Lab exercise 1
Low-Level Programming

Dávid Danaj, Salahuddin Asjad, Valentin Lemenut

TDT4258

Abstract

This is a report for the first assignment in the subject TDT4258 - Low-level programming. The assignment is about creating a small game using a microcontroller. We will go into details about the assignment in the next section. After that we will go through the needed knowledge about the assignment and the development board in the background and theory section. Thereafter we will guide you through the functionality of the game with some code snippets. In the end we will present the result with a conclusion on how efficiency the energy saving modes are.

1 Introduction

The practical goal of the lab exercises is to implement a small game for the EFM32 development board which has a microcontroller, a display and audio. We also have an access to a small prototype gamepad with buttons and LEDs so we can perfectly make our own simple computer game by programming the microcontroller which allows us to control the I/O-components we have. Furthermore, energy efficiency of our solution is an important goal of the course and this is why we have a specific component on our development board to measure the realtime power consumption of all components we have, which allows us to optimize our programs to minimize our power consumption.

In this first exercise, we will have to familiarize ourselves with some of the widely used development tools from GNU. We will learn how to write the startup code for ARM processors, we will write it in assembly language so we can control what is done very precisely. We will learn how to program GPIO in order to use the gamepad button to control the LEDs, we will achieve this by using interrupts.

We will have to turn on the LEDs independently with all the buttons and then we will be able to program other functionalities to make the LEDs react in a more fancy way.

2 Background and Theory

To solve the problem we will need to use our technical knowledge with microcontrollers and assembly programming. We will have to read the reference manual of the EFM32 development board to get the required knowledge about the microcontroller we are using. [2] The reference manual provides with important information about the different modules and peripherals in the EFM32 development board. Since the microcontroller is based on a 32-bit ARM Cortex M3 processor, you will also find the Quick Reference to ARM Thumb-2 quite useful when programming in assembly. The Quick Reference contains available instructions for memory access, arithmetic operations, logical operation et cetera.

2.1 Hardware Description

As mentioned in the previous section, we are using a EFM32 development board to solve the problem. The EFM32 microcontroller familiy is one of the most energy friendly solution of any other 8-, 16- and 32-bit microcontrollers available. The development board consists of a high performance 32-bit ARM Cortex M3 processor, which offers many benefits for the developers like ultra-low power consumption and fast interrupt handling. The processor is featured with both ARM Thumb and Thumb-2 instruction set architectures which makes some of the instructions up to 32 bits.

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

2.1.1 Main Peripherals Description

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

There are 2 analog input for miscellaneous signal, with the first one we can read directly the signal and with the other one the signal goes first in a differential operational amplifier with ground as reference before reaching the MCU. There is also a low pass filter included in this input. We also have two specific inputs for the audio signal; one for the

