**NTNU – Trondheim**
Norwegian University of
Science and Technology

TDT4258 Low-Level Programming
Laboratory Report

# Exercise 2 - Audio player

*Group 8:*

Emil Braserud
Wesley Ryan Paintsil
Erblin Ujkani
Andreas Varntresk

January 22, 2018

# Contents

# 1. Overview

This report is written for the second exercise in the course TDT4258 Low-Level Programming. Throughout the course all the exercises are done on a EFM32GG prototype board. The requirements for this exercise is to generate sound effects using the on-board digital to analog converter(DAC) and amplifier on the development board. A button pressed plays a specific sound effect. Two different solutions were implemented, a baseline solution and an improved solution. The baseline solution uses busy-waiting to detect a timer owerflow and the improved solution utilizes interrupts. C is the expected programming language and a make-file has to be generated to compile and upload the program to the prototype board.

The main objective is to compare the power efficiency of the two versions, and if possible make further improvements to the improved solution. This report is structured so that the two solutions are described in their own chapter and a last chapter describes the power measurements done with the two different solutions. The first chapter explains how the sounds are generated, as well as how the dependent peripherals are set up.

## 1.1. Sound synthesizer

In order to play sounds one has to somehow generate sound samples. There are many different ways one could do this, in this solution constant samples were generated from a PC and then stored in memory. This section discusses how this was done and presents how the sound was sent over to the DAC, as well how the timer was set up.

### 1.1.1. Sound generation

The sound samples were extracted with a python script which takes care of scaling the audio samples to its correct resolution. The resolution which are used are 8-bit and in the full scale 12-bit resolution of the DAC. The DAC can work in either single-ended- or differential mode, in this case single-ended samples were used. In the code, a typedef struct called *Sound* is used to store information of the different sounds as shown in the given code. The parameters it has is an array of the samples and an integer which stores the number of samples stored in the array. Another Sound typedef struct with 8-bit samples was also used in order to save more space.

**sounds.h: Sound structs**

```
1 typedef struct Sound {
2   uint32_t length;
3   uint16_t samples[];
4 } Sound;
5
6 typedef struct Sound_8bit {
7   uint32_t length;
8   uint8_t samples[];
9 } Sound_8bit;
```

The python script is attached in the appendix A. It takes in an .csv file with extracted samples from an audio file which is generated using a free open source digital audio software called Audacity.[1] The samples derived from the csv document are stored as linear values from -1 to 1. In order to convert the samples as 12- or 8-bit with single ended mode it has to move its zero point to the half of the maximum value of the bit resolution.
Figure 1.1 shows the differences between single ended and differential for a 12-bit signal.

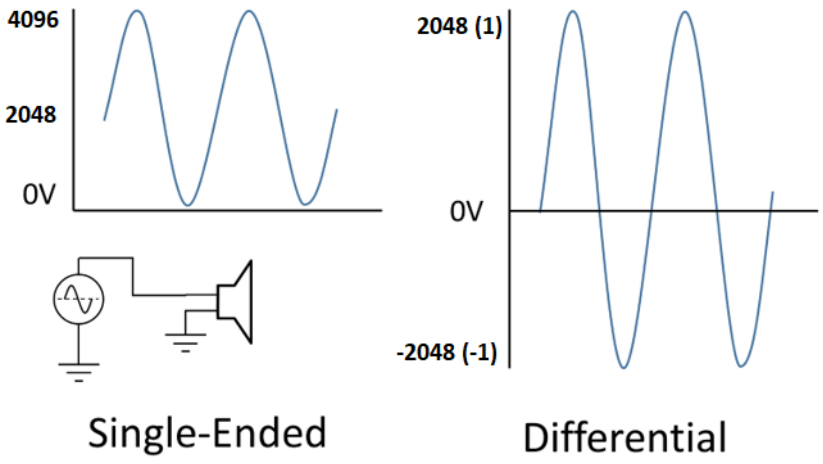



Figure 1.1.: Conversion of the sample sounds for 12 bit[4]

