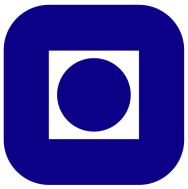

NTNU

TDT4258 - Energy Efficient Computer Systems

Exercise 2

Group 13

Mathias Ose & Øyvind Robertsen

March 10, 2014

Abstract

In this exercise, the group expanded upon the knowledge acquired in Exercise 1 [5] and learned about timer-based interrupts and generating audio using a DAC. As in the previous exercise, an ARM compatible GNU toolchain was used to program the EFM32GG development board. Both group members learned a lot about using the C language to program microcontrollers as well as about peripherals on the development board we had not previously used. To comply with exercise requirements [2, p. 42], we implemented three different sound effects and three different songs/melodies. We also implemented several abstraction layers above simply writing samples to the DAC, allowing us to "compose" new songs by listing individual notes, i.e. `C4 E4 G4 C5 E5 G5` and defining how long each note should be played. With respect to energy efficiency, we utilized the EFM32GGs energy saving modes to achieve low energy consumption whilst not playing songs.

Contents

1 Introduction

In this exercise we take the step from Assembly programming to C programming on the EFM32GG-DK3750. Having already learned how to implement simple GPIO handling in exercise 1, we now move on to the built-in Digital-Analog Converter (DAC) to generate sound. This is to be combined with GPIO input and timerbased interrupts to allow different sound effects to play on different button presses.

As in exercise 1, we will also keep in mind that a secondary but important objective of the exercise should be to try to keep energy efficiency high. Both the energy level when playing sound and the energy level while idling should be observed and measures taken to reduce it.

The primary objective of this exercise, in addition to familiarizing ourselves with new peripherals on the development board and C programming, is implementing at least three short sound effects that can be used in a game, as well as at least one song/melody that plays upon startup.

2 | Description & Methodology

This section describes the development process, use of tools, debugging techniques and details of the implementation. As listing entire implementations spread across several files for each development iteration would clutter the report and take away from the subject matter being discussed, we will only list code or formulas relevant to the task/section being described, not complete implementations. For details on the entire implementation, we refer to the source code attached to this report.

2.1 Development process

Most of the work on this excercise was carried out at the computer lab (room 458, IT-Vest, NTNU). The subject staff has equipped the lab with several workstations running Ubuntu 12.04 LTS. Each workstation is connected to an EFM32GG-DK3750 development board via USB. The toolchain necessary for cross-development is also preinstalled on each workstation. The subject staff also provided a project framework, containing amongst other things a Makefile with some convenient rules configured, a complete linkerscript and a header file containing many usefull memory addresses defined as pointer constants.

To facilitate some off-site development and to ease versioning and teamwork, the project was put under version control, see section 2.2.1. The development process itself was very iterative, with a clear goal emerging after having successfully experimented with generating sound.

2.1.1 Devices

The device used in this exercise is the same as in the previous exercise, the EFM32GG-DK3570 produced by Silicon Labs. It has a built in Digital-Analog Converter which is the primary focus of the exercise.

While the DAC is physically connected to the main processing unit, it functions like a peripheral that must be manually activated before it can be used. A standard $3.5mm$ audio jack socket is available, meaning a normal set of headphones or earplugs can be connected to listen to the DAC output.

As in exercise 1 the only source of user input is the 8-button gamepad peripheral connected to the development board's GPIO pins. In this exercise the 8 LEDs on the gamepad are not the primary means of output, but they are still available if needed.

