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

TDT4258 - Low-level programming

# Second Lab Exercise Report

*Group 15:*

Dávid Danaj
Salahuddin Asjad
Valentin Lemenut

October 19, 2015

**Abstract**

This is a report for the second assignment in the subject TDT4258 - Low-level programming. The assignment is about creating sound effects using a microcontroller. In this assignment we have created four different sound effects which are represented with four different buttons. To make the microcontroller energy effecient we have implemented interrupts and deep sleep mode into the system. Using interrupts and deep sleep mode, the system uses 2000x less power during idle state and approximately 30% less power when playing a sound. This says something about how important these functionalities are to be implemented into a microcontroller program.

# 1 Introduction

The practical goal of the lab exercises is to implement a small game for the EFM32 development board which has a microcontroller, a display and audio effects. We also have access to a small prototype gamepad with buttons and LEDs so we can perfectly make our own simple computer game by programming the microcontroller which allows us to control the I/O-components we have.

Today, microcontrollers are widely used because they fit into almost every task. It is easy to find low-end microcontrollers for small applications and high-end microcontrollers for more complex ones. Microcontrollers have a big advantage that the design and hardware cost can be kept to the minimum level, because of the wide range of microcontrollers in the market.

Microcontrollers are usually built to do a specific repeating task which often makes only small piece of complex system. This allows engineers to create a microcontroller, which perfectly fits needs of a task from hardware as well as software point of view. Size of these systems provides possibilities to fit them nearly everywhere. Since microcontrollers are getting smaller and smaller, they are often integrated into places where it is difficult or impossible to get. They are often found isolated from other systems where their exclusive power source is a battery. In cases like that, it is extremely important to focus on minimization of power consumption, which prolongs lifespan of a device.

Furthermore, energy efficiency of our solution is an important goal of the course and this is why we have a specific component on our development board to measure the realtime power consumption of all components we have, which allows us to optimize our program to minimize the power consumption. Most microcontrollers are intended to work in systems where power consumption matters. Also in our case we want an efficient application, that uses minimum of power to keep the system going. That's why we have implemented different ways of accomplish that, which will be described later.

The second exercise is the next step, where we build on the experience we got from the first exercise. Now we will implement sound effects for our game which can be played according to the pressed button. For this exercise we will use four buttons on right side of a gamepad. Pressing each of these buttons will play different short sound. Unlike the previous exercise we will write the code in the C programming language. In order to achieve this, we will have to use GPIO control and interrupt handling again but we will also have to use the microcontroller's DAC and timers for sound generation. In the last part we will also use the DMA for a better energy efficiency.

# 2 Background and Theory

To solve the problem we will need to use our technical knowledge with microcontrollers and C programming. Like in the previous exercise we will have to read the reference manual of the EFM32 development board to get the required knowledge about the microcontroller we are using. [2] The reference manual provides with important information about the different modules and peripherals in the EFM32 development board.

## 2.1 Hardware Description

As mentioned in the previous section, we are using a EFM32 development board to solve the problem. The EFM32 microcontroller familiy is one of the most energy friendly solution of any other 8-, 16- and 32-bit microcontrollers available. The development board consists of a high performance 32-bit ARM Cortex M3 processor, which offers many benefits for the developers like fast interrupt handling and ultra-low power consumption. The development kit supports five different energy modes named from EM0 to EM4. These modes make it easy for us to choose dependent on the task. In case of energy mode 0 (EM0), the CPU is running and all peripherals are active. While in case of energy mode 4 (EM4), all chip functionalities are disabled, but can be turned on by GPIO pin wake-up or Power-On Reset for example.

The Development Kit is intended to be a complete platform for developing applications for the EFM32 microcontroller. The kit includes important components such as 16 MB NOR Flash memory for non-volatile storage and 4 MB SRAM to provide fast memory access to the processor. The embedded debugger allows applications to be downloaded and debugged directly. A set of useful peripherals are provided, and custom hardware can be developed on the prototyping area, where all the microcontroller's I/O pins are made available. For our purpose, we are connecting the gamepad to some of the available GPIO pins.

### 2.1.1 Main Peripherals Description

As seen in Figure 2.1, the development kit has a lot of peripherals available. We can shut down most of them in order to prevent excess current leakage when we don't use them.