### 1.1.2. Playing sounds

Before one sends the samples to the DAC, one has to initialise it in the correct configuration. Firstly the Clock Management Unit(CMU) clock for the DAC peripheral has to be enabled which is located in bit 17 of *CMU_HFPERCLKEN0* register:

**dac.c: CMU DAC enable**

```
1     *CMU_HFPERCLKEN0 |= CMU2_HFPERCLKEN0_DAC0;
```

Thereafter the DAC clock has to be prescaled to the correct value, and the single-ended mode has to be set accordingly as discussed in 1.1.1. The conversion mode of the DAC is set to continuous mode which means that the DAC channels will drive their outputs continuously with the data in the data registers. Also the the output of both channels (left and right) have to be enabled (from section 29.4 [3]):

**dac.c: DAC enable**

```
1     *DAC0_CTRL          = 0x50010;
2     *DAC0_CH0CTRL       = 1;        /*enable channel 1*/
3     *DAC0_CH1CTRL       = 1;        /*enable channel 2*/
```

After being done with the initialization, a continuous stream of samples can be sent to the data channel register (*DAC0_CH0DATA* and *DAC1_CH0DATA*). These registers are 12 bit wide and they will be overwritten with new samples for each sampling period. This sampling period will be derived in the next section.

### 1.1.3. Deriving the timer period

The value at which the *TIMER1_CNT* overflows is determined by the period set in the *TIMER1_TOP* register, in our case it is set to match the sampling frequency of the

sample sounds that are 8 kHz. In order to set the period correctly one has to find out the at which rate the core clock runs at and how many bits the timer registers are. From section 20 in [3], we find out that the core clock runs at 14 MHz and the timer registers are 16 bit wide. From this we get that *TIMER1_CNT* register had to be set to 1750, under follows the calculation for the period:

$$\text{1 tick of timer: } \frac{1}{f_{core}} = \frac{1}{14 \text{ MHz}} \quad (\approx 71.43 \text{ ns})$$

So in order to have the period of 8 kHz in the timer: (where $\tau$ is the number of ticks to be set in *TIMER1_CNT*)

$$\frac{1}{14\ MHz}x = \frac{1}{8 \text{ kHz}}$$
$$\Downarrow$$
$$x = \frac{1}{8 \text{ kHz}} \cdot 14 \text{ MHz} = \underline{1750} \tag{1.1}$$

The value found from (1.1) does not the exceed the amount the *TIMER1_CNT* register can hold which is 65535, so a clock prescaling is not necessary.

## 1.2. Baseline Solution

The baseline solution plays sounds via a polling/busy-waiting method. This way the main program polls the timer counter register, *TIMER1_CNT*, until it reaches a certain value or overflows. In this solution minimum 3 GPIO buttons will activate and play a specific sound and a start up melody will be played at the start of the program. The GPIO's are also triggered by polling.

### 1.2.1. Initialization

The following section will describe the initialization process of the different peripherals.
**GPIO**
Firstly the GPIO clock in CMU is set up and activated.
This was done by setting the 13th bit in the *CMU_HFPERCLKEN0* register. This is done by shifting and bitwise "OR" the register with the 13th bit. (see section 11.5.18)[3]. This is done so there is no interference with the other bits in the register. The drive strength of the GPIO port A is set to 20mA by writing *0x2*to the *GPIO_PA_CTRL* register. The pins 0-7 is set as inputs for PORT C, and the pins 8-15 is set as outputs on PORT A in accordance with the connected gamepad.

**gpio.c: GPIO initialization**

```
1    *CMU_HFPERCLKEN0       |= CMU2_HFPERCLKEN0_GPIO;
2    *GPIO_PA_CTRL           = 0x02;
```

```
3     *GPIO_PA_MODEH          = 0x55555555;
4     *GPIO_PA_DOUT           = 0xFF00;
5     *GPIO_PC_MODEL          = 0x33333333;
6     *GPIO_PC_DOUT           = 0xFF;
```

