NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4258 Energy Efficient Computer Design
Laboratory Report

Exercise 1 - ARM Assembly

Group 6:

Torbjørn Viem Ness
Marjeris Romero

February 9, 2015

Abstract

The goal for the exercise was to get familiar with the EFM32GG development kit, the ARM Thumb 2 instruction set and assembly programming and debugging. The task was to write an assembly program that would read input from an array of physical buttons on a gamepad connected to the development board (fig. 0.1), and use it do something interesting with the LEDs on the gamepad.
We chose to have each group of four buttons (up, down, left, right) control one light dot, which then could be turned on or off and moved right or left by its own button group. This was implemented with interrupts handlers in order to make the task energy efficient, and the MCU was put to deep sleep after processing each keypress.
This worked really well, and resulted in the MCU spending most of the time in deep sleep, using only ~2μA on average when no keys were pressed (LED current excluded).

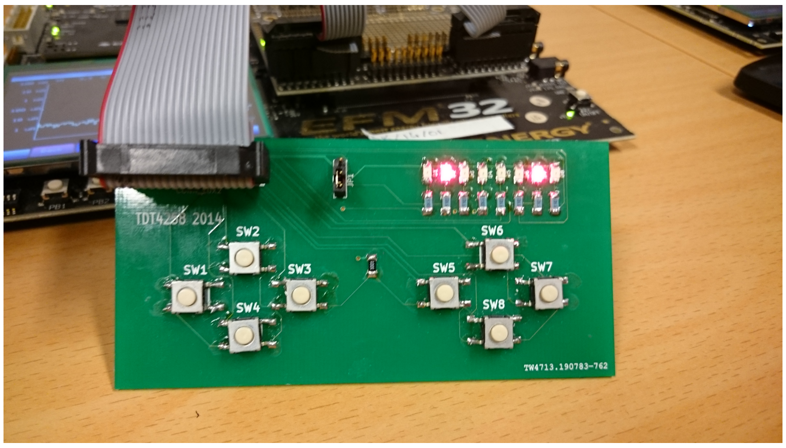

Figure 0.1: The gamepad with our application running

1 Introduction

Computer systems in all sizes and shapes are becoming more and more used in a lot of different areas, and there's no indication that this is a trend likely to turn anytime soon. In order to support this development, both hardware and software needs to be as energy efficient as possible. In addition to helping the environment, being energy efficient also gives longer lifetime for battery powered devices.

The goal of this lab exercise was to write an assembly program that takes input from the gamepad buttons, and uses it to control the row of LEDs on the gamepad. Part of the goal was also to understand the use of the GNU-toolchain, how to use the Makefile, linker, assembler and GNU Debugger.
The learning outcome of this exercise is to get knowledge of programming General Purpose I/O components, how to handle interrupts in assembly, and different considerations around how to write the most efficient code. For more details around the exercise please read the specifications in the course compendium[3].

The names of the constants and register mode bits that were used in this exercise are written in *EMPHASIZED UPPERCASE TEXT*, registers are written in **BOLD UPPERCASE TEXT** and more details about them can be found in the manual for the EFM32GG microcontroller[2].

2 Background and Theory

The EFM32GG is a 32-bit microcontroller with an ARM Cortex-M3 processor core and focus on low power applications. The Cortex-M3 is a 32 bit pipelined RISC processor that supports the ARM Thumb/Thumb-2 instruction set[1].

Among other things it has several GPIO (General Purpose Input/Output) pins which are memory-mapped. That means they can be configured by writing binary data to special memory locations, and input values can be read just like any other memory. This makes it relatively easy to write code that can interact with the outside world.

After a reset the pins to be used have to be initialized to the correct mode (output or input). The **GPIO_Px_MODEL/GPIO_Px_MODEH** registers are used to configure the behavior of the pins. **MODEL** controls pins 0-7, while **MODEH** controls pins 8-15. When a pin is configured as an output, the value of the **GPIO_Px_DOUT** register will set the pin high or low. Refer to the manual [2] for more information about the registers.

The EFM32GG-DK3750 is a development platform for the EFM32GG MCU family, with integrated debugging and trace capability. It includes a 3.5 inch TFT-LCD 320x240 pixel RGB color display with an energy monitoring tool to track the power consumption of your application.

2.1 Interrupts

A interrupt is a signal to the processor that needs immediate attention. This signal can come both from software or hardware. The processor should then stop from all activities and handle the interrupt, this is called a interrupt service routine or ISR. When the interrupt handler is done, the processor returns to its normal activities.

3 Methodology

3.1 Programming

Programming a microcontroller is actually mostly about writing the correct values into the different registers, in order to configure the various modules you need.
An example of how to program a register (using assembly code) is this:

```
ldr r4, =GPIO_PA_BASE     // 1: Load base address
mov r2, #0x2              // 2: Set the contents
str r2, [r4,#GPIO_CTRL]   // 3: Store the result
```

3.1.1 Reset handler

In order to make it work, the first thing that had to be done was to enable the modules that would be used later in the exercise. This is a one-time operation just to initialize the MCU and set the different variables as required, so it makes sense to do all this in the reset handler.
The reason this needs to be done, is that the MCU is optimized for low power, and therefore all unnecessary clocks and modules are powered down and disabled by default. Only the I/O controllers (and other peripherals) that are explicitly enabled are powered up and clocked.

First the clock management unit (CMU) was configured. The CMU can control different clock sources and enable/disable clock signals to peripherals. To enable clock for the GPIO controller, bit 13 in **CMU_HFPERCLKEN0** (High Frequency Peripheral clock) must be set to '1'.

