NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4258 Low-Level Programming
Laboratory Report

TDT4258 Exercise 2

Group 10:

Anders Sildnes
Rasmus Stene
Ole Sørlie

October 17, 2016

1 Overview

This report outlines our solution for how to create an energy-efficient digital synthesizer using the EFM32 Giant Gecko ARM Cortex-M3 microcontroller. Our synthesizer is written in standalone C[1], and we use a DK3750 development board for interacting with users. By using the buttons on a DK3750 prototyping board, users can select between different sound effects, as well as an opening melody and two other sound patterns. The sound effects are generated using the 12-bit analog-to-digital (ADC) converter of the microcontroller. All the sound effects, as well as the sound synthesizer it self, can be used later in other projects.

1.1 Baseline Solution

There are many different ways to generate a sound. In this exercise we use several different methods, each with their own strengths and weaknesses. Our baseline solution implements the synthesizer without using timers or interrupts, and instead relies on a busy-wait and polling mechanisms.

An intuitive approach for generating sound is to use a function to generate a sine wave, and simply change certain parameters of the function in order to achieve different frequencies. The output of the function would then be passed on to the DAC. However, computing sine waves is computationally expensive. With high samples rates, this can end up being too much for the Cortex-M3 processor to handle.

Instead of generating sound samples "on-the-fly", one can use predefined look-up tables. The advantage of look-up tables is that the values are already precomputed, meaning that there is no longer a need to use heavy mathematical functions. This also make it easier to accurately create sound effects, since you can generate the tables using external tools. The bigger the tables, the more real or dynamic sounds you can make. The downside is that it puts a bigger strain on both bus and memory, leaving less room for other IO units and components. In the end, one has to choose between sound quality and processing costs. Hybrid solutions are also possible, i.e. using tables in conjunction with generators. These make it possible to save storage of big tables, at the cost of increased complexity and potentially loss of accuracy.

Because of its cheap computational cost, we opted with lookup-tables and generated all the necessary sounds needed beforehand. To create the tables, we used sine waves and simply varied the frequency of the wave to produce different tones. Furthermore, we implemented standard piano key frequencies and labelled them according to their

[1] i.e. without any Operating System or external dependencies

scientific name. A standard piano consists of 88 keys, ranging from 27.5 Hz to 4186 Hz. The frequency of each key can be calculated using equation 1.1.

$$f(n) = 2^{\frac{n-49}{12}} 440Hz \tag{1.1}$$

In order to make the look-up tables for the piano keys, we used a simple Matlab script, as shown in Listing 1.1

```
Spers = 44100;
Sample = 0;
fileID = fopen('Pianokeys.txt','w');
for c = 1:88
    Freq = 2^((c-49)/12)*440
    SoundLength = round(Spers/Freq)

    fprintf(fileID,'Sound Key%d = {%d,{',c,SoundLength);

    for s = 1:SoundLength
        %The mod function ensures that the we do not exceed 2*pi in the
        %sinus function and cause unwanted harmonics

        Sample(s) = floor((sin(mod((2*pi*Freq*(s-1)/Spers),2*pi))*0.5+0.5)*
    255);
        fprintf(fileID,'%d',Sample(s));

        if (s<(SoundLength))
            fprintf(fileID,',',Sample);
        end
    end

    fprintf(fileID,'}};\n',c);

end
    fclose(fileID);
```

Listing 1.1: Piano frequency generator Matlab script.

The script uses a sinus function in order to generate look-up tables with different frequencies based on the sample rate, the wanted frequency and the set amplitude. The amplitude was set to a maximum of 255, which allowed us to store the samples in the look-up table as 8-bit integers, which would save space compared to having to use 16-bit integers. The script generates look-up tables for all 88 keys in a standard piano and places them in a separate file. This is done using the sinus function, and the result is an array which contains sample values from 0 to 255, where 127 represents the "0" of the sinusoidal waveform. The number of samples in the array is determined by the "Spers" variable, which is set as 44100 samples per second, and Freq, which is the frequency of the desired piano key. Most of the keys generated from Matlab were later implemented in the design, with the exception of the lowest 27 keys, because the size of the look-up table increases as the frequency lowers, resulting in very large tables for the lowest keys.

```
((t<<1)^((t<<1)+(t>>7)&t>>12))|t>>(4-(1^7&(t>>19)))|t>>7 % 1023
```

Table 1.1: An example "byte beat".