**TIMER**

The timer likewise requires the clock to be enabled which is done by an or operation. Then the *TIMER1_TOP* has to be decided. 1.1

timer.c: Timer initialization

```
1     *CMU_HFPERCLKEN0 |= CMU2_HFPERCLKEN0_TIMER1;
2     *TIMER1_TOP       = TIMER_PERIOD;
```

### 1.2.2. Polling of timer

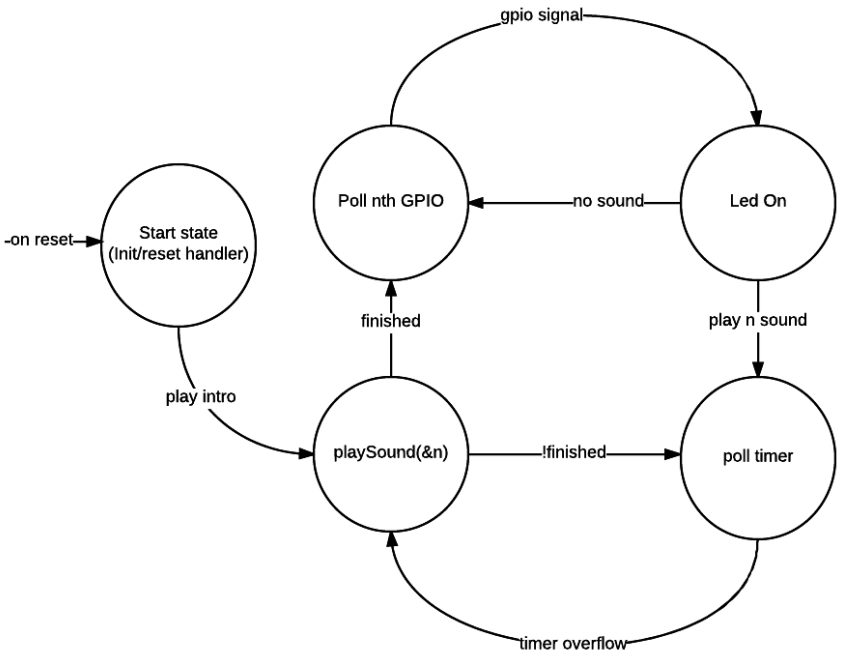



Figure 1.2.: The state machine of the basleine solution

The implementation of busy-wait polling was done by a combination of while loops. The main loop constantly checks whether a button has been pressed. When the playback function is called the array containing the sound samples is incremented by one when the timer overflows. During the playback, the *TIMER1_CNT* will constantly be polled till it overflows, when the overflow occurs new samples will be sent to the DAC and the

timer will reset and count again. The code under shows the polling implementation of the audio playback in the baseline version and a state machine is shown in figure 1.2.

sounds.c: Baseline solution

```
1  void playSound(Sound* name)
2  {
3    uint16_t idx = 0;
4    while(idx < name->length)
5    {
6      if(!(*TIMER1_CNT))
7      {
8        *DAC0_CH0DATA = name->samples[idx];
9        *DAC0_CH1DATA = name->samples[idx];
10       idx++;
11     }
12   }
13 }
```

## 1.3. Improved Solution

The improved solution puts the processor to sleep when no buttons have been pushed or there is no sound to be played. To do this "0b110" is written to the SCR register, this writes '1' to the SLEEPDEEP and SLEEPONEXIT bits, which make the CPU enter deep sleep when exiting the interrupt service routine. When a button is pressed an exception handler picks the right sound based on what button was pressed, sets up the DAC and timer, and then starts the timer. When the timer overflows it generates an interrupt, the interrupt handler then sends the next value to the DAC and resets the timer. When all samples of the sound file are played the timer and DAC are stopped and turned off. Each time a sound is played SLEEPDEEP is disabled and enabled again after the sound has finished. This is done because enabling deep-sleep mode will disable all high-frequency clocks which are necessary for timer and DAC to work. So therefore the system is only put in the sleep mode (not deep-sleep) during playback as shown in the state machine in figure 1.3.

