NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4258 Energy Efficient Computer Design
Laboratory Report

Energy efficient LED controller

Peter Aaser
Audun Sutterud
Jonatan Lund

November 11, 2015

Abstract

Embedded computing systems are being deployed on an increasingly larger scale, and it is expected that this trend will continue in the forseable future. It follows that there is a great market for developing software for embedded computing systems. In contrast to general-purpose computing, embedded computing often places strong constraints on energy consumption, performance, and cost. In order to implement these qualities in a system, software developers must make extensive use of low-level hardware functionality, and specific knowledge of the hardware components and technology used is required.

The course TDT4258 Energy Efficient Computer Systems gives an introduction to microcontroller programming on a practical and theoretical level, with a strong focus on energy-efficiency. Through lectures and three comprehensive exercises the students get theoretical knowledge as well as hands-on experience. The development platform used is the EFM32GG-DK3750 development kit from Silicon Labs, and the tools used include energyAware Commander and the GNU Compiler Collection. In the first exercise, the students connect the prototyping board to a specially built gamepad with LEDs and buttons, and write a small program that lets the user to control the gamepad LEDs through the buttons.

During the exercise we learned to connect the prototyping board to development software, load binary programs, and monitor program energy consumption. On the development end we learned about the GNU Compiler Collection, how to debug running code using the GNU Debugger with GDBServer, and how to program the Cortex-M3 microprocessor using the Thumb-2 instruction set. The exercise program was implemented in several steps, first using a simple polling loop, then a more sophisticated technique with interrupts, and finally with a simple countdown clock. Through this process we observed a great improvement in energy-efficiency.

Contents

1 Introduction

Embedded computing systems are being deployed on an increasingly larger scale, and it is expected that this trend will continue in the forseable future. It follows that there is a great market for developing software for embedded computing systems. In contrast to general-purpose computing, embedded computing often places strong constraints on energy consumption, performance, and cost. In order to implement these qualities in a system, software developers must make extensive use of low-level hardware functionality, and specific knowledge of the hardware components and technology used is required.

The course TDT4258 Energy Efficient Computer Systems gives an introduction to microcontroller programming on a practical and theoretical level, with a strong focus on energy-efficiency. Through lectures and three comprehensive exercises the students get theoretical knowledge as well as hands-on experience. The development platform used is the EFM32GG-DK3750 development kit from Silicon Labs, and the tools used include energyAware Commander and the GNU Compiler Collection.

In the first exercise we will connect the prototyping board to a specially built gamepad with LEDs and buttons, and write a small program that lets the user to control the gamepad LEDs through the buttons. The purpose of the exercise is to introduce the development kit hardware, to become familiar with writing low-level energy-efficient software, and to learn software development using the GNU-toolchain. All code will be written in assembly using the Thumb-2 instruction set, and no operating system will be running. An important exercise goal is that the program should be energy-efficient, and thus we need to utilize low-level hardware functions to improve the power consumption. To learn different techniques to improve the power consumption, we are encouraged to read the relevant reference manuals and data sheets.

2 Background and Theory

This section gives a brief introduction to the EFM32GG-DK3750 and various hardware components that were involved in the exercise. In addition, it will also introduce some concepts and underlying theory.

2.1 The EFM32GG-DK3750

The EFM32 Giant Gecko Development Kit (EFM32GG-DK3750) is a development platform for prototyping and development of applications. The attached microcontroller unit (MCU) is an ARM Cortex-M3 CPU core, built for demanding environments where cost, power consumption and performance are central issues. All the peripherals provided with the EFM32GG-DK3750 are built with focus energy efficiency. The board also features an on-board debugger, making it easy to both download programs and even debug them at run-time.

2.2 The Cortex-M3

The MCU of the EFM32GG-DK3750 is an ARM Cortex-M3. The Cortex-M3 is a 32-bit RISC processor combining high performance and excellent response time with low cost and power consumption. Among its notable features is a Harvard Architecture, the Thumb-2 instruction set, and efficient 32-bit multiplication and division.

2.2.1 Nested Vectored Interrupt Controller (NVIC)

The Cortex-M3 comes with an NVIC, which is the hardware that handles interrupts. Interrupts are events that originate on peripheral devices or special software instructions, causing the normal flow of a program to change and some special code to be executed. This special code is referred to as an *interrupt handler*, and performs some necessary action depending on the interrupt. The NVIC organizes several interrupt handlers in a table of addresses called the Interrupt Vector Table (IVT). When an interrupt is registered, the execution will jump to the handler specified by the corresponding entry in the vector table, as shown in figure 8.4. Upon a reset, the processor will start executing the reset handler.

