NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4258 Energy Efficient Computer Design
Laboratory Report

Energy Efficient Sound Generation

Group 24:

Peter Aaser
Audun Sutterud
Jonatan Lund

November 11, 2015

Abstract

The course TDT4258 Energy Efficient Computer Systems gives an introduction to energy-efficient microcontroller programming.Through lectures and three comprehensive exercises the students get theoretical knowledge as well as hands-on experience.

In this second exercise of the course we implement two different ways of handling audio generation for the EFM32GG utilizing the Digital Analog Converter using C with no operating system. We show how the different approaches differ in sound quality, energy consumption and size. To control the output we also implement functionality to handle button inputs, mapping them to various songs and sound effects.

Contents

1 Introduction

In this exercise we will write a program for the EFM32GG-DK3750 that generates sound effects by utilising the microcontroller's DAC (Digital-to-Analog Converter). A prototype gamepad will be connected that allows the user to control the sound by pressing different buttons. We need to make at least three different sound effects and a start up melody that can be used in a game. Extra points will be given for an energy efficient solution. The program is written in C, and should run directly on the board with no operating system involved. As an aid to setting up the hardware, some skeleton code was provided with function stubs and several macros for useful register addresses.

We will use the GNU Compiler Collection to compile and link source files. As there are several useful C libraries we may want to link, we need to pay extra attention to the linking process. We also need to program several board peripherals, including the DAC, the PRS (Peripheral Reflex System), an on-board timer, and the GPIO (General-Purpose Input/Output) pins. The purpose of the exercise is to learn about C programming, GPIO control in C, interrupt handling in C, and to generate sound effects using the microcontroller's DAC and timers.[1]

2 Background and Theory

This chapter contains background theory relevant to the exercise. The first part is a brief introduction to the various hardware components of the EFM32GG-DK3750 that were used in this exercise. The second part details energy optimization techniques in modern computing systems. Finally, a small section outlines theory of sound waves and music.

2.1 Overview of EFM32GG-DK3750

Through the exercises in this course we are working with the Silicon labs EFM32GG-DK3750 prototyping board (figure 2.1), from here on referenced as EFM32GG. The EFM32GG hardware components we used in this exercise include the following:

- Clock Management Unit
- Timer/Counter
- Digital to Analog Converter
- General-Purpose Input/Output pins
- Gamepad Prototype
- Energy Management Unit
- Low Energy Timer
- Peripheral Reflex System
- Direct Memory Access controller

A more comprehensive map of the EFM32GG hardware components is shown in figure 2.2.

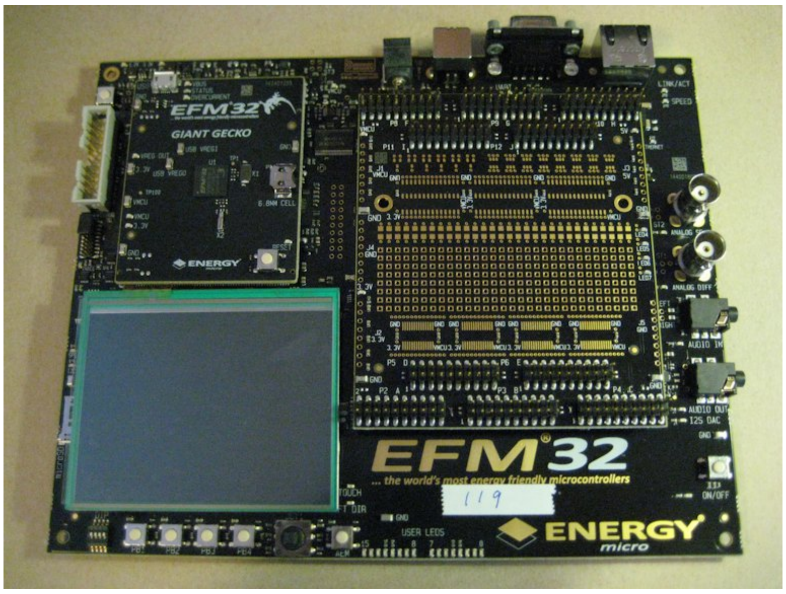
Figure 2.1: The EFM32GG prototyping board

