NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4258 Low-Level Programming
Laboratory Report

TDT4258 Exercise 1

Group 10:

Anders Sildnes
Rasmus Stene
Ole Sørlie

September 19, 2016

1 Overview

The number of transistors per chip increase every year but the power density stays the same[1]. Batteries are getting better, but in return demands grow bigger. The challenge of the modern day computer architect is to develop energy efficient systems while at the same time keeping or increasing the speed of the system.

This paper shows a simple application of switching eight LED lights on and off with eight switches, one for each light. We demonstrate how one can optimize for energy usage by using an interrupt-driven method rather than polling. In addition we configure the hardware to use as little energy as possible by disabling certain features. The hardware is made up of an EFM32GG DK3750 development board with an accompanying EXP32 prototyping board. The development board has 128 kB SRAM memory for memory and 1 MB flash-memory for permanent storage. The core architecture runs ARM Cortex-M3, so our work is done in assembly thumb2 syntax. Benchmark results are obtained using Silabs's provided software and tools.

1.1 Baseline Solution

We set each pin on our General Purpose I/O (GPIO) controller to use a pull-down configuration. Pull-down means that when the switch is off, it is connected to ground, and thus it uses slightly less current when buttons are not being held down compared to the more common pull-up configuration. Now whenever a button is pushed, we can read the current from our input gate on the GPIO controller. In assembly, this is seen as a bit having the value '1'. We read each value from the prototype buttons as eight consecutive bits at a given location in memory, defined by the documentation[2]. The LED buttons are read and toggled the same way by reading/writing to another location in memory. Thus, our polling method is simply reading 8 bits from the buttons and writing them to the LED lights section as shown in the state diagram in Figure 1.1. This is a rather energy inefficient solution, because the GPIO pins are polled at a frequency close to that of the system clock frequency. This means that the processor is constantly doing some work while it is just waiting for a change to occur on the GPIO register. Parts of the code for the baseline solution is shown in Listing 1.1.

The first section of the code contains the reset handler. This is the point in the code where the CPU will start executing code after a reset has occurred. No instructions are executed in the reset handler, and the function simply branches to the next part of the program code called "init". In this function the GPIO is activated in the Clock Management Unit before the GPIO on port A connected to the LEDs are configured as

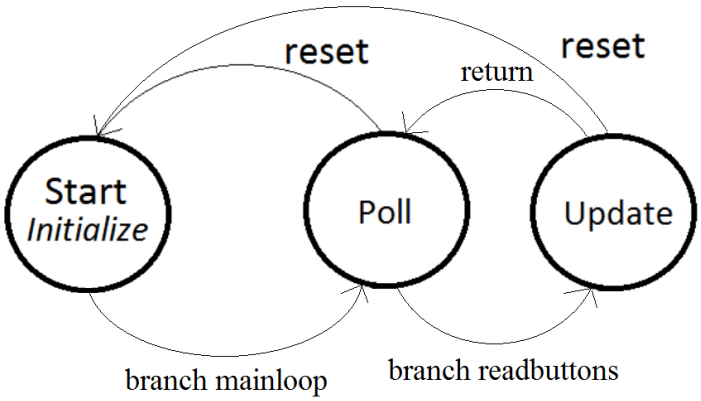

Figure 1.1: State diagram of the polling solution.

push-pull output with maximum drive strength. The last part of the init function sets the pins on port A as inputs with internal pull-up before branching to "mainloop".