2.2.2 Processor mode and privilege levels

The Cortex-M3 can run software at two privilege levels: *privileged* and *unprivileged*. The privilege levels are used by the processor hardware to restrict running software to certain

resources only. This is necessary in order to provide memory protection, for instance. In privileged mode, running software has access to all instructions and resources, while in unprivileged mode access is restricted according to some board configuration registers.

The software privilege level depends on whether the processor is running in *thread mode* or *handler mode*. In thread mode the software runs at either privileged or unprivileged level depending on the CONTROL register, while in handler mode the software always runs at the privileged level. The processor will switch from thread mode to handler mode on any exception or interrupt. This ensures that the interrupt handlers run in privileged mode with access to all instructions and memory. In an operating system, kernel code would typically run in handler mode and user space programs would run in thread mode.[4]

2.2.3 Registers

The Cortex-M3 makes several registers visible to the programmer. These include the general-purpose registers R0 through R12, the Stack Pointer (SP/R13), the Link Register (LR/R14), the Program Counter (PC/R15), and the Program Status Register (PSR).

General-purpose registers

ARM Cortex-M3 contains thirteen 32-bit general purpose registers, labelled R0-R12. When an exception occurs the CPU automatically pushes the registers R0-R3 to the stack. Registers R4-R12 need to be handled manually.

Stack Pointer (SP)

Register R13 is reserved for the stack pointer. The stack pointer is used to point to the last stacked item in the stack memory. The processor contains two stacks: the *main stack* and the *process stack*. In thread mode the control register's second bit indicates which stack pointer to use.

- 0 = Main Stack Pointer
- 1 = Process Stack pointer

Link Register (LR)

LR is register R14 and is used to store information about subroutines, function calls and exceptions.

Program Counter (PC)

PC is register R15. PC holds address for the current program. Since instructions has to be halfword aligned the first bit is always 0. When the processor is reset, the PC loads the default value from the reset vector.

Program Status Register (PSR)

The PSR is used to stored states after an instruction. For instance, if the add instruction produces an overflow this will be indicated by bit[28]. PSR's bit assignments can bee seen in figure 8.1.

2.2.4 Instruction set

The Cortex-M3 implements the Thumb-2 instruction set, a 32-bit instruction set commonly used for ARM processors. It is an extension of the older 16-bit Thumb instruction set, and contains a mix of both 16-bit and 32-bit instructions. The additional 32-bit instructions allow it to cover the functionality of newer ARM microprocessors.[1]

2.3 On-board components and peripherals

Through the exercises in this course we will work with the Silicon labs EFM32GG-DK3750 prototyping board 8.3. There are several on-board components and peripherals of special interest, including the General-Purpose Input/Output pins (GPIO pins) controlling external peripheral devices, the gamepad prototype, the Clock Management Unit (CMU), and the Energy Management Unit (EMU). This section will give a brief introduction to those devices. For a map of the full set of peripherals, see figure 8.2.

2.3.1 General-Purpose Input/Output pins (GPIO pins)

The GPIO pins is what makes it possible to connect external peripherals to the EFM32GG prototyping board. The pins are organized into ports of 16 pins each. It is possible to configure each pin individually for either input or output, in addition to configuring more advanced features such as drive-strength or pull-up resistors.

2.3.2 The Gamepad

The gamepad peripheral is connected to the GPIO pins on port A and C using a Y-shaped ribbon cable. It has eight buttons and eight LEDs connecting the pins to ground, making it possible to provide both input and output. In addition, the gamepad also has a jumper which allows us to toggle whether the amperage consumed by the LEDs will be measured.

2.3.3 Clock Management Unit (CMU)

The CMU controls on-board oscillators and clocks. These components consume a significant amount of power, as a clock signal needs to be generated for all active on-board peripherals. The CMU is highly configurable, allowing the clocks to be turned on and off on an individual basis. By default, the clocks for all components are turned off and need to be manually enabled for the components that will be used.[3]

2.3.4 Energy Management Unit (EMU)

The EMU manages the different low energy modes in the EFM32GG. Being a MCU with focus on energy efficiency, the Cortex-M3 has five distinct energy modes, from EM0 (run mode) where the CPU and all peripherals are active, to EM4 where the CPU and most peripherals are disabled. What components are active on different modes is detailed in figure 8.2. In addition to handling the energy modes, the EMU can be used to turn off the power to unused SRAM blocks.[3]

2.3.5 DMA Controller

The Direct Memory Access (DMA) controller is a hardware component that can transfer data between memory and I/O devices, without direct interaction from the CPU. An alternative to using a DMA controller for I/O is to do programmed input/output, where the CPU either uses interrupts or busy-wait to continually service an I/O device. DMA usually benefits over this approach as it allows the CPU to perform other work, or to enter low energy modes.

