NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4258 Low-Level Programming
Laboratory Report

Exercise 1

Group 5:

Marius C. K. Gulbrandsen
Antoni Climent Muñoz
Andrea Mazzoli

October 18, 2019

Contents

1 Introduction

This exercise aims to write a program for the board EFM32GG using assembly as the programming language. In particular we need to control the LEDs on the gamepad through the buttons on the gamepad itself. The idea is to turn on the LEDs by pressing the buttons. We used two different approaches to accomplish this goal, one using a "polling´´ method and the other using an interrupt mechanism. In this report the implementation, code and energy measurements will be explained in terms of advantages and disadvantages of the two different approaches.

The idea of the simplest approach is to determine when the gamepad is ready to send data by interrogating the microcontroller continuously in a round robin sequence. This approach is called *polling*, because of the continuous *polls* of the button states. When the terminal has data to send, it sends back an acknowledgment and the transmission begins. So in this case one of the LEDs will be turned on if one of the buttons are pressed.

In contrast of the second approach, we have the interrupt-driven system in which pressing one of the buttons on the gamepad generates an interrupt. This means the CPU does not need to continuously poll the button states and can do something else while waiting for an interrupt. A logical implementation when waiting for an interrupt would therefore be to make the CPU go to sleep as to use less power. The interrupt-process is managed by an interrupt handler. In this case the requirement of the exercise is to turn on or off (on in our case) a LED on the gamepad itself. After the handler is executed, the main program resumes running from where it was stopped, which will be *go back to sleep*.

It is expected less current draw from the interrupt approach, because the processor would be in sleep mode while the buttons are not pressed, while in the polling approach the processor is always running. Using Interrupts we are also able to use the different energy modes that are available on the board, this makes it possible to save even more power.

2 Solutions

2.1 Baseline Solution

The GPIO clock in the Clock Management Unit (CMU) was enabled and the drive strength of the LEDs set to HIGH. Pins 8 to 15 of port A were set to output, these are the pins that correspond to the LEDs on the gamepad. The LEDs are then initialized to being turned off by writing "0xFF00´´ to the GPIO_DOUT register. Lastly the buttons are set to input and the internal pull-up resistors enabled.

At this point, we were able to control the LEDs by writing into the GPIO_PA_DOUT register. It was decided to read the values of the input and to write it as an output. For each button pressed, the corresponding LED is lit. This was done by mapping the button press to the corresponding LED. The code for mapping the LEDs was implemented in a main function as shown in the code from *polling.s*.

```
159 main:
160
161         ldr r0, =GPIO_PC_BASE                   // load base address
162         ldr r1, [r0, #GPIO_DIN]                 // load button values
163         lsl r1, r1, #8                          // shift into LED registers
164         ldr r2, =GPIO_PA_BASE                   // load DOUT base
165         str r1, [r2, #GPIO_DOUT]                // store LED values
166
167         b main                                  // main loop
```

polling.s

There is a drawback to this solution, and that is the CPUs continuous polling of whether any buttons have been pushed to change the state. The behaviour of the system is described in the state machine in figure 2.1.

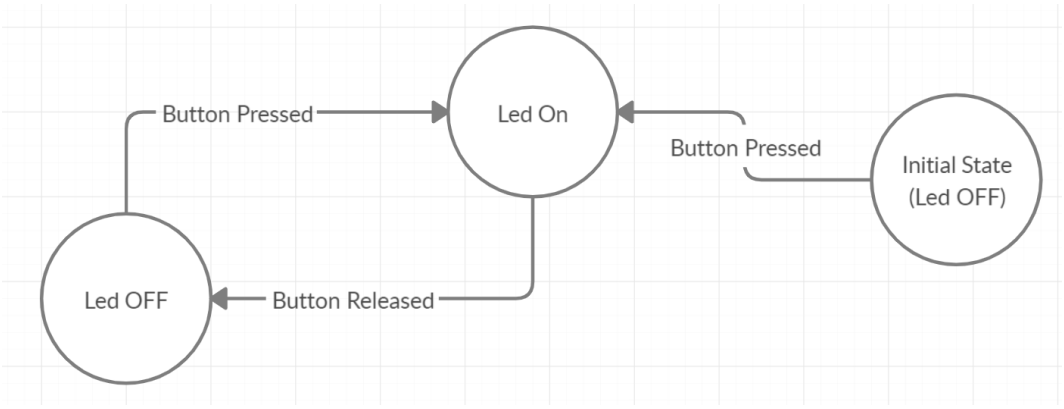

Figure 2.1: Baseline state machine.