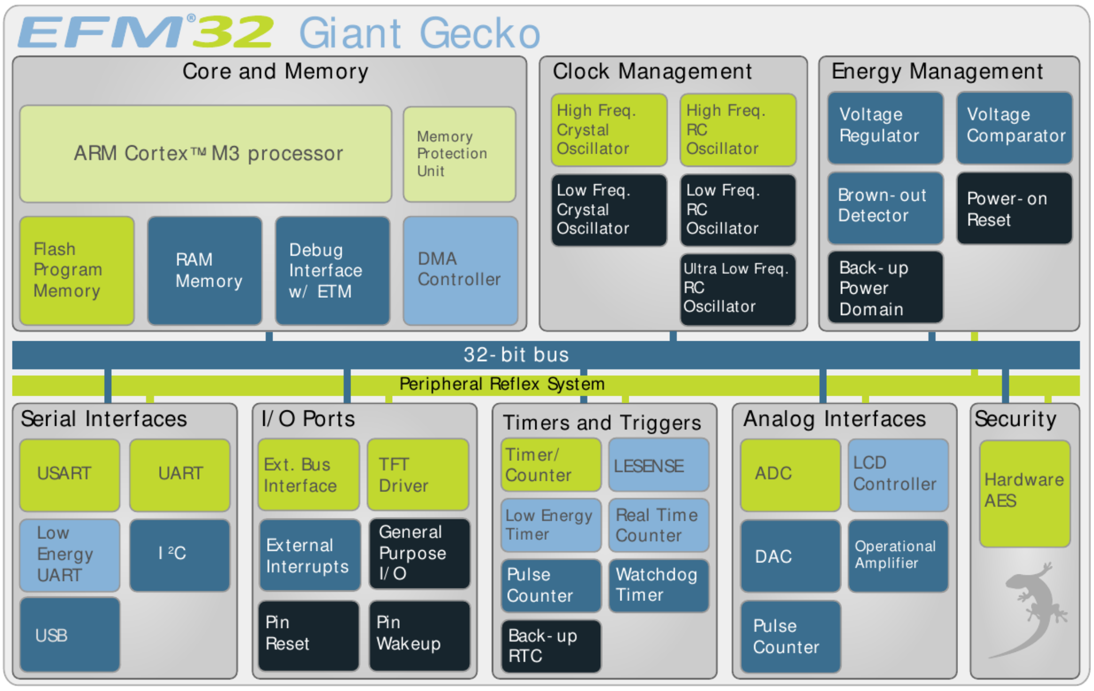

Figure 2.2: Map of the EFM32GG hardware components

2.2 Clock Management Unit

The Clock Management Unit (CMU) controls the on-board oscillators and clocks, and produces clock signals that drives the core modules and all the peripherals. The CMU can be used to enable or disable clock signals for the different peripherals on an individual basis. It possible to reduce the power consumption significantly by only enabling the clock for the peripherals that will be used. By default, the clock signals for all peripherals is disabled.

The CMU manages several system clocks in order to provide different clock signals to core modules and peripherals. These clocks include the High Frequency Clock (HFCLK), the High Frequency Core Clock (HFCORECLK), and the High Frequency Peripheral Clock (HFPERCLK).[6]

2.2.1 High Frequency Clock

HFCLK drives two prescalers that generate HFCORECLK and HFPERCLK, in addition to driving the CMU itself. With separate prescalers it is possible to configure the clock speeds of the core modules and the peripherals individually. The HFCLK itself can be driven either by one of the high-frequency oscillators (HFRCO and HFXO), or by one of the low-frequency oscillators (LFRCO and LFXO).

2.2.2 High Frequency Core Clock

HFCORECLK is the clock that drives the core modules, which consists of the CPU and tightly coupled modules such as the DMA. The frequency of HFCORECLK can be changed dynamically by writing to CMU_HFCORECLKDIV, and the clock can be disabled for individual modules by clearing bits in CMU_HFCORECLKEN0.

2.2.3 High Frequency Peripheral Clock

HFPERCLK is the clock that drives the high-frequency peripherals. The frequency of the HFPERCLK can be changed dynamically by writing to CMU_HFPERCLKDIV, and the clock can be disabled for individual peripherals by clearing bits in CMU_HFPERCLKEN0.

2.2.4 Low Frequency Clocks

The HFCLK is only available in energy modes EM0 and EM1 (see section 2.7), and entering lower energy modes than EM1 will disable most core modules and peripherals. To drive components in EM2, the EFM32GG uses several special clocks, including the Low Frequency A Clock (LFACLK) and the Low Frequency B Clock (LFBCLK). Notably, the LFACLK clock drives the LETIMER module described in section 2.8.

2.2.5 Oscillators

To generate the clock signals and drive the clocks described above, the EFM32GG has five on-board oscillators:

- 1 - 28 MHz High Frequency RC Oscillator (HFRCO)
- 4 - 48 MHz High Frequency Crystal Oscillator (HFXO)
- 32.768 KHz Low Frequency RC Oscillator (LFRCO)
- 32.768 KHz Low Frequency Crystal Oscillator (LFXO)
- 1 KHz Ultra Low Frequency RC Oscillator (ULFRCO)

Depending on the application, the board components can be run either fast or slow by driving clocks with different oscillators. The oscillators differ in having different start-up times. For instance, the crystal oscillators might need several hundreds of microseconds to become completely stable.[3]

2.3 Timer/Counter

The Timer/Counter module (TIMER) can be used to count events and trigger actions in other peripherals, and to trigger interrupts after a specific time interval. It has a 16 bit counter that is either incremented or decremented depending on the *counter mode* of the TIMER. The TIMER is driven by the HFPERCLK, and can be prescaled by setting the PRESC bits in TIMERn_CTRL to an integer n between 0 and 10. This will result in a TIMER frequency of HFPERCLK/2^n. The timer can be started and stopped by writing to the bits START and STOP in the register TIMERn_CMD, or by receiving signals from other peripherals through the PRS (see section 2.9).