2.3.6 Peripheral Reflex System (PRS)

The PRS is a network connecting the different peripheral modules together, allowing them to communicate directly without involving the CPU. The peripheral modules send each other *reflex signals* routed by the PRS. On receiving such a signal, a peripheral module may perform some specific action depending on the signal received. By relieving the CPU of work, the PRS system can be used to improve energy efficiency. It is also suited for time-critical operations as it involves no variable-time software overhead.

2.4 Energy consumption

When reasoning about the expenditure of a device it is useful to break down energy consumption into static and dynamic consumption. Dynamic consumption is the power consumed as a function of the clock rate. A processor running at a higher clock rate will in general expend more energy. The static consumption is the energy a device consumes regardless of clock frequency, for instance the small current that keeps SRAM from flipping over when not in use.

In this task the main challenge is to reduce both static and dynamic power consumption to a bare minimum while still providing the necessary services for the buttons and LEDs to work.

2.5 Memory

2.5.1 Memory-mapped I/O

The EFM32GG uses memory-mapped I/O. With memory-mapped I/O, a range of memory addresses is mapped onto device registers. This means that the registers can be read

and written using ordinary load/store instructions. All configuration of the EFM32GG is done by writing to memory-mapped registers.[3]

In order to control the gamepad and the board components, we use memory-mapped I/O to read and write different registers. For instance the GPIO pins are configured with the addresses starting at 0x40006000, and the CMU is configured with the addresses starting at 0x400c8000.

2.5.2 Register addressing

We specify registers with their base address followed by their offsets. The base address marks the start of a memory segment dedicated to a certain functionality. In the text we will refer directly to the memory address of a functionality using its symbolic name as specified in the reference manual. For instance, EMU_MEM_CTRL refers to the 32 bits that control memory handling in various energy modes. However in the code we will have to compute the memory addresses by adding an offset to the base address of a functionality.

2.6 Tools

2.6.1 GNU tool chain

The GNU tool chain is a collection of programming tools used for software development. The GNU tool chain is produced by the GNU project and is free software. The components of the GNU tool chain used in this course are make (build-automation tool), ld (linker), as (assembler), and the GNU debugger.[2]

3 Methodology

In this chapter we give an informal description of our working process, with links to a more detailed description in the appendix.

3.1 Performance measurement

Since energy efficiency was the focal point of the exercise measuring energy expenditure from our programs was necessary. For simple measurements we could utilize the LCD screen on the development board shown in figure 8.3, however for a more useful analysis we utilized the eAProfiler program that was installed on the lab computers. For our analysis we used the jumper on the gamepad to stop energy consumption from the LEDs effecting the energy consumption readout.

On very low amperage, around 1 micro-ampere the signal was affected by noise, enough to make readouts unreliable, but not enough to hide efficiency gains at the micro ampere level.

In order to reduce randomness all measurements were taken after running the EFM32GG for several hours during development in rapid succession to avoid temperature differences skewing the results.

3.1.1 A simple program

No group members had any prior experience with programming microcontrollers, so we decided on an incremental approach, focusing first on the basics, getting a simple program to run. The bare essentials for a program is simply a reset handler, a small piece of code that will be executed as soon as a reset is triggered. To do this we simply added the address for our code at the reset location specified in the NVIC vector diagram. Making the program to compile and run was also a good test run of the tool chain, making sure we had all the required pieces to get code from our development PC to runnable code on the MCU.

3.1.2 A simple LED write

Our basic program did not do anything more than starting up, so in order to make sure we had code that was running as it was supposed to, and to do the first step on mapping buttons to LEDs we started to write code to activate the LEDs.

Our first step was to activate the peripheral clock for GPIO by setting bit 13 of the HFPERCLKEN0, which we did by loading it with a value of 1 left shifted 13 times.

In order to activate the LEDs we had to set drive strength to high and set pins 8-15 of port A to output. With the LEDs activated we could control them by writing to GPIO_PA_DOUT. Being active low, simply writing 0x0 to GPIO_PA_DOUT caused every light to light up.

3.1.3 Activating the buttons

In order to do more serious testing with the LEDs, we activated the buttons on the gamepad. First we had to activate pins 0 to 7 of port C, and then enable internal pin-up. Now we could read the bits from GPIO_PC_DIN to determine which buttons were pressed.

3.1.4 A simple polling loop

While the goal was to write an energy efficient program to handle button inputs we started off with the simplest approach, building a solution iteratively. In addition, implementing the program in the most naive way we could think off would serve as a very useful baseline to gauge effectiveness of the improvements we would do to efficiency in later iterations. Our first working program was therefore a simple loop continuously copying the contents of the button register and writing it to the LED register. This gave us the adequate functionality, but the processor did a lot of wasteful reads and writes giving a rather lacklustre efficiency performance.