In addition to lookup-tables we implemented sound patterns from generators. The idea is taken from Heikkilä's blog[1], where he features many patterns, referred to as "byte beats", generated by users online. The idea is to perform bit-shift and arithmetic operations on an unsigned integer to produce sound patterns[2]. An example pattern is given in Table 1.1. "t" is a 16-bit unsigned integer which is repeatedly incremented by one, with a return to 0 on overflow. The advantage of the "byte beats" is that they are fast to compute and take little storage space. The downside is that they are difficult to produce and modify.

In order to store the look-up tables produced by the Matlab script, we created two structs, typedefined as Sound and SoundFile. The piano keys themselves would be defined as Sound, while a pattern of piano keys would be defined as SoundFile. The definitions of Sound and SoundFile is shown in Listing 1.2.

```c
typedef struct Sound {
    uint32_t SoundLength;
    uint8_t SoundData[];
} Sound;

typedef struct SoundFile {
  uint32_t SoundFileLength;
  Sound* SoundFileData[][2];
} SoundFile;
```

Listing 1.2: Defining Sound and SoundFile.

The Sound struct consists of two variables, SoundLength and SoundData. SoundLength is simply how many samples there are in in the look-up table for each piano key. SoundData is an unspecified array which contains all the samples. Listing 1.3 contains an example Sound for the piano key C7. As can be seen, the value 21 is the number of samples, while the other values are values generated by the Matlab script.

```c
Sound C7 = { 21, {127, 164, 199, 226, 245, 254, 251, 238, 214, 183,
                  147, 109, 73, 41, 17, 3, 0, 8, 26, 53, 87}
};
```

Listing 1.3: The piano key C7 represented as a Sound type.

The SoundFile struct works similarly to the Sound struct. The variable SoundFileLength specifies how many piano keys the sound effect is made up of. SoundFileData contains a list of piano keys as well as a duration variable for each key. The duration

[2]Our code yields compiler warnings on these patterns because of unbalanced parenthesises, we hope this is okay due to the trivial nature of these patterns.

variable indicates how long each key is to be played when the SoundFile is read. The duration variable is casted to an unsigned 32-bit integer pointer by the help of a #define, as is shown in listing 1.4. This is to make it easier to make the sound effects, with an example for the sound effect Imperial March shown in Listing 1.5.

```
#define T (void*)(uint32_t)  //Typecast to uint32_t and to pointer because
   SoundFileData expects a pointer
```

Listing 1.4: Typecasting for the SoundFile pointer.

As seen in Listing 1.5, the sound effect Imperial March consists of 36 Sounds, where about half of them are the key "Q", which is simply silence. Next to each key is a T, followed by a number, which indicates the duration of each key.

```
SoundFile ImperialMarch = {36, {{&G4,T 400},{&Q,T 80},{&G4,T 400},{&Q,T 80},
                                {&G4,T 400},{&Q,T 80},{&DE4,T 240},{&Q,T 80},
                                {&AB4,T 80},{&Q,T 80},{&G4,T 400},{&Q,T 80},
                                {&DE4,T 240},{&Q,T 80},{&AB4,T 80},{&Q,T 80},
                                {&G4,T 400},{&Q,T 400},{&D5,T 400},{&Q,T 80},
                                {&D5,T 400},{&Q,T 80},{&D5,T 400},{&Q,T 80},
                                {&DE5,T240},{&Q,T 80},{&AB4,T 80},{&Q,T 80},
                                {&G5,T 400},{&Q,T 80},{&DE5,T 240},{&Q,T 80},
                                {&AB5,T 80},{&Q,T 80},{&G5,T 400},{&Q,T 80}}
};
```

Listing 1.5: Imperial March implemented as a SoundFile type.

The program code for the baseline solution is relatively simple. All the look-up tables for the piano keys are stored in a separate file called "soundfiles.c", while the main program runs in "main.c". A simple flowchart which gives a basic overview of the baseline solution is shown in Figure 1.1. In order to enable and configure the DAC we used a separate function called "setupDAC", as shown is Listing 1.6. This function first enables the DAC in the Clock Management Unit, before configuring the DAC frequency as:

$$\frac{14}{32} MHz = 437.5 KHz \tag{1.2}$$

as well as configuring the DAC to output samples continuously. Finally, the function enables both channel 0 and channel 1 of the DAC.

```
void setupDAC()
{
    *CMU_HFPERCLKEN0 |= (0x01 << 17);    //Enable DAC clock
    *DAC0_CTRL = 0x50010;                //Set prescaler
    *DAC0_CH0CTRL = 0x01;                //Enable channel 0
    *DAC0_CH1CTRL = 0x01;                 //Enable channel 1
}
```

Listing 1.6: How the DAC was configured.

