NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4258 Energy Efficient Computer Design
Laboratory Report

Exercise 2

Group 27:

Eirik Prestegårdshus
Tor Braastad Gylder

April 27, 2015

Abstract

Today the microcontrollers has become so small and energy sensitive that energy-efficient programming has become a huge factor. Energy-efficient programming gives the board a longer lifespan so it can be used places that it hasn't been used before. This assignment is about learning how to program different sound waves in C code to the processor, and keep the energy consumption as low as possible. The group need to use their experience and knowledge about energy-efficient coding and sound waves. This task gives more deeper understanding in how the different energy modes works on the board, as well as debugging their code. The first task is to make a sound when pressing a button on the gamepad. Here the GPIO setup from the earlier assignment might be handy. While programming, it would be a good idea to shut down the TIMER after every sound. After this the programming of the different sound effects comes in. On every interrupt the energy consumption was around 2.3 mA, by enabling deep sleep between the interrupts the power consumption fell to an average of 2.41 µA. That way the processor would sleep while nothing happens and save power.

1 Introduction

Microcontrollers are almost in every electrical driven part around you. The number of battery driven devices multiply by the day. In order to get the most out of the device, it has to be made in a smart manner so it can behave properly whilst being as power efficient as possible. There're numerous ways to write programs to the chip, but they might differ a lot in terms of efficiency. This report will focus on coding in C utilizing interrupts and sleep states when there's no need for action.

This is the second part of a three part assignment. The first part was an introduction to assembly code and learning how to handle the processor. This part on the other hand is an introduction to coding in C with the same principles of memory management and interrupts. The desired outcome of this exercise is to be able to make a few different sound effects, using a number of different frequencies. To accomplish this the program needs to have an internally synthesizer and a DAC. Ideally it should not be using too much memory due to accessing the flash is costly. The processor should also be able to use the buttons to change the effects as well as using the lights as in the previous task.

2 Theory

In order to make the processor play sound it'll need the the following; Timer interrupts, Digital-to-Analog-Converter(DAC) as well a programmed synthesizer and it will also utilize the GPIO from the previous project. This chapter will give some background information about this.

2.1 Sound Effects

A synthesizer in the software will enable the board to generate custom sound waves at different frequencies and amplitudes. Sound effects can be created by a set of waves in succession. The properties of the sound will depend on frequency, amplitude and waveform that oscillates in a media. The amplitude defines the strength of the sound while the frequency defines the tone. The waveform will decide the harmonic overtones, this will impact how static the set frequency will sound. E.g. a squared wave will have a lot of harmonic overtones, whilst a pure sine will have none. The squared wave will sound a lot more static as it has a lot of harmonics which our ears will register as higher frequencies. In case the decision is to make a custom waveform we will have to consider the above. A pure sine wave will sound terrible, and so will a squared wave. The desired waveform might be a compromise between the two.

There are more ways of making sound. The other way of creating a sound effect is by iterating over a huge array containing the entire sound, e.g. a song or an effect. The sound wave has to be imported in some way and converted to an array of values fitted for the DAC. If the program were to import a 1 second effect with a sampling frequency of 44kHz, this would be 44000 values. Furthermore let's say the values are of size 2 bytes(16 bits), the one second effect is suddenly 88 kilobytes large. If the processor were to play a 2 minute song the memory needed will quickly become very much. In terms of energy efficiency, this might not be the best choice.

2.2 Software

2.2.1 GCC & GDB

The GNU Compiler Collection (GCC) will compile the selected file and check for warnings. This also enables debugging and creates an object file (which is later used for linking files). There are different ways to debug the code while testing it on the board. One way is to use GNU Emacs with GDB commands, where the user can check the code and the debugging at the same time. An other way is to use GDB directly in the

terminal, and connect it to a server(JLinkGDBServer). The interface is more simplicity, but it all comes down to what the user is more experienced with.

