NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4258 Low-Level Programming
Laboratory Report

Exercise 2 - Report

Group 8:

Matias Varnum Christensen
Magnus Melseth Karlsen
Vebjørn Isaksen

October 17, 2016

1 Overview

This second exercise will give us a further introduction to the EFM32GG-DK3750 developer board, and how to write C code for the EFM32 microcontroller using the Thumb 2 instruction set without an operating system. Our final delivery includes a solution for detecting button presses using the supplied GPIO gamepad board, and playing sounds using the DAC.

Our original intent was to render the audio to be played on the fly. The plan was to have two timers. One running at 16000 Hz and copying the data from an sample ring-buffer to the DAC. And another timer running at 30 Hz that would render the audio in large chunks and keep a bit ahead of where the other timer was reading. This would have given us the ability to change the aspects of the audio effects programmaticly and mixing multiple audio effects together. We managed to get this system working, and it worked perfectly when we tested it by letting the system fill the buffer once in advance. Unfortunately we had problems when trying to render the audio in real time.

A representation of the problems with rendering sound samples on demand can be seen in Figure 1.1. The problem arises when we cannot generate and write the samples to memory fast enough. In the buffer, we have a *read cursor* and a *write cursor*. The read cursor points to where in the ring-buffer we should read the next sample from, while the write cursor points to where we should write the next sample. To get a reliable audio output, the write cursor should always be ahead of the read cursor. In Figure 1.1, the blue part is the samples, while the green graph is [write cursor − read cursor], meaning how far ahead the write cursor is of the read cursor. When it goes below zero, the write cursor is no longer in front of the read cursor, which results in a failure to play the desired audio. We plotted this data using *dump binary memory samples 0x200010b4 0x20002ff4* in GDB and writing a Python script to unpack and plot the data. This script proved to be very useful for debugging the rendering system by plotting the waveforms. We attempted to improve the speed of the audio rendering system by modifying it to use fixed point arithmetic instead of floating point, using a fixed point library we found [1]. This improved the speed of the audio rendering quite a bit, but not enough to be fast enough for real-time audio.

We created two different implementations as per the requirements of the exercise. The first implementation uses a *timer* to generate the sound effects, and *polling* and *busy-wait* to detect button presses, while the second implementation uses *interrupts* and deep sleep mode. We attempted to implement the use of the *DMA* but where unable to get it working in time.

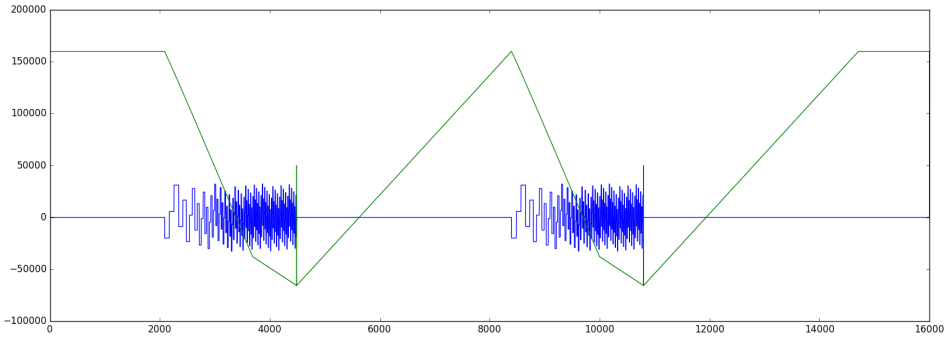

Figure 1.1: The problem with rendering in real time.

Offset	Bit Position																													
0x044	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset															0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access															RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Name															DAC0	ADC0	PRS	VCMP	GPIO	I2C1	I2C0	ACMP1	ACMP0	TIMER3	TIMER2	TIMER1	TIMER0	UART1	UART0	USART2	USART1	USART0

Figure 1.2: The CMU_HFPERCLKEN0 register.

1.1 Setting up the board

1.1.1 Enabling Clock of Components

To use some of the components, we have to enable their clock. This allows the clock to reach the different components, ultimately "turning them on". It is a good idea to only enable those that are going to be used, as every additional component will use power.

To enable the clock of the components we need, we need to change the *CMU_HFPERCLKEN0* register, shown in Figure 1.2. We then write 1 to the desired bit position in the register. The bit positions for the components we used in this exercise are shown in Table 1.1.

Component	Bit position
GPIO	13
Timer1	6
Timer2	7
DAC	17

Table 1.1: The bit position to enable or disable clock in the CMU_HFPERCLKEN0 register.

Component	Bit position
GPIO ODD	1
GPIO EVEN	11
Timer1	12
Timer2	13

Table 1.2: The bit position to enable or disable the handling of interrups in ISER0 register.

1.1.2 NVIC

The NVIC needs to be set to enable the handling of interrupts from the GPIO and the timers. The GPIO is split into odd and even pins. Since we use some odd and even pins, we enable both. To do this, we change the *ISER0* register. We write 1 to the deisired bit position in the register, shown in Table 1.2.