There is also a LCD screen which basically makes it easier to monitor different cases under runtime. For instance we can use the screen to keep track of the power consumption via the Advanced Energy Monitor (AEM) included in the kit. The Advanced Energy Monitor is capable of measuring currents in the range of 100 nA to 50 mA. For

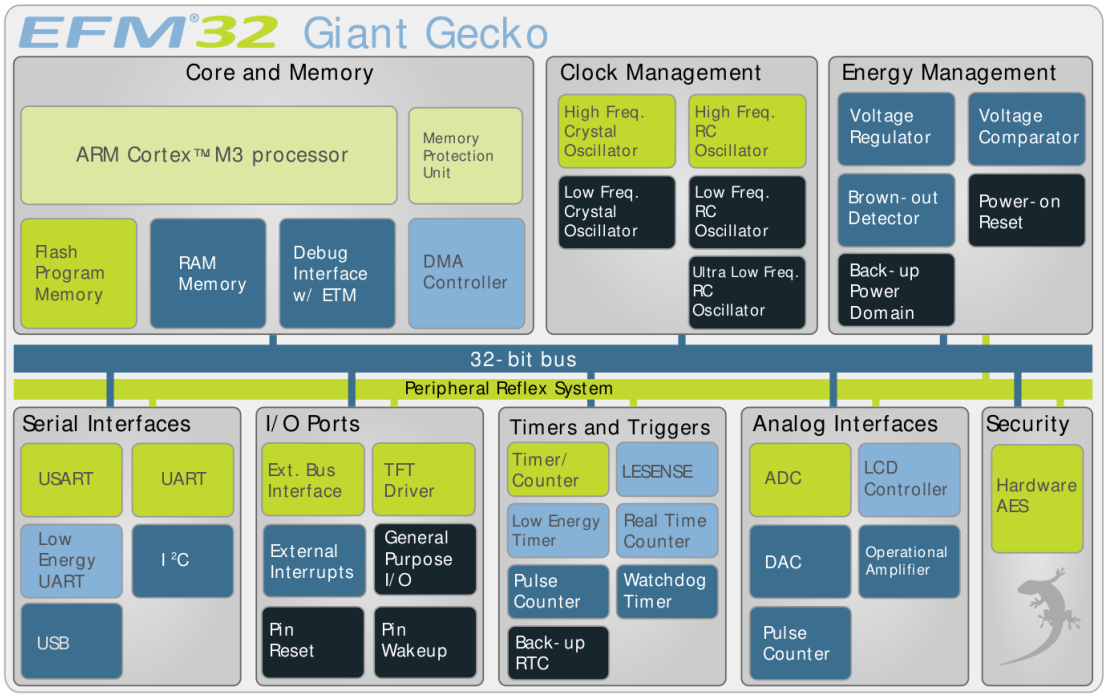


Figure 2.1: Block Diagram of the Development Kit.

currents above 200 uA, the AEM is accurate within 100 uA. When measuring currents below 200 uA, the accuracy increases to 1 uA.

This exercise is mainly focused on playing audio samples. We have two specific inputs for the audio signal, one for the input audio and the other for the output signal. The bandwidth is up to 27 kHz for the output, and we can choose between two different sources. In this exercise we will only need the output because we only generate sound from the memory and not an external source. The connection with external devices is made through the audio out jack plug.

To control what the microcontroller will do under runtime, we are using a gamepad. The gamepad is connected to the prototyping board on the development board.

### 2.1.2 Peripherals Description specifically useful for the exercise 2

#### Timers

The TIMER (timer/counter) keeps track of timing and counts events, generates output waveforms and triggers timed actions in other peripherals. Most applications have activities that need to be timed accurately with as little CPU intervention and energy consumption as possible. The flexible 16-bit timer can be configured to provide PWM waveforms with optional dead-time insertion for e.g. motor control, or work as a frequency generator. The Timer can also count events and control other peripherals through the PRS, which offloads the CPU and reduce energy consumption. [2] We use a

timer in up-count mode (i.e. counter counts up until it reaches the value in TOP value, where it is reset to 0 before counting up again). We simply have to synchronize our counter with the clock peripheral. The timer can raise interrupt on particular events, we will use it on counter overflow. Overflow is set when the counter value shifts from the TOP to the next value when counting up. In up-count mode the next value is 0. It is important to know that the default clock runs at 14 MHz, and timer uses 16 bits register for saving TOP value. Therefore, it is necessary to divide clock provided for a timer, so interrupts occur less often.