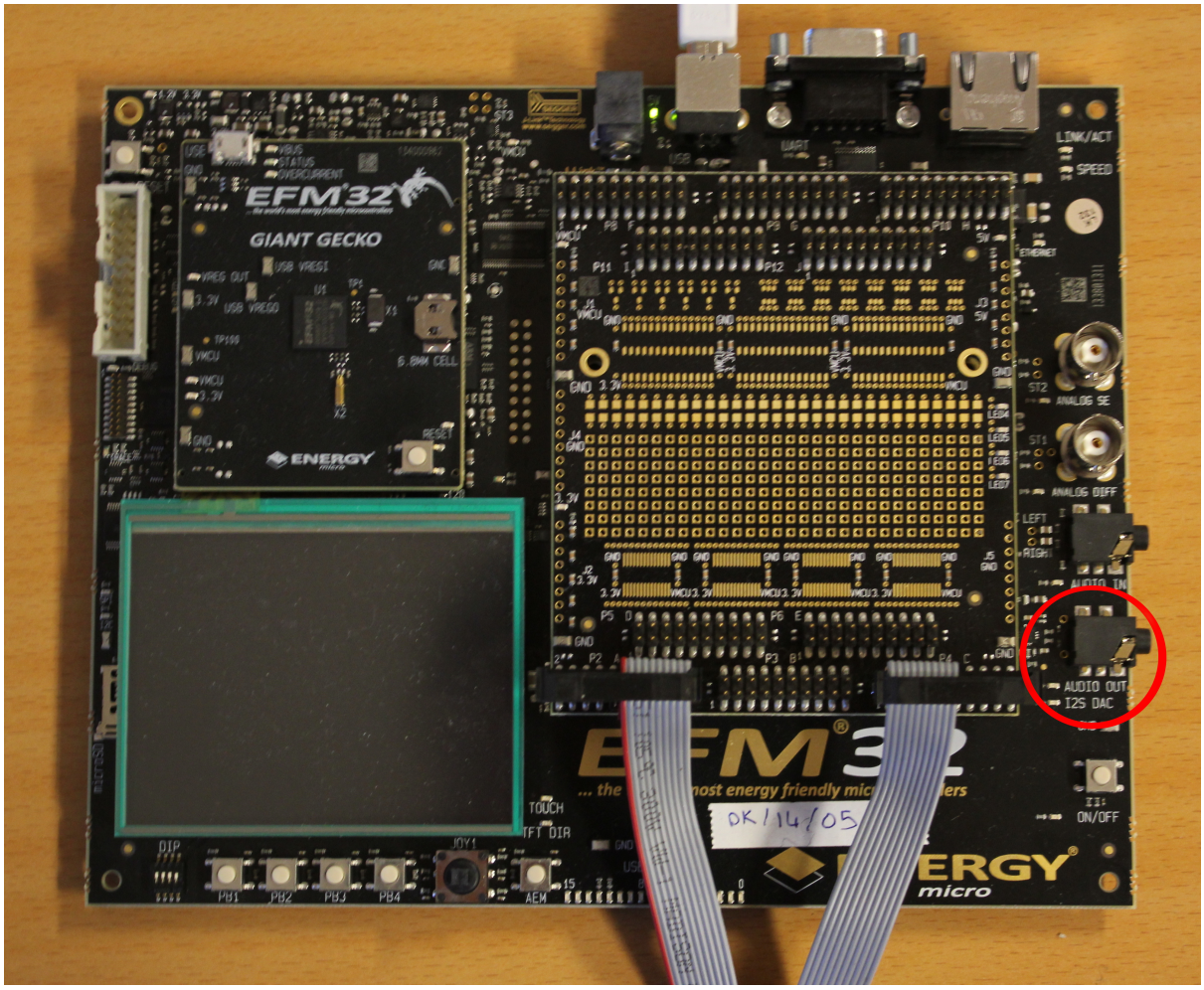

Figure 2.1: The EFM32GG-DK3570. Highlighted with red is the audio output from the DAC. The gamepad peripheral is connected by the cables emerging on the bottom of the picture.

2.2 Project setup & toolchain

The sample code handed out by he subject staff had all files gathered in the same directory. Being avid supporters of neatly organized code, we chose to restructure the project layout. Listing 2.1 shows a general outline of the directory structure we opted for. Later in the development process, an additional `support` directory was added to contain some Python code for pre-generating the source data for sound.

Listing 2.1: Revised project structure

```
exercise2
|-- build
|   |-- <compiled and assembled object files>
|-- exe
|   |-- <binary/executable build artefacts>
|-- Makefile
|-- lib
|   |-- efm32gg.ld
|   |-- libefm32gg.a
|-- report
|   |-- <Report files>
|-- src
|   |-- dac.c
|   |-- efm32gg.h
|   |-- gpio.c
|   |-- interrupt_handlers.c
|   |-- main.c
|   |-- timer.c
|   |-- wfi.s
|-- support
|-- music.py
    |-- scale.py
```

The Makefile was also amended to support this new layout, and a `make run` rule was added, simplifying the build process.

2.2.1 Tools

Cross-development toolchain

As mentioned earlier, the workstations at the lab are pre-equipped with the tools required for cross-development of ARM-based microcontrollers. A version of the GNU development toolchain tailored specifically for ARM-based build targets was used for compiling/assembling, linking and copying throughout the development process. Specifically, the GNU AS [5, p. 4] utility was used for assembling ASM source files into object files, the GNU LD [5, p. 4] for linking objectfiles and creating a debuggable executable, GNU Objcopy [5, p. 5] for creating a clean, flashable binary from the executable made by LD and last but not least, GNU Make [5, p. 5] for automating the entire build process. The utilities mentioned thus far as well as their usage was described in great detail in our report from exercise 1.

Since code in this exercise would mostly be written in C, a new utility was introduced, namely GNU CC (henceforth referred to as GCC) for compiling/assembling C source files into object files which could then be linked into the final executables/binaries. It's usage is described in listing 2.2. Compiling C source with GCC actually consists of GCC compiling our C code into it's Assembly equivalent, with varying levels of optimization added, followed by GCC using AS to assemble the generated ASM source. Among the flags used for compiling this project, `-std=c99`, defining which C version we're using, and `-mcpu=cortex-m3 -mthumb`, defining which architecture we're targeting are worth noting.

The build process differs from the one in the previous exercise in one more crucial

aspect; the linking step. Since we're programming in C and not explicitly defining a reset subrouting and a interrupt vector table as in exercise 1, some startup code must be linked into our program. GCC is clever, and allows us to use it for linking as well. Linking with GCC provides some useful features over linking with LD directly, such as easier dynamic linking. Using GCC for linking is shown in listing 2.3. The flags used in the linker step that are worth noting are `-lcs3 -lcs3unhosted`, which and responsible for linking the necessary CodeSourcery startup library code into our executable, `-lefm32gg` which links the constants defined in `efm32gg.h` into our program and `-lc` which tells GCC to look for functions not defined in our source in the system-supplied C standard libraries.

Listing 2.2: GCC compilation usage

```
arm-none-eabi-gcc -mcpu=cortex-m3 -mthumb -g -std=c99 -Wall -c
    ↪ <source_file.c> -o <output_file.o>
```

Listing 2.3: GCC linking usage

```
arm-none-eabi-gcc -T <linkerscript.ld> [<arg.o>] -o <output.elf>
    ↪ -mcpu=cortex-m3 -mthumb -g -lgcc -lc -lcs3 -lcs3unhosted
    ↪ -lefm32gg -Llib
```

Other tools

- Vim as our main editor
- Git for version control, and GitHub as a centralized repository host
- LaTeX for typesetting the report
- Python 2.7 for generating samples, see section 2.5

2.3 Reimplementing exercise 1

As recommended by the compendium, we started exercise 2 by reimplementing exercise 1 in C. This was for the most part a breeze, since it is possible to do inline arithmetic and R/W operations are much simpler. For example, the single line of C in listing 2.4 is equivalent to something like 5 lines of Assembly at the bare minimum, as seen in listing 2.5.