2.4 Digital-to-Analog Converter

The Digital-to-Analog Converter (DAC) converts digital values to analog signals. The digital values are fed into the DAC through the two 12-bit input channels DACn_CH0DATA and DACn_CH1DATA, and is converted to output voltages on the two output channels. By feeding the output from the DAC through the on-board amplifier and into external speakers, the EFM32GG can be used to create sounds. For more information about sound synthesizing in general, see section 2.11.

2.5 General-Purpose Input/Output pins

The General-Purpose Input/output (GPIO) pins makes it possible to connect a wide range of 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.

2.6 Gamepad Prototype

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 that allows us to toggle whether the amperage consumed by the LEDs will be measured during energy profiling. An image of the gamepad is shown in figure 2.3.

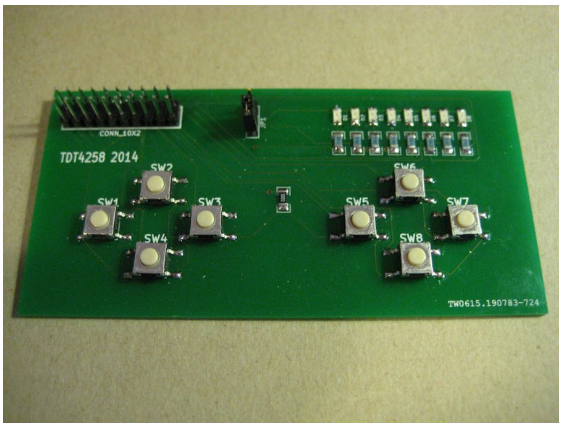

Figure 2.3: The prototype gamepad

2.7 Energy Management Unit

The Energy Management Unit (EMU) manages the different low energy modes in the EFM32GG. The EFM32GG has the capability to turn off power to board components at runtime by switching between five distinct energy modes. These modes range from EM0 (run mode) where the CPU and all peripherals are active, to EM4 where almost everything is disabled. The hardware component map in figure 2.2 shows what components are active in the different energy modes by showing them with separate colors. A description about the color scheme for each energy mode, can be found in figure 2.4. In addition to handling the energy modes, the EMU can also be used to turn off the power to unused SRAM blocks.[6]

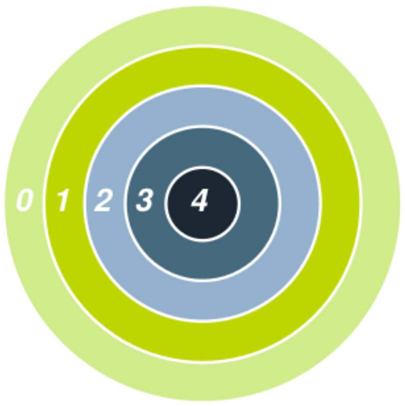

Figure 2.4: Color scheme for the individual energy modes

2.8 Low Energy Timer

The Low Energy Timer (LETIMER) is a 16-bit timer available in energy modes down to EM3. Similar to the TIMER module (section 2.3), it can be used to generate interrupts after specific time intervals, to generate PRS signals for other peripherals, and to generate arbitrary waveforms on its two output pins. The fact that the LETIMER is available in EM3 makes it especially useful for applications where energy consumption is at a minimum. As mentioned in section 2.2, the LETIMER is driven by the Low Frequency A Clock.

The LETIMER can be started and stopped by setting the START and STOP bits respectively in register LETIMER0_CMD. It has an internal counter that is decremented each cycle, and that can be read from register LETIMER0_CNT. The counter value is by default reloaded to `0xffff` when it underflows, but this can be changed by setting COMP0_TOP in LETIMER0_CTRL and writing a new reload value to LETIMER0_COMP0.

2.9 Peripheral Reflex System

The Peripheral Reflex System (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.

2.10 Energy Optimization

This section introduces relevant theory of energy optimization, and lists several energy optimization techniques that are used in modern computing.

2.10.1 Static and dynamic power

When considering the power consumption of an electronic CMOS integrated circuit, it is useful to split the power into two main parts: *dynamic power* and *static power.*

Dynamic power is the power needed to switch a circuit between two states. It depends on the clock frequency and activity level of a circuit, and is proportional to the square of the circuit voltage. In a clocked circuit it can be expressed as

$$P_{dynamic} = \alpha C V_{DD}^2 f$$

where α is the activity level, C is the capitance, V_{DD} is the voltage, and f is the clock frequency of the circuit.