3.2 Interrupts

In order to stop the processor from wastefully reading or writing every cycle we started implementing an interrupt based approach. In order to generate interrupts for the NVIC to handle we first had to set port C as interrupt source by writing to GPIO_EXTIPSELL, then we had to set interrupt generation for pushing and releasing by writing to GPIO_EXTIPFALL and GPIO_EXTIPRISE. Finally after setting up how GPIO interrupts would be done we activated interrupt generation by writing to GPIO_IEN.

With interrupts enabled, whenever an interrupt is triggered the NVIC will cause the processor to jump to the GPIO handler specified by the address in the IVT, as mentioned in chapter two.

3.2.1 Handling interrupts

Our interrupt handler is set up to do two things when called. First it reads which button has flagged the interrupt from GPIO_IF and writes it to the LED register, then it writes to GPIO_IFC to clear the interrupt. Still, simply implementing interrupts did not make the program more efficient as it was still looping waiting for interrupts.

3.2.2 Efficient interrupts

In order to achieve efficiency we utilized the energy modes of the EFM32GG, turning off unnecessary processor use when waiting for a button to be pushed. In order to be able to react to input even when the processor is turned off, the EFM32GG relies on interrupts which restore the EFM32GG to EM0. This meant we could put the processor to deep sleep, only running after a button push to change the LED. In order to put the processor back to sleep after handling an interrupt we simply set the processor to sleep on exiting interrupt handler mode by writing to SCR.

3.3 Further improvements

3.3.1 Powering down RAM

Having achieved efficient interrupt handling we consulted the energy optimization application note which suggested powering down unused SRAM. Since our program was very basic we turned off three out of the four 32kb blocks, the lowest amount possible by writing to EMU_MEM_CTRL. We also had to edit our linker script to reflect the reduced amount of available RAM, without doing this the program did not run at all.

3.3.2 Entering energy mode 3

The reference manual describes the difference between EM2 and EM3 as having the two low frequency peripheral clocks LFACLK and LFBCLK on or off, so in order to reduce energy consumption further we wrote to CMU_LFCLKSEL.

3.4 Implementing a clock

Having gotten a fairly efficient button to LED mapping we next implemented a simple clock. In order for a clock to work in EM3 the only time available was the LETIMER which could run from the ultra low frequency crystal oscillator (ULFRCO). While the LETIMER offers some advanced functionality we settled on simply counting down from a set value and triggering an interrupt on overflow, and making the interrupt handler decrement the LED bits once.

After setting the LETIMER to generate interrupts we also had to enable interrupt handling for LETIMER interrupts by writing to ISER0, like we did with the GPIO interrupts.

Setting the count down value very low and making the clock tick very fast also gave us a nice source of steady interrupts we could use to gauge energy consumption.

In order to test how interrupts impacted performance we set the countdown value to one, generating an interrupt for every LFRCO oscillation. Next we tried duplicating the instructions for the handler, trying with the normal execution, then adding redundant executions, trying three, five and seven executions, as well as doing nothing but clearing

the interrupt flag. This would help us gauge how code optimization could effect efficiency for frequent interrupts.

4 Results

4.1 Final program

Our final program works as LED a toggler combined with a clock. Each button is mapped to a single LED. When a button is pushed the corresponding LED is toggled. In addition a clock steadily decrements the LED value.

4.2 Measurements

4.2.1 Polling

With our polling implementation as a baseline we measured about 4.9mA, regardless of button input. As expected, the polling method consumes a lot of energy since it constantly runs instructions whether they are needed or not. The result are shown in figure 4.1.

4.2.2 Interrupt

Our wake on interrupt based program fared much better, averaging around 1.5μA, and expending about 83μA when a button was held down. This is shown in figure 4.2.

4.2.3 SRAM blocks

In search of further improvement we powered down idle SRAM blocks to save more power, resulting in around 1μA, a pretty substantial improvement from the simple interrupt based program. However, with a button held down the difference was negligible as shown in figure 4.3.

4.2.4 Energy mode 3

Turning off LFBCLK and setting LFACLK to use the ULFRCO set the device to energy mode 3, but we could measure no performance difference at all.

4.2.5 Interrupt efficiency

When duplicating the instruction for the handler, making it do redundant actions we received the readings in table 4.1.

Handler executions	cycles per interrupt	μA
1	16	2
1	1	8.5
3	1	14.2
5	1	20.5
7	1	26.6

Table 4.1: Energy cost with different handler executions

We see a strong correlation between interrupt handler instruction size and efficiency. The amount of energy spent on executing instructions is much larger than the energy spent generating interrupts and handling them.

We would get better data if we could make an empty handler, but then there would be no way to clear the flags, making the processor unable to exit handler mode.

4.2.6 Noise

In figure 4.2 and 4.3 a lot of noise can be seen when the board is in deep sleep. We could not find any specification about the measurement range and precision. It is unclear if the noise was caused by to low precision on the sensor, or if the other components on the board affects the sensor.

