NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4258 Low-Level Programming
Laboratory Report

Exercise 1

Group 4:

Sigve S. Nordgaard
Thomas Aven
Lasse A. Eggen

September 19, 2017

1 Overview

In this exercise we have written very simple programs to control LEDs by pressing buttons on a gamepad using GPIO. This consists of two very similiar programs that control LEDs as an 8-bit binary counter by using two approaches: busy-waiting/polling and interrupts.

To improve on energy efficiency we enabled deep sleep by going to energy mode 2 for our interrupt based program.

We have also included ex1_custom in addition to the two solutions, that is code written to use both button interrupts and timer interrupts. We did not manage to make the hardware timers work exactly as we want, but have still included the code, as the LEDS are managed in a slightly different way.

1.1 Project structure, Makefile

Our project code is structured into three directories; base, improved, and custom. To build and upload, one would maneuver into the right directory, and run "make", then "make upload". We experimented with having everything in one directory, and a more complex Makefile (e.g. "make base && make uploadbase" with pattern matching), but decided this was the most convenient for someone doing the testing (we included this Makefile in the root of our project as Makefile.example, to show what it would look like). We have also created a core directory that includes our common files, as to avoid having duplicates (this is reflected in changes in the Makefile and importing using "../core/efm32gg.s"). Thus, for each different project, the directory will only contain the ex1.s file and its corresponding Makefile.

1.2 Baseline Solution

The counter program is controlled by buttons:

- SW1: Increment counter
- SW2: Reset counter
- SW3: Decrement counter

A counter is set to 0. Every time the increment button is pushed, the counter increases by 1, when the decrement button is pushed it is decremented by 1 and reset to 0 when the reset button is pressed. This value is reflected on the eight LEDs on the gamepad, keep in mind that the LEDs have the lowest significant bit to the left, whilst words on

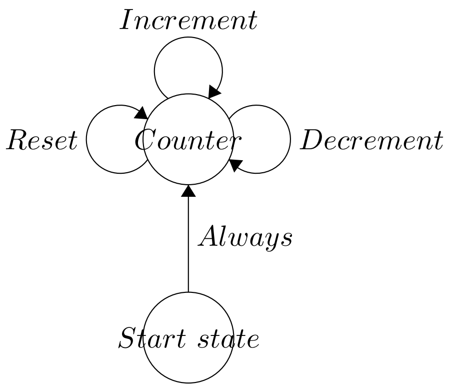

Figure 1.1: State machine of counter program.

the EFM32GG are in little endian format. When the 8-bit value overflows into bit 8 of the 32-bit register, the counter will necessarily restart (this overflow is thus a bug becoming a feature, as we don't care about bits other than 0-7).

The solution was implemented without interrupts by keeping the state of a button press in a register at all times. This makes sure that whenever the increment or decrement button is pushed, the corresponding action is performed only once - when the button is first activated. Otherwise, the subroutine will notice that the button was already pushed during the last loop cycle.

Figure 1.1 shows how the program's states are after any action. At reset/start of the program the state transitions to the Counter state. Every action transitions to the Counter state. All Counter states are physically represented by the LEDs. Values from 0 to 255 are possible. The counter wraps around, i.e. decrements from 0x00 to 0xFF and increments from 0xFF to 0x00.

More specifically for the implementation, we are using registers r5-r8 to keep global state variables, instead of using RAM. r5 and r6 keep the base of GPIO_PA and GPIO_PC, respectively, r7 keeps the state of a button push, and r8 keeps the binary counter. This persists through interrupts. We have also added a few extra symbols like SW{1-3} to improve readability.

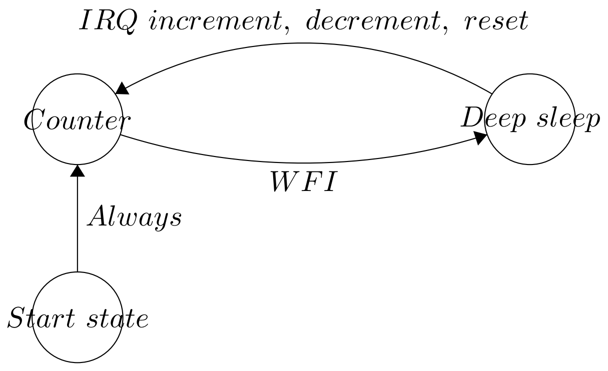

Figure 1.2: State machine of counter program with interrupts and deep sleep.