Static power is power that dissipates due to leakage in the transistors of a circuit. It is linearly proportional to the circuit voltage, but independent of frequency. Static power can be expressed as

$$P_{static} = I_{leak} V_{DD}$$

where I_{leak} is the current leaking through the transistors.

The total power used by the circuit is the sum of both dynamic and static power,

$$P_{total} = P_{dynamic} + P_{static} \ .$$

[2]

2.10.2 EFM32GG Energy Optimization Techniques

This section gives an overview of several ways to reduce the static and dynamic power consumption in the EFM32GG.

Clock Gating

Clock gating is a technique that reduces dynamic power consumption by disconnecting the clock from unused circuits. The EFM32GG uses automatic clock gating, and supports manual clock gating of both core modules and peripherals on an individual basis (see section 2.2). Some energy can be saved by turning clocks on and off as the individual peripherals are needed.[5]

Disabling RAM blocks

In the EFM32GG there is a static power dissipation in the RAM blocks of approximately 170 nA per 32 KB block. This constitutes a considerable amount when the device is in EM2 or EM3 energy modes. The RAM blocks can be disabled individually, and this is explained in section 2.7.[5]

Increasing CPU frequency

By increasing the clock frequency of the CPU it is possible to reduce its overall static power consumption. An increased frequency usually means that the CPU will finish its computations sooner, allowing it to stay in low energy modes for a larger fraction of the time. Increasing the frequency will also cause an increase in dynamic power consumption, but this is canceled out by the similar reduction in processing time. There are some exceptional cases where the CPU processing speed is limited by wait-states in bus protocols. In such a case, the frequency should be increased within the imposed bounds.[5]

Optimizing peripheral clocks

The EFM32GG has several clocks and oscillators running at different frequencies. By using a slower clock for the peripherals, the associated dynamic power consumption is reduced. It is optimal to selecting the slowest clock that satisfies the application requirements. Clock prescalers in the CMU or the individual peripherals can be used to adjust the frequency further. Clocks and oscillators are controlled by the Clock Management Unit, described in section 2.2.[5]

Reducing Bias Current of Analog Peripherals

Most analog peripherals in the EFM32GG use something called a *bias current*. It is possible to reduce this bias current to reduce power consumption, but this will affect the analog performance.[5]

Cache Optimization

Several optimizations can be made by utilizing the caches of the EFM32GG. When a memory read or instruction fetch is satisfied from the cache, the processor avoids possible wait-states and memory reads and energy is saved. To optimize use of the 512 byte instruction cache of the EFM32GG, loops with a large number of instructions should be avoided. It is possible to measure cache misses and cache hits by reading performance counter registers.[5]

Compiler optimizations

As a rule of thumb, using a higher compiler optimization-level will result in more energy-efficient code.

2.11 Sounds Generation

When interpreting a sound as a signal, we speak about properties of the sound such as frequency, period, and amplitude. Here frequency and period (the inverse frequency) describes the pitch of a sound, while amplitude describes the loudness. In a typical speaker, a voltage signal is converted directly into sound waves. Here the voltage signal is an accurate model of the sound wave and has the same frequency and amplitude. The range of sounds that can be perceived by a human being is from 20Hz to 20KHz.[1]

2.11.1 Simple music theory

In the western world the chromatic scale is the standard way to describe music. The chromatic scale is a set of 12 ordered frequencies such that the twelwth note has twice the frequency of the first, and that the spacing between the notes is equal on a logarithmic scale. The common ground note in western music is the A_4 with the frequency of 440Hz. The number refers to the octave, such that A_5 at one octave higher will have twice the frequency of A_4, 880Hz. The note is the letter, and can be sharpened with a #. The 12 notes on a chromatic scale will be referred to in table 2.1. There is no difference between a note with a # and one without, they all have the same relation to their predecessor.

The frequency of a note is given by the following equation:

$$f = A_4 * 2^{D/12}$$

Here we have used A_4 as the ground note, and D is the difference. For instance a third octave C sharp $C\#_3$ is 7 tones lower than A_4, so its frequency f will be

$$f_{C\#_3} = f_{A_4} * 2^{-7/12}$$

1	2	3	4	5	6	7	8	9	10	11	12
A	A#/H	B	C	C#	D	D#	E	F	F#	G	G#

Table 2.1: Chromatic scale

3 Methodology

This chapter details the methods used and the process with which we achieved our results. The chapter starts with setup process with each component. Later we describe our method to generate songs and how they are stored in memory. At the end we describe our different optimizations.

3.1 Initial Approach

In order to get an introduction to programming the DAC we created a program intended to play a single continuous note. This would be an easy introduction to C programming, and would give us a good performance baseline with regards to energy consumption.