1.1.3 GPIO and Gamepad

To be able to read the buttons from the gamepad and write to the LEDs, we have to configure some of the GPIO pins. In our case, the LEDs are connected to GPIO port A (GPIO_PA), occupying pins 8 to 15. The buttons are connected to GPIO port C, occupying pins 0 to 7. The pins for the LEDs are configured as output pins, with a high drive strength. The button pins are configured as input pins.

1.1.4 Timers

To use the timer, we first set the period of the timer. The period can be calculated as shown in the example in Equation 1.2. The period is written to the memory address given in *TIMERn_TOP*.

To tell the timer to generating interrupts, we set *TIMERn_IEN* to 1. Finally, we write 1 to *TIMERn_CMD* to start the timer.

1.1.5 DAC

To use the DAC, we need to set the output mode and the prescaler setting. We do this by writing to the *DACn_CTRL* register. In our implementation, we set the DAC output mode to *PIN*, which outputs to the pin, but disables other output options. We set the prescaler to 101, which equals 5. This gives a frequency of 437.5 KHz, as shown in Equation 1.1.

The only thing left to do is enabling the left and right channel. This is done by writing the value 1 to both *DAC0_CH0CTRL* and *DAC0_CH1CTRL*. We can now play sounds by writing samples to *DAC0_CH0DATA* and *DAC0_CH1DATA*.

$$\frac{14\text{MHz}}{2^5} = 437.5\text{KHz} \tag{1.1}$$

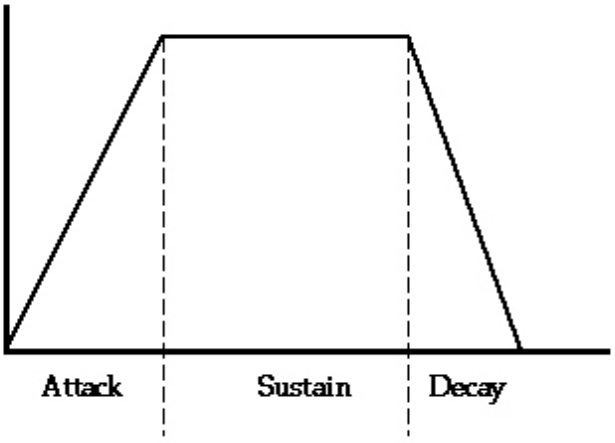

Figure 1.3: The way sound is represented in our code.

1.1.6 Pre-rendering the sounds

Since we ran into problems when rendering sound on the fly, we chose to render them at boot. This is done by sampling the sound, which later will be sent to the DAC.

The way the sound is represented in our code, is by dividing each sound into three parts, as shown in Figure 1.3. The attack, sustain and decay parts are independently sampled, and placed together in memory. The system is capable to have a separate frequency and length in each part and transition between them. We are also able to generate sound using several different source waveforms. These are sinewave, squarewave, sawtooth and noise. These parameters allow us to produce a large variety of different sounds.

Since the board does not have a FPU, calculating samples using *float* values ended up taking a lot of computational power. To improve performance, it would be smarter to do calculations on integers. That is why we chose to include a third-party library for fixed point arithmetics. A fixed-point library by Ivan Voras and Tim Hartrick [1] was chosen. This allows us to store numbers with a fixed number of digits after the radix point, while still being able to do computations on them in an acceptable time.

1.2 Baseline Solution

The baseline solution uses *polling* and *busy-wait* to retrieve the state of the buttons. The state of these buttons are constantly checked, and if the state of one button changes, a sound will be played. Each button can be customized with the 8 different sounds we have implemented.

The sound is played by copying the pre-rendered sounds from memory and into the DAC. One sample will be transferred to the DAC, and the DAC will play this sample.

Another one will be written into the DAC, and the process is repeated. To control the rate of data being written to the DAC, we use a *timer*. The timer executes 8000 times in a second, or every 1750 tick (see Equation 1.2).

$$\frac{14\text{MHz}}{8000} = 1750 \tag{1.2}$$

By constantly checking the state of the gamepad, more power is consumed compared to techniques like *interrupts*. By using the CPU to transfer the samples from memory to the DAC, more power is consumed compared to using the *DMA*, which can transfer the sounds more effectively.

1.3 Improved Solution

The improved solution uses a technique called *interrupts* to retrieve the state of the buttons. When no button is pressed or released, the board will enter a power saving mode. This greatly reduces power consumption, since the processor only does work when a button is pressed or released.

When a button is pressed or released, the MCU will exit its low energy mode, and the exception vector table will point to where in memory our handler code is located.

When a button is pressed, a sound will be played. As with the Baseline Solution, each button can be customized with the 8 different sounds we have implemented.

The timers does not run when the system is in deep sleep mode. We therefore temporarily disable deep sleep when starting to play an audio effect. We do this by writing *0b0100* to *SCR*. Once the effect is done playing, we reenable it by writing *0b0110* to *SCR*.

