TDT4258 - Exercise 1

Magnus Halvorsen Julian Lam

February 10, 2014

Abstract

As the world today becomes more and more digitialized and depended on computers for efficiency, embededded systems have become increasingly popular. One of the main reasons for this is the microcontroller, a small computer on a single integrated curcuit. Since microcontrollers are flexible and programmable, you can use the same microcontroller in several different embedded systems. Thus reducing project and product cost, since you do not have to spend time designing a special purpose processor.

But due to the vast increase of computer usage and the use of computers in extreme environments, power consumption and longtivity has become a major challenge within embedded computer design.

This report covers the first exercise in the course TDT4258 Energy Efficient Computer Systems at NTNU. The main purpose of this exercise is to get an introduction and general understanding of the architecture of *the ARM Cortex-M3 microcontroller* and *the EFM32GG DK3750 development board*. Thus this report will guide you through our process of making a simple program that uses the microcontroller to turn on LEDs when a button is pressed. We will also present the design choices made in order to use as little power as possible.

Contents

List of Figures

1 Introduction

The components used in this assignement is the *EFM32GG Giant Gecko microcontroller* combined with the *ARM Cortex-M3 processor*, connected to a *gamepad* containing eight LEDs and buttons. The programming is done in assembly and can be uploaded to the board by usb and a program called *eA-Commander*. The processor supports the ARM Thumb-2 instruction set, so all assembly code must be written in it.

The microcontroller is connected to the gamepad with a ribbon cable split into two parts, one for the buttons and one for the LEDs. The parts are connected to two different I/O ports on the microcontroller, port C and A, respectively. The buttons are connected to pin 0-7 on port C and the LEDs are connected to pin 8-15 on port A.

Both the micocontroller and the processor are designed to be highly energy efficient, and thus support five different energy modes. Energy mode 0 provides full functionality, while energy mode 4 provides little functionality but uses a lot less power. Thus, in order to save power, we wish to keep the microcontroller in as high energy mode as possible at all times.

We were provided with a framework which contained two assembly files, one containing useful constants and one containing enough assembly code to set up the microcontroller. The framework also contained a makefile which could be used to assemble, link and upload the code to the microcontroller. We were also provided with a compendium[3], containing useful information about the system, and reference guides[2][1].

1.1 Assignment

The assignment is as follows:

- Write an assembly program which enables a user to control the LEDs in some way by pressing the buttons.
- Write an interrupt rotuine (an interrupt handler) for reading the buttons.
- Use a Makefile to compile and upload the program to the microcontroller.
- Use the energy monitor to see the power requirements of your program. Analyze and discuss (or implement) improvements.

The solution provided in this report have completed all of the listed conditions, and have also added some extra functionality. In addition to simply using the buttons to control the LEDs, they are also manipulated by interrupts from a countdown timer. This extra functionality will be described in more detail in the following sections.

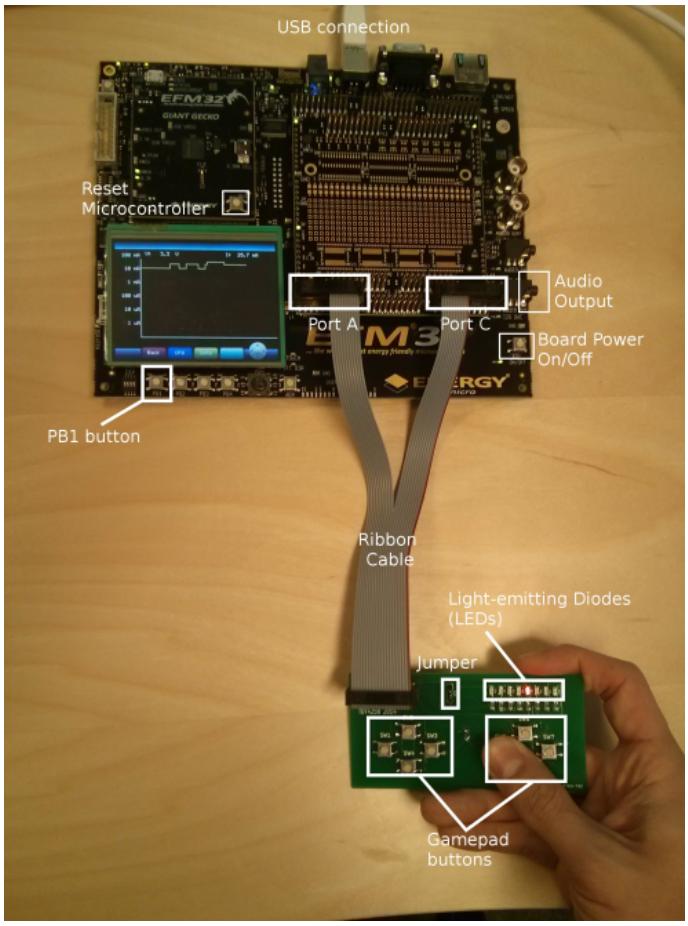

Figure 1: The EFM32GG microcontroller and gamepad[3]

