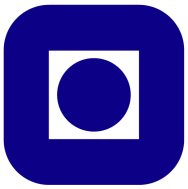

NTNU

TDT4258 - Energy Efficient Computer Systems

Exercise 1 - Buttons and LEDs

Group 13

Mathias Ose, Øyvind Robertsen and Jørn-Egil Jensen

February 10, 2014

Abstract

In this exercise, the group learned the basics of developing and running programs on the EFM32GG-DK3750 development board, including communication between the board and a prototype gamepad. Each group member acquired knowledge of the internal workings of the EFM32GG microcontroller, of programming for the ARM Cortex-M3 processor and of the GNU-toolchain. We chose to implement a simple way of controlling the LEDs on the gamepad, a one-to-one mapping between pressing $button_n$ and LED_n turning on. Additionally, we implemented a simple animated sequence of flashing LEDs. All code was written in ARM assembly using the Thumb-2 instruction set. To achieve energy efficiency, we implemented a solution using interrupts, as opposed to busy-loop polling.

Contents

1 Introduction

In this exercise we take a very hands on approach to learning microcontroller programming in Assembly. This should give us some valuable knowledge about the inner workings of the microcontroller, and be a good foundation for doing more advanced things in later exercises.

The device used in this exercise is called EFM32GG-DK3750. It is produced by Silicon Labs for developing and testing programs for embedded systems running ARM processors. Programs are written and compiled on a personal computer then flashed to the development board via USB. The development board has multiple I/O capabilities, but in this exercise we only utilize the GPIO pins. The development board also features a built in screen that can plot current spent by the microcontroller and allows some settings to be edited live. Amperage can also be monitored on the connected PC using provided software.

The primary objective of the exercise is to use a gamepad peripheral as an input device, send input to the device using it's GPIO functionality, handle the input, then send some sort of output based on the input back through some other GPIO pins to the gamepad where they will activate some LEDs.

A secondary but important objective of the exercise is to observe the energy efficiency of the system when running the program, and see what measures can be taken to reduce it. Using interrupt-based handling instead of continuous polling is especially useful to reduce power consumption, as it allows the microcontroller to sleep until input happens, drastically reducing the consumed power. Both methods will be tried and results will be analyzed.

2 Description and Methodology

This section describes the development process, use of tools, debugging techniques and details of the implementation.

2.1 Development Process

The exercise was primarily done in the computer lab (room 458 at IT-Vest, NTNU). We chose to use git for version control and GitHub for repository hosting. This allowed us to do minor work off-site, followed by testing at the lab.

The lab had already been set up with multiple workstations running Ubuntu 12.04 LTS with the neccessary software already installed, connected via USB to an EFM32GG-DK3750 development board. A support framework was made available for download that contained some things necessary start programming, such as a vector table and some subroutine headers. This allowed us to get to work on the exercise itself more or less immediately, instead of being caught up in setting things up ourselves.

The exercise description was made available in a compendium also containing instructions on how to write and run the program. Also described was how to use the GNU Debugger to debug. A student assistant was also available a few hours every week to answer questions and provide assistance.

The program itself was developed iteratively with no clearly defined goal, with lots of trial and error along the way. Following the advice in the exercise description, we chose to start by familiarising ourselves with the toolchain and the support files supplied by the subject staff. Subsequently, we implemented a basic version of the program using busy-loop polling, then a version based on interrupts. Finally, we improved energy efficiency by implementing automatic return to a less power intense energy mode after interrupt handling.

2.1.1 Devices

The EFM32GG-DK3750 (figure 2.1) connects to a personal computer via USB, shown on the top of the picture. On the device there are multiple GPIO pins. Wires are connected to some of these, and on the other end connected to the gamepad (figure 2.2). Also integrated on the development board is a energy monitoring unit and a screen that can show an energy monitoring graph.

The gamepad peripheral features 8 LEDs that are toggled via 8 GPIO pins and 8 buttons that control 8 other GPIO pins. The gamepad also has a jumper that can be toggled between two settings. Only if it is "enabled", the amperage powering the LEDs will be registered by the EFM32GG-DK3750 energy monitoring unit.