3.1.1 GPIO Setup

The first step was porting the assembly code we had previously written to control LEDs and clocks over to C. Having set up the LEDs and buttons would also make it easier to debug the code by writing values to the LEDs to discern the current program state. Rather than starting with a simple polling loop we felt confident enough to go straight to interrupts for the LEDs and buttons, letting the program idle in a loop between interrupts.

The code we used to set up the GPIO pins for input and output is shown in code listing 3.1 and 3.2 respectively. In addition to the steps shown, the clock signal for the modules needed to be enabled in the CMU.

```
*GPIO_PC_MODEL = 0x33333333;  // Set pins 0-7 as input pins
*GPIO_PC_DOUT = 0xff;         // Enable pull-up resistors
*GPIO_EXTIPSELL = 0x22222222; // Set interrupt generation for pin 0-7
*GPIO_EXTIFALL = 0xff;        // Set interrupt on 1->0
*GPIO_EXTIRISE = 0xff;        // Set interrupt on 0->1
*GPIO_IEN = 0xff;             // Enable interrupt generation
```

Listing 3.1: Setting GPIO pins 0-7 as input

```
*GPIO_PA_MODEH = 0x55555555; // Set pins 8-15 to output
*GPIO_PA_CTRL = 2;           // Set high drive strength
```

Listing 3.2: Setting GPIO pins 8-15 as output

3.1.2 DAC Setup

The setup code we wrote for the DAC closely followed the instructions in the compendium, as shown in code listing 3.3. With this setup, the DAC would continuously read both its 12 bit channels and update the output voltage. The next step to generate sound was to continuously write samples to the channels emulating an audible waveform.

```
*DAC0_CTRL = 0x50000; // Set a DAC clock division factor of 32
*DAC0_CTRL |= 0x10;   // Set continuous mode
*DAC0_CH0CTRL = 1;    // Enable left channel
*DAC0_CH1CTRL = 1;    // Enable right channel
```

Listing 3.3: Setting up the DAC

3.1.3 TIMER setup

We chose to base our initial implementation on interrupts, writing a new sample to the DAC whenever an interrupt occurred. The sampling rate could then be set by setting the frequency of the interrupts. To generate interrupts, we used the TIMER module, setup as described in listing 3.4. Note the variable `period` used to set interrupt frequency.

```
*TIMER1_TOP = period; // Set period between interrupts
*TIMER1_IEN = 1;      // Enable interrupt generation
*TIMER1_CMD = 1;      // Start the timer
```

Listing 3.4: Setting up the timer to generate interrupts

Finally, an interrupt handler was written that used a simple conditional statement to write a square waveform to the DAC channels, oscillating the DAC voltage between 0 and 1000. Our first attempt was outside of the audible frequency range, but by adjusting the timer we could reduce the amount of interrupts to control the frequency.

3.2 Efficient Interrupts

After setting up everything we needed to generate a simple tone we decided to do some preliminary optimizations in our program. Rather than using the TIMER module we would use the LETIMER with a max speed of 32678 Hz, more than ample for our needs. As the LETIMER module was available in EM2 and EM3, this would allow us to potentially save energy by entering EM2 between the interrupts.

3.2.1 LETIMER and LFACLK Setup

As the LETIMER is driven by the LFACLK clock, we needed to configure the LFACLK clock before using the LETIMER. The LFACLK was configured using the code shown in listing 3.5, and the LETIMER was configured using the code shown in listing 3.6.

Choosing Low Frequency Oscillator

To drive the LFACLK we had the choice of either using the Low Frequency Crystal Oscillator (LFXO) or the Low Frequency RC Oscillator (LFRCO). After some initial testing we found that the LFRCO was unstable due to temperature differences, producing a slight variation in the pitch of the generated sound. LFXO was stable enough for our purposes. However, it had a longer start-up time, and we were aware that it might introduce delays into our program when waking up from low energy modes. [4]

```
*CMU_OSCENCMD |= (1 << 8);     // Start the LFXO
*CMU_LFCLKSEL &= ~(0x3 << 0); // Select LFXO to drive LFACLK
*CMU_LFCLKSEL |= (2 << 0);     //
*CMU_LFACLKEN0 |= (1 << 2);   // Enable clock signal for LETIMER

// Enable clock for the Low Energy Peripheral Interface
*CMU_HFCORECLKEN0 |= (1 << 4);
```

Listing 3.5: Setting up the LFACLK to drive the LETIMER

```
*LETIMER0_CTRL |= (1 << 9); // Set period between interrupts
*LETIMER0_COMP0 = period;   //
*LETIMER0_CMD |= (1 << 0);  // Start the LETIMER
*LETIMER0_IEN |= (1 << 2);  // Enable interrupt generation
```

Listing 3.6: Setting up LETIMER to generate periodic interrupts

3.3 Generating Songs

We decided early in the process that we wanted to use the EFM32GG to play songs. We explored several different methods to create songs with the EFM32GG, and eventually decided on two different approaches. In the first approach, pre-synthesized samples were loaded on to the board at compile time. In the second approach, distinct notes coded as 16 bits words were loaded on to the board at compile time, and then synthesized at run-time by a software synthesizer. The result was two different implementations: The *on-board synthesizer* and the *sample-based music player*, described in section 3.3.2 and 3.3.3 respectively.