In this exercise, we are using low-energy timer to generate sound waves. The LETIMER is a down-counter that can keep track of time and output configurable waveforms. Running on a 32.768 Hz clock the LETIMER is available in EM2. [2] With use of LETIMER instead of traditional TIMER we are able to put a microcontroller into deep sleep mode (EM2) which further reduces power consumption. LETIMER is used for generating interupts on counter underflow. We can precisely set timing by seting TOP value into letimer counter register. This value is also set after every interrupt. We use these interrupts to change a value of Digital to Analog converter. Simplified, we use it to play a sound.

### DAC

It can convert digital values to analog signals at up to 500 kilo samples/second and with 12-bit accuracy. The DAC can generate high-resolution analog signals while the MCU is operating at low frequencies and with low total power consumption. Using DMA and a timer, the DAC can be used to generate waveforms without any CPU intervention. [2] Here, we use it at a single-ended output. We simply have to give the digital value we want to play to the DAC and it converts it into the analog signal that we need for the audio output.

### PRS

The PRS (Peripheral Reflex System) allows configurable, fast and autonomous communication between the peripherals. Events and signals from one peripheral can be used as input signals or triggers by other peripherals and ensure timing-critical operation and reduced software overhead. Without CPU intervention the peripherals can send reflex signals (both pulses and level) to each other in single- or chained steps. The peripherals can be set up to perform actions based on the incoming reflex signals. This results in improved system performance and reduced energy consumption. [2] We only use this peripheral for the advanced part of the exercise: Using DMA for Feeding the DAC. The PRS contains 12 interconnect channels, and each of these can select between all the output reflex signals offered by the producers. The consumers can then choose which PRS channel to listen to and perform actions based on the reflex signals routed through that channel. The reflex signals can be both pulse signals and level signals. Synchronous PRS pulses are one HFPERCLK cycle long, and can either be sent out by a producer (e.g., ADC conversion complete) or be generated from the edge detector in the PRS

channel. Level signals can have an arbitrary waveform (e.g., Timer PWM output). [2]

#### DMA

This peripheral is only needed for the advanced part of the exercise as well. The DMA controller can move data without CPU intervention, effectively reducing the energy consumption for a data transfer. The DMA can perform data transfers more energy efficiently than the CPU and allows autonomous operation in low energy modes. The DMA controller has multiple highly configurable, prioritized DMA channels. Advanced transfer modes such as ping-pong and scatter-gather make it possible to tailor the controller to the specific needs of an application. What the DMA Controller should do (when one of its channels are triggered) is configured through channel descriptors residing in system memory. Before enabling a channel, the software must therefore take care to write this configuration to memory. When a channel is triggered, the DMA Controller will first read the channel descriptor from system memory, and then it will proceed to perform the memory transfers as specified by the descriptor. The descriptor contains the memory address to read from, the memory address to write to, the number of bytes to be transferred, etc [2] It is useful in our project because we can send sound data from memory to the DAC without need to use a CPU each time. We only use one channel so we do not care about channel priority. We use our DMA in ping pong mode: the controller performs a DMA cycle using one of the data structures (primary or alternate) and it then performs a DMA cycle using the other data structure. The controller continues to switch from primary to alternate to primary... until it reads a data structure that is invalid, or until the host processor disables the channel. We need to fill one of this two buffers while we are sending data to the DAC with the other. This mode is a good solution for streaming data for high-speed peripheral communication. It is quite difficult to fully understand all the principles of the DMA and this is the reason why we have not been able to complete the extended part.