Listing 2.4: C

```
*GPIO_PA_DOUT = *GPIO_PC_DIN << 8;
```

Listing 2.5: ASM

```
LDR R4, =GPIO_PA_BASE
LDR R5, =GPIO_PC_BASE
LDR R8, [R4, #GPIO_DIN]
LSL R8, R8, #8
STR R8, [R5, #GPIO_DOUT]
```

However, there was one part that was not easier in C than in Assembly. We were made aware that the WFI instruction was not as readily available on this level as one might have hoped. In order to make the development board sleep, we would still have to use that instruction directly as an Assembly instruction.

To achieve this we had a couple options. One was using inline Assembly to call the single instruction. The other option, which we opted for, was to create an Assembly file containing only a subroutine consisting only of the single instruction, called simply wfi. Then a function prototype `void wfi(void)` was declared in the C code, and the Makefile edited to include compilation of the Assembly file and the linking together of this new compiled file with the others. Thus the WFI instruction became available in C as `wfi()`.

Listings 2.6, 2.7, 2.8 and 2.10 show the Assembly and C code necessary for a reimplementation of our Assembly code from exercise 1.

Listing 2.6: wfi.s

```
.global wfi
.thumb_func
wfi:
    WFI
```

Listing 2.7: main.c

```
#include <stdint.h>
#include "efm32gg.h"

/* Function prototypes */
void wfi(void);
void setupNVIC(void);

int main(void) {
    /* call GPIO setup function */
    setupGPIO();
    /* Enable interrupt handlers */
    setupNVIC();
    /* Go to sleep*/
    wfi();
    return;
}

void setupNVIC() {
    /* Enable interrupt handlin for GPIO IRQs */
    *ISER0 = 0x802;
}
```

Listing 2.8: gpio.c

```
void setupGPIO() {
    *CMU_HFPERCLKEN0 |= CMU2_HFPERCLKEN0_GPIO; /* enable GPIO clock*/
    *GPIO_PA_CTRL = 2; /* set high drive strength */
    *GPIO_PA_MODEH = 0x55555555; /* set pins A8-15 as output */
    *GPIO_PC_MODEL = 0x33333333; /* Set pins C0-7 as input */
    *GPIO_PC_DOUT = 0xFF; /* Enable internal pull-up*/
    /* Interrupt config */
    *GPIO_EXTIPSELL = 0x22222222;
    *GPIO_EXTIFALL = 0xFF;
    *GPIO_EXTIRISE = 0xFF;
    *GPIO_IEN = 0xFF;
    *GPIO_PA_DOUT = 0xFF00;
}
```

Listing 2.9: interrupt_handlers.c

```
#include <stdint.h>
#include <stdbool.h>
#include "efm32gg.h"

/* Commmon GPIO_Handler */
void GPIO_Handler() {
    /* Clear interrupt flag */
    *GPIO_IFC = *GPIO_IF;
    /* Shift input 8 bits left to use as output */
    *GPIO_PA_DOUT = *GPIO_PC_DIN << 8;
}

/* GPIO even pin interrupt handler */
void __attribute__ ((interrupt)) GPIO_EVEN_IRQHandler() {
    GPIO_Handler();
}

/* GPIO odd pin interrupt handler */
void __attribute__ ((interrupt)) GPIO_ODD_IRQHandler() {
    GPIO_Handler();
}
```

In the end, it turns out the `wfi()` instruction was actually not needed, because we set up the system to go to sleep automatically upon returning from an interrupt handler. Since an unintended interrupt always occurs at restart, the system goes to deep sleep anyway as soon as this interrupt has been handled. Never the less, learning to integrate Assembly instructions in C was a useful lesson, and if we ever do figure out what causes the interrupt on reset and manage to disable it, we will need `wfi()` again.

Listing 2.10 shows the function we used in the final program to write to the System Control Register.

Listing 2.10: Writing 1 to the second bit of the System Control Register enables automatic sleep on return from interrupt handlers.

```
void setupSleep(int arg) {
    *SCR = arg;
}
```

2.4 Configuring the DAC and the timer

As with the GPIO controller, both the the DAC and the timer has to be configured before they can be used. With guidance from the compendium [2], this was for the most part smooth sailing.

2.4.1 DAC

Setting the DAC up for use consists of three short steps. We gathered these steps in a function `setupDAC()` in the `dac.c` file. First up is enabling the DAC clock. This is done by enabling bit 17 in the High Frequency Peripheral Clock Enable 0 register (CMU_HFPERCLKEN0), as seen in listing 2.11

Listing 2.11: Enabling DAC clock

```
*CMU_HFPERCLKEN0 |= (1 << 17);
```

Next up is configuring the DAC control register to prescale the DAC clock based on the HFPERCLK frequency and enabling output to the amplifier in continuous mode. The prescaling part of this is done by writing to the PRESC bits in the control register. The prescaling formula is shown below. Listing 2.12 shows the configuration itself.

$$f_{DAC} = f_{HFPERCLK} * \frac{1}{2^{PRESC}}$$

Listing 2.12: Configuring

```
*DAC0_CTRL = 0x50010;
```

The final configuration step for the DAC is enabling output on one or both audio channels. How to accomplish this by writing to the channel control registers is shown in listing 2.13

Listing 2.13: Enabling output to both audio channels

```
*DAC0_CH0CTRL = 1;
*DAC0_CH1CTRL = 1;
```

To test DAC functionality, we fed a countinuous stream of varying data to the DAC data register using a for-loop in the GPIO interrupthandlers. This way, we could have a sound play each time one of the buttons on the gamepad was pressed. Using this technique blocks the CPU from doing anything else while the handler is still running, so the need for timer-based interrupts presented itself quite clearly.