The last thing to configure was the GPIO, which enables the buttons and the LEDs. First the GPIO is enabled in the Clock Management Unit. Then the LEDs are set to high drive strength, pins 8-15 on port A is set as outputs, and all the LEDs are set to off. Then pins 0-7 on port C is set as input with internal pull-up resistor. The GPIO configuration setup function is shown in Listing 1.7.

```
void setupGPIO()
{
    *CMU_HFPERCLKEN0 |= CMU2_HFPERCLKEN0_GPIO;  //Enable GPIO clock
    *GPIO_PA_CTRL = 0x02;                      //Set high drive strength on LEDs
    *GPIO_PA_MODEH = 0x55555555;               //Set pin 8-15 of portA as output
    *GPIO_PA_DOUT = 0xff00;                    //Turn all LEDs off
    *GPIO_PC_MODEL = 0x33333333;               //Set pin 0-7 of portC as input
    *GPIO_PC_DOUT = 0xff;                      //Enable internal pullup resistor
}
```

Listing 1.7: Configuring the GPIO.

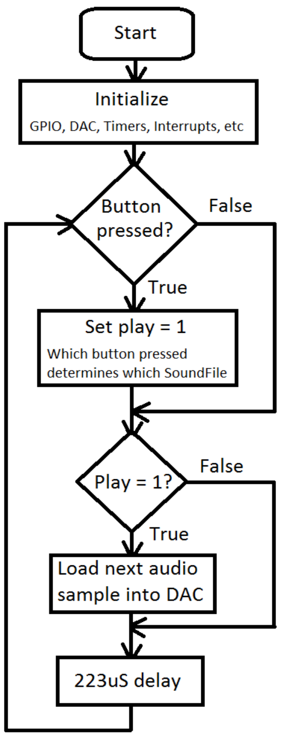
Start
Initialize
GPIO, DAC, Timers, Interrupts, etc
Button pressed?
False
True
Set play = 1
Which button pressed determines which SoundFile
Play = 1?
False
True
Load next audio sample into DAC
223uS delay

Figure 1.1: Program flowchart for the baseline solution.

Since the baseline solution was not supposed to use any interrupt functions and instead rely on busy-wait, we were not able to put the device in any lower energy state, because in order to wake the device from this mode we have to use some kind of interrupt.

In order to implement a busy-wait function, and also time how often to update the DAC without the use of a timer we used a simple delay function called "delay". This function takes an unsigned 32-bit integer as an argument and simply runs an empty loop depending on the argument given, as shown in Listing 1.8. Because the DAC is fed from the mainloop, and since we have a sample rate of approximately 44 100 samples per second, we use the delay function so that the correct number of samples were fed to the DAC. By changing the delay, one can change the pitch of the sound sent to the DAC.

```c
void delay(uint32_t duration)
{
    uint32_t i = 0;
    do {
        i += 1;
    } while (i < duration);
}
```

Listing 1.8: Simple delay function used in the main loop.

An excerpt from our main function is shown in Listing 1.9. The “aButtonIsEnabled” is toggled to prevent race conditions with one button push generating multiple signals or reading more than one button at a time.

```c
int main()
{
  // setup hardware...
  while(1) {
    if(*GPIO_PC_DIN != INPUT_EMPTY) {
        if(aButtonIsEnabled == 0) {
            resetSoundPlayer();
            setupDAC();
            buttonPressHandler();
            aButtonIsEnabled = 1;
        }
    }
    else {
        aButtonIsEnabled = 0;
    }

    if(ByteBeatIndex > 0 || CurrentSoundFile > 0) {
        playCurrentSound();
    }
    else {
        shutdownDAC();
    }
}
```

Listing 1.9: Excerpt from "main.c" for the baseline solution.