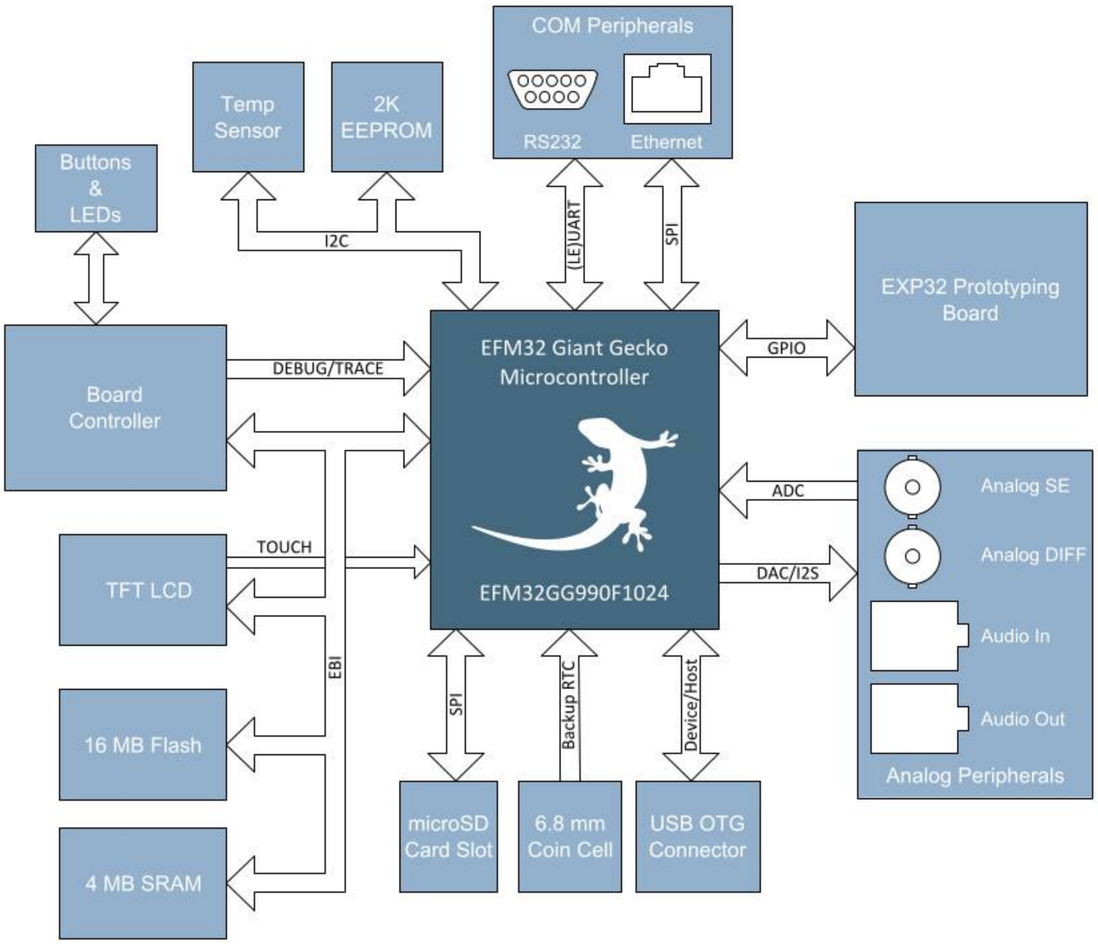

Figure 2.1: Block Diagram of the Development Kit

input audio and the other for the output signal. The bandwidth is up to 27 kHz for the output we can choose between two different sources. For the input the cutoff frequency is 20 kHz. The connection with external devices is made through jack plug. There is also a LCD screen which basically makes it easier to monitor different cases under runtime. For instance we can use the screen to keep track of the power consumption via the Advanced Energy Monitor (AEM) included in the kit. The Advanced Energy Monitor is capable of measuring currents in the range of 100 nA to 50 mA. For currents above 200 uA, the AEM is accurate within 100 uA. When measuring currents below 200 uA, the accuracy increases to 1 uA. Even though the absolute accuracy is 1 uA in the sub 200 uA range, the AEM is able to detect changes in the current consumption as small as 100 nA The measurement bandwidth of the AEM is 60 Hz when measuring currents below 200 uA and 120 Hz when measuring currents above 200 uA.

This kit also includes an RS232 level converter together with a DSUB-9 connector is provided for serial communication between the EFM32 and an external device via an UART. We can also communicate via an ethernet protocol, using a RJ-45 connector. But in this lab project we will use the USB Micro-AB Connector to communicate with the computer.

To control what the microcontroller will do under runtime, we are using a gamepad. The gamepad is connected to the prototyping board on the development board. In the following Figure 2.2, we can see how the ports are set up on the prototyping board. The gamepad is connected to the MCU through portA (pins 8-15 for the LEDs) and portC (pins 0-7 for the buttons).