The 8 LEDs on the gamepad are controlled by pins 8-15 of port A. This has to be set to output, using mode *PUSHPULLDRIVE*, push-pull output with drive strength set by *DRIVEMODE.* This was done by writing 0x555555555 to **GPIO_PA_MODEH**. Drive strength was then set to 20 mA writing 0x2 to **GPIO_PA_CTRL**.
The LEDs are active-low (*they light up when the voltage of the connected MCU pin goes to GND*) as seen in the schematics in the exercise description [3], so they were turned on by writing to '0' to **GPIO_DOUT** in the desired position.

The switches are connected to pins 0-7 of port C. This had to be set as input with internal pull-up, using the *INPUTPULLFILTER* mode in **GPIO_PC_MODEL**.

Interrupts had to be set separately at falling and rising edge for the switches. This was done by writing '1' to registers **GPIO_EXTIFALL** and **GPIO_EXTIRISE**, in the position of each pin to enable the interrupt for - in this case it was best to only trigger interrupts on falling edge (on keypress), so **GPIO_EXTIRISE** was not set. Then interrupts for pin 0-7 had to be enabled by writing 0xFF to **GPIO_IEN**, and finally interrupts had to be enabled globally by writing '1' to bit positions *1* and *11* of the **ISER** registry.

After initializing, the microcontroller was sent to deep sleep mode. This was done by writing the correct bits to register **SCR** (System Control Register) to select *DEEPSLEEP* as the low power mode, and set automatic sleep on return from interrupt handler, using the *SLEEPONEXIT* bit. Then, the instruction "wfi" can be used to make the MCU enter sleep mode and wait for an interrupt signal.

3.1.2 Interrupt handler

The interrupt handler is the code that executes when an event triggers an interrupt - e.g. when a key is pressed. In this project, the GPIO interrupt handler took care of everything related to keypresses; read the state of the buttons, determined which LEDs should be lit and finally updated the GPIO output register for the LED port.
To keep things simple and manageable, we decided to put most of the interrupt handler code in a separate file (*"fancy.s"*), and then jump to the top instruction there for all the register manipulation. After determining which LEDs should be turned on, the result was written back to the correct register and the LED port was updated with the new output values.

3.2 Testing

The first step of testing the implementation, was to assemble it and step through the program to see if everything behaved as intended. For this, the GNU debugger (gdb) was used to run single instructions and display the results. JLinkServer was used for communications between gdb and the development board, and the source file was viewed in *vim*. The code was assembled using the delivered Makefile and running command "make".

When testing, it was important so try all combinations of keypresses the program was likely to encounter, to see if it behaved as intended.
During the tests, the power consumption was also observed using the energy monitor on the development board - it was useful to see when the CPU went to sleep after execution, and a helpful tool for finding out where to optimize the code.

4 Results

4.1 Energy consumption

When no keys were pressed, the power consumption was around 2μA, and with one button pressed it went up to 83μA (+ $\sim 80\mu$A for each additional simultaneous button). That was probably current draw directly from the switches, and nothing we could do much about in software (apart from reducing the pull-up strength).

To further reduce power consumption, one could lower the supply voltage to the MCU - our development kit was able to keep running as low as 1.9V.
For example we found that a 1V supply voltage reduction (down to 2.3V from 3.3V) led to almost 40% reduced current.

4.1.1 CPU and sleep modes

While experimenting with different implementations we discovered that when it comes to power consumption - every instruction counts. Putting the CPU into sleep mode after executing the necessary instructions greatly reduced the current measured on the energy monitor of the development board. This part of the power consumption is possible to optimize, and can be improved quite a bit by spending extra time analyzing the code to find more efficient instructions and algorithms for a given task.
Also the energy consumption can be reduced by operating the MCU at a lower clock frequency when the task isn't very demanding.

4.1.2 LEDs

The LEDs were clearly the most power-hungry part of the design, but with a few clever tricks they too can be tamed to reduce the consumption. One of which is to multiplex, and only turn one LED on at a time, instead of all of them (then the maximum current draw will be limited to that of a single LED).
Another (and related, but perhaps more important and interesting) trick is to take it one step further and use PWM (Pulse-Width Modulation). Then by adjusting the period of time the LEDs stay on for each cycle one can greatly reduce the average power draw, with the added bonus of being able to dim the light. In many cases this can also be done entirely in hardware, so the CPU only needs to wake up when something needs to be changed. As long as the swithing is done fast enough, the eyes are unable to tell the difference

Both these methods require some extra hardware to be enabled, however on average the energy savings on the LED side are far greater than the cost of operating them.

5 Conclusion

We chose to have each group of four buttons control one light dot, which then could be turned on or off and moved right or left by its own button group. This was implemented with interrupts handlers in order to make the task energy efficient, with the MCU being put to deep sleep after processing each keypress.
This worked really well, and resulted in the MCU spending most of the time in deep sleep mode, using only $\sim 2\mu$A on average when no keys were pressed (LED current excluded).

Further improvements would include PWM for dimming (and thereby reducing consumption from the LEDs), but the time and hardware features of the gamepad board were a bit limiting.
There were probably other modules in the MCU that could have been turned off for extra energy savings, and the clock frequency could have been reduced significantly while still keeping sufficient performance to complete the task.

5.1 Evaluation of the Assignment

Overall the assignment was interesting, however it wasn't always easy to find descriptions and good documentation about how to do different things in assembly - for example, we got a tip about using the equal sign operator (=CONSTANT) when loading addresses and constants into registers, but this wasn't mentioned anywhere in the available documentation.

Bibliography

[1] *Cortex-M3 Reference Manual*. Energy Micro, 2011.

[2] *EFM32GG Reference Manual*. Silicon Labs, 2013.

[3] *Lab Exercises in TDT4258 Energy Efficient Computer Systems*. Department of Computer and Information Science, NTNU, 2014.