2 Description and Methodology

This section covers how the assignment was solved. This was done in the following steps:

- Upload provided test program to the board.
- Turn on a LED upon startup.
- Enable simple interrupts.
- Read input from the buttons upon a interrupt, and set LEDs accordingly.
- Manipulate the input for more advanced LED patterns.
- Manipulate the input by using a countdown timer.
- Reduce the power consumed.

2.1 Upload provided test program to the board

The first thing we had to do in order to familiarize ourself with the system was by simply loading a provided .bit file to the board. This was done by using the program eACommander to flash the board, which provided a simple LED manipulation program.

Afterwards we uploaded the provided framework to the board. The provided makefile contained commands to assemble, link and upload the program. In order to make development faster and easier, we added an *make all* command to the makefile, which executed all the mentioned commands.

2.2 Turn on a LED upon startup

Now as we had managed to upload our program to the microcontroller, we figured the best way to see if we understood the system was to make the LEDs light up upon start up.

This was really straight forward, since we were provided with a guide in the compendium:

- Enable the general-purpose input/output (GPIO) clock, which was done by writing to the high frequency peripheral clock register (HFPERCLKEN0) in the clock manangement unit (CMU).
- Set high drive strength to port A, which controls the LEDs, by writing to the GPIO_PA_CTRL register.
- Enable pin 8-15 in port A for output, by writing to the GPIO_PA_MODEH register.
- Activate the LEDs by setting the register GPIO_PA_DOUT to low.

2.3 Enable simple interrupts

Now as we got the LEDs to light up we decided to manipulate the LEDs by using the buttons, thus we had to enable interrupts. Again the compendium provided us with an direct guide:

- Setting the input port for external interrupts to port C, by writing 0x22222222 to the GPIO_EXTIPSELL register.
- Enable interrupts to happen when input pin 0-7 goes low (i.e. when a button is pressed down), by writing 0xff to GPIO_EXITFALL.
- Enable interrupt generation on pin 0-7, by writing 0xff to GPIO_IEN.
- Enable the handlers in the exception vector table by writing 0x802 to the ISER0 registry.
- Clear the interrupt by writing 0xff to GPIO_IFC in the interrupt handler, so the interrupt handler will not be repeatedly called.

We then moved the code which turned on the LEDs to the gpio_handler, and saw that they lightened up upon a button press.

2.4 Read input from the buttons upon a interrupt, and set LEDs accordingly

With the interrupt handler working correctly, we now wanted to extend the handler to lighten up LEDs when their respective button was pressed. This was simply done by reading from port C's GPIO_DIN register (from now on we will denote port X's GPIO with GPIO_PX). Since the buttons are connected to pin 0-7, and the LEDs are connected to pin 8-15, we had to left shift the value in GPIO_DIN by eight bits. After that we simply wrote the result to GPIO_PA_DOUT register, setting the respective LED.

2.5 Manipulate the input for more advanced LED patterns

Since the program now provided the minimum needed features to satisfiy the assignment, we now wanted to manipulate the input byte to more a more complex pattern. Thus, when you pushed a button the respective LED should lighten up, and the complementary LED. E.g. if you pushed button one, LED one and eight should lighten up, and if you pushed button five, LED four and five to lighten up.

To do this we added a function called reverse_byte. This function takes one 32-bit value as input, and reverses the first 8 bits. E.g. if the first 8 bits are 11010000, the function outputs 00001011.

The value from the GPIO_PC_DIN register is used as input to the function, and then the output value and the original value are logically and'ed. The result is then left shifted eight bits, and written to GPIO_PA_DOUT.

The algorithm in reverse_byte took a bit more effort to develop, and the GNU debugger (GDB) was used extensively to read the registers and see how the different instructions manipulated the bits. The algorithm works as follows:

1. Put the original value into r1.
2. Iterate through step 3-6 eight times.
3. Rotate right the original value one bit into r2.
4. Right shift the orginal value.
5. Right shift r2 (32 - number of iterations) bits.
6. Add the value to r3.
7. return r3

2.6 Manipulate the input by using a countdown timer

Another extension we wished to add to the program was to control the LEDs with a timer. We decided to program it so that when you pushed a button the respective LED and its complement LED should light up, and then for every timer interval the active LEDs should be shifted one to the right - until it reached the end of the line of LEDs.