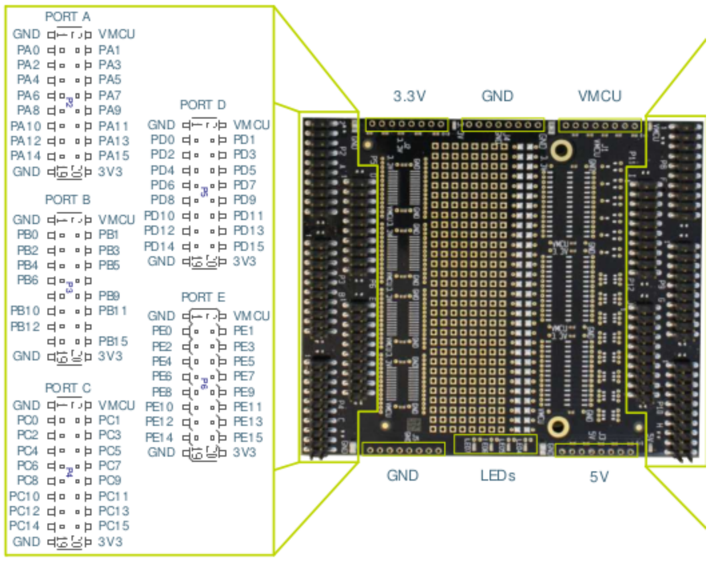

Figure 2.2: The prototyping board

2.2 Important concepts

An important point in this presentation of the hardware we have is that some of these peripherals can generate output. We simply have to check the signal of the pin `PE0`. When a GPIO (gamepad) interrupt occurs, and the interrupt is caused by a falling edge of PE0, the interrupt flag register in the board controller should be read to determine which peripheral caused the interrupt. We will use this concept of interrupts at the end of our first exercise, to optimize the resource usage. An interrupt is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention. An interrupt alerts the processor to a high-priority condition requiring the interruption of the current code the processor is executing. The processor responds by suspending its current activities, saving its state, and executing a function called an interrupt handler (or an interrupt service routine, ISR) to deal with the event. This interruption is temporary, and after the interrupt handler is finished, the processor continues in execution of the stopped code. [1] We can enable interrupts we want to use. We will use it in order to read the state of the buttons on the controller so we do not need to always read their state and just enable the interrupt and create a handler for this. The exception vectors specifies where in memory handlers for exceptions and interrupts are located.

In order to prevent the kit to use too much energy we can power on only the I/O controller we will use. For instance, we will only use the GPIO controller in this exercise so we enable its clock. We can also use the sleep mode. When the CPU is no longer doing anything useful (e.g. waiting for button input) we will stop the CPU and will wake it up when it receives an interrupt, and go back to sleep after the interrupt handler returns.

We will use the GNU Toolchain, GNU is a project with the main goal to provide a free and open operating system with all accompanying tools for software development. This is why we will use GNU AS as assembler, GNU LD as linker, GNU Make to build the project and the GNU Debugger (GDB) to debug the program. It is a powerful program which gives many possibilities to monitor the program execution, stop the execution and inspect the contents of registers and memory. It also gives us 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 is happening in closer detail. [3] 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 gdbserver. The gdbserver lives in the middle between the development board and GDB. [3] We will run GDB in Emacs to be able to see which line in the source code is being executed. Emacs allows use to see everything that happens with GDB. To connect the development board through gdbserver we just have to run the .gdbinit file which is provided.

3 Methodology

The first thing we had to do was to power on the development board and connect it to the development desktop.

The second thing we had to do was to enable the GPIO controller, which is done with help from Clock Management Unit (CMU). We have to enable the GPIO controller because the gamepad communicates with the development board through the GPIO controller. By default, only I/O controllers that are being used has its clock enabled. Since we are not using all the I/O controllers, it is not a good design to have all of them clocked and waste a lot of power. So we had to store the value 1 in bit position 13 in the `CMU_HFPERCLKEN0` register, which is representing the GPIO to enable the controller. After enabling the GPIO controller, we put a 20mA drive current on the port A via the register `GPIO_CTRL`. By putting a 20mA drive current we are setting the drive mode to high. Since we want the 8 to 15 pins, which is port A, to be outputs refering to the Figure 3.6 in the compendium [3], we are setting the `GPIO_MODEH` register to push-pull output mode. In the same sense we also want the 0 to 7 pins, which is the port C, to be inputs refering to the same Figure 3.6. It is done by setting the `GPIO_MODEL` to input enabled with filter mode, so `GPIO_DOUT` determines pull direction (we set them as pull-up through the `GPIO_DOUT` register). [2] (p.766-767).