It is also worth mentioning that we later created a corresponding `disableDAC()` function for convenience, which reverts `enableDAC()` in order to prevent the DAC from using power.

2.4.2 Timer

To make non-blocking sound-generating possible, we would need a timer. The EFM32GG has several timer peripherals available, all of which can be configured to generate interrupt requests at given intervals. As long as we could keep the handler for those interrupts short and optimized, playing one sample each time a timer interrupt is generated would allow the processor to perform other work in between each timer interrupt.

As suggested in the compendium [2, p. 42], we wanted to play 44100 samples per second. The core clock runs at $14Mhz$, so to accomplish this, we would need the timer to generate an interrupt every 317 clock cycles (see formula 2.1).

$$\Delta C = \frac{f_{HFCORECLOCK}}{f_{samplerate}} = \frac{14000000}{44100} \approx 317 \tag{2.1}$$

The procedure for configuring the timer (TIMER1) was well documented in the compendium [2, p. 40], and the steps are outlined below. We gathered all the configuration steps into a `setupTimer()` function in `timer.c`. A corresponding `disableTimer()` function was also made.

First of all, the timer must be enabled in the clock management unit (CMU). As usual, a bit, in this case bit 6, has to be set in the High Frequency Peripheral Clock Enable 0 register (HFPERCLKEN0). See listing 2.14

Listing 2.14: Enabling the timer in the CMU

```
*CMU_HFPERCLKEN0 |= CMU2_HFPERCLKEN0_TIMER1;
```

The Timer Counter Top Value register (TIMERn_TOP) is used to configure the interval at which the timer generates interrupts. Listng 2.15 shows how this is done with an interrupt interval of 317 clock cycles.

Listing 2.15: Setting interrupt interval

```
*TIMER1_TOP = 317;
```

Lastly, as with the GPIO interrupts, interrupt generating has to be enabled in the Interrupt Enable register (TIMERn_IEN). This is shown in listing 2.16

Listing 2.16: Interrupt generation

```
*TIMER1_IEN = 1;
```

2.5 Generating sound

After spending a lot of time testing the DAC and figuring out what we could and could not do with the microcontroller and DAC itself, we had to decide on what to implement.

One thing became apparent was that it would not be feasible to do sine wave manipulations on the fly, as floating point operations are slow on the EFM32GG and would not be able to keep up. We instead opted to pre-generate samples of sine waves of different frequencies on a PC. The different samples could then be loaded into the memory on the development board and used to produce sound.

With this setup we can easily create long songs as sequences of reusable notes instead of one long continuous sample. In other words to play the sequence "C D C D C" for 5 seconds we would not need a sequence of five seconds worth of alternating samples, just two short sets of samples, play one repeatedly for 1 second, then switch to the next note and play that for one second, and so on.

2.5.1 Sample generator script

The sample generator script was written in Python for convenience. It uses a dictionary with note-frequency as key-value pairs, sourced from the equal-tempered scale tuned to $A4 = 440Hz$ [3].

The script is fed strings with sequences of note names, a sort of primitive musical sheet, looks up the frequencies of the relevant notes and for each generate samples for one period of a sine wave of that frequency when played at $44.1kHz$. The samples are then scaled and shifted to produce a wave oscillating between `0` and `0xFF`.

Listing 2.17 shows the Python function that takes values from the Python `math.sin()` function and formats them to the specifications required by the DAC. Figure 2.2 shows the generated samples for the notes A4 and B4. Notice how a different number of samples is produced for different notes. As the frequency of a note increases, the period of the corresponding wave decreases.

Listing 2.17: Sample generation

```
def sine_samples(frequency=440.0, framerate=44100):
    period = int(framerate / frequency)
    steps = range(period)
    framerate = float(framerate)

    k = 2.0 * pi * frequency
    l = [sin(k * float(i % period) / framerate) for i in steps]
    l = [0.5 * x for x in l]
    l = [x + 0.5 for x in l]
    l = [0xFF * x for x in l]
    l = [int(x) for x in l]

    return l
```

The use of 8 bits instead of 12 was deliberate and seemed entirely advantageous. Since the sine curve the samples produce is so simple, there was no apparent difference in sound quality when decreasing from 12 bit samples to 8 bits. This also allowed us to store the samples as `uint8_t` instead of `uint16_t`, halving the memory required for storing music.