The mainloop function simply branches to another function called "readbuttons", writes the result from that function to the LEDs and then branches back to the the beginning of mainloop again. The readbuttons function reads the GPIO corresponding to the buttons and then shifts the results 8 positions to the left. This is because the buttons are pins 0-7 on port A while the LEDs are pins 8-15 on port C. The program then returns back to where it was in the mainloop function. The result is that when you push a button on the control pad, one LED turns on, and when you let go of the button, the LED turns off.

```
1  // Reset handler
2    .globl   _reset
3    .type    _reset, %function
4    .thumb_func
5  _reset:
6    //Do nothing
7    b init
8
9    .thumb_func
10 init:
11   //Configure CMU to enable GPIO
12   ldr r1, =CMU_BASE
13   ldr r2, [r1, #CMU_HFPERCLKEN0]
14   mov r3, #1
15   lsl r3, r3, #CMU_HFPERCLKEN0_GPIO
16   orr r2, r2, r3
17   str r2, [r1, #CMU_HFPERCLKEN0]
18
19   //Store often used addresses in reg 4-6
20   ldr r4, =GPIO_PA_BASE
```

```
21    ldr r5, =GPIO_PC_BASE
22    ldr r6, =GPIO_BASE
23
24    //Set all pins as push-pull with defined drive strength
25    ldr r2, =0x55555555
26    str r2, [r4, #GPIO_MODEH]
27
28    //Set maximum drive strength
29    mov r2, #2
30    str r2, [r4, #GPIO_CTRL]
31
32    //Set all LEDs off
33    ldr r2, =0xFFFF
34    str r2, [r4, #GPIO_DOUT]
35
36    //Set all pins as inputs and with internal pullup
37    ldr r2, =0x33333333
38    str r2, [r5, #GPIO_MODEL]
39    ldr r2, =0xFF
40    str r2, [r5, #GPIO_DOUT]
41
42    b mainloop
43
44    .thumb_func
45    mainloop:
46      bl readbuttons
47      str r0, [r4, #GPIO_DOUT]
48      b mainloop
49
50    .thumb_func
51    readbuttons:
52      ldr r0, [r5, #GPIO_DIN]
53      lsl r0, r0, #8
54      mov pc, lr
```

Listing 1.1: Baseline solution code.

Some energy could be saved in the baseline solution without using interrupts by reducing the number of times the buttons are polled each second, and instead enter a delay routine after each poll. This could for instance reduce the time between each poll to 20 ms, which would be suitable for a controller intended to be used by a human. If the delay routine consists of NOP instructions or instructions which consumes less energy than the polling routine, then some energy can be saved using this method.

1.2 Improved Solution

With the polling solution, the system continuously polls the buttons checking if any changes have occurred. A better way of doing this would simply be to have the buttons "tell" the system when a button has been pushed. This can be accomplished through interrupts. An interrupt basically "interrupts" the current program execution in order

to execute some predefined interrupt subroutine, before returning back to where it was in the main program. This allows the processor to do something else while waiting for a button to be pressed, for instance entering a lower power state. A lower power state more or less shuts down features not in use in order to conserve energy.

The EFM32GG has 5 energy modes, EM0-EM4, where EM4 is the lowest energy mode. EM0 is the normal mode, where all peripherals are active and the CPU consumes around 200uA/MHz. EM1, also known as "sleep mode", all peripherals are still active, but the CPU is sleeping, reducing the current consumption to 50uA/MHz. EM2, called "deep sleep", turns off some of the peripherals, but leaves some low energy peripherals still available. In this mode only the low frequency oscillator is available, and the CPU and RAM is retained. In this mode the current consumption is as low as 1.1uA/MHz.

EM3, the "stop mode", turns off the low frequency oscillator as well, while still retaining CPU and memory. The CPU can still be awoken through asynchronous interrupts, and the current consumption can be as low as 0.9uA/MHz. EM4 is the mode where the device is "shut down", with a current consumption as low as 20nA. In this mode almost all chip functionality is turned off, and CPU and memory is not retained (memory can be retained through Backup RTC). The device can only be awoken from this state through reset, a power cycle, GPIO pin wake up or backup RTC.