2.2.2 Interrupts handling in C

By using C programs you can easily control the hardware directly by using pointers to the memory mapped I/O registers. This part is crucial for changing the synthetic sound and handling the buttons on the gamepad. C code gives an easier way for the program to "lookout" for buttonpress (see code below). With interrupts the program is going to a sleep mode and waiting for something to happen e.g a buttonpress, timer bit etc. With pulling you cant use sleep modes. So in other words, it's a lot more energy efficient to use interrupts.

```
//Enables interrupt on Even
void __attribute__ ((interrupt)) GPIO_EVEN_IRQHANDLER()

//Enables interrupt on ODD
void __attribute__ ((interrupt)) GPIO_ODD_IRQHANDLER()
```

2.2.3 Sound Generator: DAC

The Digital Analog Converter (DAC) creates an analog signal based on digital inputs. The EFM32GG has an internal DAC and the DK3720 board has an amplifier to the Audio Out port. This way we can extract the sound wave created by the code and make it behave as intended. The internal DAC are controlled by setting 1(on) or 0(off) to *DAC0_CH0CTRL* and *DAC0_CH1CTRL*. It can also run in every energy modes except mode 4.

2.3 Timers

The timer interrupt will trigger a new value to be sent to the DAC. This will efficiently work as a sampling frequency with a frequency equal to the one divided by the time between interrupts. The timer clock can be turned off to pause new values being sent to the DAC, which will pause the sound. Additional improvements may be to turn the DAC off when Timer isn't used, as they're both needed to generate sound.

2.3.1 Low energy timer

Low Energy Timer (LETIMER) can keep track of time and output configurable waveforms. The LETIMER works like the normal timer, but it's available in energy mode 2. Although it's not as good as the regular timer, it's very practical to use when the program needs to be energy efficient. *LETIMER0_CMD* and *LETIMER0_IEN* must be set to 1(on) to enable LETIMER.

2.4 Energy Efficiency

The EFM32GG has five different energy modes(2.2). Each mode shuts down different parts of the board to lower the energy consumption as showed in the figure 2.1 below. When programming you can enable WakeForInterrupts(WFI). This gives us the possibility to change between different sleep modes when an interrupt occur. That way the program can access the different blocks to fulfill its tasks.

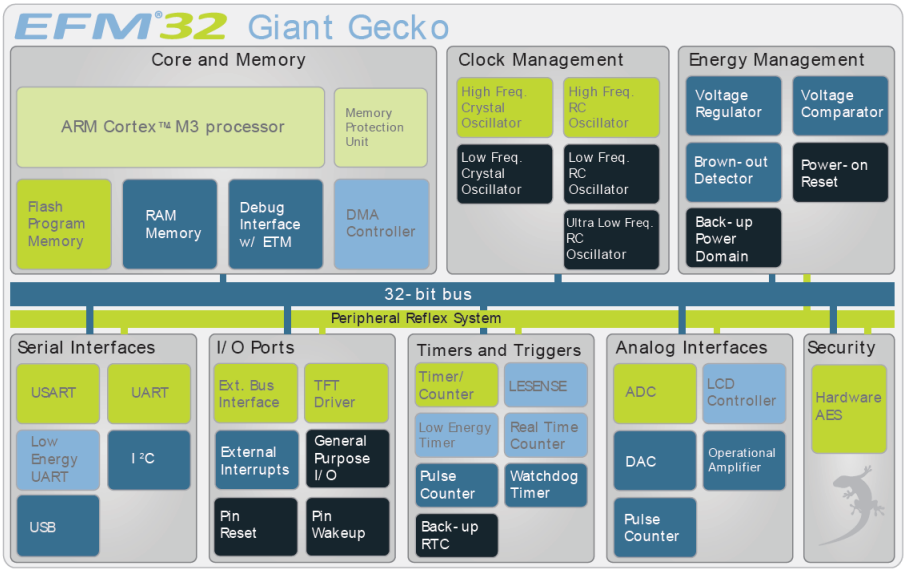

Figure 2.1: Board blocks & their energy mode [1]

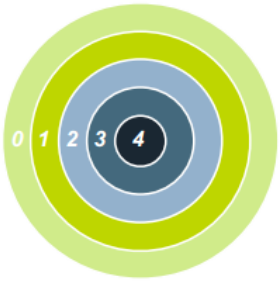

Figure 2.2: The different energy modes [1]

3 Methodology

The method used in this project use the following parts of the EMF32; TIMER1, LETIMER0, GPIO, DAC and EMU. To achieve full functionality at once is hard, it will in the end be build up of many smaller parts.