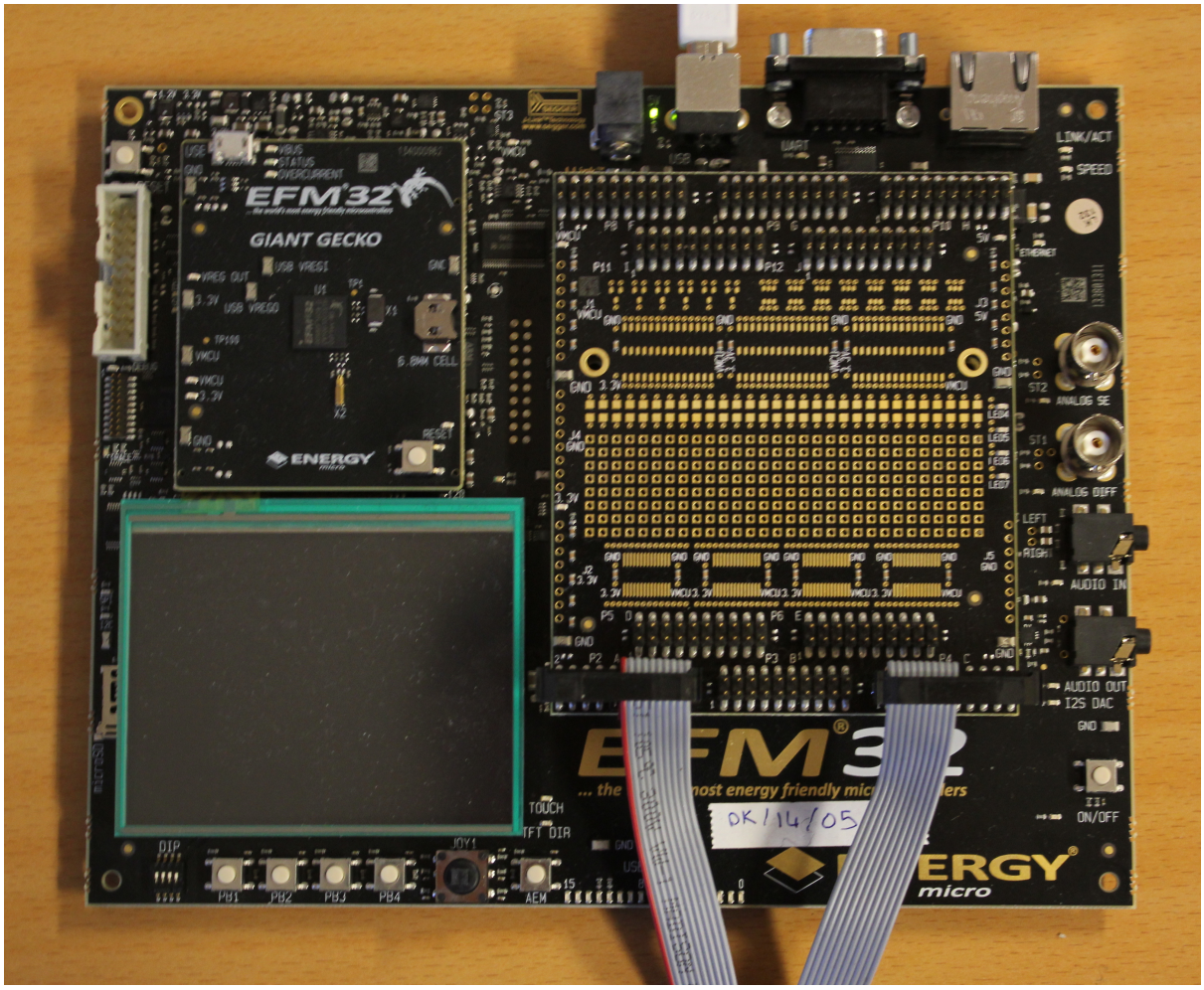

Figure 2.1: The EFM32GG-DK3750

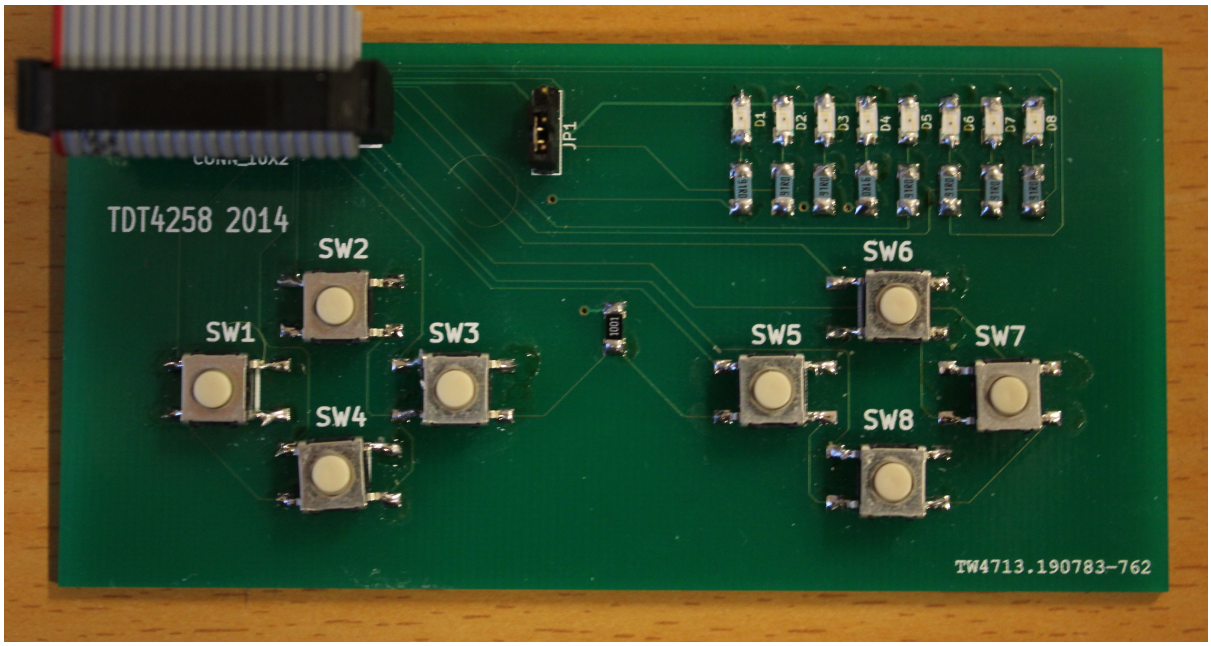

Figure 2.2: The gamepad

2.2 Project setup and toolchain description

After downloading and untarring the support files for this exercise, we chose to reorganize the project structure slightly. We decided upon the directory layout in listing 2.1

Listing 2.1: Directory layout

```
exercise1
|-- Makefile
|-- NOTES
|-- efm32gg.ld
|-- lib
|   |-- efm32gg.s
|   |-- vector.s
|-- report
|   |-- ...
|-- src
    |-- main.s
```

Following this, we made the necessary edits to the Makefile, updating all rules with the new directory layout. As can be seen from listing 2.1 we added a `vector.s` file to the lib directory. This contains the vector table defined at the top of the Assembly file supplied by the subject staff. In addition to the directories listed above, we also added `build/` and `exe/` directories, to contain build artefacts. We configured the VCS to ignore these files, and used a `make clean` rule to remove them.

2.2.1 GNU Toolchain & GNU Make

Throughout the exercise a version of the GNU toolchain made especially for cross development on microcontrollers based on ARM embedded processors was used. The toolchain was preinstalled on the lab computers, but is freely available online if one wishes to install it on a personal computer. The following is a brief description of how we used each of the tools in this exercise.

GNU AS

To assemble our program code into object files we used an ARM-specific version of the GNU AS assembler, a general example of usage is shown in listing 2.2

Listing 2.2: Assembler usage

```
arm-none-eabi-as -mcpu=cortex-m3 -mthumb -g -o <output.o> <source.s>
```

The result is a GDB-debuggable object file with code conforming to the Cortex-M3 instruction set based on the assembly code in the source file.

GNU LD

To link the object files assembled by AS into an executable, an ARM version of GNU LD was used. Syntax in listing 2.3

Listing 2.3: Linker usage

```
arm-none-eabi-ld -T <linkerscript.ld> -nostdlib -o <output.elf>
```

```
↪ <arg0.o> <arg1.o> <arg2.o>
```

The linkerscript supplied defines how the memory on the microcontroller is to be used.

