TDT4258 - Exercise 2

Magnus Halvorsen Julian Lam

March 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 second exercise, and builds on the first exercise, in the course TDT4258 Energy Efficient Computer Systems at NTNU. The main goal of this exercise is to use the *the EFM32GG DK3750 development board* to produce sound, by using the boards Digital to Analog Converter (DAC), a timer, and interrupts. This report will guide you through process of creating a simple program that plays different sounds when a button is pressed. We will also present the design choices made in order to minimize the power consumption.

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 C and can be uploaded to the board by usb and a program called *eACommander*. The processor supports the ARM Thumb-2 instruction set, which must be specified to the compiler.

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.

The DAC outputs sound to via the audio output, which can be listened to by connected a standard headset.

Both the micocontroller is designed to be highly energy efficient, and thus support five different energy modes. Energy mode 0 provides full peripheral availability, while energy mode 4 disables most of the peripherals 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 five C files and a header file containing useful constants, providing a simple LED program in C. 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 a C program that runs directly on the development board (without support of an operating system) and which plays different sound effects when different buttons are pressed. Each generated sound effect will be a sound effect you may use in the final game. You have to make at least three different sound effects (for example, cannon shot, target hit, player win etc.). Make a start up melody which can be played when the game begins
- Use the energy monitor to see the power requirements of your program. Analyze and discuss (or implement) improvements.

The solution we provide does the following: if you press one of the three leftmost buttons the board plays either the Star Wars main theme, a cannon shot or a game over song, depending on which button was pressed.

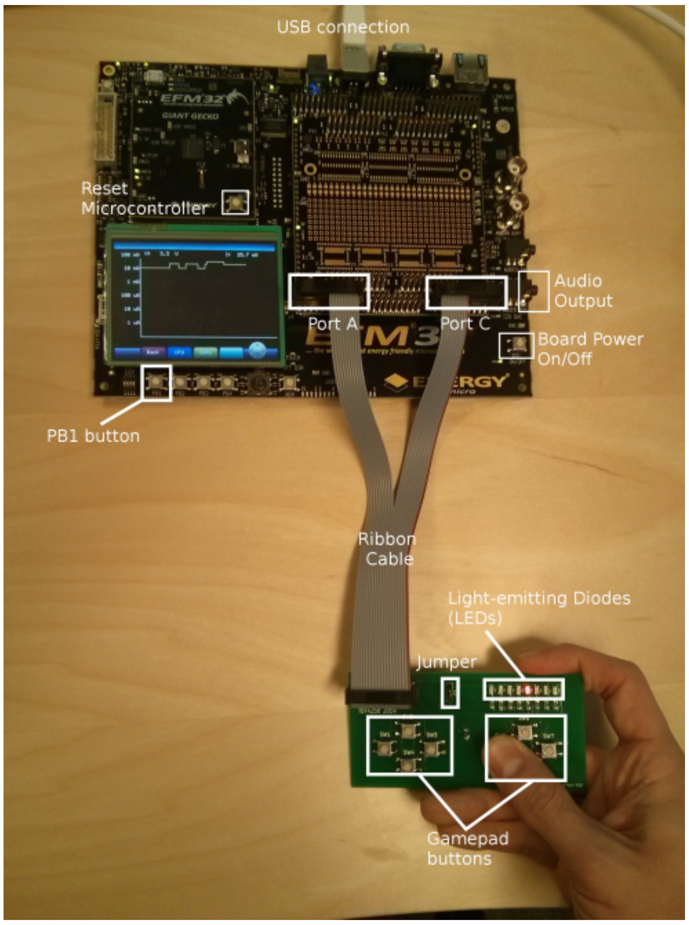

Figure 1: The EFM32GG microcontroller and gamepad[3]

2 Description and Methodology

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

- Setup General Purpose Input/Output and interrupt.
- Set up the Digital to Analog Converter (DAC).
- Set up a timer to feed the DAC with sound samples.
- Create sound samples.
- Reduce the power consumed.

2.1 Setup General Purpose Input/Output and interrupt

Basically the same method that was used in the previous assignment was used here as well.

- Enable the general-purpose input/output (GPIO) clock gate, 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.
- 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.
- Handle any GPIO interrupts by creating the C functions __attribute__ ((interrupt)) GPIO_EVEN_IRQHandler() and __attribute__ ((interrupt)) GPIO_ODD_IRQHandler().
- Clear the interrupt by writing 0xff to GPIO_IFC in the interrupt handler, so the interrupt handler will not be repeatedly called.