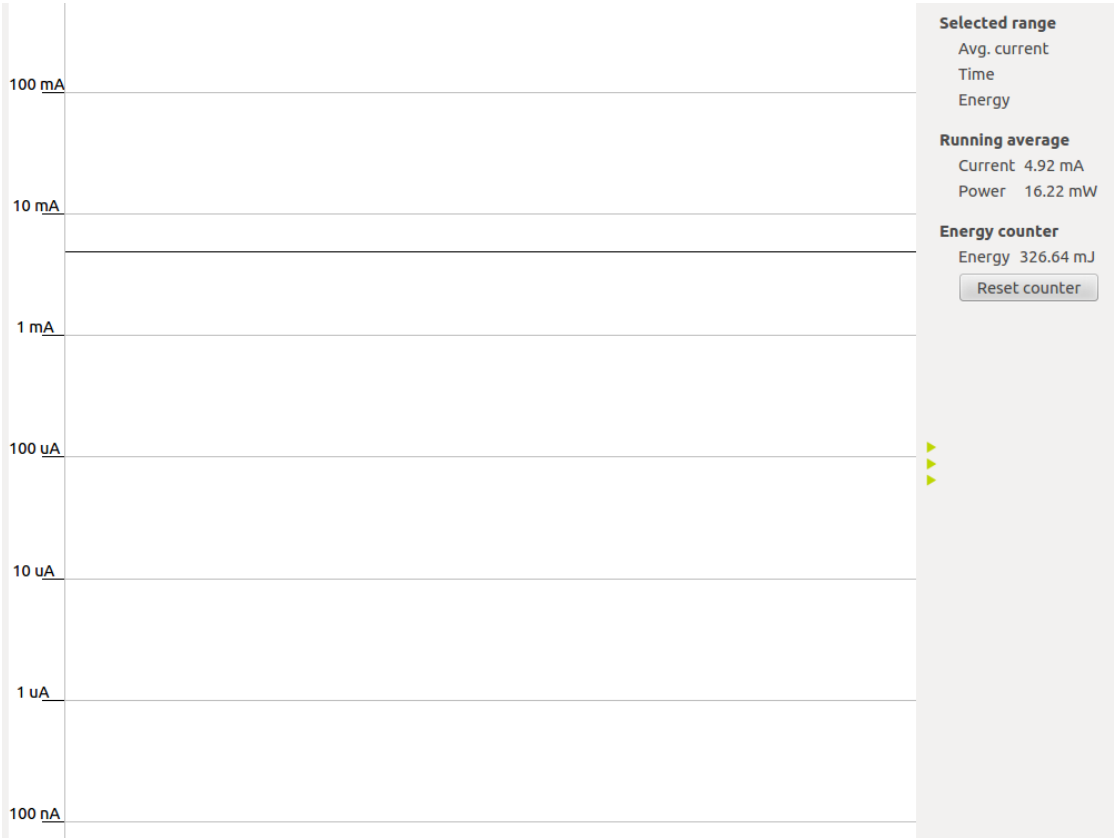

Figure 4.1: Energy consumption with polling

Figure 4.2: Energy consumption with interrupts

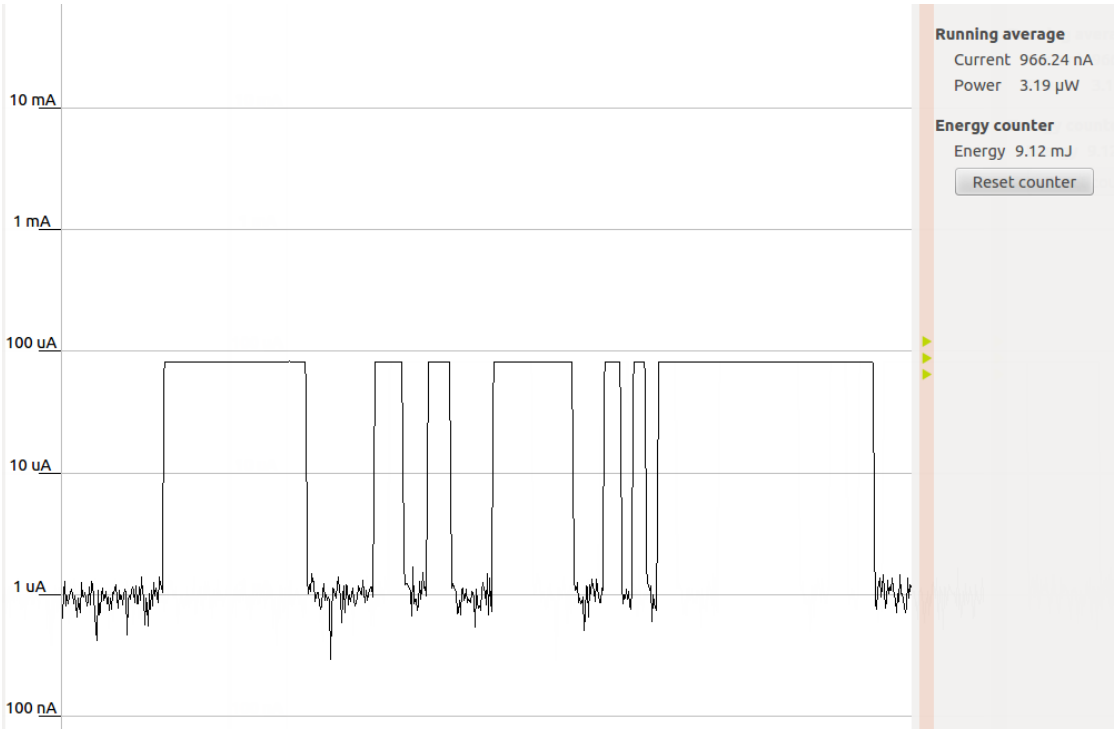

Figure 4.3: Energy consumption with interrupts, and SRAM blocks powered off

5 Discussion

We believe we have presented a pretty decent solution to handling LED and button mapping. Our program runs very efficiently, and while improvements are probably still possible the device uses so little power that the power source will probably fail before the energy is drained.

We tried to implement DMA to read from memory and display on the LEDs using the LEUART and PRS, but in the end it proved too challenging since none of us had any prior experience.

We also considered using the PRS to map buttons to LEDs, but we do not know whether this would have been possible, and if it would have saved power.

6 Conclusion

In this exercise we have gone from knowing almost nothing about microcontrollers and energy efficiency to implementing an energy efficient, although fairly primitive clock.

We have shown how energy consumption of a device can vary a great deal based on how it is implemented, by several orders of magnitude. We have also gained a much deeper understanding about how the processor works very close to the metal, and how it interfaces with various peripherals.

6.1 Evaluation of the Assignment

This was a very useful exercise for us, and on the technical side everything worked out smoothly except for eACommander from time to time being unable to halt the processor to flash memory.

7 Appendix A

7.1 A simple program

7.1.1 Register aliasing

In order to make writing code simple we used some more mnemonic aliases for the registers, making it simpler to access registers.

```
    CMU .req R4
    GPIO_LED .req R5
    GPIO_btn .req R6
    GPIO .req R7
    EMU .req R8

    ldr GPIO_LED, =GPIO_PA_BASE
    ldr GPIO_btn, =GPIO_PC_BASE
    ldr CMU, =CMU_BASE
    ldr EMU, =EMU_BASE
    ldr GPIO, =GPIO_BASE
```

7.1.2 enabling peripheral clock