GNU Objcopy

To upload the executable to the development board, we need a clean binary file, void of metadata. This was achieved with the GNU objcopy utility. Syntax is as follows:

Listing 2.4: Objcopy usage

```
arm-none-eabi-objcopy -j .text -O binary <input.elf> <output.bin>
```

GNU Make

Having to manually assemble, link and copy every time we want to test a modification in the code quickly becomes tedious. Thankfully, the handed out files included a Makefile with build rules automating all these tasks, allowing us to use the command `make` as an end-to-end solution for creating new binary files. An `upload` rule was also included. We chose to combine the `make all` rule and the `make upload` rule into a single `make run` rule, allowing us to assemble, link, copy and upload in a single step.

2.2.2 Other tools

- Although the exercise description suggested Emacs as editor of choice, the authors opted for Vim instead.
- Git for version control
- LaTeX for typesetting the report

2.3 Debugging

As with most development processes, debugging played an essential part throughout our work on this exercise. Since we opted to use Vim as our main editor, the GDB capabilities built into Emacs were unavailable to us. Instead, we ran GDB directly from a terminal. In order to debug the program running on the development board, we needed a bridge between our local GDB session and the board. Luckily for us, the development board is equipped with a SEGGER JLink debugging interface. With the supplied `.gdbinit` and the `JLinkGDBServer` program installed on the lab workstation, we could start a GDB instance connected to the program running on the development board using the command `arm-none-eabi-gdb <elf-file>`. Our use of GDB in this exercise did not extend beyond stepping through instructions, placing breakpoints and inspecting register and memory values.

2.4 Implementation

This section describes the implementation of our program and the choices we made through each iteration.

2.4.1 Register Convention

In an early stage of development it became apparent that the limited number of registers available posed a challenge for programmers used to having unlimited variables available in higher level languages. After investigating how the ARM register convention was defined, a new convention was defined on top of the ARM convention to fit our purposes, and some aliases were added to the program to make the convention easier to use.

Register	Alias	Description
R0		Reserved for subroutine argument by ARM convention
R1		Reserved for subroutine argument by ARM convention
R2		Reserved for subroutine argument by ARM convention
R3	W	Reserved for subroutine argument by ARM convention Used for the countdown for the wait subroutine.
R4	GPIO_O	Used for addressing GPIO_PA_BASE (LED outputs)
R5	GPIO_I	Used for addressing GPIO_PC_BASE (button inputs)
R6	GPIO	Used for addressing GPIO_BASE
R7	T0	Used to hold temporary variables
R8	T1	Used to hold temporary variables
R9	T2	Temporary variable
R10		Unused
R11		Unused
R12	IP	Reserved for Intra-Procedure-call by ARM convention
R13	SP	Reserved for Stack Pointer by ARM convention
R14	LR	Reserved for Link Register by ARM convention
R15	PC	Reserved for Program Counter by ARM convention

Table 2.1: Register convention

We implemented this using the `.req` directive. The syntax is as follows:

Listing 2.5: Register aliasing

```
name .req register
T0 .req R7
```

2.4.2 Bare essentials

In the spirit of developing iteratively, our first goal was to create a minimal, compiling program that we could flash to the development board and test our workflow, described in chapter 2. With the modifications we made to the project layout, such a minimal program would look something like this:

Listing 2.6: A minimal program

```
.syntax unified
.include "lib/efm32gg.s"
.include "lib/vector.s"

.section .text
// Reset handler
.globl _reset
```

```
.type _reset, %function
.thumb_func
_reset:
    // Aliases
    W .req R3
    GPIO_O .req R4
    GPIO_I .req R5
    GPIO .req R6
    T0 .req R7
    T1 .req R8
    T2 .req R9

    // Load GPIO base addresses into the relevant registers
    LDR R4, =GPIO_PA_BASE
    LDR R5, =GPIO_PC_BASE
    LDR R6, =GPIO_BASE

// GPIO_Handler
.thumb_func
gpio_handler:
    B .

// Dummy handler
.thumb_func
dummy_handler:
    B .
```

At this point, there was still some confusion as to how one should start a GDB session interacting with the program running on the development board. The compendium originally suggested a nonexistent `gdbserver.sh` script, and the error was not corrected until we had moved on to the next phase.

