NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4258 Low-Level Programming
Laboratory Report

Exercise 2

Group 23:

Maciej Ambrozek
Matej Mnoucek
Jonas Peis
Emanuel Roos

October 18, 2019

1 Overview

The goal of this exercise was to utilize the DAC of EFM32 board and use it to play some sound effects. The main difference between this exercise and the Exercise 1 is that the mentioned tasks were completed in C programming language and were programmed directly without any help from an operating system. There also was a focus on code style and the overall code quality. In order to keep the level of it acceptable, the Linux coding style is enforced. Furthermore, each supplied function and the description of it's purpose is documented in comments.

The solution consists of three projects. The first project is called **Generator**, is programmed in Python and serves the purpose of generating sound samples. We wanted to test a wide variety of sounds and therefore great effort was put in this part. The generator is able to produce samples from *sine wave*, *square wave* and *sawtooth wave*, can compose a melody from note notation (e. g. sequences like A4, C5, G3...), play it in mentioned waves and also convert a standard *.wav* file into samples. The sample rate and amplitude parameters are customizable. The generator outputs a standard C header file (the format is shown in 1.0.1) that can be included in the C code and used as a sample source for the program. This project serves as a base for the baseline solution and also the improved solution. We have also experimented with generating samples on the chip or even on the fly but that turned out to be inefficient, too computationally heavy and difficult to maintain/develop further.

```c
typedef struct {
   uint16_t sampleRate;
   uint16_t sampleCount;
   uint8_t samples[];
} Sound;
```

Source code 1.0.1: The format of structs with samples that generator outputs

Baseline solution, similarly to the previous exercise, uses infinite loop and polling which continuously checks for the current time and moves samples from memory to DAC registers accordingly. The time is supplied by EFM32's hardware timer. In addi-

Button SW1	Scooter - Maria
Button SW2	Deep purple - Smoke on the Water
Button SW3	Laugh
Button SW4	Scream
Button SW5	Explosion
Button SW6 Button SW7 Button SW8	Stops audio output

Figure 1.1: Table of what every button does

tion the playback of several supplied sounds is controlled by GPIO inputs i.e. buttons. The available sounds are: *Scooter - Maria* in sawtooth wave, *Deep purple - Smoke on the Water* in sine wave and three resampled wav files that plays various sounds (laugh, scream and explosion). After bootup the board remains silent and user can play a sound by pressing and holding one of the first fives buttons. The corresponding LED is also lit when a button is pressed and serves as visual feedback to the user. All the states are described in Figure 1.1.

In addition to the baseline solution, **Improved solution** with a focus on energy efficiency is delivered as well. The main difference is that it uses interrupts instead of continuous polling and also several other optimisations that help to further reduce energy consumption. It also includes the same sound set as the first board but there is a slight difference in functionality. After pressing one of the first five buttons the corresponding sound plays and keeps repeating forever. Therefore, any of the remaining buttons can be used to deactivate the sound.

1.1 Baseline Solution

As it is mentioned in the **Introduction** chapter baseline solution is build around polling. The essence of creating programs for EFM32 in C is still the same like it was while programming in Assembly. Most of the interaction between the chip and code is done by reading from or writing to a register. Hence, this operation is crucial for our solution. Thankfully, using registers from the C code is really straightforward and we have also been provided with a header file that contains memory addresses of most of the important registers. See Source code 1.1.1.

In order to use GPIO for detecting button presses and also output to LEDs it was necessary to perform similar setup like in Exercise 1. However, in addition to it, setting up hardware timer for in-time sample delivery and also configuring DAC in order to output sound was neccessary. The hardware timer and DAC setup code can be seen in 1.1.2.

```c
// Writting to a register
*GPIO_PC_DOUT = 0xff;
// Reading from a register
uint32_t value = *GPIO_PC_DIN;
```

Source code 1.1.1: Example of reading and writing to a register

The main polling logic takes place in the *ex2.c* file. Thanks to the fact that reading from timer counter register `TIMER1_CNT` worked unreliably, we came up with an alternative solution. Instead of reading `TIMER1_CNT` we detect timer overflow from `TIMER1_IF` register. Then at each overflow we supply a new sample from the pre-generated arrays to the DAC. The polling loop also includes a call to function `playSound()` which checks button states and plays corresponding sounds.

1.2 Improved Solution

The main difference between **Basic** and **Improved** solution is ditching the energy-wasting polling loop and utilizing interrupts. The internal logic stays the same only the code parts for button state checking and for supplying DAC with samples were moved to interrupt handlers. There is also a new function called `setupNVIC()` which setups proper interrupt generation and handling for the hardware timer and DAC. This enables us to utilize deep-sleep and other energy consumption optimization techniques. The interrupt handler for timer interrupts is shown in 1.2.1.

```
// Function for enabling DAC
void enableDAC()
{
        // DAC0_CH0DATA goes from 0 to 4095 (12 bits)

        // Enable the DAC clock
        *CMU_HFPERCLKEN0 |= CMU2_HFPERCLKEN0_DAC0;

        // Prescale DAC clock
        *DAC0_CTRL = 0x50010;

        // Enable left and right audio channels
        *DAC0_CH0CTRL = 1;
        *DAC0_CH1CTRL = 1;
}

// Function to setup the timer
void enableTimer(uint16_t period)
{
        // Enable clock to timer
        *CMU_HFPERCLKEN0 |= CMU2_HFPERCLKEN0_TIMER1;

        // Write the period to register
        *TIMER1_TOP = period;

        // Enable timer interrupt generation but no execution
        *TIMER1_IEN = 1;

        // Start the timer
        *TIMER1_CMD = 1;
}
```