In order to do this we had to use one of the built in timers on the microcontroller. We decided to use the Low Energy Timer (LETIMER0), in order to use as little power as possible. It took a lot of work to get the timer working, and much of the effort was spent reading and understanding the EF32GG reference manual. GDB was also used extensively to confirm and deconfirm our understanding of how the timer worked by reading various register values.

The program was implemented by enabling the timer in the gpio_handler every time a button is pressed and saving the LEDs status in a global register. So every time the counter reached zero, we shifted the register containing the LEDs status one bit to the left and writing it to the GPIO_PA_DOUT register. When the timer reached zero after the eight time we stoppped it.

In order to use the timer we had to do the following steps:

- Enable the Low Energy clock in the CMU_HFCORECLKEN0 register.
- Enable LETIMER0 clock in the CMU_LFACLKEN0 register.
- Set the ULFRCO as the clock source for LETIMER0, by writing to the CMU_LFCLKSEL register.
- Set COMP0 as top value (the value the timer starts at) and enable buffered mode in the LETIMER0_CTRL register.
- Write 0x1 to LETIMER0_REP0 and LETIMER0_REP1.
- Enable interrupt for when REP0 reaches zero.
- Set the top value by writing it to LETIMER0_COMP0.
- Start the timer when a GPIO interrupt occurs.

With this setup the timer is reset to the top value every time it reaches zero, as long LETIMER0_REP1 is written to in each interval. Thus in the LETIMER0 interrupt handler will have to check a global counter to see if the timer have counted down eight times. We used a global register for this.

2.7 Reduce the power consumed

We used three main strategies to reduce the power consumed:

- Use interrupts, instead of letting the processor constantly check which buttons are pressed.
- Using the Low Energy Timer as timer, and connecting it to the ULFRCO clock, which uses very little energy.
- When the processor is done handling an interrupt, it executes the Wait For Interrupt (WFI) instruction. Putting it in sleep mode until an interrupt occurs.

3 Results

This section covers the testing of the program, the resulting functionality of the program and its power consumption.

3.1 Result

3.1.1 Functionality

The final version of the program works as follows:

- When you press a button(s), the respective LED and its complement will light up.
- The timer is enabled upon a GPIO interrupt, and will count down to zero eight times.
- Every time the timer reaches zero, the lighted LEDs are shifted one LED to the right.

3.1.2 Power consumption

Without any power optimizations the program used 4.3 mA. By putting the processor to sleep by using the WFI instruction this number was lowered to 1.5 mA. Finally, by setting the LETIMER0 clock source to ULFRCO we managed to lower the energy usage to 1.4 mA.

3.2 Testing

The testing of the program was either done manually (pressing buttons and observe the result) or with GDB. The general method was testing manually to see if the program did what we intended. If this failed, and there were no obvious reasons for this, we debugged with GDB by reading the values of critical registers.

Power consumption was tested by reading the energy usage of the board, using the built-in energy meter.

4 Feedback

We found the assignment very exciting and learned a lot from it. The compendium was very informative and, supported by the student assistent, helped a lot in getting a fundamental understanding of how the system worked. The only complaint we have is that in order to do fulfill the minimum requirement of the assignement, you do not really have to use the reference manual at all. So a possible improvement would be to try to involve the reference manual a bit more directly, since you learn a lot by understanding how to use it.

5 Conclusion

All of the requirements given by the assignment has been fulfilled. The described program allows you to manipulate the LEDs by pressing the buttons on the gamepad. We have also extended the program with extra functionality, which included a more complex LED pattern and changing the pattern on invervals using a countdown timer. Interrupts are activated for both GPIO events and timer events, thus allows the processor to do other operations instead of constantly polling the GPIO and timer. We have also designed the program to be as power efficient as possible, by letting the processor sleep while waiting for interrupts and using the Low Energy Timer for countdown timing.

An improvement that could be made to the program, is removing the use of global values. When keeping track of the number of timer interrupts we save it in register 7, and hope that it will not be used somewhere else in the program. While this works in the programs current state, it is not good design if the program is to be extended with more functionality.

But we are quite satisfied with the resulting program. Making it have given us a lot of insight on the EF32GG architecture, given us a basis for low-level coding and reasoning, and how to read reference manuals for computer components.

References

[1] EFM32GG Reference Manual
http://cdn.energymicro.com/dl/devices/pdf/d0053_efm32gg_reference_manual.pdf.

[2] Cortex-M3 Reference Manual
http://www.silabs.com/Support%20Documents/TechnicalDocs/EFM32-Cortex-M3-RM.pdf

[3] TDT4258 - Compendium
Provided on itslearning