2.2 Improved Solution

To improve the efficiency of the baseline solution, an interrupt routine was implemented. The processor will therefore only need to turn LEDs on or off when it gets an interrupt. When the processor get the interrupt, a interrupt handler will be called. In the handler the actual code for switching the LEDs will be executed. When the code is executed the handler returns back to where the processor left off. This saves a lot of resources in terms of energy saving. Also it leds the processor free to do other tasks and not keep polling the LEDs like in the baseline solution.

The implementation of the improved solution starts with the same initializing as the polling solution in terms of enabling the clock, input/output pins etc. There is though some extra initializing to be done for enabling the interrupts. First the interrupt is cleaned, because it was causing problems on the initializing of the LEDs. Then interrupt is enabled by writing to the registers shown in the code from *interrupt.s*.

```
138         ldr r3, =0x22222222                 // load value
139         str r3, [r1, #GPIO_EXTIPSELL]       // store in EXTISPELL
140
141         ldr r3, =0xFF
142         str r3, [r1, #GPIO_EXTIFALL]        // store EXTIFALL
143         ldr r3, =0x00
144         str r3, [r1, #GPIO_EXTIRISE]        // store in EXTIRISE
145         ldr r3, =0xFF
146         str r3, [r1, #GPIO_IEN]             // store in IEN
147
148         ldr r1, =ISER0                      // load base address
149         ldr r3, =0x802                      //
150         str r3, [r1]                        // store in ISER0
```

interrupt.s

Energy saving mode 3 was implemented as well. This saved a lot of energy which is described in detail in chapter 3. To save even more energy RAM blocks were disabled and the clocks in register CMU_LFCLKSEL were turned off. The processor now goes to deep sleep and only wakes up on an interrupt call to execute LED functionality, before returning to sleep again. This was done by writing to the SCR register and using the *wfi* command (wait for interrupt).

The LEDs work in a different way from the baseline solution. In the interrupt handler first the interrupt is cleaned before making two sets of *if*-checks by reading the interrupt flag to figure out which button was pushed. If button one or two was pushed, it will map the button push to the corresponding LED and switch between LED one and two lighting up. If either of the other buttons are pushed the *else*-statement will be executed, which turns the other six LEDs on and off by XORing them. The flowchart of the implementation is shown in figure 2.2.

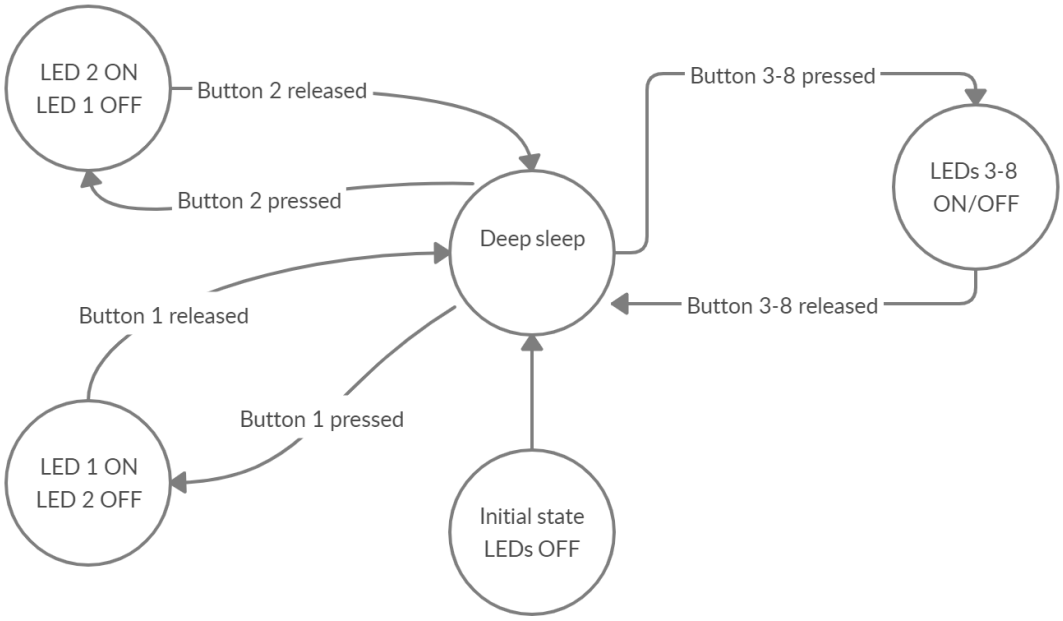

Figure 2.2: Interrupt state machine.