To enable the clock we write to bit 13 of HFPERCLKEN0 by left shifting 1 by 13 and storing it.

```
    mov r2, #1
    lsl r2, r2, #CMU_HFPERCLKEN0_GPIO
    orr r1, r1, r2
    str r1, [CMU, #CMU_HFPERCLKEN0]
```

7.1.3 enabling LEDs

To enable LEDs we write 0x55555555 to GPIO_PA_MODEH

```
    mov r1, #0x55
    orr r1, r1, r1, lsl #8
    orr r1, r1, r1, lsl #16
    str r1, [GPIO_LED, #GPIO_MODEH]
```

We set drive strength to high by writing the second bit of GPIO_PA_BASE to 1

```
  mov r1, #0x2
  str r1, [GPIO_LED, #GPIO_CTRL]
```

7.1.4 enabing buttons

To set pins 0-7 as input we write 0x33333333 to GPIO_PC_MODEL

```
mov r1, #0x33
orr r1, r1, r1, lsl #8
orr r1, r1, r1, lsl #16
str r1, [GPIO_btn, #GPIO_MODEL]
```

Enabling button pull-up

```
mov r1, #0xff
str r1, [GPIO_btn, #GPIO_DOUT]
```

The polling loop continuously reads which buttons are pressed from GPIO_PC_ and writes them to

7.2 Interrupt based implementation

Setting port C as interrupt source by writing 0x22222222 to GPIO_EXTIPSELL

```
  ldr r1, =0x22222222
  str r1, [GPIO, #GPIO_EXTIPSELL]
```

Set 1-¿0 and 0-¿1 to trigger interrupt by writing 0xFF to GPIO_EXTIPFALL and EXTIPRISE

```
  // Set interrupt on 1->0
  ldr r1, =0x0ff
  str r1, [GPIO, #GPIO_EXTIFALL]

  // Set interrupt on 0->1
  str r1, [GPIO, #GPIO_EXTIRISE]
```

Activating interrupt generation by writing 0x6 to ISER0

```
  ldr r1, =ISER0
  ldr r2, =0x802
  str r2, [r1]
```

7.3 Handling interrupts

To handle GPIO interrupts the handler reads from GPIO_IF interrupt flag register, and stores it in the GPIO_IFC, interrupt flag cleared register.

```
    ldr r1, [GPIO, #GPIO_IF]
    str r1, [GPIO, #GPIO_IFC]
```

7.4 Efficient interrupts

Enabling deep sleep on exiting interrupt handler by writing 0x6 to SCR, the system control register

```
    ldr r1, =SCR
    mov r2, #6
    str r2, [r1]
```

Waiting for interrupt

```
program_loop:
    wfi
    b program_loop
```

The loop is not strictly necessary as the processor will not execute anything after hitting the wfi instruction.