3.3.1 Memory Limitations

The EFM32GG board has 128 kB of SRAM and 1024 kB of flash memory. This puts limitations on how a song is stored in the memory on the EFM32GG. Considering storing the samples of a song sampled at 44100 Hz where each individual sample is 2 bytes large and the song duration is d seconds. The size of the song can be calculated with the formula

$$\text{Song size (bytes)} = 88.2 \cdot d$$

. To find the longest song with a size that will fit in SRAM, we set $88200d = 128kB = 131072B$ which yields $d = 1.49$. Thus the longest song that will fit in SRAM is shorter than 1.5 seconds! This can be remedied somewhat by choosing a lower sample rate and possibly utilizing flash memory, but the EFM32GG will be hard pressed to store any quality song much longer than 30 seconds. Our two approaches uses different techniques to evade the memory limitations.

3.3.2 On-board Synthesizer

To overcome the memory limitations, this implementation uses the EFM32GG as a synthesizer to generate square waveforms. A song is stored in memory as integer arrays. The integer array contains information to create different tones. Two integer arrays produces a song with two channels.

Our first thought was to save all the samples of a song.

Song Representation

In our solution we chose to represent a song as an array of 16 bit int values. These arrays is generated by a python script and placed in the source code. Inside these 16 bits we store information about pitch, octave, amplitude and duration. In table 3.1 the bit partitions are described.

Duration	Amplitude	Octave	Pitch
5	3	4	4

Table 3.1: Bit partitioning

Each pitch value is represented by a value from 0 to 11. The Octave can take the values 0-10. Note A with octave 0 is lowest possible at 27.5 Hz, and note A with octave 10 the highest at 28160 Hz. The DAC's lowest values is too low hear, because of this we calculated the amplitude with the equation 3.1. In addition we only had 3 bit to store the amplitude values, and as a result we calculated it exponentially. Duration takes values between 0-32. In our program we define the duration of the shortest note. The duration value is multiplied with the defined duration value and returns each notes duration given in milliseconds. By changing the defined duration value we can increase or decrease the song's speed.

$$x = 2^{amplitude+5} \tag{3.1}$$

Song Playback

To play a song the synthesizer wakes up 32768 times every second. Each tone is played with square waves. After calculating each tone's frequency the synthesizer turns the DAC on and off, to match the tone frequency. This method has limitations when it comes to reproducing sounds, but this method is very space efficient.

3.3.3 Sample Based Music Player

An issue with the synth is that in order to generate more advanced waveforms a program must utilize expensive trigonometric calculations to express them. Although the EFM32GG does have the capability to generate a sine wave on the DAC, in order to play actual music a more advance approach is required. To achieve this we decided to do the expensive calculations on a personal computer, using a synthesizer to sample a sine wave and output an array that could be compiled. Although the limited DAC sound quality made it possible to use a fairly low sample rate with no discernable quality difference we settled on using the maximum sample rate the LFXO clock could grant us, 32768 samples per second.

Song Representation

In order to play back sound at a given sample S with B bits accuracy the required data is S*B per second. for 12 bits accuracy at a 32768 samples per second that equates 48kB for a second of music, unfeasible for the 128kB of SRAM the EFM32GG has. Additionally, in order to simplify access to sample the 12 bit values would be loaded into 2 bytes, effectively making each 12 bit sample take 16 bits of storage. In order to save space we therefore decided to create samples of the frequencies we needed into short segments that could be repeated as long as we wanted the sound to play. To implement this idea we created an array with 100 sample pointers as a lookup table for notes, where every note was assigned its own number, initializing only the pointers that pointed to a note that would be played. A song could then be encoded as an array of instructions to access various notes and repeat the sample for a set duration.