In the end, we play sounds by iterating indexes over current sounds, selected from the main loop. Sounds are played in a fashion similar to what is shown in Listing 1.10. The return values for “getFromSoundTable()” are fed to both DAC channels.

```c
uint16_t getFromSoundTable()
{
    if (sampleIndex > CurrentSoundFile->SoundFileLength - 1) {
        resetSoundPlayer();
        return 0;
    }

    trackA++;
     // If we have reached the end of a tone, reset the counter and increment
     trackB
    if (trackA == CurrentSoundFile->SoundFileData[sampleIndex][0]->
    SoundLength) {
        trackA = 0;
        trackB++;
    }

    // if we have reached the end of a tone
    if ((trackB*(CurrentSoundFile->SoundFileData[sampleIndex][0]->SoundLength
    )/33) >
         CurrentSoundFile->SoundFileData[sampleIndex][1]) {
        trackB = 0;
        sampleIndex++;
    }


    return CurrentSoundFile->SoundFileData[sampleIndex][0]->SoundData[trackA
    ];
}
```

Listing 1.10: Excerpt from “soundPlayer.c” for the baseline solution.

1.2 Improved Solution

Our improved solution generate sounds in the same way as the baseline solution, but instead implements timer- and button-interrupts so that the CPU can do something else while it is not playing sounds. This will be important in Exercise 3 when a game will be running together with the sound-effects. The advantage of not using busy-wait as was used in the baseline solution is that it is possible to let the CPU sleep while we wait for a button to be pressed. In the busy-wait version, the CPU simply runs some trivial process while we wait for a button to be pressed, which in turn wastes energy.

The DAC is configured in the same was as in the baseline solution. The timer is configured using the function “setupTimer”. As with the DAC setup, the Clock Management Unit is first configured to enable the timer. Then the timer overflow limit is set to a given period, which for us is “2”, i.e. interrupting every second cycle. We use the LETIMER0 from the development board with a prescaling factor of DIV32. This does not yield exactly 44100 samples per second, but we are still able to generate sounds with high resolution. Finally, we enable the timer interrupt and enable the timer itself. The timer setup function is shown in Listing 1.11.

```
    *CMU_HFCORECLKEN0 |= 0x10; // enable EBI (chapter 14)
    *CMU_LFCLKSEL |= 0x2; // LowFrequencyCLocKSelect (choose crystal
    oscillator as source)
    *CMU_OSCENCMD |= 0x10; // Toggle the crystal oscillator
    *CMU_LFACLKEN0 |= 0x4;  // toggle LETimer page 152
    *LETIMER0_CMD = 0x2; // Clear LETimer
    *LETIMER0_CTRL |= (0x2 << 8); // Restart the clock (RTC) after reaching
    COMP0

    // Parameters that affect timer frequency:
    *CMU_LFAPRESC0 |= (0x5 << 8); // prescaler: divide by n (bits 11 to 8)
    *LETIMER0_COMP0 = 2; // generate interrupt for every count

    //Clear interrupt
    *LETIMER0_IEN = 0x4; // Enable REP1 interrupt
```

Listing 1.11: Configuring the timer.

```
void setupTimer(uint16_t period)
{
    *CMU_HFPERCLKEN0 |= (0x01 << 6);    //Enable TIMER clock
    *TIMER1_TOP = period;               //Set the period of the timer
    *TIMER1_IEN = 0x01;                 //Enable the timer interrupt
    *TIMER1_CMD = 0x01;                 //Enable the timer
}
```

Listing 1.12: Configuring the timer.

The GPIO is configured in the same was as in the baseline solution, but with the addition of a couple more lines of code. The added code simply selects the buttons as a source of interrupts, sets the interrupt to trigger at a falling edge, enables the GPIO interrupts and then clear all GPIO interrupt flags. The GPIO setup function for the improved solution is shown in Listing 1.12.

```
void setupGPIO()
{
    *CMU_HFPERCLKEN0 |= CMU2_HFPERCLKEN0_GPIO;  //Enable GPIO clock
    *GPIO_PA_CTRL = 0x02;               //Set high drive strength
    *GPIO_PA_MODEH = 0x55555555;        //Set pin 8-15 of portA as output
    *GPIO_PA_DOUT = 0xff00;             //Turn all LEDs off
    *GPIO_PC_MODEL = 0x33333333;        //Set pin 0-7 of portC as input
    *GPIO_PC_DOUT = 0xff;               //Enable internal pullup resistor
    *GPIO_EXTIPSELL = 0x22222222;       //Set buttons as external interrupts
    *GPIO_EXTIFALL = 0xff;              //Interrupt triggers at falling edge
    *GPIO_IEN = 0xff;                   //Enable GPIO interrupts
    *GPIO_IFC = 0xff;                   //Clear all GPIO interrupt flags
}
```

Listing 1.13: Configuring the GPIO.