7.5 Further improvements

To disable SRAM blocks we write 0x7 to the EMU_MEMCTRL to set all POWER-DOWN bits to 1.

```
    mov r1, #0x7
    str r1, [EMU, #EMU_MEM_CTRL]
```

To reflect the changed memory layout we change the linker script, specifically, ram LENGTH is now 32kb

```
MEMORY
{
  rom (rx) : ORIGIN = 0x00000000, LENGTH = 1048576
  ram (rwx) : ORIGIN = 0x20000000, LENGTH = 32768
}
```

To disable the low frequency peripheral clock LFBCLK, and set LFBCLK to use ULFRCO we write to CMU_LFCLKSEL

```
    mov r1, #0x1
    lsl r1, #0x10
    str r1, [CMU, #CMU_LFCLKSEL]
```

To set up LETIMER we first need to change the value we wrote to ISER0 to also set bit 26 to 1 in addition to odd and even interrupts.

```
    mov r2, #1
    lsl r2, #26
    ldr r3, =0x802
    orr r2, r2, r3
    ldr r1, =ISER0
    str r2, [r1]
```

Then we set up the LE interface clock so we could configure LETIMER

```
    mov r1, #0x10
    str r1, [CMU, #CMU_HFCORECLKEN0]
```

We set up LETIMER

```
    // Setting comp0 value, i.e which val we want to count down from
    ldr r1, =0x100
    str r1, [LETIMER, #LETIMER_COMP0]

    // We set the comp0 to be top val
    mov r1, #1
    lsl r1, #9
    str r1, [LETIMER, #LETIMER_CTRL]

    // We set the LETIMER to interrupt on underflow
    mov r1, #4
    str r1, [LETIMER, #LETIMER_IEN]
    // str r1, [LETIMER, #LETIMER_IFS]

    // We start LETIMER
    mov r1, #1
    str r1, [LETIMER, #LETIMER_CMD]
```

We handle the interrupts that will now be generated

```
    .thumb_func
letimer0_handler:
    ldr r1, [LETIMER, #LETIMER_IF]
    str r1, [LETIMER, #LETIMER_IFC]
    // Count down
```

```
    ldr r1, [GPIO_LED, #GPIO_DOUT]
    lsr r1, #8
    sub r1, #1
    lsl r1, #8
    str r1, [GPIO_LED, #GPIO_DOUT]

    bx lr
```