3.0.1 GPIO handler

One of the requirements of this assignment was also to implement interrupt capability into the system. Without an interrupt mechanism, the program keeps executing a while loop checking to see if a button has been pressed. This is called polling and is not an efficient design method. In this assignment, we are only noticing it by the power consumption measurement, but when we are working on larger systems, it is much better to let the program do something else while waiting for an interrupt from a button. In this case the interrupt for the buttons is called GPIO interrupt.

Whenever a GPIO interrupt is triggered, the CPU will jump to the GPIO interrupt handler, where the code functionality for the program is provided.

First in the GPIO interrupt handler we have to get the interrupt source. In other words, we check which button invoked the interrupt. Depending on which button it is, the program will branch to the appropriate section. The comparison and branch functionality is shown in Code 1.

```
ldr r2, [r1,#GPIO_IF]
cmp r2, #0x00000010 // SW5
beq SW5
cmp r2, #0x00000020 // SW6
beq SW6
cmp r2, #0x00000040 // SW7
beq SW7
cmp r2, #0x00000080 // SW8
beq SW8
```

Listing 1: Comparison and branch functionality

Dependent of which button was pressed, it can branch to 4 different sections:

- A branch to `SW5` will make the LEDs rotate to left side. This is maybe the most tricky part of the exercise, we wrote some useful comments in the code to understand what we are doing, but basically we have to make a rotation to the right of the state of the pins 8-15 (because LEDs are in the opposite order on the pad, so it will make same rotate to the left). We just have to be careful with the bits on the 'edge' of this 8-15 range, it means that if there is a bit from the LEDs state on the seventh pin after rotation we have to move it to the fifteenth bit.
- A branch to `SW6` will turn on the LED D8, which is the rightmost LED on the board. We simply need to put the pin 15 on port A at 0, without changing the state of the other pins.
- A branch to `SW7` will make the LEDs rotate to right side. It's basically the same as for SW5 but we rotate in the other way so we have now to be careful with a possible bit at 1 on the sixteenth pin and move it to the eighth.
- A branch to `SW8` will turn off the LED D8, which could be turned on by branch to SW6. Similarly to SW6 we just need to put the pin 15 on port A at 1, without changing the state of the other pins. ...

Each of these branches will then again branch to end_handler where it will be taken care of outputting the functionality to GPIO_DOUT and then clear the interrupt. After that it will return back to the function call.

If the interrupt source doesn't match any of the comparisons, the value 0x0000FF00 will be outputted to GPIO_DOUT. This will then turn off all the LEDs. Since the buttons from `SW5` to `SW8` are included in the comparison in Code 1, the LEDs will only go to reset whenever we press the buttons from `SW1` to `SW4`, which are the four buttons on the left side of the gamepad.

```
mov r3, #0x0000FF00
b end_handler
```

Listing 2: Turn off LEDs

3.1 Testing

The main test we performed during this exercise was to check that we controlled the LEDs in the way we wanted to and to check the power consumption. We just loaded the program and tried to push the buttons and looked at the state of the LEDs on the gamepad, and we did it again and again until we concluded that it worked as we expected after correcting the program each time.

On the other hand, we tried different power strategy for the device and we were able to monitor the power consumption on the screen. We made the program run with and without the sleeping mode activated for instance, and we compared the power consumption during the specific sequence; we let the program run without any LED activated, then we activate one LED after the other until we reach the number of five LEDs switched on. We performed this test with our last version of code, the only difference between the two programs was the sleeping mode. The results are shown in the next section.