1.3 Improved Solution

To reduce the power consumption of the micro controller when running the program the Wait For Interrupt instruction is passed to the processor. The clock stops and execution suspends until an interrupt signal is received. This decreases power consumption drastically.

For the improved solution, we are using interrupts instead of an internal state to keep track of button presses. This means that our solution does only one thing in the main loop "wfi", to deep sleep. The microcontroller will then sleep until it is woken up by interrupts triggered by the buttons, which will jump to handlers in IRQ#1 or IRQ#11 according to whether it is GPIO_EVEN or GPIO_ODD. (1) We use the same interrupt handler for both, which makes it jump to the read_buttons label. Here we do just about exactly the same as in the base solution, with the exception being that we no longer have to keep the state of a button press in mind, as interrupts will, by configuration, only be fired when there is a transition into the asserted state.

To set up deep sleep, we enable energy mode 2 by writing 0x6 to the SCR-register. This makes the power consumption marginal in comparison to using interrupts with busy-waiting in the main loop (e.g. "eor r1, r1, r1") instead of going to sleep.

2 Energy Measurements

For both of our solutions, we decided to keep JP1 in the lower position, to make sure that the power consumption from the game pad did not affect our results. Thus we are only providing power consumption from the Cortex-M3 itself, and not including the power needed for the LED lights etc. We used eAProfiler with an AEM sampling period of 100ms, and took screenshots from the VM to obtain measurements for current consumption.

Our base solution, which uses polling and busy-waiting, can be seen in figure 2.1. We estimate that the current consumption here is a steady 4-5mA, which makes sense considering that every cycle of the CPU is spent executing a fodder loop, while polling to see if buttons are pressed. One thing to notice, is that we would see a jump of about $\tilde{8}$0µA whenever we pushed a button (this is also true for the interrupt solution). After a bit of speculation, we ended up concluding that this was due to the fact that the buttons are active low, and that this affects the power consumption in some manner irrelevant to our own code on assertion.