## 2.2 Important concepts

### 2.2.1 Sound Wave Synthesis

The sound that we can hear is the result of oscillating vibrations which have two important properties: frequency and amplitude. Every single sound we can hear is the result of the combination of a lot of different frequency and amplitude sines signals. For more simplicity here, we will use another form of signals: the sawtooth signal (it is a ramp which grows until it reaches its maximal value and then it goes back directly to its initial value and so on). Humans are able to hear sound with frequency up to 20 KHz so according to the Shannon theory we should use a sample frequency at least twice as big. Each period of the sound we play is divided in a certain number of sample. Each sample must be written in a continuous stream to a data register in the DAC. This is easiest done by setting up a timer interrupt that gives us an interrupt every time we need to push a new sample to the DAC.

### 2.2.2 Interrupts vs polling

An important point in this presentation of the hardware we have is that some of these peripherals can generate output. We simply have to check the signal of the pin `PE0`. When a GPIO (gamepad) interrupt occurs, and the interrupt is caused by a falling edge of PE0, the interrupt flag register in the board controller should be read to determine which peripheral caused the interrupt. We will use this concept of interrupts through all this exercise, to optimize the resource usage. An interrupt is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention. An interrupt alerts the processor to a high-priority condition requiring the interruption of the current code the processor is executing. The processor responds by suspending its current activities, saving its state, and executing a function called an interrupt handler (or an interrupt service routine, ISR) to deal with the event. This interruption is temporary, and after the interrupt handler is finished, the processor continues in execution of the stopped code. [1] We can enable only those interrupts we want to use. We will use it in order to read the state of the buttons on the controller so we do not need to always read their state and just enable the interrupt and create a handler for this.

We use the same method for implementing a timer. Active polling of a current value of the timer could be used to achieve the same result. However, timer peripheral has built-in functionality for generating interrupts, which we can use to avoid this impractical polling. The exception vectors specifies where in memory handlers for exceptions and interrupts are located.

### 2.2.3 Sleep modes

In order to prevent the kit to use too much energy we can power on only the I/O controller we will use. For instance, we will only use the GPIO controller, the DAC, the PRS, the DMA and two timers in this exercise so we just enable their clock. We can also use the sleep mode. When the CPU is no longer doing anything useful (e.g. waiting for button input) we will stop the CPU and will wake it up when it receives an interrupt, and go back to sleep after the interrupt handler returns. The Energy Management Unit (EMU) manages all the low energy modes (EM) in EFM32GG microcontrollers. The energy modes range from EM0 to EM4, where EM0, also called run mode, enables the CPU and all peripherals. The lowest recoverable energy mode, EM3, disables the CPU and most peripherals while maintaining wake-up and RAM functionality. By default, EM0 is activated. It is the highest activity mode, in which all functionality is available. EM0 is therefore also the mode with highest energy consumption. In this exercise we will switch between Energy mode 0 and Energy mode 2, which means that the high frequency oscillator and the high frequency peripheral and MCU clock trees are inactive. Energy mode 2 is used during idle state when processor is waiting for a button press. We use sleep mode in between low-energy timer interrupts as well, to minimize power consumption while playing a sound.

### 2.2.4 Software Toolchain

A big difference with the previous exercise is that we will use C language rather than assembly language. It is a higher level programming language which allows use more flexibility and a more efficient and readable code. For instance it is easier to access the hardware since we only need to use pointers that point to the memory mapped I/O registers.

We will use the GNU Toolchain, GNU is a project with the main goal to provide a free and open operating system with all accompanying tools for software development. This is why we will use GNU GCC as compiler, GNU LD as linker (with more object files and useful libraries as previously), GNU Make to build the project and the GNU Debugger (GDB) to debug the program. It is a powerful program which gives many possibilities to monitor the program execution, stop the execution and inspect the contents of registers and memory. It also gives us the possibility to single step the program execution, which executes line by line from the original source code, so that the user can monitor what is happening in closer detail. [3] As we do cross development, we must have a possibility to run the debugger on a PC while the program we are debugging is being executed on the development board. To do this, the debugger communicates through a program called gdbserver. The gdbserver lives in the middle between the development board and GDB. [3] We will run GDB in Emacs to be able to see which line in the source code is being executed. Emacs allows use to see everything that happens with GDB. To connect the development board through gdbserver we just have to run the .gdbinit file which is provided.