Parts of the code for the improved solution is presented in Listing 1.2. As with the baseline solution, no instructions are executed in the reset handler apart from a branch to the "init" function. As with the baseline solution, the init function configures the CMU to enable the GPIOs and configures the GPIO for the LEDs and the buttons. In addition, it configures the buttons as interrupts, which points to the GPIO handler function in the interrupt vector table. The interrupt routine triggers on both rising and falling edges caused by pressing and releasing the buttons.

```
1  // Reset handler
2    .globl   _reset
3    .type    _reset, %function
4    .thumb_func
5  _reset:
6    //Do nothing
7    b init
8
9    .thumb_func
10 init:
11   //Configure CMU to enable GPIO
12   ldr r1, =CMU_BASE
13   ldr r2, [r1, #CMU_HFPERCLKEN0]
14   mov r3, #1
15   lsl r3, r3, #CMU_HFPERCLKEN0_GPIO
16   orr r2, r2, r3
```

```
17    str r2, [r1, #CMU_HFPERCLKEN0]
18
19    //Store often used addresses in reg 4-6
20    ldr r4, =GPIO_PA_BASE
21    ldr r5, =GPIO_PC_BASE
22    ldr r6, =GPIO_BASE
23
24    //Set all pins as push-pull with defined drive strength
25    ldr r2, =0x55555555
26    str r2, [r4, #GPIO_MODEH]
27
28    //Set maximum drive strength
29    mov r2, #2
30    str r2, [r4, #GPIO_CTRL]
31
32    //Set all LEDs off
33    ldr r2, =0xFFFF
34    str r2, [r4, #GPIO_DOUT]
35
36    //Set all pins as inputs and with internal pullup
37    ldr r2, =0x33333333
38    str r2, [r5, #GPIO_MODEL]
39    ldr r2, =0xFF
40    str r2, [r5, #GPIO_DOUT]
41
42    //Configure interrupt for buttons
43    ldr r2, =0x22222222
44    str r2, [r6, #GPIO_EXTIPSELL]
45    ldr r2, =0xFF
46    str r2, [r6, #GPIO_EXTIRISE]
47    str r2, [r6, #GPIO_EXTIFALL]
48    str r2, [r6, #GPIO_IEN]
49    ldr r2, =0x802
50    ldr r1, =ISER0
51    str r2, [r1]
52
53    //Configure SCR
54    ldr r1, =SCR
55    ldr r2, =0b00110
56    str r2, [r1]
57
58    b mainloop
59
60    .thumb_func
61    mainloop:
62      wfi
63      b mainloop
64
65  // GPIO handler
66  // The CPU will jump here when there is a GPIO interrupt
67
68      .thumb_func
69  gpio_handler:
70    ldr r0, [r5, #GPIO_DIN]
```

```
71    lsl r0, r0, #8
72    str r0, [r4, #GPIO_DOUT]
73    ldr r1, [r6, #GPIO_IF]
74    str r1, [r6, #GPIO_IFC]
75    bx lr
```

Listing 1.2: Improved solution code.

In the last part of the init function, the System Control Block register is configured so that the device can enter sleepmode and that it will go back to sleep after an interrupt has occurred. In the mainloop function, a wfi instruction is executed, which tells the processor to sleep until an interrupt occurs.

Once a button is pressed, the CPU will awake and jump to the GPIO handler routine. Here it reads the button GPIO port register, shifts the result 8 positions to the left, and then outputs it on the GPIO port connected to the LEDs. Afterwards the interrupt flag is cleared and the program exits the routine. It then returns to its sleep state and waits for a new interrupt from the GPIO pins.

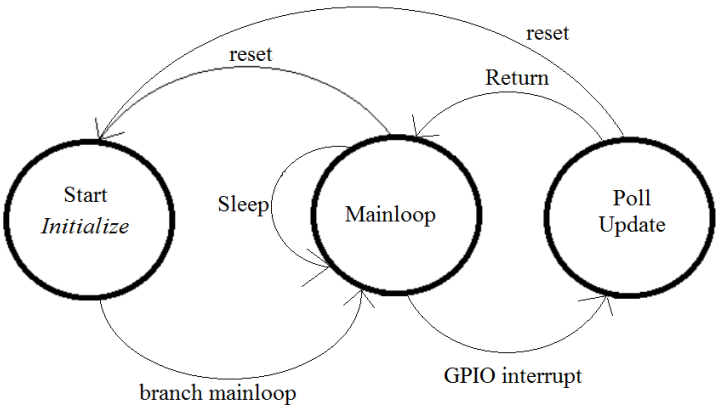

Figure 1.2: State diagram of the improved solution.

The state diagram for the improved solution is shown in Figure 1.2.

1.3 Improved solution with LED patterns

In addition to adding interrupts, we implemented more button-functionality. Each button received a handler with a an action for the LED lights. The features are described in table 1.1. The functionality is implemented primarily as a series of bit-wise operations to track and edit the state of the LED lights.

Button	Action
Left	Move LED lights to left
Up	Increase LED light-strength
Right	Move LED lights to left
Down	Decrease LED light-strenght

(a) Left buttons

Button	Action
Left	Light up an extra LED light
Up	Light all LED lights
Right	Light down an extra LED light
Down	Return to starting state (one LED light)

(b) Right buttons

Table 1.1: The prototype board buttons and their corresponding actions

2 Energy Measurements

We used the EnergyAware Profiler in order to measure the power consumption of both the baseline, improved and improved with fancy LED solution. We measured the power consumption when the device was in idle as well as when a button was pressed. In all cases the LEDs were driven with maximum drive strength, or around 20mA per LED. Figure 2.3 shows how much energy usage each drive strength uses for one LED light.