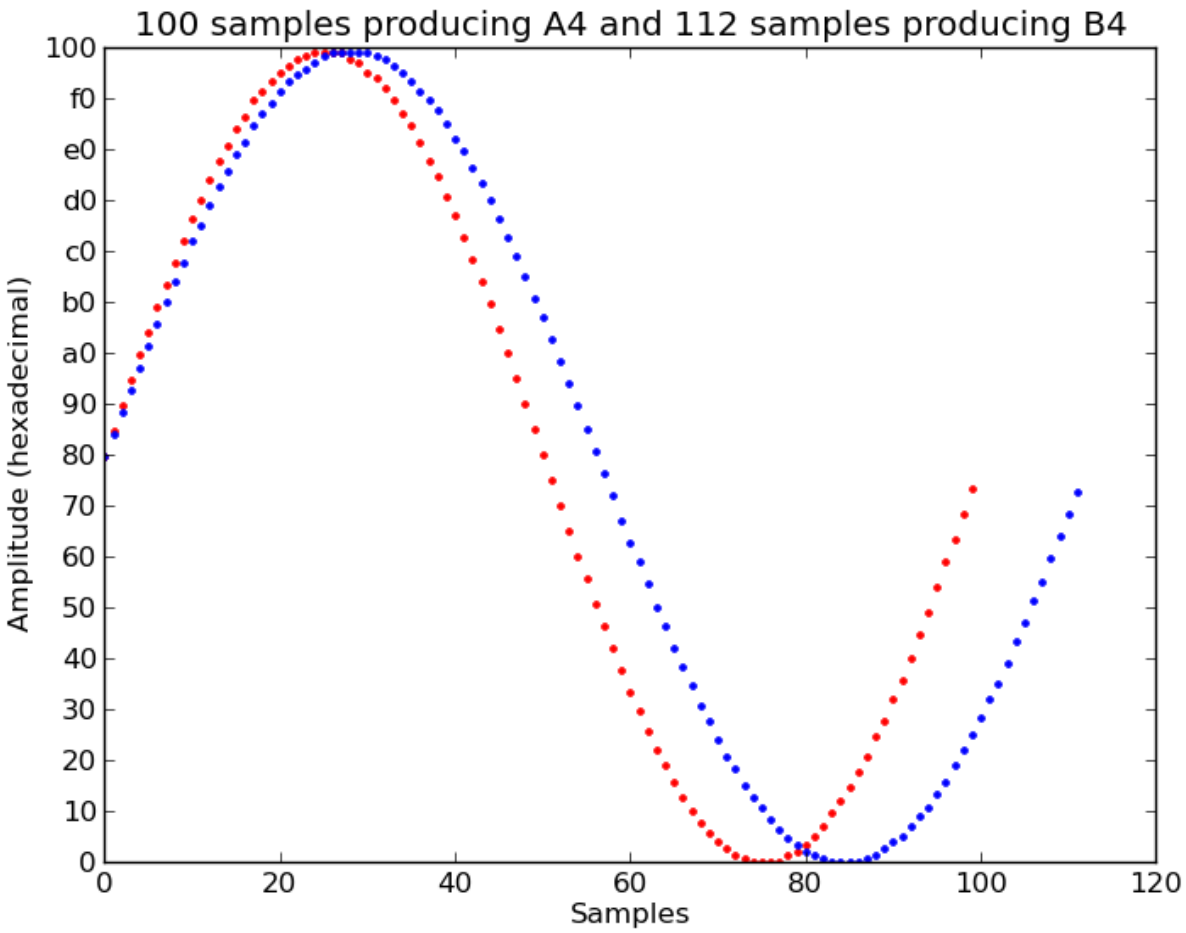

Figure 2.2: Plot of samples producing the notes A4 (red) and B4 (blue) when played at $44.1kHz$.

The script was made to print notes and songs in the form of C structs, as well as supporting variables and functions for playing music. Declarations were written to a header file and implementations to a .c file. The script could then be run directly on the command line, as well as added to the beginning of the `make run` routine, integrating the Python script with the rest of the flashing process.

Listing 2.18: Excerpt from music.h

```
typedef struct Note {
    uint16_t num;
    uint8_t samples[];
} Note;

typedef struct Song {
    uint8_t length;
    Note* notes[];
} Song;
```

Listing 2.19: Excerpts from music.c

```
Note A4 = { 100, { 127, 135, 143, 151, 159, 166, 174, 181, 188, 195,
    202, 208, 214, 220, 225, 230, 234, 239, 242, 245, 248, 250, 252, 253,
     254, 254, 254, 254, 252, 251, 248, 246, 243, 239, 235, 231, 226,
    220, 215, 209, 203, 196, 189, 182, 175, 167, 160, 152, 144, 136, 128,
     120, 112, 104, 96, 89, 81, 74, 67, 60, 53, 47, 41, 35, 30, 25, 20,
    16, 12, 9, 6, 4, 2, 1, 0, 0, 0, 0, 2, 3, 5, 8, 11, 15, 19, 23, 28,
    33, 39, 44, 51, 57, 64, 71, 78, 86, 94, 101, 109, 117 } };

Song CANON = { 52, {&A5, &FS5, &D5, &A4, &FS4, &D4, &A3, &CS4, &E4, &A4,
     &CS5, &E5, &FS5, &D5, &B4, &FS4, &D4, &B3, &FS3, &A3, &CS4, &FS4, &
    A4, &CS5, &D5, &B4, &G4, &D4, &B3, &G3, &D3, &FS3, &A3, &D4, &FS4, &
    A4, &B4, &G4, &D4, &B3, &G3, &D3, &A3, &CS4, &E4, &A4, &CS5, &E5, &A5
    , &A5, &A5, &A5} };
```

With this setup we created six "songs". Three short sound effects which could be used in a game setting, three were longer melodies adapted from existing works, namely "Binary Sunset" by John Williams, "Sweet Child of Mine" by Guns N' Roses and "Canon in D" by Johann Pachelbel.

2.5.2 Playing the sounds

The system was configuired to idle in deep sleep mode with no peripherals active except the GPIO listener.

Once an even or odd GPIO interrupt is registered, the timer interrupt generator and DAC are enabled, depending on what the input was from the GPIO peripheral, one of the songs will be selected based on the input with a helper function. Since the input peripheral has eight buttons and we only used six different melodies, the last two buttons were used to play "Canon" at slower and faster rates.