2.4.3 Light Output

Having familiarised ourselves with the toolchain, we moved on to the next phase; manipulating the lights on the gamepad. It should be noted that we at this point had not yet decided on the functionality of our program. We knew however that it would involve the LEDs on the gamepad.

The gamepad was connected to the development boards GPIO ports A and C, with port A intended for output and port C for input. From the gamepad schematics [2, p. 26] we saw that we would have to send a logical low signal on the relevant pins on GPIO port A to pull down the signal from V_{CC} and light the LEDs. There is some setup required however. Every function on the microcontroller is governed by a clock, and interfacing with the GPIO ports is no exeption. We would have to enable the clock for GPIO in the CMU (Clock management unit). This is done by setting bit 13 in the CMU High frequency peripheral clock enable register (CMU_HFPERCLKEN0).

Listing 2.7: Enabling GPIO clock in the CMU

```
LDR T2, =CMU_BASE
LDR T0, [T2, #CMU_HFPERCLKEN0]
MOV T1, #1
LSL T1, T1, #CMU_HFPERCLKEN0_GPIO
ORR T0, T0, T1
STR T0, [T2, #CMU_HFPERCLKEN0]
```

Next up is configuring the drive strength of port A, this is done by writing to the port control register (GPIO_PA_CTRL). As described in the EFM32GG reference manual, section 32.5.1 [3], this is a two bit register with four possible values. In spirit of energy efficiency, we chose the lowest at $0.5mA$ drive current.

Listing 2.8: Set drive strength

```
MOV T0, #1
STR T0, [GPIO_O, #GPIO_CTRL]
```

Final setup requirement is enabling push-pull output on pins 8-15 on port A. Writing `0x55555555` to the port pin mode high register (GPIO_PA_MODEH) accomplishes this. [3, p. 767]

Listing 2.9: Enable output

```
LDR T0, =0x55555555
STR T0, [GPIO_O, #GPIO_MODEH]
```

With configuration out of the way, we could enable LEDs by writing to the port A data out register (GPIO_PA_DOUT).

Listing 2.10: Enabling LEDs

```
LDR T0, =0x0000
STR T0, [GPIO_O, #GPIO_DOUT]
```

Through each step outlined above, we used GDB to inspect register and memory values.

2.4.4 Polling implementation

Having successfully configured and enabled LEDs, we moved on to reading input from GPIO port C. At this point we decided that the functionality of our program would be a simple mapping between each button and a corresponding LED. This simple functionality would allow us to focus more on energy efficiency.

Having already enabled the GPIO clock in the CMU, setting up pins 0-7 on port C for input required little work. Simply enabling input in the port C pin mode low register (GPIO_PC_MODEL) and writing `0xFF` to the port C data out register did the trick.

Listing 2.11: Enabling input on port C

```
LDR T0, =0x33333333
STR T0, [GPIO_I, #GPIO_MODEL]
LDR T0, =0xFF
STR T0, [GPIO_I, #GPIO_DOUT]
```

Subsequently, we implemented a main loop, constantly polling the port C data in register (GPIO_PC_DIN), and pushing processed data to the port a data out register (GPIO_PA_DOUT). Seeing as pressing a button pulls the corresponding pin to ground, and LEDs are enabled by setting a pin low, all we had do to map the press of a button to the enabling of an LED, was shift the 8 least significant bits in the bitstring from GPIO_PC_DIN 8 bits left and write the resulting string to GPIO_PA_DOUT.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1

Table 2.2: Example of bitstring read from GPIO_PC_DIN with SW1 button pressed

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 2.3: Example of bitstring required to write to GPIO_PA_DOUT to enable LED_1

Listing 2.12: Main loop

```
main:
    LDR T0, [GPIO_I, #GPIO_DIN]
    LSL T0, T0, #8
    STR T0, [GPIO_O, #GPIO_DOUT]
    B main
```

2.4.5 Interrupt implementation

As required by the exercise description, and as a necessary component in increasing energy efficiency, our next move was reimplementing our programs functionality using interrupts. As the vector table defined in our `lib/vector.s` file already contained the necessary entries, all we had to do was enable interrupt generation for the GPIO. First of all we needed to configure which port would generate interrupts in the external interrupt port select low register (GPIO_EXTIPSELL). From section 32.5.10 in the EFM32GG reference manual [3] we saw that we had to write the string `0x22222222` to the register to configure port C as the source port across all pins.