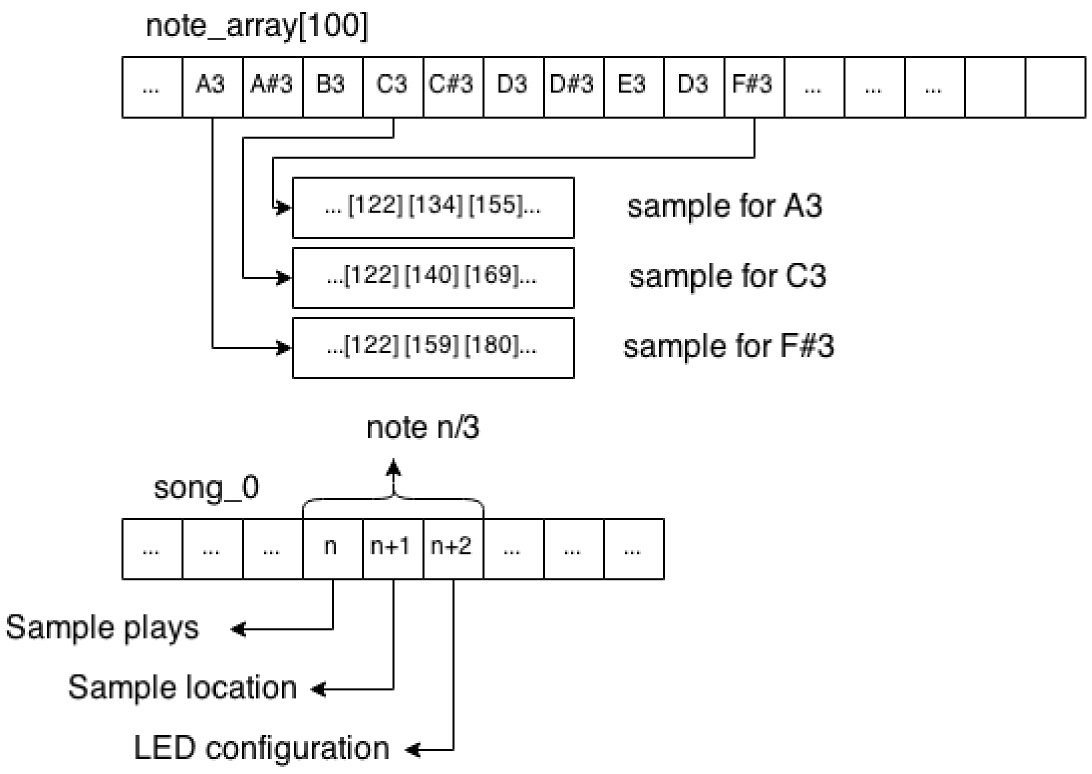

Figure 3.1: A map of the memory layout for the sample based approach

```c
typedef struct Player
{
    uint16_t song;          // Number of current song
    uint16_t song_len;      // Total notes for current song
    uint16_t notes_played;  // Keeps track of notes
    uint16_t sample_total;  // Times current sample should play
    uint16_t sample;        // Times current sample has played
    uint16_t sample_i;      // Position in current sample
    uint16_t note_len;      // Length of current sample
    uint16_t* note;         // Location of current sample
    uint16_t LED;           // Current LED configuration
} Player;
```

Song Playback

In order to play a song we bundled the necessary attributes to keep track of what notes to play and their tempo together in a Play struct. To initialize a song play_init() is called with the song to be played as the argument. This sets up a playback. After initializing calling the play() function will output the next sample to the DAC. The play() function keeps track on the current position of a playback, reading from an array of triplets to load the next note whenever the current one has been finished.

```
void play(void){
    // Load the next sample
    if(player.sample_i < player.note_len-1){
        player.sample_i++;
    }
    else{
        player.sample_i = 0;
        if(player.sample < player.sample_total){
            player.sample++;
        }
        else{
            player.sample = 0;
            if(player.notes_played < player.song_len){
                next_note();
                (*GPIO_PA_DOUT) =  player.note_len << 8 ;
            }
            else{
              disableLETIMER();
              disableDAC();
            }
        }
    }
    // Write the sample to the DAC
    *DAC0_CH0DATA = player.note[player.sample_i];
    *DAC0_CH1DATA = player.note[player.sample_i];
}
```

3.4 The Final Program

In order to string together our different approaches to a single coherent program we made a sound selector using the GPIO functionality. Different sounds were mapped to different buttons such that pressing a mapped button would start a song. In addition we wrote functionality to toggle the LETIMER and DAC on and off, as well as toggling different sleep behavior and switching between synth and sample playback. By necessity we also implemented a silencing function to make the board stop playing and power down extra functionality. The functionality of the different buttons is shown in table 3.2.

Button 1	Sample-based player
Button 2	Synthesizer (Tetris theme)
Button 3	Synthesizer (Super Mario theme)
Button 4	Sound effect
Button 5	Sound effect
Button 8	Enter EM2

Table 3.2: Button map

3.5 Optimization

In order to reduce energy consumption we had two areas to focus on, efficiency when playing and standby mode. In order to make the board go back to sleep after finishing a playback or being manually halted we implemented several toggle functions in order to put the board to sleep when not used. When actively playing we tried different energy modes to observe whether maintaining the cache in EM1 would lead to greater energy efficiency compared to going down to EM2, effectively flushing the cache.

3.5.1 Sleeping Between Interrupts

During music playback the board receives 32.768 interrupts each second. In the time between interrupts we placed the board in sleep mode. We tried different energy modes to measure the energy consumption and the result can bee seen in 4.1.