8 Figures

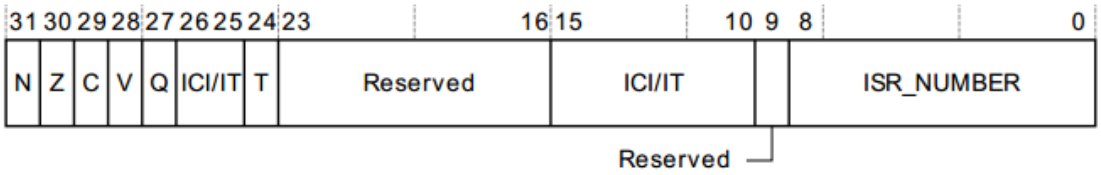

Figure 8.1: PSR bit assignments

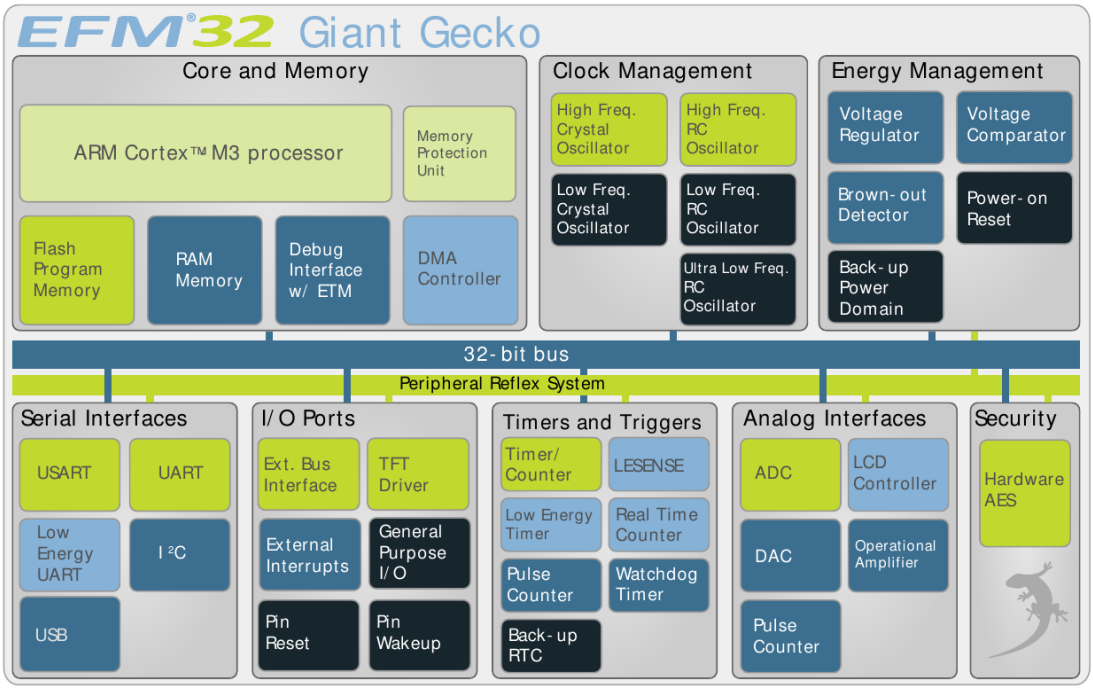

Figure 8.2: Block diagram of EFM32 Giant Gecko

Figure 8.3: EFM board with gamepad

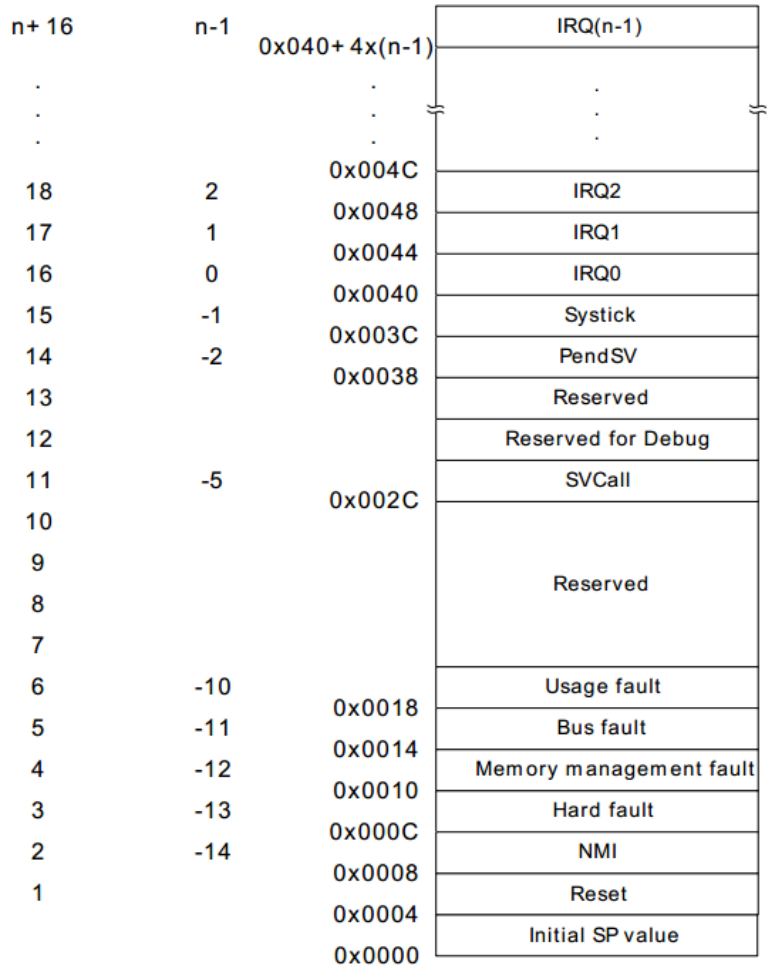

Figure 8.4: Exception vector table

Bibliography

[1] ARM. *Cortex-A8 Technical Reference Manual*, 2010. Available at `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/DDI0344K_cortex_a8_r3p2_trm.pdf`.

[2] Free Software Foundation. www.gnu.org, 2014.

[3] Silicon Labs. *EFM32GG Reference Manual*, 2014. Available at `http://www.silabs.com/Support%20Documents/TechnicalDocs/EFM32GG-RM.pdf`.

[4] Energy Micro. *CortexEFM32GG Reference Manual*, 2011. Available at `http://www.silabs.com/Support%20Documents/TechnicalDocs/EFM32-Cortex-M3-RM.pdf`.