NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4258 Energy Efficient Computer Design
Laboratory Report

Exercise 1

Group 27:

Eirik Prestegårdshus
Tor Braastad Gylder

February 8, 2015

Abstract

This assignment is about learning to program and compile assembly code to the processor. While doing this the group is also experimenting with the power consumption in their software design, and using their knowledge to alter the setup for better energy saving. In this task the group will get an understanding with the tools and get familiar with these.

The first task is to get a led light up when a button is pressed on the gamepad. After this it's all about the energy saving.

The first code was a basic loop. It worked, but the energy consumption was around 3.6 mA. When interrupts was introduced the power consumtion rose to an average of 4.6 mA. Finally deep sleep was utilized in the design software. That way the processor would sleep and only be interrupted by a button-press. During this button-press it would figure out which LEDs to light up. The average power consumption without button press was 1.5 µA, and with button press it was 83 µA.

1 Introduction

This is the first of three parts of making a simple video-game with a custom controller, and it's a practical assignment in the class TDT4258. In the first part the goal is to program an ARM processor to manage the controller. It's required to write assembly code and compile it to a binary file which is read by the processor. Ideally the controller should have some kind of logical behavior, for example light a LED upon button-press, and use as little power consumption as possible.

2 Background and Theory

This part focuses on programming assembly code for the ARM Cortex-M3 processor.In this case we use the GPIO (general Purpose I/O) to control the processor on the EFM32GG DK3750 development board.

2.1 Hardware

2.1.1 EFM32GG DK3750

The EFM32GG DK3750 development board is used in this exercise. All of I/O controllers on the board are memory mapped. Which gives you the ability to program the controller by reading and writing in the special memory locations. More documentation can be found in [8]

2.1.2 ARM Cortex-M3

The main component on the development board is the ARM Cortex-M3 processor. This processor supports the ARM Thumb instruction set and consist of a 32 bit pipelined RISC processor. More documentation is given in [7]. The GPIO can change its pins to work as inputs or outputs specified by the user.

2.1.3 Gamepad

The gamepad is connected to the board through input ports 0-7 on the PC connections and output ports 8-15 on the PC connections.

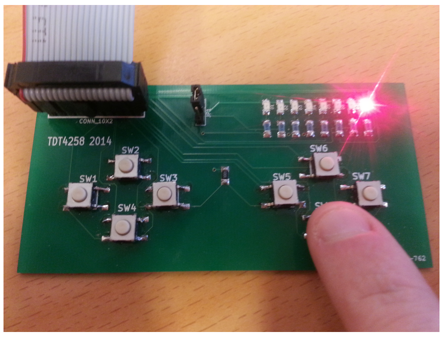

Figure 2.1: Gamepad

As you can see in figure (2.1) it also contains led lights in the upper right corner.

2.2 Software

GNU software is widely used by Linux machines. The GNU tools used in this exercise are:

- GNU AS: Makes an assembly file. Manual [4].
- GNU LD: Links object files together to write a .elf file. Manual [5].
- GNU Make: Compiles/assembles and link the files into an executable file. Manual [6].
- GNU Debugger: Debugger program that runs while the program is being executed. Manual [3].

3 Methodology

3.1 Assembly Coding

When coding assembly there's a lot you need to take into consideration that might not be so obvious to everyone. For example, when using a part of the device, it's always the need of turning it on by changing some register somewhere. The actual logic of the gamepad is very simple, it's simply turn on a LED when there's a button pressed.

The first step is enabling the functions it's going to use. Let's take polling as an example, the reset handler [2] is included in the appendix. How polling works is described in the next section. In this example we first have to enable clocking for the parts the program need to use. This is the first paragraph of the code.

The second step is enabling parts of the GPIO. The buttons has to be set as input and the lights as output.

Finally, it's possible to define the logic of the gamepad. In the polling example we simply take the input read on the port of the buttons and output it to the LEDs. lsl is a shift operation. It's needed to shift the bits received by 8 because the lights use bits 8-15 and the bits received are stored in the first 0-7 bits.

By using a loop we can do this more frequent than the eye can see, it will then seem like it happens simultaneously. The loop is infinite so the program never stops running.

```
        loop:
        ldr r4, [r3, #GPIO_DIN]              //Read buttons
        lsl r2, r4, #8

        str r2, [r1, #GPIO_DOUT]             //activate lights
        b loop
```

3.1.1 Polling versus Interrupts

Polling is a term for a program which continuously sample the status of an external device. Polling occur when the program very frequently have to read the ports from the buttons presses rather than waiting for a button-press before doing anything. By using a principle called interrupting, waiting for a button-press is easily implemented. Interrupts is basically an interrupt from the main code(reset handler), and can be triggered by ports on the GPIO. When an interrupt is triggered the program will jump into the belonging interrupt handler and do whatever is written there. After the interrupt handler is done, it'll return to the stage the program had before the interrupt. This brilliant mechanic allows the program to do nothing when there's no interrupts called. It can even be told to sleep to reduce it's power consumption a lot.