# 3 Methodology

In this exercise we have to be able to play different sounds for our game according to which button is pressed. We therefore have to use the GPIO to know the state of the buttons and a timer for creating sound waves. We also need to create sounds we play and to store them. Eventually we need to convert the digital data of the sound into an analog one in order for players to hear this sound which is the role of the Digital to Analog Converter (DAC). We will also see in this chapter that there is a more efficient solution using the DMA to send data to the DAC directly rather than with use of the CPU. Interrupts are a key point of our solution.

### 3.0.5 GPIO

We can reuse a part of the first exercise code (initialization of GPIO for instance and how to handle their interrupts for instance) because it is not complicated to translate from assembly language to C language. It is even easier to do so because now we can directly use the pointer to access to the register in memory, this is much more convenient than using LDR and STR instructions in assembly language. The rest of the initialization is done like in the previous exercise. The handler is different. We set up a flag for the sound we want to play according to the button we press: we just need one variable which can take four different values because we only use the four buttons on the right side of the gamepad. We will use this variable in a main loop to know which sound we want to play.

### 3.0.6 Timer

We have decided to use low-energy timer (LETIMER) peripheral, which is also available in a deep sleep mode. That allows us to switch MCU completely off, while still keeping track of time and waking MCU up when needed. To run LETIMER, low energy peripheral interface clock should be allowed first. Then we have to enable low frequency crystal LFXO and set it as a low frequency clock. When this is done, we can enable clock of LETIMER and set compare value, top value and enable interrupts from LETIMER. Interrupts also have to be enabled in NVIC.

### 3.0.7 Sound data

Our sound data is basically a sequence of ten single frequencies which lasts a few seconds in total. We created four different sequences for different parts of the game. For each sequence we have two arrays to store the data: one for the frequency we want to play and another one for the duration of each frequency. These two arrays need to have

the same dimension. We choosed the values of these arrays a bit by instinct, but with frequencies around the 440 Hz and for the duration we just wanted to be sure that the total sequence does not last more than a few seconds.

### 3.0.8 Playing sound with use of DAC

As for all the peripherals we use, we need to create a setup function for the DAC. We use both channels of the DAC. The clock for the DAC has to beenabled first, using HFPERCLKEN as usual. Then we setup the DAC control register with DAC output to pin enabled, use a clock division factor of 32 and enable Control Registers to use them for data conversion. The idea of our solution is that when a button is pressed we use a fucntion that for each cell of the array for the sound, we calculate some useful parameters for the function which will send data to the DAC, here are the main parameters:

– samples_in_period = (SAMPLE_RATE / freq);
– dac_addition = (DAC_MAX - DAC_MIN) / samples_in_period;
– number_of_periods = SAMPLE_RATE * time_milisec / samples_in_period;
– number_of_periods /= 1000;

We want to play our sounds period by period because it is the smallest time window we can use. It is therefore important to know how many samples we need to send during each period. We then need to know the incrementing step we add to the previous sample to create the sawtooth signal and we eventually need to know how many periods we will need to play to have the complete duration of the sound. Once we have all these parameters we can start the timer to get its interrupts which call the function to send a sample to the DAC. In this function we have to be sure that we have not played the total duration of the sound yet and we add the incrementing step to the value of the DAC data (if we are at the end of a sawtooth we need to get the DAC value back to its initial value) and so on. Once we have finished to play the last period of the sound we stop the timer to stop playing the sound. We also disable DAC channels to lower power consumption when DAC is not being used.

### 3.0.9 Playing sound with use of the DMA

#### Setting up the DMA

DMA is an alternative solution to send the sound to the DAC to avoid to use CPU each time. We have only done the setup for the DMA, and started to try to use it. As specified in the background, we use both DMA and PRS for this part so we need to enable their clock. The PRS need a timer to be triggerred so we use TIMER0 for this (we then need to setup TIMER0 the same way that we've done it with TIMER1). We also need to set up the channel descriptor. In the end destination pointer we use the adress of the CH0 of the DAC and for the end adress of the source we do not set it up now because it will depend on which sound we want to play so we will use it with the playing function. Last part of the descriptor is the channel_config, we only need to set up the ping pong mode at the beginning of the channel because we want to keep

the other option at their value by default: adress increment (byte), data size (byte) and arbitration (no need because we only use one channel of the DMA).