An excerpt from “main.c” is shown in Listing 1.13.

```c
int main()
{
    while (1) {
        __asm("wfi");
    }
}
```

Listing 1.14: Excerpt from “main.c” for the improved solution.

Notably, the main loop is much simpler to read since we are allowed to write the program in an event-driven fashion, where each interrupt has a corresponding input action.

1.3 Implementing DMA

Implementing Direct Memory Access would have enable us to potentially save some more energy because this would have made it possible to feed the DAC with samples without using the CPU. However, we were not sure if it would be possible to implement the DMA with the way we had constructed our sounds. Since all sound effects are built up using several smaller look-up tables arranged in an array, this means that they can, as far as we could determine, not be fed via the DMA to the DAC, because the CPU has to process them. If our sound effects were stored just as continious samples in a large look-up table instead, if would have been possible to implement the DMA to read the look-up table directly from memory and then pass on the samples to the DAC. This could have been done by modifying the Matlab script and make it generate entire successions of piano keys for a set duration and in a set sequence, however it would have been a rather complicated function, and time did now allow us to try this method out.

2 Energy Measurements

We measured the energy consumption while the device was in idle as well as when the device was playing a tune. Originally we had a LED light up corresponding to which button was pressed, but in order to conserve energy we decided to not use any of the LEDs in the final implementation.

2.1 Baseline Solution

Figure 2.1 shows the power consumption from when the baseline solution is in idle, in other words when no sound effect is playing. As is seen from the figure, when the CPU is busy-waiting the power consumption is 3.8mA. when the CPU is busy-waiting, we also turn off the DAC, and only turn it on again when we are ready to play a sound effect.

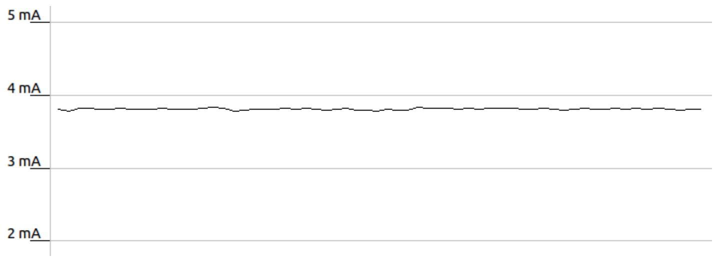

Figure 2.1: Power consumption when the baseline solution is in idle.

Figure 2.2 shows the power consumption when playing a sound effect from a look-up table, in this case the "Laser" sound effect. The peak current consumption when playing the sound effect was around 5.1mA.

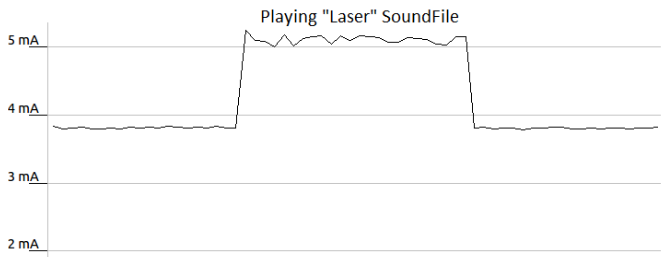

Figure 2.2: Power consumption when the baseline solution is playing a sound effect from lookup table.

Figure 2.3 shows the power consumption when playing a sound effect from a generator function. As can be seen, the peak power consumption in this case is around 4.8mA, which is 0.3mA less than using a look-up table. This is most probably due to the processor not having to work as hard in order to retrieve sound values from the look-up table.

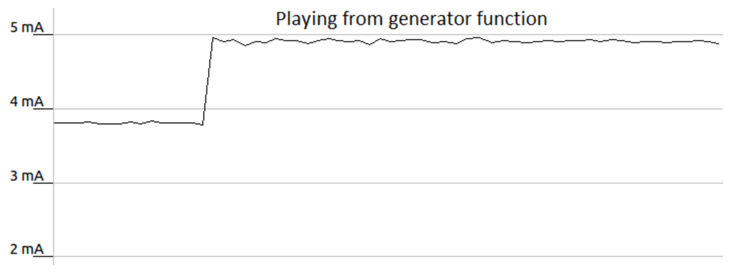

Figure 2.3: Power consumption when the baseline solution is playing a sound effect from generator function.

2.2 Improved Solution

In order to conserve power in the improved solution, we disable both the timer and the DAC and enter EM2 when the device is in idle. When the button is pressed it triggers an interrupt which wakes the processor and allows the program to continue. Just before a sample is loaded into the DAC after waking up, the timer and DAC are enabled. Figure 2.4 shows the power consumption of the improved solution when in idle is around 1.7uA.