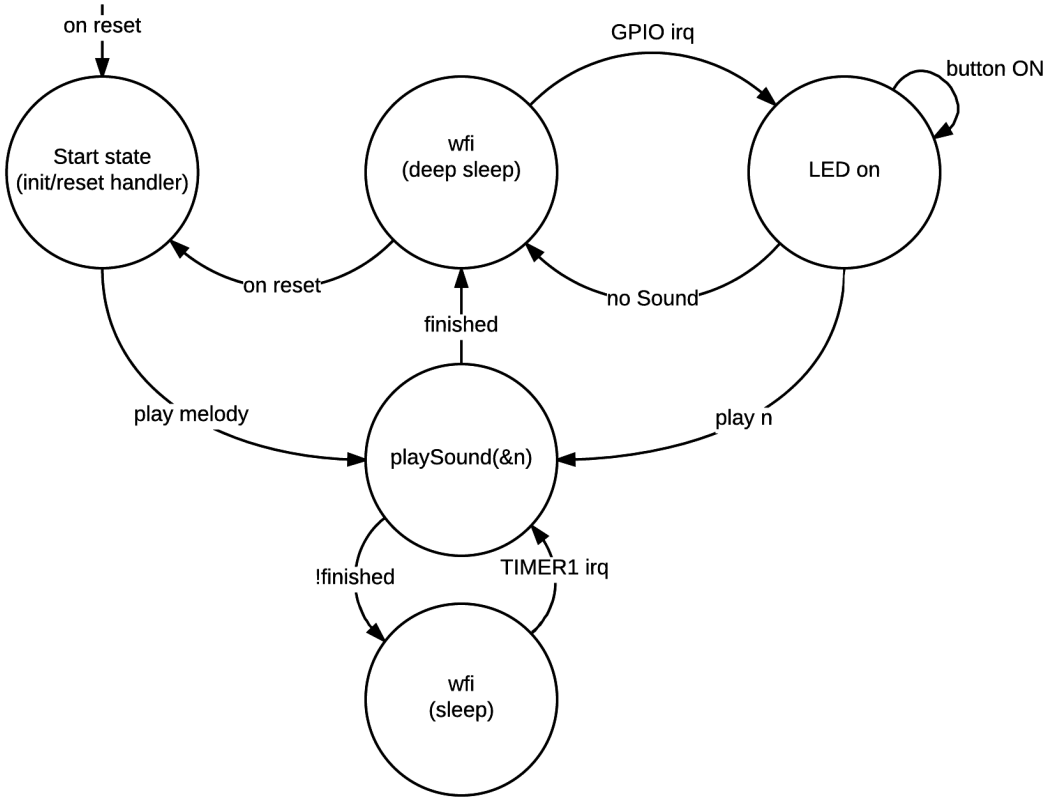



Figure 1.3.: The state machine of the improved solution

### 1.3.1. Initialization

The initialization is almost identical to the baseline approach with the exception of interrupt enabling. Enabling the external interrupts on GPIO pins on PORT C required the value *0x22222222* to be written to *GPIO_EXTIPSELL*(external interrupt select low register). According to the EFM32GG-RM datasheet section 32.5.10, this will select PORT C as the trigger for the interrupt flags. [3]

**gpio.c: GPIO interrupt**

```
1    *GPIO_EXTIPSELL     = 0x22222222;
2    *GPIO_EXTIRISE      = 0xFF;
3    *GPIO_IEN           = 0xFF;
```

The next step consists of setting 'rising' edge triggers on the interrupts by writing 0xff to the *GPIO_IEN*, *GPIO_EXTIFALL* registers. *0xff* is written because only pin 0-7 are relevant. (see section 32.5.13,32.5.12,32,5,14) [3]. Similarly the hardware *Timer1* reqires the *TIMER1_IEN* register t be set, to enable interrupts.

**timer.c: TIMER interrupt**

```
1    *TIMER1_IEN         = 0x1
```

To generate the interrupts for *GPIO_ODD* and *GPIO_EVEN* *TIMER1* the value *0x1802* has to be written to the address *ISER0*, where the value 0x1802 correspond to the IRQs lines of the *GPIO_EVEN*,*GPIO_ODD* and *TIMER1* (see section 4.3.1) [2]. The interrupt handlers will clear the interrupt flag to ensure that the interrupt are not called repeatedly.