To complete our configuartion we have to disable the single requests (i.e., does not react to data available and wait for the buffer to be full), enable buffer-full requests, select PRS channel 0 as input to start the conversion, enable DAC conversions to be started by a high PRS input signal, enable interrupt from DMA channel 0 and eventually enable channel 0enable DMA.

#### Using the DMA

It was not possible for us to use the DMA to really send signal to the DAC by the lack of time but we will try to explain our idea here. Once the DMA is correctly configured we mainly need to change the source destination for the sound and give the address of the next sound. We should change this value each time we want to play another frequency (next cell of the array), we should therefore do this in the main. For each odd step, the alternate source address is getting the frequency of the next sound we want to play (because of ping-pong mode), and vice versa for even number. The primary source address will read the next sound we want to play because we are in ping pong mode.

## 3.1 Testing:

We did some tests to check if the microcontroller is working as we want. We have provided some of the scenarios and the observations below.

**Scenario 1:** Press several buttons at the same time:
**Expected result:** Only the sound from the last pressed button is played, and the previous sound is stopped immediately after a new button is pressed.
**Observed result:** The same as expected.

**Scenario 2:** Press the same button several times:
**Expected result:** The sound is repeated immediately right after the button is pressed again.
**Observed result:** The same as expected.

**Scenario 3:** Check that the state goes back to EM2 after playing a sound:
**Expected result:** After playing a sound, the state goes back to deep sleep mode (EM2).
**Observed result:** Based on power consumption, we assume that microprocessor goes back to deep sleep mode after playing a sound.

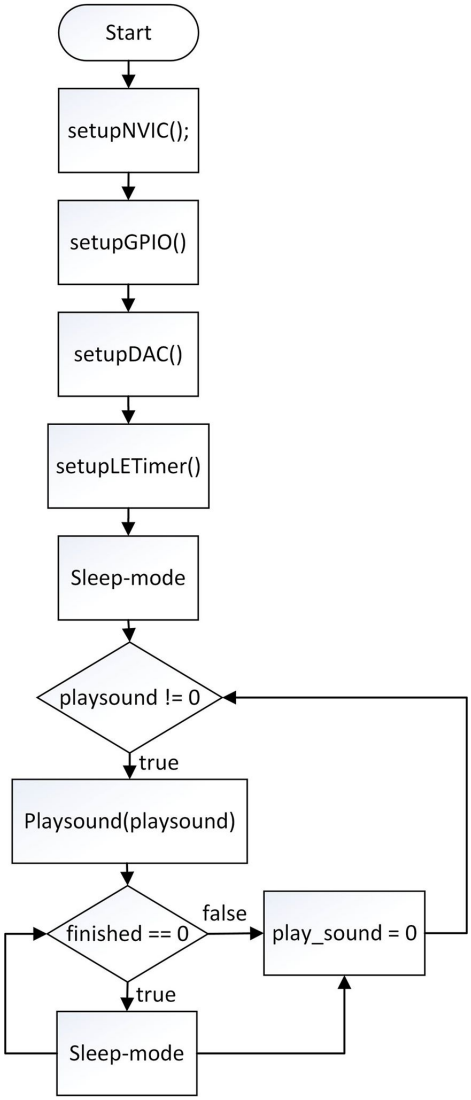


Figure 3.1: Flow chart of the main functions and the loop.

# 4 Results

In this exercise we have managed to play a predefined sound after pressing a button. Generated sound corresponds to an expected sound.

With our final implementation we have achieved huge improvement in terms of power consumption when comparing to a first prototype. To improve a power consumption we have used deep sleep mode (EM2) with cooperation with low-energy timer. To further improve consumption while processor is in idle state, waiting for a button pressed, we have also disabled DAC channels. Results of measurements with different optimization techniques implemented are shown below.

| Index | Description | idle state | playing state |
|---|---|---|---|
| 1 | Without sleep | 4.0mA | 4.5mA |
| 2 | Sleep while waiting for button press | 1.6mA | 4.5mA |
| 3 | Sleep while waiting for button press and during playing | 1.6mA | 3.3mA |
| 4 | Deep sleep (EM2) | 0.2mA | 3.3mA |
| 5 | Deep sleep + turning off/on DAC | 0.002mA | 3.5mA |

Figure 4.1 shows the power consumption we got earlier in the exercise phase. On idle state, the power consumption was on about 4mA and while playing a sound the power consumption increased up to 4.5mA. At this point of stage we couldn't say that the microcontroller was working in an efficient way, becuase the microcontroller was consuming power when it didn't had any reason to.