Listing 2.13: Configuring GPIO_EXTIPSELL

```
LDR T0, =0x22222222
STR T0, [GPIO, #GPIO_EXTIPSELL]
```

Next up, as we were interested in generating interrups on both rising and falling edges (pushing and releasing the buttons), we had to configure that. The external interrupt rising/falling edge trigger registers (GPIO_EXTIRISE/FALL) are described in section 32.5.12 in the EFM32GG reference manual. [3]

Listing 2.14: Rising/falling edge

```
LDR T0, =0xFF
STR T0, [GPIO, #GPIO_EXTIRISE]
STR T0, [GPIO, #GPIO_EXTIFALL]
```

Having ensured the proper interrupt flags would be raised, we now had to write `0xFF` to the GPIO interrupt enable register (GPIO_IEN) in order for interrupts to be generated.

Listing 2.15: Enable interrupts in GPIO_IEN

```
LDR T0, =0xFF
STR T0, [GPIO, #GPIO_IEN]
```

The final configuration step consisted of enabling our `gpio_handler` subroutine to handle both odd and even interrupts. This is done by writing to the interrupt set-enable register (ISER0), which is a Cortex-M3 register. Setting bits 11 and 1 high, would accomplish our goal.

Listing 2.16: Configuring ISER0

```
LDR T0, =0x802
LDR T1, =ISER0
STR T0, [T1]
```

As a final part of the reset subroutine, we added a branch to an infinite main loop doing no actual work to keep the processor busy while waiting for interrupts. Afterwards, a proper interrupt handler was implemented.

Listing 2.17: GPIO interrupt handler

```
.thumb_func
gpio_handler:
    //Clear interrupt flag
    LDR T0, [GPIO, #GPIO_IF]
    STR T0, [GPIO, #GPIO_IFC]

    // Perform actual signal processing
    LDR T1, [GPIO_I, #GPIO_DIN]
    LSL T1, T1, #8 // Shift input 8 bits left
    STR T1, [GPIO_O, #GPIO_DOUT]
    BX lr
```

2.4.6 Energy optimization

At this point in the development process, our program could perform every task we required of it. The one exception was energy efficiency. Up until now, our program consumed as much energy while idle as it did while handling interrupts. The functionality of our program was so simple that there was little to no room for improving power consumption in the interrupt handler. Due to our program returning to the main loop after handling an interrupt, the power consumption remained the same, even though the program wasn't actually doing anything.

As described in section 3.2.5 in the compendium [2], the EFM32GG has several different energy modes it can operate within. Normal operating mode is energy mode zero (EM0). In energy mode two, many of the microcontrollers functions that we had previously enabled, would be inactive. In other words, if we could have the microcontroller enter EM2 while it was not handling interrupts, energy efficiency could be greatly improved.

Listing 2.18: Enabling automatic deep sleep after interrupt handling

```
MOV T0, #6
LDR T1, =SCR
STR T0, [T1]
WFI
```

The instructions in listing 2.18 are the requirements for enabling automatic deep sleep in the system control register (SCR). The WFI instruction puts the microcontroller into

deep sleep manually if deep sleep has been enabled in the SCR. We first tried putting these instructions at the end of our `reset` subroutine to no success. Hours upon hours were spent stepping through instructions and placing breakpoints in GDB, making small changes and repeating the process. Finally, slightly inexplicably, we were successfull by moving the instructions to a separate subroutine, branching to this subroutine from the end of our reset subroutine.

3 Results and Tests

3.1 Program

The final version of the program features a simple button-to-LED mapping. The gamepad (seen on figure 2.2), has 8 buttons and 8 LEDs, both sets indexed 1 through 8. Pushing button N activates LED N. The nature of the program allows any number of buttons to be simultaneously pressed and the corresponding LEDs to be activated.

3.1.1 Testing the program

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

Button functionality test

This procedure tests the implemented functionality of our program. Pushing button N should light LED N.