**ex2.c: NVIC Interrupt handling**

```
1 void setupNVIC()
2 {
3   *ISER0 |= 0x1802; /* Enable interrupt handling for peripherals(GPIO
      and timer 0x1802) */
4 }
```

# 2. Results

## 2.1. Program

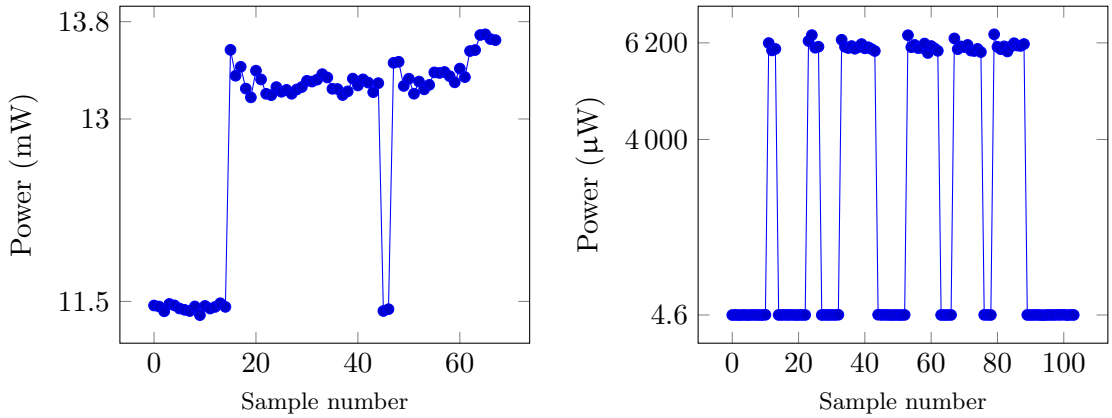
The final program consists of four sounds and one start up melody. After initiation/reset a start up melody will be played, and when the melody is finished the board enters deep sleep mode and remains there until a button is pressed. Four of the gamepad buttons each play one sound effect, and only one sound can be played at the time.

## 2.2. Sound quality

The group observed a drastic improvement in sound quality when the interrupt solution was implemented. This is mainly because the baseline solution polls the timer count register and it may sometime miss some of the timer overflows. This introduces jitter , which causes some samples to be sent at different rates. The improved based solution on the other hand, utilizes precise timer interrupts which makes the samples play at correct interval and hence generates clearer sound.

## 2.3. Energy Measurements

As in exercise 1, this section will be used to compare the difference in energy consumption between the baseline solution and the improved solution.
For the baseline solution, the microcontrollers idle power consumption is around 12mW and 15 mW during playback.
In the improved solution interrupts were used which makes it possible to use sleep mode when the system is idle, which leads to a massive decrease in power consumption. When in idle mode, the improved solution uses about 4.5uW which is around 1440 times less energy consuming compared to the baseline solution. During playback the power consumption stays at about 6.2 mW which is twice as power efficient as the baseline solution.




(a) Baseline

(b) Improved

Figure 2.1.: Power consumption comparison

# 3. Conclusion

The baseline and the improved solution gave totally different results in view of the power consumption and sound quality. As from the previous exercise. An interrupt solution in terms of power consumption is always superior against a polling solution as it drastically reduces the overall static power consumption. Using interrupts also made the timer periods more precise than the polling version, and this made the sound quality much better. Further improvements could have been made to the improved solution, such as using the DMA to feed the samples to the DAC directly without intervening with the CPU which would result in higher energy efficiency. This could have been implemented if there was more time to study and setup the DMA.