3 Energy Measurements

The EFM32GG-board offer the possibility to put the microcontroller in dierent energy modes, depending on what resources are needed. The idea is to go to sleep while the processor is waiting for an interrupt signal and thus, save power by not needing to check continuously if the buttons are pushed. The CPU wakes up when the interrupt arrives, and goes back to sleep after the interrupt handler returns.

Energy modes are controlled by the Energy Management Unit (EMU). The EMU manages all the low energy modes in the EFM32GG microcontroller. Each energy mode manages clocks and how much of the various peripherals are available. The five different modes are EM0 through to EM4. EM0, also called the active mode, is the energy mode in which any peripheral function can be enabled and the Cortex-M3 core is executing instructions. EM1 through EM4, also called low energy modes, provide a selection of reduced peripheral functionality that also lead to reduced energy consumption. The Reference Manual for EFM32GG describes the interrupt functionality as being on in EM1, EM2 and EM3, but off in EM4. This means that in our exercise we are only able to use the first 3 different modes in order to save as much energy as possible.

3.1 Baseline

The current draw of the baseline solution was $3.33mA$ as shown in figure 3.1.

Figure 3.1: Baseline current draw for the polling method.

3.2 Energy mode 0

Running the interrupt solution with no energy saving implemented it draws $4.32mA$ as show in in figure 3.2.

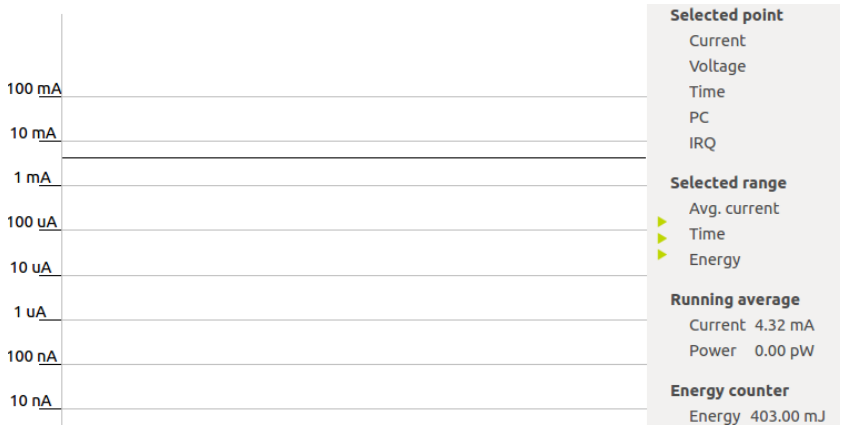

Figure 3.2: Current draw in energy mode 0.

3.3 Energy mode 1

Implementing energy mode 1 puts the processor to *sleep*. The measured current draw in this mode was $1.25mA$ shown in figure 3.3. This is a factor of ≈ 3.5 times less current draw than EM0.

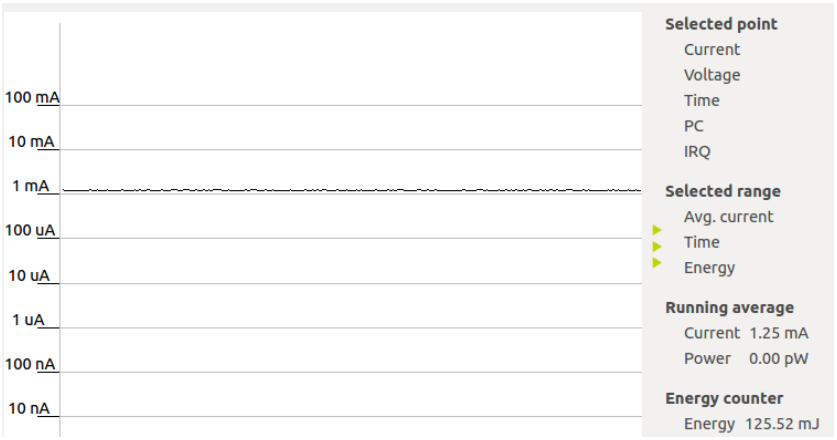

Figure 3.3: Current draw in energy mode 1.