Listing 2.20: Helper function for mapping input to song number

```
int map_input() {
    int input = ~(*GPIO_PC_DIN);
    for ( int i = 0; i < 8; i++) {
        int match = input & (1 << i);
        if ( (1 << i) == match ) {
            return (i+1);
        }
    }
    return 0;
}
```

Listing 2.21: GPIO handling from interrupt_handlers.c

```
void GPIO_Handler() {
    timer_cleanup();
    GPIO_interrupt_clear();
    GPIO_LED(); // will stay on until end of song

    int SW = map_input();
    if (SW == 1) {
        playSong(&JUMP, 0x027F);
    } else if ( SW == 2) {
        playSong(&PEWPEW, 0x3FF);
    } else if ( SW == 3) {
        playSong(&ONEUP, 0x71f);
    } else if (SW == 4) {
        playSong(&THATSNOMOON, 0x3FFF);
    } else if (SW == 5) {
        playSong(&SCOM, 0x24FF);
    } else if (SW == 6) {
        playSong(&CANON, 0x17FF);
    } else if (SW == 7) {
        playSong(&CANON, 0x5FFF);
    } else if (SW == 8) {
        playSong(&CANON, 0xFFF);
    } else {
        playSong(&SCOM, 0x24FF);
    }
}

void __attribute__ ((interrupt)) GPIO_EVEN_IRQHandler() {
    GPIO_Handler();
}

void __attribute__ ((interrupt)) GPIO_ODD_IRQHandler() {
    GPIO_Handler();
}
```

When `playSong()` is called, the song reference and playback speed is written to static variables declared in `music.h`. Then deep sleep mode is disabled, the DAC and timer interrupt generator is enabled, and the timer is started.

Listing 2.22: Excerpts from music.c

```
void setSong(Song* song, uint16_t note_length) {
    current_song = song;
    current_note_length = note_length;
}

void playSong(Song* song, uint16_t note_length) {
    setSong(song, note_length);

    setupSleep(0b010);
    setupDAC();
    setupTimer();
    startTimer();
}
```

The timer interrupt generator will trigger an interrupt every 317th clock cycle (see formula 2.1). In the timer interrupt handler several counters are managed. `c` keeps track of whether it's time to start playing the next note. `note_c` keeps track of whether the end of the song has been reached yet.

`i` is used to select the next sample to be played every interrupt. The current note and sample index (offset from 0) to be played is played with the `note()` function which plays the same sample on both DAC channels.

When the end of the song is reached, the function disables the timer and DAC, turns of the LEDs, then sets up deep sleep and returns from the function, which activates deep sleep.

Listing 2.23: Timer interrupt handler from interrupt_handlers.c

```
void __attribute__ ((interrupt)) TIMER1_IRQHandler() {
    /* Clear interrupt flag */
    *TIMER1_IFC = 1;

    if ( c >= current_note_length ) {
        c = 0;
        note_c++;
    } else {
        c++;
    }

    if ( note_c >= current_song->length ) {
        timer_cleanup();
        *GPIO_PA_DOUT = 0xFFFF;
        disableDAC();
        disableTimer();
        setupSleep(0b110);
        return;
    }

    Note* n = current_song->notes[note_c];
    int offset = (i % n->num);
    note(n, offset);

    i++;
}
```

The program would easily allow for different samples to be played on the two channels simultaneously. But musically it was not easy to make this sound good, so we opted to play the same on both channels.

Listing 2.24: Convenience functions for playing samples

```
void note0(Note* n, int offset) {
    *DAC0_CH0DATA = n->samples[offset];
}

void note1(Note* n, int offset) {
    *DAC0_CH1DATA = n->samples[offset];
}

void note(Note* n, int offset) {
    note0(n, offset);
    note1(n, offset);
}
```

3 Results

3.1 Program

The final program features three effects and three melodies. The 8 buttons on the gamepad each play one sound, with three of the buttons playing the same melody at different speeds. Only one sound can be playing at any time, and a LED will stay active as long as the sound is playing, indicating which sound is playing.

3.1.1 Testing the program

All tests require possession of an EFM32GG-DK3750, a purpose built gamepad connected to the board on GPIO ports A and C and a computer capable of compiling for ARM based platforms and flashing software to the development board. In addition some sort of analog speaker with a $3.5mm$ stereo audio plug must be connected to the board in order to hear the output.

When calling `make run`, `support/music.py` will be run as a Python program first, and it is important that the PC is set to run it as Python 2.7 code. Alternately, the Python command could be removed from the Makefile, as the finished output of the script has been included with the source code.

Trigger sound by button test

This procedure tests the implemented functionality of our program. Each button should trigger a sound unique to that button press. The three highest numbered buttons trigger the same note sequence, but at different speeds.

Procedure:

1. Compile and flash the program to the board by running `make run` from the exercise2 directory

2. Wait for the board to reset properly. On reset the board should play a melody, then go silent. No LEDs should be active, and the energy monitoring screen should be fluctuating around $1.9uA$.

3. Test pressing different buttons. Each should play a different sound, and an LED should be active as long as the sound is playing, indicating which button was pressed.

3.2 Energy efficiency

As in exercise 1, when discussing energy efficiency, we will be focusing on amperage, since it's more relateable to, among other things, battery capacity.

3.2.1 Readings

In the final iteration of our implementation, we were able to have the microcontroller enter energy mode 4 with no extra peripherals enabled, allowing it to idle at about $2\mu A$. Figure 3.1 shows a log-scale plot of the amperage while idle and figure 3.2 shows the active amperage.