# A. Python code

Listing A.1: Python code for generating Sound structs in c code

```
#!/usr/bin/python3

import csv

def num(s):
    try:
        return int(s)*128+128
    except ValueError:      #just in case
        return int(float(s)*128)+128

def main():
    import csv

    results = []

    with open('win.csv', 'r', newline='') as f:
        reader = csv.reader(f)
        for counter, row in enumerate(reader):
            results.append( num((row[:1][0])) )
    print(results)

    with open("win.txt", "w") as outputfile:
        outputfile.write("Sound laser = {")
        index = 0
        for item in results:
            outputfile.write(" %s, " % item)
            index += 1
        outputfile.write("}};\n")
        indexByte = index // 8
        outputfile.write("Number of samples: %s\n" % index)
        outputfile.write("Number of bytes: %s" % indexByte)

if __name__ == "__main__":
    main()
```

# B. Block diagram

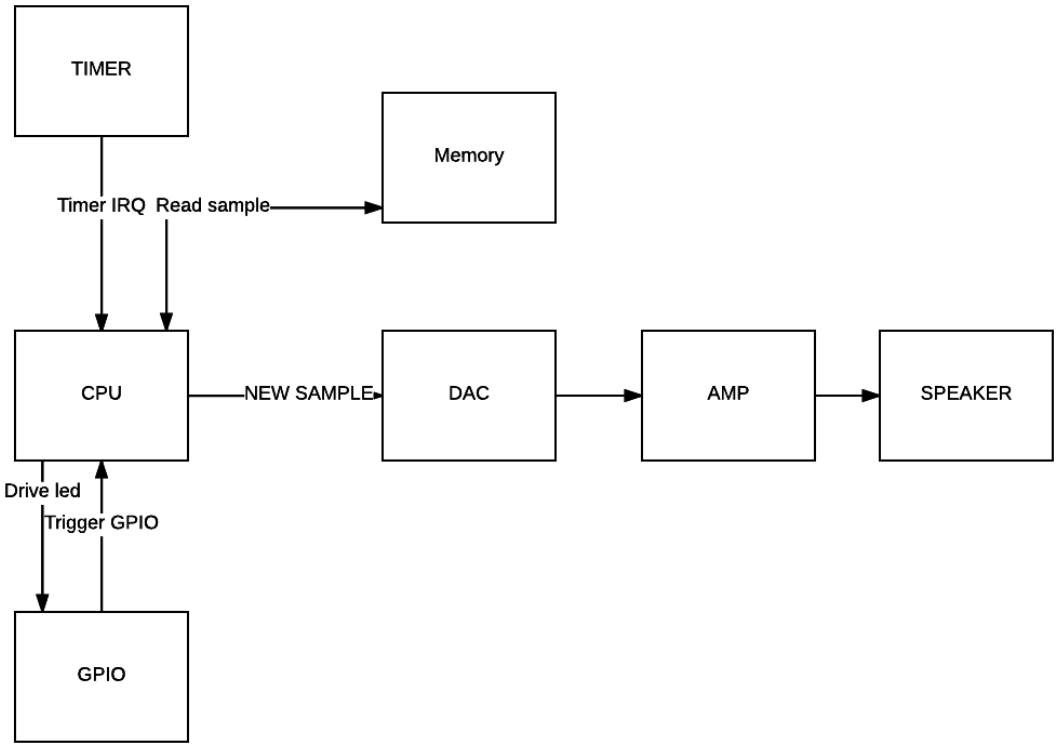



Figure B.1.: Block diagram of solution

# Bibliography

[1] Audacity. Home — audacity, March 2017. http://www.audacityteam.org/home/, (Accessed on 02/10/2017).

[2] Silicon Labs. Cortex-m3 reference manual, feb 2011. (Available at https://www.silabs.com/documents/public/reference-manuals/EFM32-Cortex-M3-RM.pdf, Accessed on 10/09/2017).

[3] Silicon Labs. Efm32gg reference manual, apr 2016. (Available at https://www.silabs.com/documents/public/reference-manuals/EFM32GG-RM.pdf, (Accessed on 10/09/2017).

[4] Silicon Labs Community : Blog : Official Blog of Silicon Labs. Build your own lightsaber (sounds) - part 3, November 2016. http://community.silabs.com/t5/Official-Blog-of-Silicon-Labs/Build-your-own-lightsaber-sounds-Part-3/ba-p/184000, (Accessed on 25/09/2017).