Source code 1.1.2: Timer and DAC setup code

```c
// TIMER1 interrupt handler
uint64_t counter = 0;
void __attribute__ ((interrupt)) TIMER1_IRQHandler()
{
        // Clear the interrupt register
        *TIMER1_IFC = *TIMER1_IF;

        // Increment counter
        counter++;

        // Play a song
        if (currentSound != NULL)
        {
                if(counter > (*currentSound).sampleCount-1)
                {
                        counter = 0;
                }

                *DAC0_CH0DATA = (*currentSound).samples[counter];
                *DAC0_CH1DATA = (*currentSound).samples[counter];
        }
}
```

Source code 1.2.1: Timer interrupt handler function

2 Energy Measurements

The two different approaches to the problem also differ in power consumption.

2.1 Baseline Solution

In the polling version the microcontroller never enters a power saving state because it constantly checks if a button is pressed and if the timer reached its previously set limit which corresponds to the time period of each audio sample. That is why the current consumption is relatively high and never falls below 4 mA.

In Figure 2.1 the current consumption of the baseline solution is shown. At the beginning until about 2s a button was pressed and a melody played by the DAC. During the following about 1.5 seconds no button was pressed which resulted in a lower power consumption of about 4.3 mA. After the trough a button was pressed again until shortly before the end where it stabilizes again at a slightly lower current consumption level. Playing a sound has only a small current consumption penalty of about 0.1 mA.

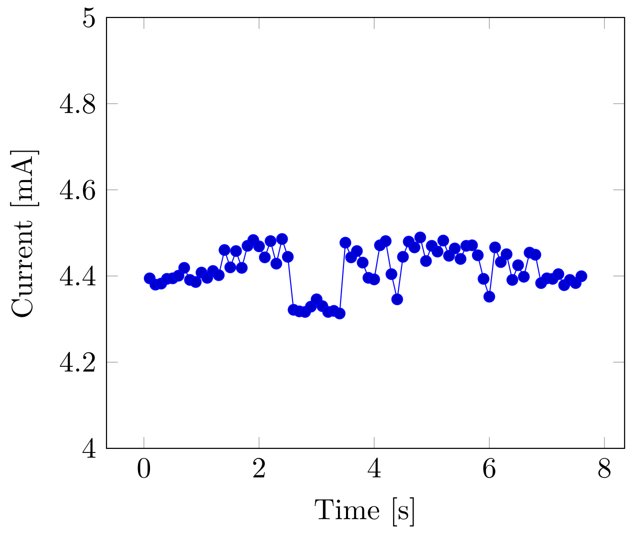

Figure 2.1: Power consumption of the basic solution without LEDs.

2.2 Improved Solution

With the use of interrupts further energy saving methods like deep sleep can be used. After power up and initialization the microcontroller enters deep sleep and consumes only about 250 µA. When a button is pressed the chip wakes up and sends the sound samples to the DAC. The timer triggers also an interrupt when a new sample for the DAC is needed. This reduces the current consumption during the active phase from about 4.4 mA (basline solution) down to about 1.9 mA.

Figure 2.2 shows the current consumption of the EFM32. At the beginning no sample is played and the microcontroller is in deep sleep. Then a button is pressed and a melody played, which results in a higher power consumption and is visible as the first peak. After that the melody is stopped and deep sleep is entered again. After a short pause this repeated a second time, starting at about 5 seconds.

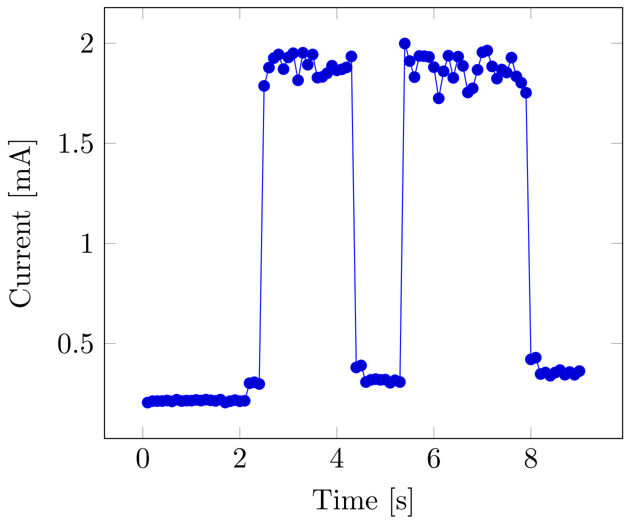

Figure 2.2: Power consumption of the improved solution without LEDs.

When it comes to other tricks that lower the power consumption, we downsampled all the sounds to 5000 samples a second, which still preserves acceptable quality and allows the timer interrupt to fire less often while playing. We also found a specific optimization related to square waves. When square wave is used than there is a possibility to push a new sample to DAC only when the value changes as the half periods are constant. However, because of the wave dependence, this technique cannot be used for generic sounds. Also generation of the samples on the device turned out to be expensive in terms of power consumption so we decided not to use it.

Other promising techniques were also tested but brought little to no difference in power consumption. There were high hopes about lowering the CPU clock but the difference turned out to be neglectable. Deactivating unused resources during the active time of the chip did not result in a measurable difference in power consumption too, therefore, mainly enabling deep sleep had a noticeable positive impact on the microcontroller current draw. Lastly, there was an idea of configuring low-energy timer which remains untested and is considered a subject for investigation in further exercises.