How to Query Platform Device Information

Do not hardcode base addresses of I/O registers or IRQ numbers. This information is known by the platform device, and can change between platforms.

You can find the I/O register base address for the TDT4258 device by using the following function:

```
struct resource *res = platform_get_resource(dev, IORESOURCE_MEM, index);
```

`dev` is the argument to the probe function. `index` is the mem index shown in table 5.2. You can find the start address of the I/O memory with `res->start` and the end address with `res->end`.

You can find the IRQ numbers by using the following function:

```
int irq = platform_get_irq(dev, index)
```

`dev` is the argument to the probe function and `index` is the IRQ index shown in table 5.2.

E.3. I/O Access

As discussed in "Linux Device Drivers", accessing registers as done in exercise 2 is not the preferred way in Linux. Instead, use the funcions ioread32() and iowrite32().

F. Troubleshooting the Development Kit

While trying to upload your program to the development kit, you may sometimes encounter the following errors:

- Unable to read from SRAM
- Unable to halt processor
- Unable to flash (or other flash-related errors)

Most of the time, reset the microcontroller via the CPU board reset button, power-cycling the entire devkit with the on-off button, or removing and re-inserting the USB cable will fix solve the problem. Confirm by launching eACommander, click the Connect button, and looking at the information displayed in the Kit Information section, it should be similar to those in Figure F.1. However, if you are still encountering the same error(s), or if it says "Not Connected" in the eACommander kit info fields, try the following:

1. Trying using a different USB port on the computer you are using. Some USB ports may be faulty or damaged.
2. Is something holding down the reset button on the CPU board? The debugger will not be able to connect if the EFM32 is in reset state, indicated by a red LED on the CPU board. In this case, remove any physical sources of obstruction on the button, or contact the course assistants.
3. Try using the Debug Unlock function. You can access this on the development kit interface by pressing the Flash button (PB3), selecting UnLoc (PB2) and then confirming Yes (PB4).
4. Have you perhaps short-circuited Vcc and ground while connecting custom hardware? (Please be careful not to do this, the boards are expensive and can get easily damaged)

If all else fails, contact the course assistants to get help, and mention the computer you are working on and the identification number on the development kit.

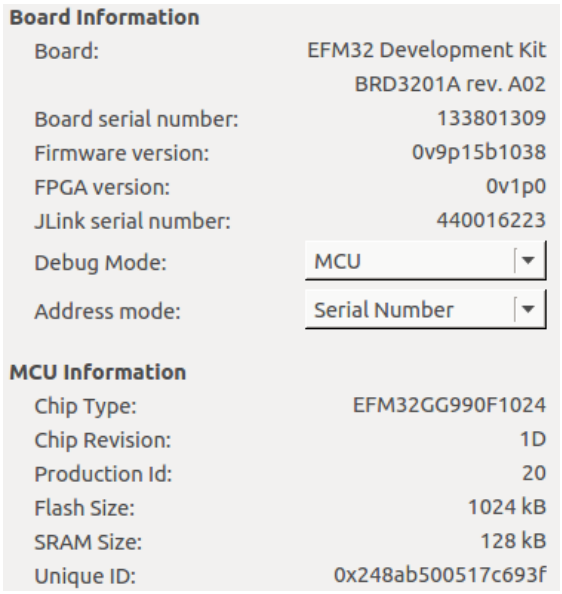

Figure F.1.: The Kit tab in eACommander with a successfully connected development kit.

Bibliography

[1] ARM. *ARM and Thumb-2 Instruction Set Quick Reference Carda*. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.qrc0001m/index.html.

[2] Jonathan Corbet, Allesandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly, 3rd edition, 2005. http://lwn.net/Kernel/LDD3/.

[3] Free Software Foundation. *GDB Manual*, 2006. infonode: gdb.

[4] Free Software Foundation. *GNU As Manual*, 2006. infonode: as.

[5] Free Software Foundation. *GNU Ld Manual*, 2006. infonode: ld.

[6] Free Software Foundation. *GNU Make Manual*, 2006. infonode: make.

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

[8] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Software Series, 2nd edition, 1988.

[9] Pengutronix. Ptxdist homepage. http://www.ptxdist.org/, 2014.

[10] Silicon Labs. *Cortex-M3 Reference Manual*, 2011. http://www.silabs.com/Support%20Documents/TechnicalDocs/EFM32-Cortex-M3-RM.pdf.

[11] Silicon Labs. *Application Note: Direct Memory Access*, 2013. http://www.silabs.com/Support%20Documents/TechnicalDocs/AN0013.pdf.

[12] Silicon Labs. *EFM32GG-DK3750 User Manual*, 2013. http://www.silabs.com/Support%20Documents/TechnicalDocs/efm32gg-dk3750-ug.pdf.

[13] Silicon Labs. *EFM32GG Reference Manual*, 2013. http://www.silabs.com/Support%20Documents/TechnicalDocs/EFM32GG-RM.pdf.

[14] Richard Stallmann. GDB debugger tutorial. http://www.unknownroad.com/rtfm/gdbtut/, 2006.