4 Results

After loading our last binary file into the board, everything was working as we wanted. We managed to accomplish what we planned to do before we started coding.

Here are the results of the power consumption:

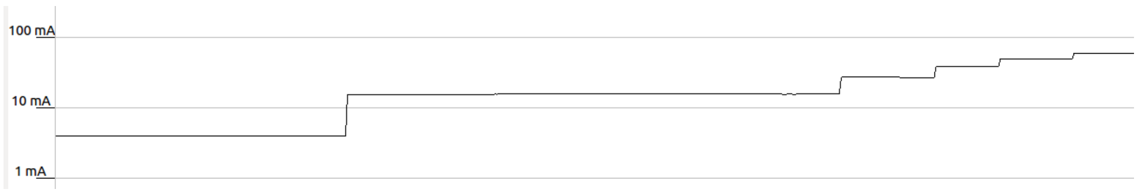

Figure 4.1: Power consumption of the board without sleeping mode

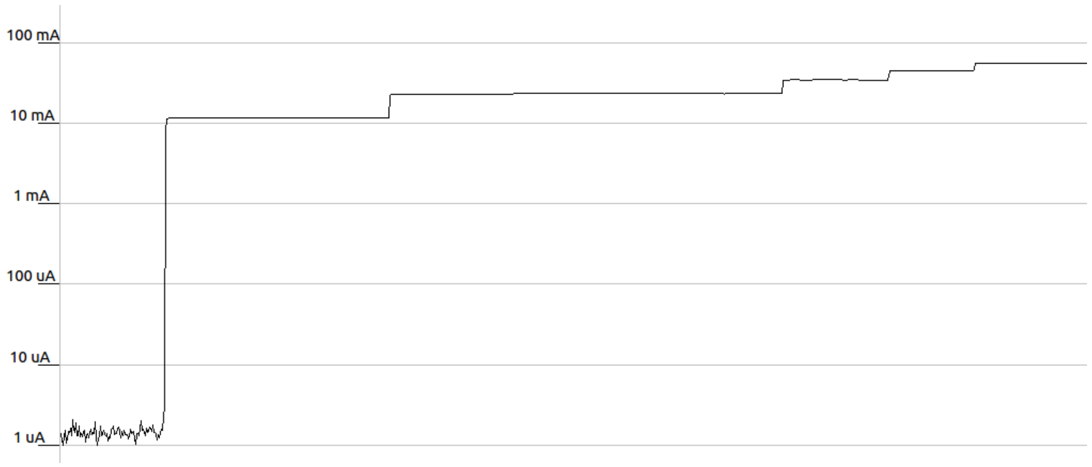

Figure 4.2: Power consumption of the board with sleeping mode activated

In the first Figure 4.2, we can see the consumption without trying to save energy. We can see that even in the idle state, the power consumption is quite high. The power consumption without any LEDs activated was about 5 mA. In the second one 4.2, we can see the consumption in the same context with sleeping mode activated. In this case, the power consumption was much lower during the idle state, about 2A. So the sleeping mode is really important for energy saving.

5 Conclusion

In this first exercise, we familiarized ourselves with some of the widely used development tools from GNU and the development board itself. We had to read datasheets carefully in order to find the precise information we needed such as the role of the registers and how to use them to achieve the functionality we wanted. We learned how to program GPIO in order to use gamepad buttons to control LEDs, we achieved this by using interrupts. Using the assembly language was also a big part of this exercise.

We also experienced the importance of implementing a sleep mode, which made it really interesting to notice the big difference between with and without sleep mode.

Finally we also had to write a technical report following a specific template in Latex, which was a useful experience in describing the concept we used in this first exercise.

5.1 Evaluation of the Assignment

This exercise was well made, the difficulty level and the time we needed to complete the assignment was accurately enough for us. Sometimes it was a bit challenging to solve different small problems, but we like it that way!

Bibliography

[1] *Linux Device Drivers, Third Edition, Chapter 10. Interrupt Handling.* O'Reilly Media, 2005.

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

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