3.4 Energy mode 2

This mode puts the processor into *deepsleep* mode. In deepsleep the current draw is significantly less as more peripherals are turned off. Figure 3.4 shows that the current draw in this mode is $1.81\mu A$. This is ≈ 2387 times less current draw than EM0, which is a significant amount of energy saving.

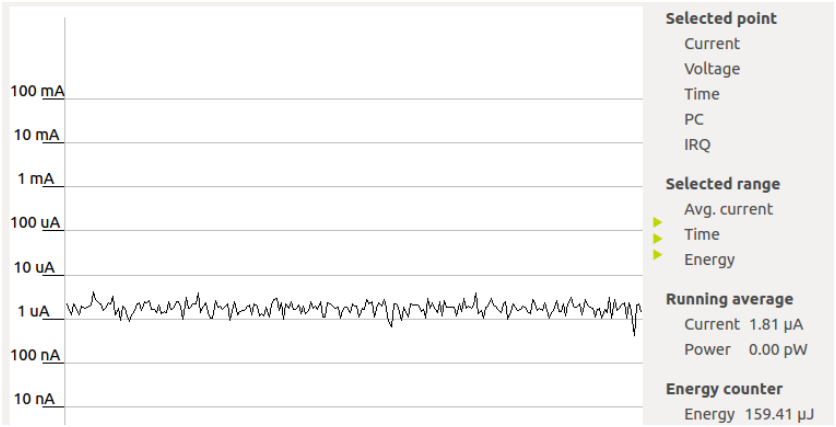

Figure 3.4: Current draw in energy mode 2.

3.5 Energy mode 3

The difference in the current between EM2 and EM3 should be $0.3\mu A$ as given by scilabs. This difference is so small it was hard to distinguish the uncertainty of the current measurement device from the actual current reduction and the difference between EM2 and EM3 was therefore negligible.

3.6 Disabling RAM

RAM blocks where disabled to lower the current draw even more. With the RAM blocks disabled the current draw went down to $961nA$ shown in figure 3.5.

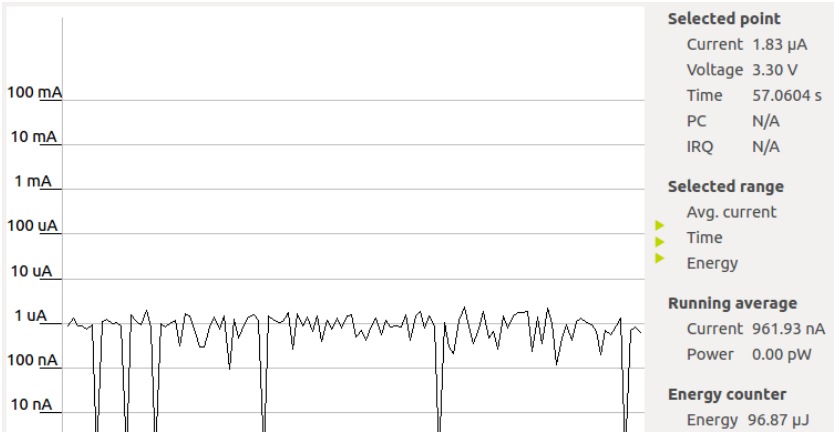

Figure 3.5: Current draw with powered off RAM blocks.

3.7 Current draw on button push

The button pushes adds to the amount of current that is drawn. How much will of course depend on how often the buttons are pressed, but it can be shown that a single button press draw somewhere around $100\mu A$ as seen in figure 3.6. Therefore, in the baseline solution on a "mA"-level it does not contribute a lot to the overall energy draw, but makes a significant impact in the low power modes.