Figure 4.1: Without sleep mode.

After implementing different methods to make the microcontroller work in a more efficient way, we could see a big improvement. The result is shown in Figure 4.2. On idle state, the power consumption was on about 1.93$\mu$A. Now the microcontroller is working 2000x more efficient when it is in idle state. In our case, an eventually high power consumption wouldn't have so much effect, since we always have additional power source. But in cases where the microcontroller is running from external power sources, these implementation can be quite critical.

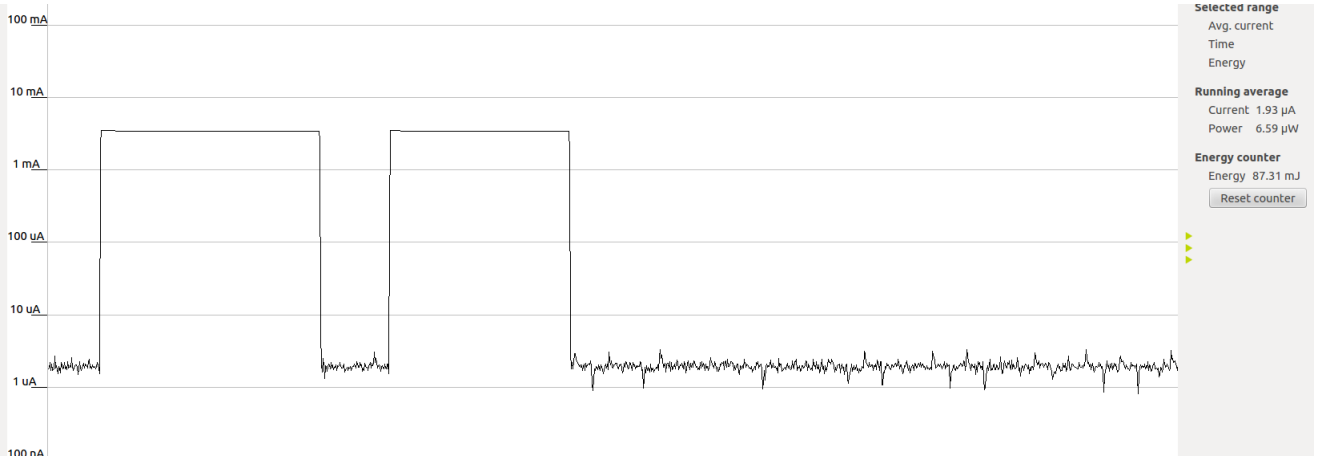


Figure 4.2: Deep sleep + turning off/on DAC.

# 5 Conclusion

In the second exercise, we have built on the knowledge from the first exercise. We started by re-coding the exercise 1 into C programming language. And then we expanded the code to accomplish the exercise 2. We had to read the datasheets very carefully in order to find the precise information we needed such as the role of the registers and how to use them to achieve the functionality we wanted. Programming DAC, GPIO, interrupts, low-energy timer and sleep-mode are some of the learnful experience we have accomplished during this exercise. Not only was the programming itself learnful, but also the result it provided. For instance the big difference between running a microcontroller with sleep mode and without is quite interesting. Finally we also wrote a technical report using the template in Latex, which also was a useful experience.

## 5.1 Evaluation of the Assignment

This exercise was well made, the difficulty level and the time we needed to complete the assignment was accurately enough for us. Sometimes it was a bit challenging to solve different small problems, like the attempt of DMA implementation, but we like it that way!

# Bibliography

[1] *Linux Device Drivers, Third Edition, Chapter 10. Interrupt Handling.* O'Reilly Media, 2005.

[2] *EFM32GG Reference Manual.* Silicon Labs, 2013.

[3] *Lab Exercises in TDT4258 Low-Level Programming.* NTNU, 2015.