Procedure:

1. Compile and flash the program to the board by running `make run` from the project root
2. Wait for the board to reset properly
3. Push each button, ensuring the corresponding LED lights up.

3.2 Energy efficiency

When we discuss energy efficiency, we will primarily be looking at the current spent (amperage). Wattage or voltage could have been investigated too, but amperage is the most interesting because it can be more easily related to things such as battery capacity. The current powering the LEDs is not registered, as the jumper on the gamepad is set to "disabled" (see section 2.1.1).

3.2.1 Readings

Using the eAProfiler monitoring tool we observed the graph oscillating around and averaging about $2.0\mu A$ while running the final iteration of the program with sleep mode enabled. Previously we had seen it average slightly lower, about $1.6\mu A$. We assume that

this is because the temperature of the room increased, thus increasing the temperature of the components in the system and causing conductivity to decrease. Unfortunately we did not have the time to run the sleeping program for an extended time in a regulated temperature, so we had to make do with the $2.0\mu A$ reading as our main data point. This reading seems to be consistent with what we read in the Energy Optimization Note, which says EM2 could run on as little as $900nA$ [4, chapter 1] and the high frequency clock has a typical consumption of up to $106\mu A$ [4, section 3.1].

Figure 3.1 shows the plot of the amperage on a log scale. First no buttons are pressed, then one, two, three and four. The amperage while these we pushed are shown in table 3.1.

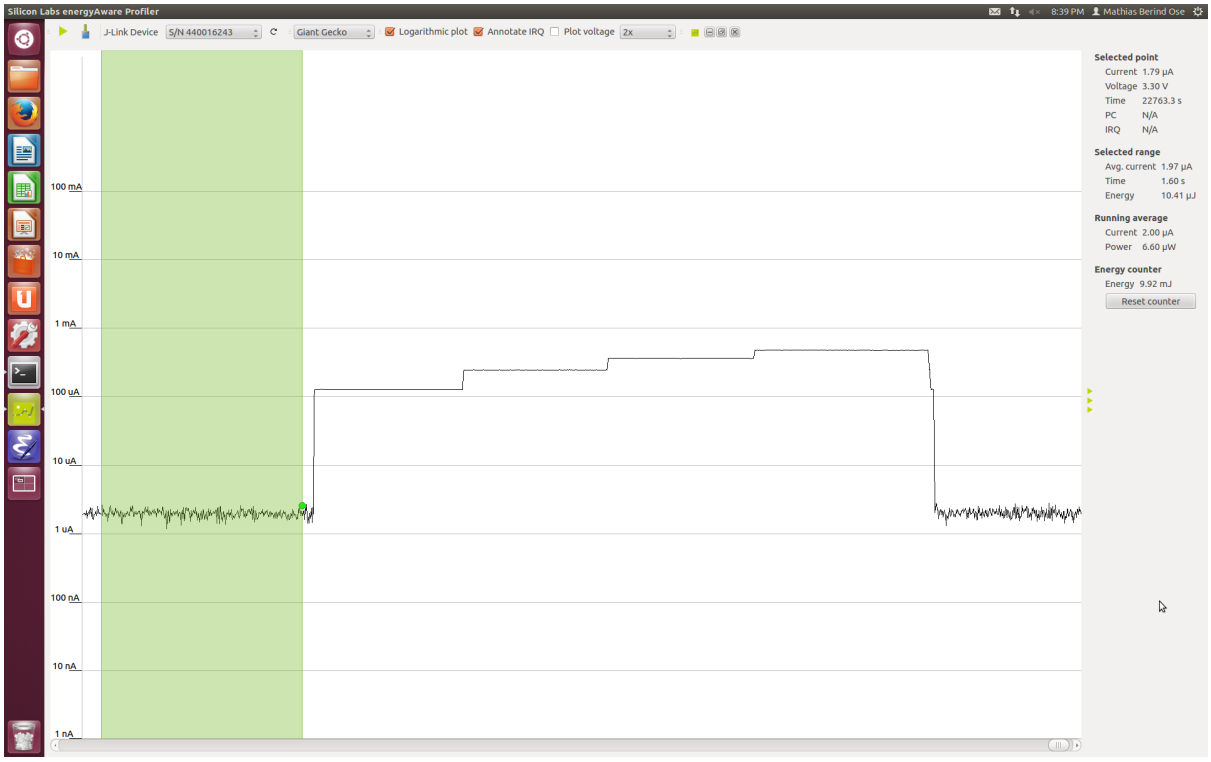

Figure 3.1: The amperage plotted by the eAProfiler tool.