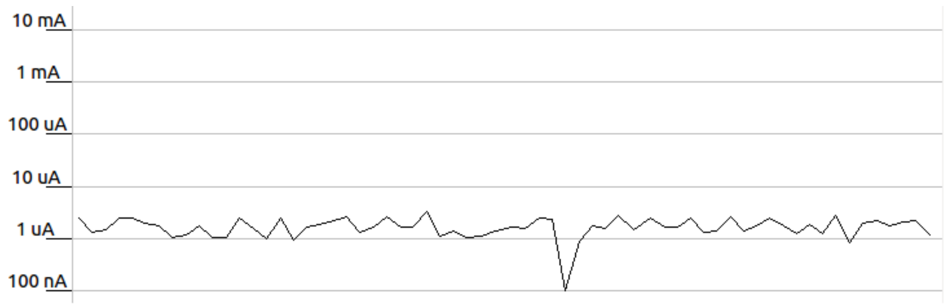

Figure 2.4: Power consumption when the improved solution is in idle.

Figure 2.5 shows the power consumption when playing the “Laser” Soundfile with the improved solution. With the look-up table, the device draws around 4mA peak current, which is much better than the baseline solution which drew 5.1mA. It is worthwhile noticing that if we removed function calls and if-statements, it is possible to use run on a lower current. For the “Laser” Soundfile it was possible to go down to 3.4 mA. We did not do this to keep the code readable and easier to extend. Also, it is important to note that our power usage is affected by the clock frequency; upping the sample rate also increases current usage.

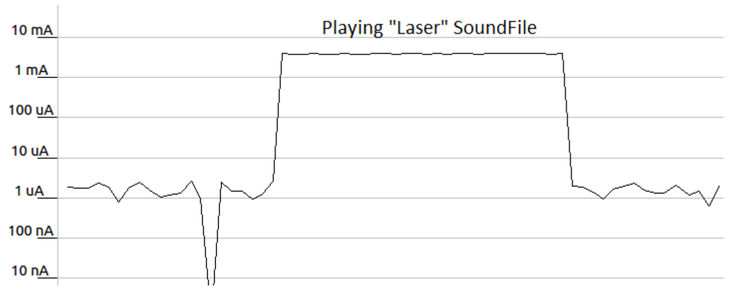

Figure 2.5: Power consumption when the improved solution is playing a sound effect from lookup table.

Figure 2.6 shows the power consumption when playing a sound effect using the generator function in the improved solution. The peak current was around 2.4mA, which is 1.6mA less than when using a look-up table.

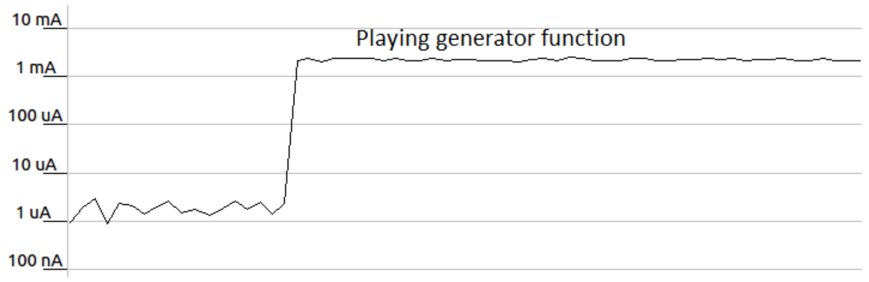

Figure 2.6: Power consumption when the baseline solution is playing a sound effect from generator function.

2.3 Conclusion

In conclusion, we see a drastic reduction in power consumption of the improved solution compared to the baseline solution. While the idle current draw of the baseline solution was 3.8mA, the idle power consumption of the improved solution was just 1.7uA. The biggest energy saving method is to utilize the sleep function. Turning off unused peripherals, such as the DAC and the timer also helps lowering the power consumption. There was also a difference in power consumption when playing sound effects from look-up tables and from generator functions, with the generator function method drawing less power than when using look-up tables.

Bibliography

[1] Ville-Matias Heikkilä. countercomplex: Algorithmic symphonies from one line of code – how and why? http://countercomplex.blogspot.no/2011/10/algorithmic-symphonies-from-one-line-of.html, October 2011. (Accessed on 10/16/2016).