The first measurement was done on the baseline solution. As can be seen from Figure 2.1, the current when the device is in idle is around 3.5mA. When a single button is pressed, the current changes to 17.7mA. This is due to the LED switching on. When multiple buttons are held, more LEDs switch on and therefore the current consumption increases. Because the CPU polls, it is unable to enter sleep mode and therefore, the no current is saved when there is no activity on the GPIO.

The second measurement was done on the improved solution. As can be seen from the graph, the idle current for the device was around 1.1uA. Once a single button is pressed, the current consumption rises to around 13mA. The majority of the power is spent driving the LED, which is driven with maximum drive strength, while the remaining is spent running the CPU. When multiple buttons were held, the power consumption expectantly rose, as can be seen from Figure 2.2.

The power consumption in the improved solution is much better when in idle mode compared to the baseline solution. This is because the improved solution can sleep when there is no activity on the GPIO. The LED patterns code is very similar to the improved solution except is has an idle of *at least* 500 uA since there is always one light on.

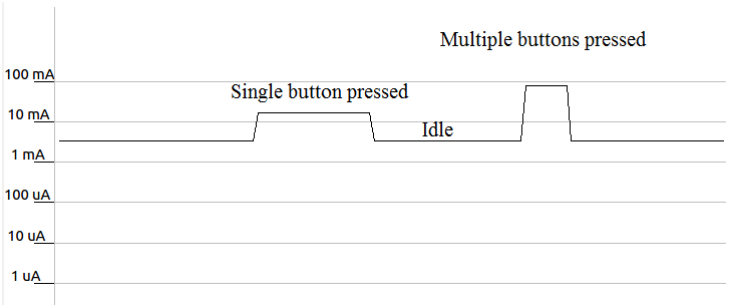

Figure 2.1: Energy consumption for baseline solution.

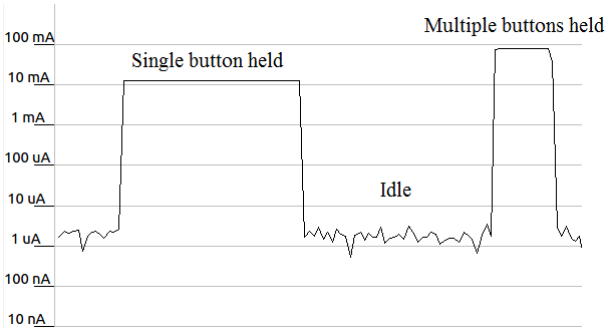

Figure 2.2: Energy consumption for improved solution.

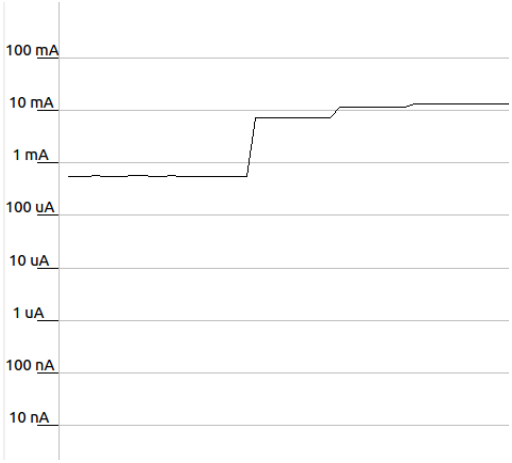

Figure 2.3: Power usage when gradually going from the lowest drive strength on the LED lights to the highest (using the improved solution with LED patterns)

2.1 Conclusion

As can be seen from the energy measurements, the improved solution requires far less energy than the baseline solution. As already explained, this is because the baseline solution wastes energy by constantly checking the state of the GPIO pins. In contrast, the improved solution simply waits for the GPIO pins to give an indication that an event has occurred, meaning that a button has been pressed or released. Compared to the baseline solution, which used 3.5mA in idle, the improved solution only used 1.1uA. In other words, when not in use, the improved solution uses 3000 times less energy than then baseline solution. In battery driven application, where every uW is shaved off at all cost, no energy can be wasted on polling.

Bibliography

[1] Robert H. Dennard, Fritz H. Gaensslen, Hwa nien Yu, V. Leo Rideout, Ernest Bassous, Andre, and R. Leblanc. Design of ion-implanted mosfets with very small physical dimensions. *IEEE J. Solid-State Circuits*, page 256, 1974.

[2] Silicon Labs. Efm32gg reference manual. It's learning.