Buttons pressed	Current $[\mu A]$
0	1.97
1	130.41
2	260.85
3	364.33
4	476.34

Table 3.1: Buttons pressed and resulting amperage

When running the build with polling instead of interrupts, the amperage averaged $3.6mA$. That means that the idling system uses 1800 times more current than the sleeping version.

3.2.2 Expected lifetime on a cr2032 battery

Representatives from Silicon Labs, the manufacturer of the EFM32GG STK have used the cr2032 "coin cell" battery as an example when talking about the energy efficiency of their products. For this reason we chose to look up some statistics about this battery online and calculate the theoretical time the development board can run on one battery. One battery manufacturer claims their battery has a typical capacity of $240mAh$.[5]

$$240mAh/2.0uA = 120000 \text{ hours} = 5000 \text{ days} = 13.7 \text{ years}$$

According to one source [1] a coin cell battery can last up to 5 years while in use before deteriorating too much. In other words the battery will deteriorate before discharging from use for this microcontroller running this sleeping program.

If the polling version of the program had been running instead, the coin cell battery would only last 2 days and 18 hours.

3.3 Discussion

The primary goal of the exercise was learning, and that was definently achieved. The group members had no significant prior experience with the technologies, only some theoretical knowledge from the TDT4160 course. But thanks to the instructions in the compendium we were able to do most of the work on our own.

The biggest problem we had was enabling sleep mode with interrupt-based I/O. After following the instructions in the compendium we had sucessfully implemented the I/O with interrupts, but the microcontroller did not go to sleep mode. Attempts at debugging revealed that removing code crucial to the I/O made the device go to sleep. It seemed impossible to solve the problem, but inexplicably it suddenly began working after moving some code about, even though it seems that should not have changed anything (See section 2.4.6). We spent hours debugging the issue, but at the time of writing we have yet to reach a conclusion as to why it worked out the way it did. One theory arose after placing the deep sleep configuration code back into the reset subroutine, and placing three `NOP` instructions above it. Doing this made everything work smoothly, leading us to thinking that perhaps the instruction pipeline has to be cleared before one can write to the SCR register. This could however not be confirmed by subject staff or inquiries into the documentation of the processor.

4 | Evaluation of Assignment

The exercise provided a very good learning experience. None of the group members could boast any significant prior experience with the technologies used in the exercise, only some theoretical knowledge from the TDT4160 course. The exercise allowed this knowledge to be used practically, and introduced many new concepts as well.

One criticism worth mentioning is the fact that we struggled with the sleep mode enabling. Even though we followed instructions and the student assistant could not see any errors when inspecting the code, the device would not go to sleep. The resolution of this problem was as inexplicabe as the cause. It seems there might be some additional criteria for going to sleep that we are not explicitly aware of, and had to stumble upon instead. Since energy efficiency is emphasised in this subject, the seemingly arbirtrariness of this is frustrating.

5 Conclusion

Both the practical goal and the learning goals of the exercise were achieved. For all of the learning goals as listed in section 3.1.1 of the compendium [2] the group members went from having little or no understanding to at least a basic a understanding of the subject. It became especially apparent just how much energy usage could be reduced by making the microcontroller sleep.

A | Acknowledgements

Course by *Silicon Labs* - 29.01.2014
Thanks to Silicon Labs, represented by Audun Nystad Bugge and Alf Petter Syvertsen, for holding a very relevant workshop for the student association Abakus, where the EFM32GG STK was used.
Group members Ose and Robertsen attended.

References

[1] Anonymous. Cr2032 lithium battery. http://www.cr2032battery.org/cr2032-lithium-battery/, April 2010.

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

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

[4] Silicon Labs. Energy optimization an0027 - application note, November 2013.

[5] POWER GLORY BATTERY TECH (HK) CO., LTD. Specification for lithium battery model: Cr2032. http://www.farnell.com/datasheets/1496885.pdf.