The listed steppes make the microcontroller generate a interrupt whenever a button is pushed, allowing us to set the DAC up for playing a sound.

2.2 Set up the Digital to Analog Converter (DAC)

The DAC was very easily enabeled by the following steppes[3]:

- Enable the DAC clock gate by writing to HFPERCLKEN0.
- Prescale the DAC to 437.5 kHZ, and enable the DAC output to the amplifier by writing 0x50010 to the DAC control register.
- Enable DAC0 channel 0 and 1 by setting their respective control registers to 1.

We also created a function to turn off the DAC, when not playing sound, to save power. This was done by simply writing 0 to the mentioned registers, except for HFPERCLKEN0.

This enabled the DAC, but in order to generate sound we have to feed it with a new sample thousands of times every second. For this we needed a timer.

2.3 Set up a timer to feed the DAC with sound samples

We decided to use the Low Energy Timer (LETIMER0) in this exercise too, in order to save power. We tried to use the Ultra Low Freq. RC Osciliator this time too, but it provided too low sample rate (1 kHz). Thus we connected the LETIMER0 to the Low Freq. RC Osciliator (LFRCO), which runs at 32 kHz. The LETIMER0 also allows us to reduce the clock rate by writing to the CMU_LFAPRESC0 register.

The following steppes were preformed to setup the LETIMER0:

- Enable the Low Energy clock gate by writing to the CMU_HFCORECLKEN0 register.
- Enable the LETIMER0 clock gate by writing to the CMU_LFACLKEN0 register.
- Enable LFRCO by writing to CMU_OSCENCMD, and connect it to the LETIMER0 by writing to the CMU_LFCLKSEL register.
- Set COMP0 as top value (the value the timer starts at) and enable free mode in the LETIMER0_CTRL register.
- Write 1 to COMP0.
- Enable interrupt on underflow.
- Start the timer when a GPIO interrupt occurs.

We also added a function to disable the LETIMER0 when we were not playing sound in order to save power, by simply disabling the LFRCO and interrupts.

This allowed us write a new sample to the DAC channels whenever a underflow occured, which meant we could generate a new sample at a rate of 32 kHz. Now we only needed samples.

2.4 Create sound samples

In order to create sound, we needed samples we could feed into the DAC. We did this by converting WAV sound files into 8 kHz, 8-bit WAV files, using a free program called Audacity[4]. Afterwards we used a simple C++ program[5] to convert the WAV files to a C char array. The mentioned C++ program is among the deliverables.

With these char arrays, containing samples, available, generating sound was very simple. On GPIO interrupt we see which button was pressed, and setup the timer and the DAC to play the respective sound. We then simply fed the DAC with a new sample on every LETIMER0 interrupt from the respective sample array.

2.5 Reduce the power consumed

As mentioned, we decided to use the LETIMER0 with the LFRCO in order to reduce power consumed. This allowed us to let the microcontroller to enter deep sleep (i.e. turn of the processor) in-between the underflow interrupts. We also turn off the DAC, the LETIMER0 and the LFRCO when sound is not played. In other words, the DAC and the LETIMER0 is set up in the GPIO interrupt handler, and disabled in the LETIMER0 interrupt handler when the end of the sample array is reached.

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 one of the leftmost buttons, it will play a sound. The first will play the Star Wars main theme, the second will play a cannon fire, and the third will play a game over sound.

3.1.2 Power consumption

Without any power optimizations the program used 5-6 mA. By putting the processor to sleep by using the WFI instruction and enabling deep sleep this number was lowered to 1.5 mA (when sound was not playing). Finally, by disabling the DAC and the LETIMER0 when sound is not playing, the power consumption ended up at 1-3 μA when no sound is playing and 500-1000 μA when sound is playing.

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 Conclusion

All of the requirements given by the assignment has been fulfilled. The described program plays three different sound when the first leftmost buttons are pressed.

We have also designed the program to be as power efficient as possible, by letting the processor deep sleep while waiting for interrupts, using the Low Energy Timer for timing the sampling rate, and disabling any uneccesary power consuming component.

We tried to improve the energy efficiency of the program by using Direct Memory Access, but failed to do so after having troubles enabling PRS pulses on LETIMER0 output.

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

[4] Audacity
http://audacity.sourceforge.net/?lang=nb

[5] Dark Fader
http://www.darkfader.net/