3.5.2 Disabling SRAM blocks

We disabled on-board SRAM blocks by writing to the POWERDOWN bits of the register EMU_MEMCTRL. However, we found that even disabling only a single SRAM block, the program no longer worked correctly: the sample-based player stopped working, and produced static noise instead of beautiful songs.

3.5.3 Compiler Optimizations

At one point the performance of the program went down as we added more sound effects. We tracked down the main cause to be a too large switch statement in the interrupt handler with five entries. We came up with the possible solution of turning on compiler optimizations, and adding the `-O1` flag to the compiler solved the problem. However, the additional compiler optimization broke the sample-based player again. The final solution was to reduce the size of the switch statement by pulling the most used cases out into an external conditional if-statement.

3.6 Testing

In order to test the functionality of our program we relied on visual and aural feedback rather than software approaches such as GDB. By using the LEDs we could write values directly to the LEDs, and with the earphones we could hear whether the sound was playing correctly or not. In order to measure energy consumption we used the on-board LCD screen for quick estimates. For higher quality measurements we used eAProfiler on a PC to read energy consumption on the board via the USB connection. We performed all final measurements in succession on a board that had been in use for several hours to minimize measurement variance.

4 Results

This chapter discusses our results and findings through this exercise. Our main focus is energy consumption and memory efficiency.

4.1 Energy Modes

During song playback the board consumes a significant amount of power. To decrease the consupmtion, we tried different energy modes between interrupts. We measured energy consumption in EM0, EM1 and EM2. The results can be seen in table 4.1.

	EM0	EM1	EM2
Synthesizer (Tetris theme)	4.7 mA	3.2 mA	3.2 mA
Sample-based player	4.9 mA	2.7 mA	2.1 mA

Table 4.1: Energy consumption with different energy modes

From the table 4.1, we can see that the synthesizer is equal effecient in EM1 and EM2. The sample-based player was most energy effecient in EM2. We are assuming the computation time of the synthesizer is long and the sleep time is short. Energy consumption during wake up seems to be higher at the lower energy modes and in this case makes EM1 cheaper, with consern to energy consumption. The sample-based player on the other hand contains easier computations and require less time awake. This makes the sleep time longer and the awake time shorter.

When the board was in EM2 and waiting for button interrupt the energy consumption ranged from 310-500 micro ampere.

4.2 Song Size

In order to compare the two implementations we needed to not only compare efficiency but also utility.

4.2.1 Synth Solution

The synth based approach yielded fairly small song sizes, the tetris theme totalled at 204 bytes. The song consists of two arrays with respectively 38 and 64 integers. Each integer had a size of 2 bytes.

4.2.2 Sample-based Solution

For the sample-based apprach we used samples averaging at around 1640 integers per array for 0.05 seconds. Clearly the samples were a lot more memory intensive. While sample size could be brought down by only sampling a single waveform we decided 0.05 second samples would be more interesting to work with since they would allow for variance in the waveforms, creating more interesting tones. The final size per sample ended up averaging around 3280 bytes, a significant cost compared to the synth approach. However, reuse of samples can offset this cost somewhat since playing a note twice comes at the same cost as the synth approach

4.3 Size vs Computation

The synth solution has a small memory footprint. This does however come at a cost. Since we only store small portion of the song in memory the rest has to be done on the fly. During music playback the board used about 4.9 mA. With the sample-based solution a larger array was stored in memory. This solution required less computation, and we measured 2.6 mA.

5 Conclusion

By implementing both a sample based playback and an on board synthesizer we were able to quantify the differences between the approaches. As expected the synthesizer drew more power, having to do more calculations than the sample based approach. On the other hand the synthesizer was much more space efficient, allowing much more versatility for a computational premium. In addition to learning a lot about playing sound, we also learned a lot about developing embedded code with C, having to deal with a very different way of organizing code than we were used to, coming from more high level languages.

Bibliography

[1] NTNU Computer Architecture and Design Group. Lab exercises in TDT4258 energy efficient computer systems, 2014. Available at `innsida.ntnu.no/sso/?target=itslearning`.

[2] Neil H. E. Weste and David Money Harris. *CMOS VLSI design : a circuits and systems perspective*. Pearson Addison-Wesley, fourth edition, 2011.

[3] Silicon Laboratories Inc. Efm32 application note an0004 - clock management unit, 11 2013. Available at `http://www.silabs.com/Support%20Documents/TechnicalDocs/AN0004.pdf`.

[4] Silicon Laboratories Inc. Efm32 application note an0016 - oscillator design considerations, 09 2013. Available at `http://www.silabs.com/Support%20Documents/TechnicalDocs/AN00016.pdf`.

[5] Silicon Laboratories Inc. Efm32 application note an0027 - energy optimization, 11 2013. Available at `http://www.silabs.com/Support%20Documents/TechnicalDocs/AN0027.pdf`.

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