3.1 Sound

The most attractive setup of generating sound for the given were decided to be a synthesizer. In the theory part it's described why. The synthesizer should have some common frequencies used by other instruments, e.g. some piano tones so it's easy to place tones in succession to make a song. The frequencies used are listed in the array *freq*. How to generate sound with TIMER1 and the DAC is described in the theory chapter. Instead of actually making the sound wave longer or shorter, it simply changes how fast it iterates through it. The way this is implemented is to change the prescaling of the timer in such fashion so it matches the desired frequency.

The array contains simply one period of the waveform. The frequency decides which tone to be played. Instead of changing the frequency of the array, the processor changes the sampling frequency in order to play different tones. New waveforms can easily be added to the array making the number of sounds theoretically infinite.

```
uint16_t array[3][40] = {{....},
{0,5,10,15,20,25,30,35,40,45,50,55,60,65,70,75,80,85,90,95,100,
95,90,85,80,75,70,65,60,55,50,45,40,35,30,25,20,15,10,5}, {...}};
```

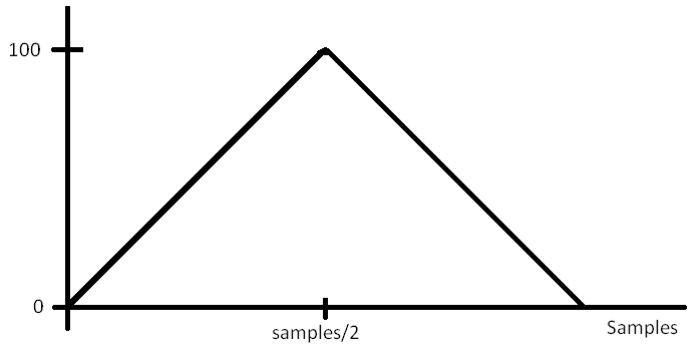

Figure 3.1: Triangle wave we coded after

3.1.1 Low Energy Timer

To iterate over tones it's best to use interrupts as well. It can of course be done by an idle loop, but this is a very bad implementation in terms of energy efficiency. The function *playSong()* starts the timer. The handler has two options, if the *songCounter* haven't reached the end; play next note. There's two different options here, either it has a note to play, or it's zero; which allows it to turn off the sound and deep sleep for a period. It's able to do this only because low frequency oscillators are turned on in energy mode 2. If the *songCounter* has reached the end, it stops itself, turns off the sound and reinitialize deep sleep. The scalability of this interpretation is great. New songs can easily be added and selected in a similar way to the selection of waveforms which uses a global variable *mode*.

3.2 GPIO

The first, and maybe the easiest is handling button presses. This is the task given in Exercise 1, what has to be done is to translate the assembly code to C code. A switch statement with each button assigned to each case can easily give a specific task to each button. Both the even and the odd interrupt handlers call the function *handleInterrupt()*. This function stores buttons presses in b and shift them 8 positions to light up the right button and then proceeds to doing a custom task based on which button/buttons. As most of the cases play a frequency directly, it needs the DAC and the TIMER. The TIMER uses a high frequency clock, therefore it needs to turn DEEPSLEEP off in the System Control Register(SCR). There are two special cases in the switch statement which require attention, no button pressed(0xffff) and default. The default will handle cases where there's no case specified, in this case it happens when there's more than one button pressed. The action taken in default is to swap mode. The mode specifies the waveform to be used for the sound. GPIO interrupts are set to both rising and falling edges. This means it'll trigger when the button is let out too. This is when the no button press will be used. This case turns off the sound, and enables sleep again.