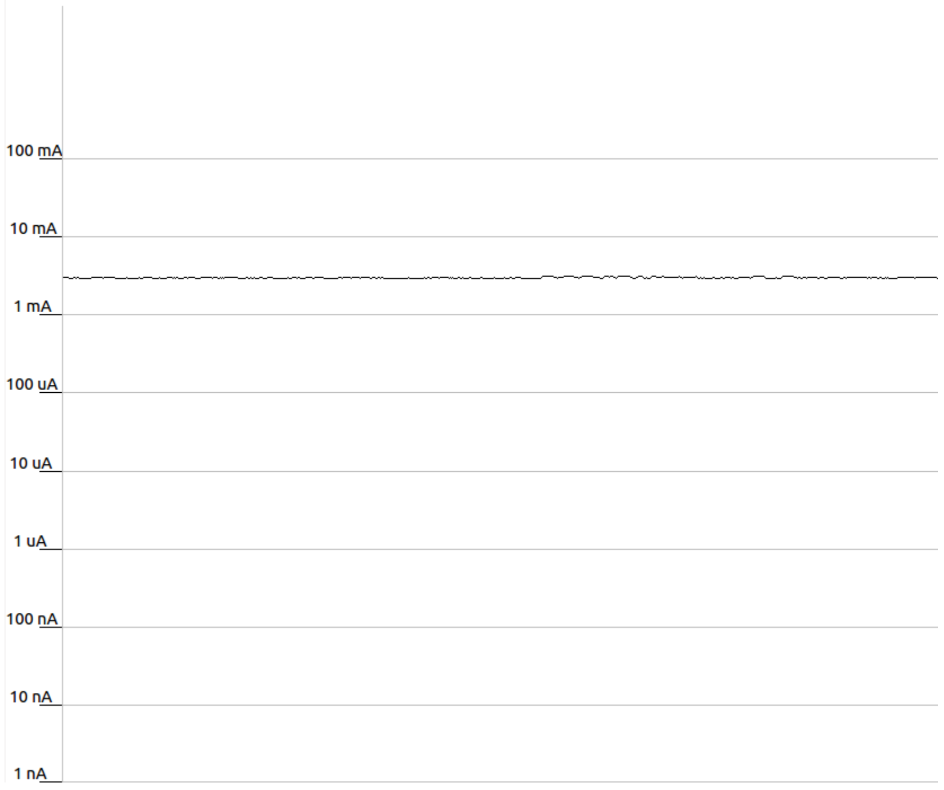

Figure 2.1: Base program current consumption.

In addition to this, the graph pictured in figure 2.1 strongly resembles the graph for power consumption that a solution with interrupts would provide, given that no sleep modes are activated (the graph is not pictured).

For our improved solution that uses interrupts, there is no noticeable difference unless one puts the CPU to sleep mode while waiting for interrupts. Thus, instead of doing some empty instruction during the main loop, we use "wfi", which essentially puts the processor to sleep until it is woken up by external interrupts (wait for interrupt). By having set up SCR, we have enabled energy mode 2 ((1), p.8) where only selected peripherals are available, giving a current consumption as low as 1.1µA. This is depicted in figure 2.2, including the previously mentioned jumps of about 80µA during button presses.

We decided not to pursue further current consumption optimization on this first exercise, as 1µA is already very low. However, according to (2), there are several other current consumption methods, like disabling RAM blocks (as we do not use RAM here). We think that this exercise, that is fairly simple, would notice much less from these methods, and that they are more relevant in later exercises.

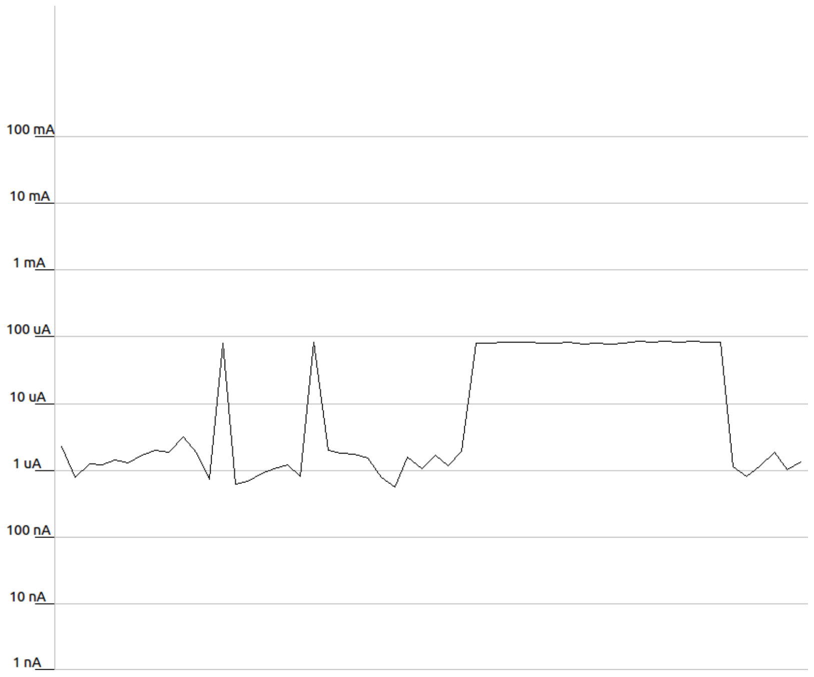

Figure 2.2: Improved program current consumption.

Show how the power consumption changes when your solution is responding to different events in different states (or usage scenarios) with plots, and comment on your findings. How does the energy efficiency of the improved solution compare to the baseline?

Bibliography

[1] E. Micro, *EFM32GG Reference Manual*. Energy Micro, 2016. [Online]. Available: https://www.silabs.com/documents/public/reference-manuals/EFM32GG-RM.pdf

[2] ——, *EFM32 Energy Optimization*. Energy Micro, 2016. [Online]. Available: https://www.silabs.com/documents/public/application-notes/an0027.pdf