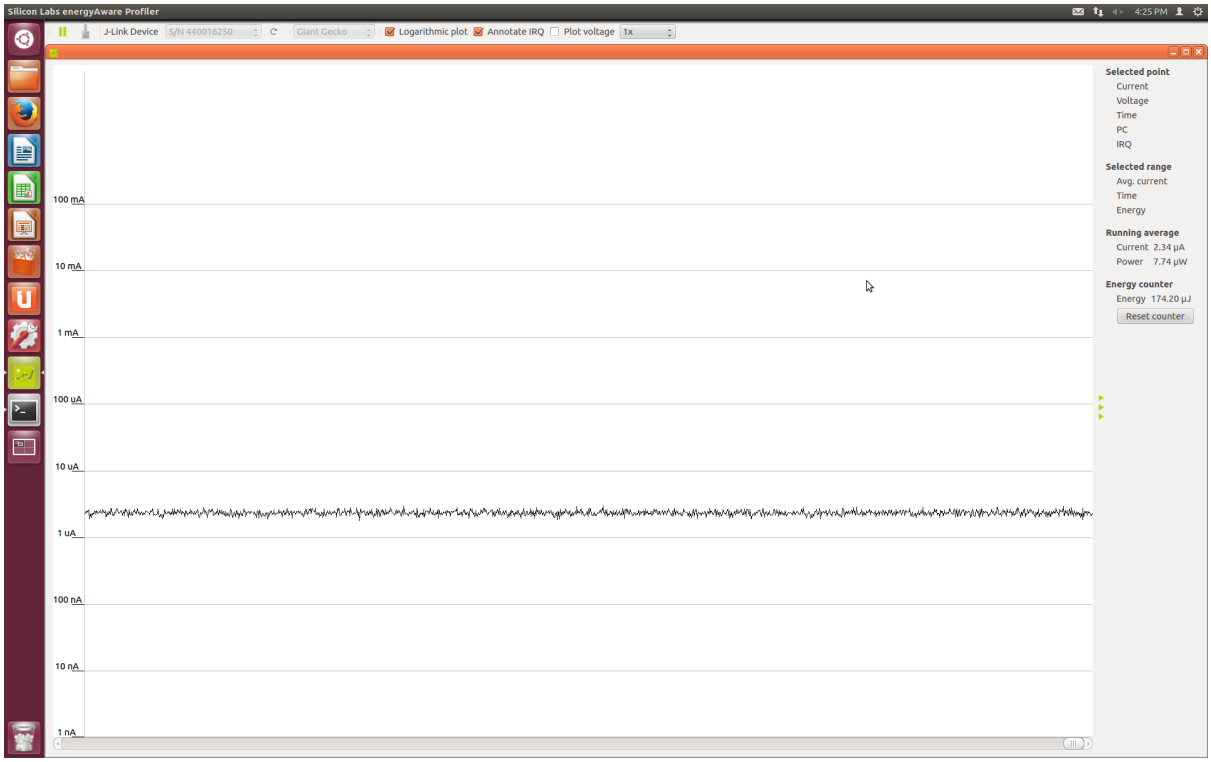

Figure 3.1: Idle amperage plotted by the eAProfiler tool.

3.2.2 Expected lifetime on a CR2032 battery

In our report for exercise 1, we used the CR2032 battery as an example in the energy efficiency section. [5, p. 13] Since our idle power consumption is at the same level in this exercise as in the previous, our conclusion from that report still stands; in idle mode, the battery will deteriorate before discharging from the microcontrollers usage. This time however, the functionality of our program is rather different, and it spends a larger amount of time not idling. As can be seen from figure 3.2, the amperage while playing is around $3.5mA$. Using the same reference battery capacity [1] as in exercise 1, and the playing amperage, formula 3.1 gives us an expected lifetime with continuous playing, of about 2 days, 20 hours.