So why is the *DACandTIMER()* placed inside the switch case and not outside? This is because a race condition that might occur. If it turns the DAC/TIMER on when it does the operation, it might raise an interrupt flag before it recognizes the DAC/TIMER should be off. There's also a bit unnecessary to turn the sound on and off about 50% of the time, since users are most likely to press one button at a time.

3.3 Energy Mode 2

The processor utilize Energy Mode 2(EM2) to save energy while it's no need for the processing unit to be running. This is done by enabling DEEPSLEEP in the System Control Register(SCR). When an interrupt is being handled the processor automatically jumps out of the sleep mode, also known as EM0. This enables the processor to run the handler and return. The processor should start to sleep every time it returns from

a handler, for this to happen the SLEEPONEXIT in the SCR should also be set. EM2 disables a feature which are crucial to the program, high frequency oscillators which is used by TIMER1. The drawback of this is that we are unable to bring the processor down into EM2 in between each new sample sent to the DAC. This requires DEEPSLEEP to be turned off when a tone is playing and not allowing the processor to sleep.

3.4 Further Improvements

The sound has a sample frequency of 40 times the desired frequency. The generation of sound might not even need a high frequency oscillator, and a low frequency oscillator could be used. The use of another low energy timer based on a low frequency clock could allow the processor to go into EM2 in between each sample. This would reduce the power usage as the time needed in EM0 will be reduced drastically, because the time needed for operation in the handler is much less then the time spent idle waiting for an interrupt.

4 Results

4.0.1 Sound

The gamepad has been programmed to have different piano tones on each button, except from the first one. After getting LETIMER to work the first button was implemented with the Super Mario intro song. The user of the gamepad can change waveform by pushing two buttons or more at the same time. The frequency bound to each button are the same, but the sound wave has changed. The three waves are; Simple "guitar"-, triangle- and square-wave. As mention earlier in section 3.1 each wave is approximate representation coded in an array. This could be solved in other ways as well, such as saving sound effects/tones as separate sound files and iterate over them over time. We thought the way we did it was simpler and better. It's easy to separate the different sound waves, as smoother the sound is the closer to a sine wave it is. In other words the triangle wave has lower pitch than the square wave, and the "guitar" wave is in between.

4.0.2 Energy consumption

In deep sleep mode (Energy mode 2) the average power consumption is 2.41 µA. This is fairly low considering that without deep sleep it would have an average about 2.3 mA. As shown in the figure below, when a button is pressed the board peaks on 2.3 mA. This happens because the board goes out of deep sleep and enables part of the board that the program needs. After every interrupt the board goes back to sleep and waits for the next one. It may be possible to make the board go to a deeper sleep mode than EM2, since LETIMER can work in EM3(REF). That way the energy consumption would lower even more, but we couldn't figure out how to enable LETIMER in this mode, so we skipped it.

4.1 Problems & Debugging

We experienced some bugs and problems during this assignment. Like pausing between tones, deep sleep after tones and unwanted interrupt flags. At first, when making the Super Mario intro song, we had troubles with finding a way to pause between each note and keep the power consumption relativity low. This was solved by enabling the low energy timer on the board, which lowered the energy consumption and enabled interrupts on timer as well.