In the interrupt version of the code [1], interrupts and sleep were enabled and used. The processor will then end up in the sleep state(wfi) most of the time, and then handle interrupts when they occur. The first two lines fix interruption flags, so it is able to interrupt more than once. The three following lines is the same as before, simply reading button register and storing it to the LED register. The last line branches back to where it was previously, which is the sleep state.

```
        mov r2, #0x6                            //Sleep state
        ldr r4, =SCR
        str r2, [r4]
        wfi
```

```
gpio_handler:
        // r1, r3, r5 are used for base addr

        ldr r2, [r5, #GPIO_IF]
        str r2, [r5, #GPIO_IFC]

        ldr r2, [r3, #GPIO_DIN]                 //Read buttons
        lsl r2, r2, #8
        str r2, [r1, #GPIO_DOUT]        //Activate lights



        bx lr
```

3.2 Compiling and the GNU Toolchain

First the written code is made to an object file through GNU AS. In GNU AS some arguments must be set (in this case, get M3 instruction set and enable debug). After making the object file you can link different object files together by using GNU LD, which outputs a .elf file. GNU LD is also able to create a binary file out of the .elf file. A binary file is required by the processor to function. For first time setup eACommander will load the binary onto the processor. For additional uploads, GNU Make can compile and update the files as long as the processor stays connected.

GDB(GNU Debugger) is a nifty tool which can simultaneously run the program on the board and debug at the same time.

3.3 Further Improvements

There are some things which could have given better results over longer time.

3.3.1 mov/ldr

In some places in the code we deliberately chose mov instead ldr. mov has some restrictions with that if there are more than 8 bits seperating the ones it won't work. mov also moves the value directly into the register, while ldr must load a value then store it.

3.3.2 Less power consumption?

There are some ways the energy consumption could have been lowered. Since all of the I/O controls on the board are memory mapped, we could e.g. have lowered the clock frequency and removed some unused sram.

4 Results

All the power results were read without the power consumption of the LEDs.

4.1 Polling

The first version of the program ran a basic loop. All the memory allocations had to be ran again every time, even if a button was pressed or not. Due to all this logic running all the time, the power consumption was almost the same every time.

Average: 3.6 mA

Button-press: 3.7 mA.

4.2 Interrupts

As a test interrupting with a loop rather than the sleep statement was ran. This led to an even greater power consumption as the processor has to keep even more parts fed with power. The difference between average and button-press was minimal here as well.

Average: 4.3 mA

Button-press: 4.4 mA.

4.3 Energy saving

When deep sleep was enabled the power consumption fell to micro stages, and only increased when a button was pressed. This is a great result compared to the previous ones as the average usage is less than a tenth of the above.

Average: 1.5 µA

Button-press: 83 µA.

4.4 Problems and Debugging

There were a few problems when writing the code. There was however one thing that did not work. A function which might be desired on the gamepad is to press the button to turn on a light, then keep it lit until the button is pressed again. The first thing that comes to mind is a XOR with the current button-press and the current light. A possible solution might be to save current LED state in a separate register and do the XOR operation with the register, rather than directly with the GPIOs register.

The GDB were also a bit hard to use. It did not work properly with the deep sleep command. Instead of reading the program counter which stayed on wfi it kept going and printed every line in the interruption handler even though no interrupts triggered. When it reached the end of the interrupt handler it kept printing question-marks.

Although there debugger was a bit buggy, it was a great resource when we had to figure out what were contained in the different registers. The command *info registers* proved to be very helpful.

5 Conclusion

There was a huge difference in power consumption between running a loop and utilizing interrupts with deep sleep. The designed code went from consuming 3.6 mA in idle mode to 1.5 µA. And did it's purpose by light a LED during a button-press.

By trying different angles to solve the task(polling and interrupting) it's possible to see the connection between power usage and code. The less parts of the processor used, the less power usage. Deep sleep disables every function not needed and basically pauses everything while waiting for an interrupt.

The exercise were given to improve the students basic knowledge of a simple processor. There is still enormous amount of functions this basic introduction hasn't touched, yet it has improved our grasp on the processor by a great extent.

Bibliography

[1] ex1.s. source code for interrupt w/deep sleep.

[2] polling.s. source code for polling.

[3] Free Software Foundation. Gdb manual, 2006.

[4] Free Software Foundation. Gnu as manual, 2006. infonode: as.

[5] Free Software Foundation. Gnu ld manual, 2006. infonode: ld.

[6] Free Software Foundation. Gnu make manual, 2006. infonode: make.

[7] Silicon Labs. Cortex-m3 reference manual, 2011.

[8] Silicon Labs. Efm32gg reference manual, 2013.