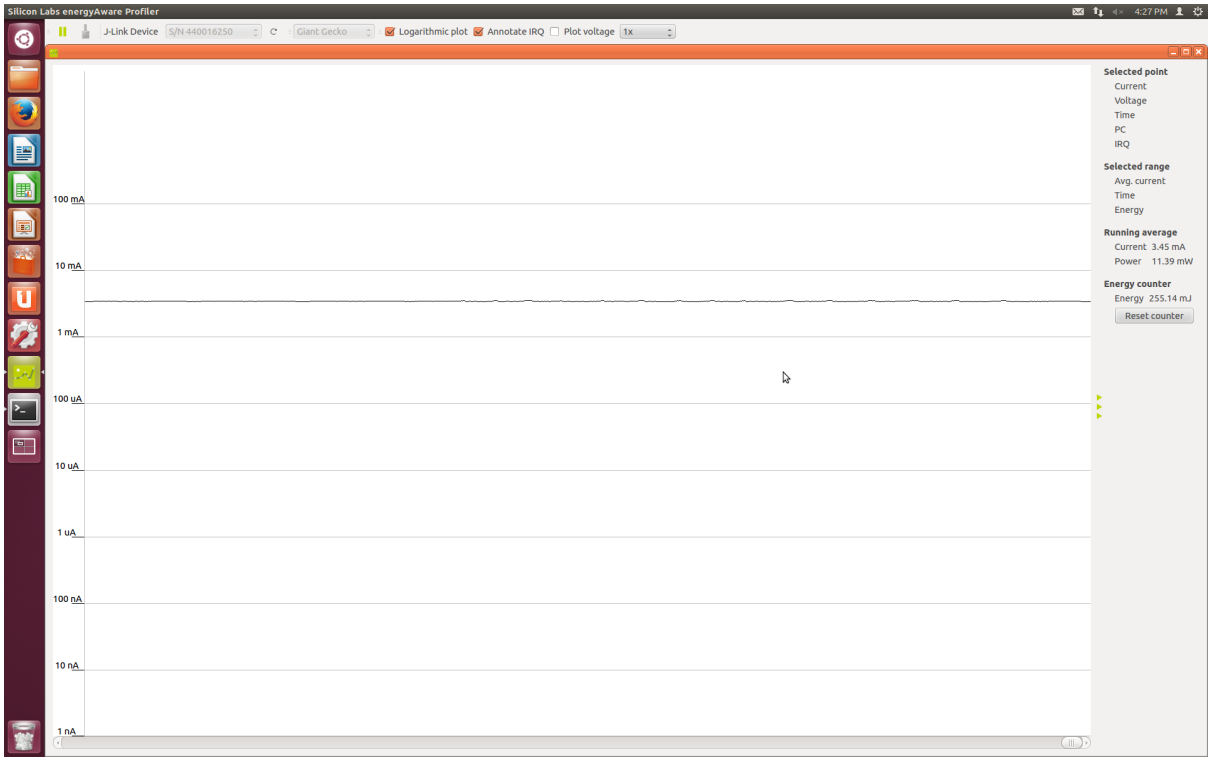

Figure 3.2: Active amperage plotted by the eAProfiler tool.

$$\frac{240mAh}{3.5mA} = 68.6hours \tag{3.1}$$

3.3 Discussion

The final iteration of our program implements a simple way of playing both short sound effects and longer musical pieces, with low power consumption while idle to boot. We learned a lot about general sound theory, DAC usage, microcontroller programming in C and about many of the relevant peripherals on the EFM32GG. Our implementation lacked in a few places however, namely energy efficiency while playing and variation of sound.

Active energy consumption

As described in section 2.5.2, we chose to disable deepsleep while each song plays. The result of this is an average amperage of $3.5mA$ while playing sounds, which is rather high. This could have been improved in several ways, given more time.

One possible way to improve active power consumption would have been to use a dfferent timer. The regular timer works along these lines; the timer incrementes its counter value once every clock cycle until it reaches the value set in the TOP register. Since the timer relies on the clock cycles from the High Frequency Peripheral Clock, this timer must be run in energy mode 0. We spent some time investigating other possibilities, amongst them using the low energy timer wich can be used while in energy mode 1 or 2, seeing as it uses one of the low frequency clocks as a source. These clocks run at

$32.768kHz$ or lower, so opting to use the low energy timer would mean sacrificing sound resolution for energy efficiency.

In the compendium the possibility of using Direct Memory Access (DMA) was mentioned as a possible method to reduce power consumption. We looked into this, but did not implement it, as our program relies on the CPU to change the current playing note. If the program had instead played songs as long continuous sample sequences, DMA would be useful as the CPU could have longer periods of time idling.

Finally, we could also have disable any unused ram, using the memory system controller (MSC). We investigated how to do this towards the end of the development process, but opted for putting in more hours working on this report instead.

Sound features

Our implementation has several convenient features, among them a very simple system for adding new songs and controlling how long a song lasts. However, as our system only implements sine waves, most music implemented, while being tonally perfect, will have little room for variation. Given more time, we could have implemented a more sophisticated system for representing songs, involving additional wavetypes (i.e. square and sawtooth) and tweakable parameters.

Unintended GPIO interrupts

As mentioned at the end of section 2.3, an unwanted even GPIO interrupt occured on every restart. We unsuccesfully tried getting it to go away, then experimented with ignoring the interrupt when no buttons were pressed. The input mapping function (listing 2.20) was written for this.

In the end, we decided to call it a feature rather than a bug and used the interrupt as our way of playing a startup song.

4 | Evaluation of assigment

This exercise provided a lot of valuable experience for both group members. Expanding on the knowledge we acquired in exercise 1, it posed enough of a challenge to be interesting, but not so much that it became overwhelming and felt unfeasible. Having some expreience with C(++) from the TDT4102 course, the exercise allowed us to use and learn more about using the C language for programming microcontrollers.

The subject staff provided valuable insight and aid during the tuition hours. One criticism that has been mentioned to the staff is the heat situation in the lab. The extreme temperature that develops by only being a few people in the room at the same time makes it really hard to work over long time periods, possibly discouraging creativity and innovation.

5 Conclusion

Both the practical goal and the learning goals of the exercise were achieved. Additional improvements to the program could have been implemented but we ran out of time. These have been mentioned in the discussion instead, so we have hopefully learned the theory of those if not how to do it practically.

Neither group member really had much relevant experience with timers, DACs, or the physics of sound in general, but the compendium [2] and the EFM32GG Reference Manual [4] were very helpful for understanding, and with enough time to consider the material most of the work was very straightforward and easy to understand.

References

[1] Anonymous. Cr2032 lithium battery. http://www.cr2032battery.org/cr2032-lithium-battery/, April 2010.

[2] Computer Architecture and Design Group. Lab exercises in tdt4258 energy efficient computer systems. Technical report, Department of Computer and Information Science, NTNU, 2014.

[3] B. H. Suits, Physics Department, Michigan Technological University. Frequencies for equal-tempered scale, a4 = 440 hz. http://www.phy.mtu.edu/~suits/notefreqs.html, 1998.

[4] Silicon Labs. Efm32gg reference manual, October 2013.

[5] Mathias Ose, Øyvind Robertsen, and Jørn-Egil Jensen. Exercise 1 - buttons and leds. Technical report, Department of Computer and Information Science, NTNU, 2014.