When coding there's a lot of small mistakes that may have an impact on the result. Simple mistakes as turning interrupt on before the entire setup caused a problem. For some reason an interrupt flag in the GPIO is raised upon reset every time. When

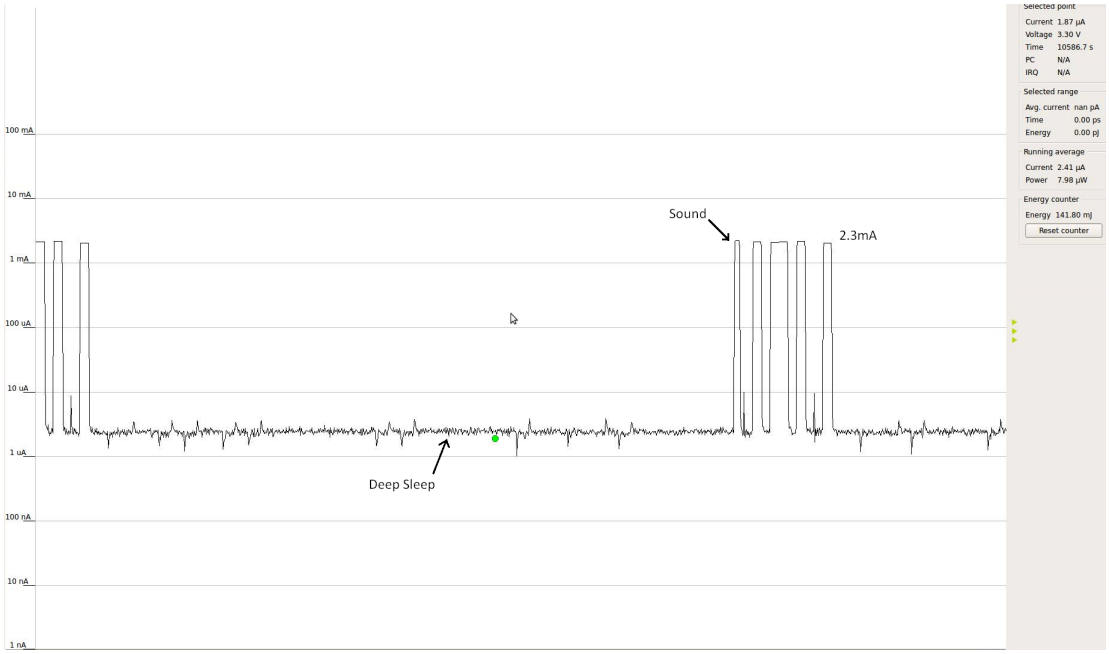

Figure 4.1: plot of energy consumption. With and without sound

interrupting were enabled before the setup completed this caused a mess and the DAC would not properly activate. There where taken two countermeasures to this problem. When setting up the GPIO, we made sure to clear the flag and align the order in such fashion that the *setupNVIC*(Enabling interrupts) were the last of the setup.

5 Conclusion

The processor is fully able to play songs or any other custom sound. The method of iterating through an array till the end is a great way to play a song. It's also very scalable, due to everything being stored in array. These arrays can easily be lengthened or added another dimension to hold more sound effects, or the use of multiple arrays in case the sound effects are of different length. What each button does can easily be modified as well. Custom action can even be given to combinations of button presses as the switch method can easily be appended with more cases.

It's easy to see the huge difference in power usage between sleeping and running. Many devices run on battery or have the desire to be eco friendly. The clue is to keep the processor as much as possible in the sleep state. The processing clock run at 14 million times a second, the calculations might need only a few clock cycles to compute. In the case of having a frequency of say, thousand, there would be 14 thousand cycles between each time computation is needed. Unfortunately for this implementation sleeping in these unused cycles couldn't be done. That's because of the TIMER using a high frequency clock, which will be shut down during deep sleep. There's a way to solve this, although it wasn't done in this report. The way is to swap out the high frequency timer with a low frequency timer, e.g. another LETIMER(Low Energy Timer). In the case with only low frequency timers, which is enabled in deep sleep, the processor will be able to save many of those unused clock cycles.

After a period of time the amount of codelines start to pile up. It can be the smallest mistakes that cause the processor to not behave properly. The workflow becomes very important in order to maintain functionality. It helps a great deal to test each added feature thoroughly before moving on with another one. Compiling and testing the functionality after each addition contributed to us seeing the mistakes early, and giving another view of the recently written code.

Bibliography

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