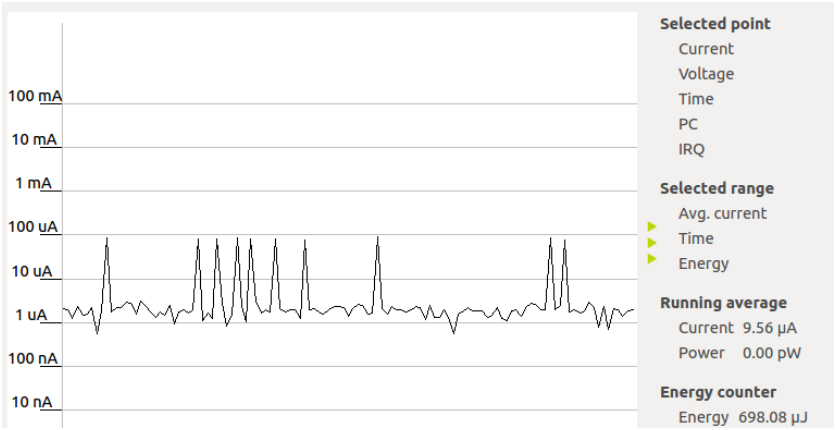

Figure 3.6: Current draw when the buttons are pushed.

3.8 Overall system

The measurements shown so far have not included the LEDs on the gamepad. Figure 3.7, 3.8 and 3.9 show the current draw of the whole system, with the current draw of the LEDs as well. With the LEDs turned on the current draw with all the other energy saving implemented is $\approx 80mA$ shown in figure 3.7.

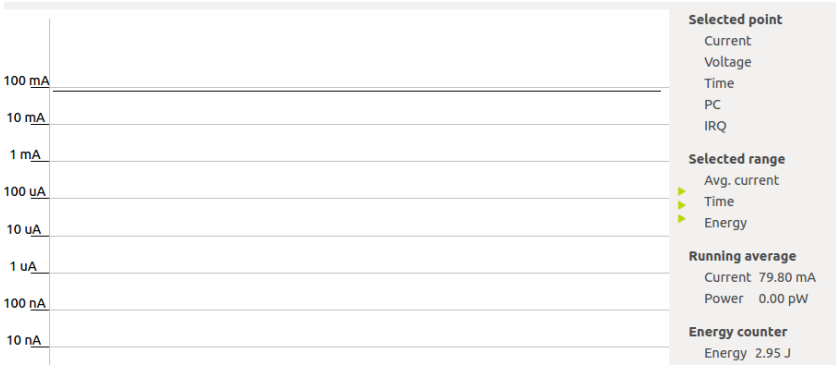

Figure 3.7: High drive strength with LEDs on.

To save the most amount of power the drive strength of the LEDs was set to the lowest level and the amount of current draw was $4.05mA$ when all LEDs are on. The system is therefore very efficent.

Figure 3.8: Lowest drive strength with LEDs on.

The best result of the current measurements was obtained when the LEDs was off and the drive strength set to lowest, together with the other energy saving implementations. This gave a current measurement of $\approx 922nA$ as shown in figure 3.9.

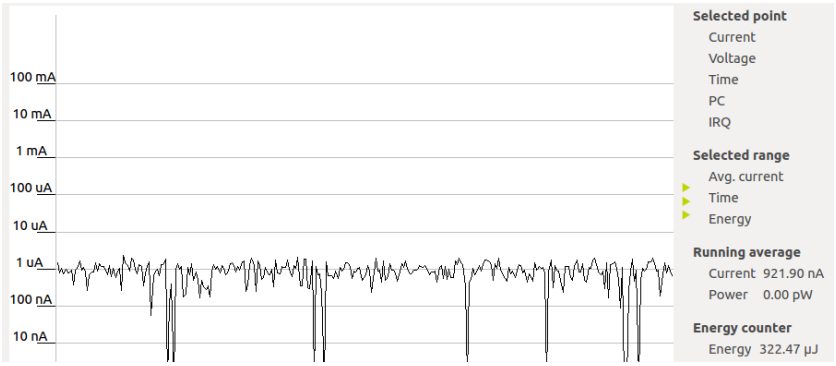

Figure 3.9: Lowest drive strength with LEDs off.

4 Conclusion

The system implementation of a system for turning LEDs on and off in a certain pattern benefited a lot from introducing an interrupt routine in terms of energy saving. In the mode where the energy saving was the highest, the difference from the baseline current draw was 3357 times less. We did not measure the power, only the current, this might make the results slightly inaccurate if the voltage varies a lot with the energy saving modes. It seems though, from the documentation of EFM32GG, that the voltage in static power draw stays stable due to the internal voltage regulators.

The code for the polling approach can be found in the folder ../exercise1/polling/ and likewise for the interrupt approach in ../exercise1/interrupt/. Each of the approaches have their own makefile and should work independently of each other.