The startup melody is run by a second timer. We set the prescaler divider of this timer to 128. This slows this timer down by a factor of 128 times and allows us to use each tick of the timer to play one audio effect in the melody.

We attempted (but failed) to implement the use of the DMA to transfer audio to the DAC. The DMA controller is capable of moving data without the use of the CPU. This can reduce overall power consumption and the load which would in other circumstances be on the CPU. A basic representation of the DMA can be seen in Figure 1.4.

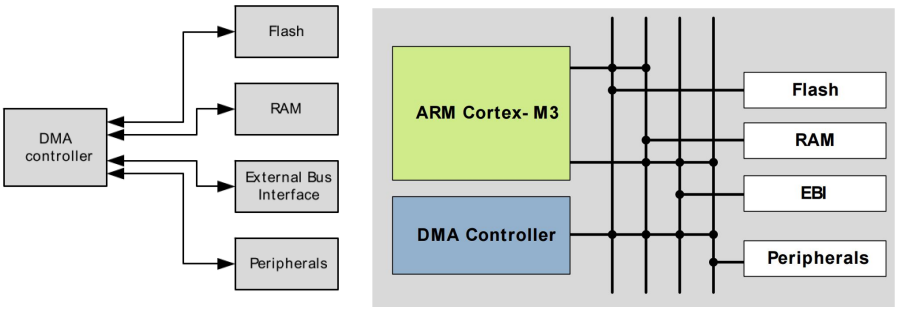

Figure 1.4: Illustration of the DMA.

2 Energy Measurements

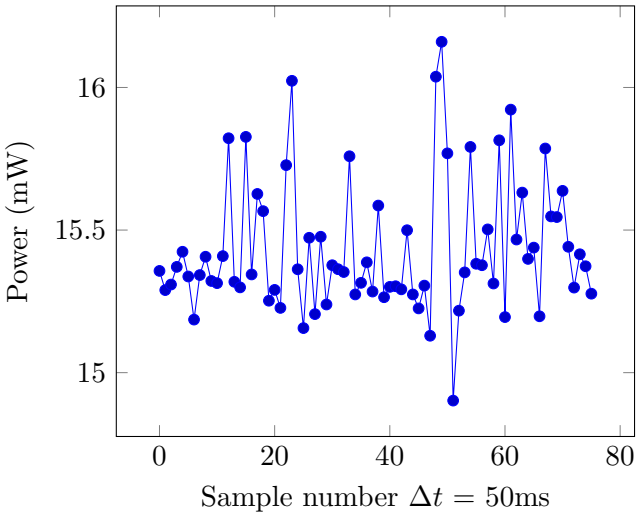

Figure 2.1: Power consumption plot using the polling (baseline) solution.

When using the baseline solution, the idle state of the board is consuming about 15.5 mW. This is represented in Figure 2.1. When a button is pressed, the power consumption spikes to about 16 mW. This is about 3 mW more than in the baseline solution from exercise 1. The reason is that in this exercise, the clock for the DAC is turned on, resulting in the DAC using power. The spikes visible in the graph are caused by the buttons being pressed and the audio being written to, and played by, the DAC.

If we now look at the improved solution, we can see a much lower power consumption. In Figure 2.2, we can see the power consumption of the board when in sleep mode. The spikes are caused by the buttons being pressed (and thereby exiting deep sleep) and the audio being written to, and played by, the DAC. This shows a power usage of about 1 mW when left untouched, and a power usage close to the baseline solution, 16 mW, when a button is pressed and sound is played.

The reduction in power consumption from the solution using *polling* to the solution using *interrupts* is enormous. If we consider 15 mW as the average power consumption when idle using polling, and 1 mW as the average power consumption when idle using interrupts, the baseline solution uses a substantial amount of power compared to the improved solution when idle. This shows that putting the board into a more suitable power mode makes a large difference on the overall power consumption.

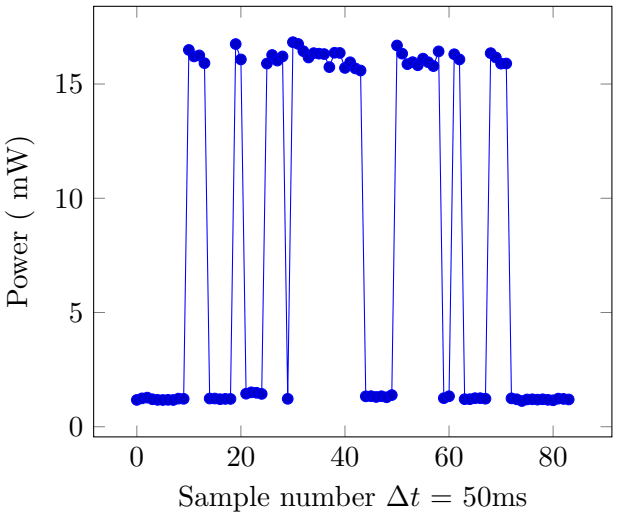

Figure 2.2: Power consumption plot using the interrupt (improved) solution.

Bibliography

[1] Ivan Voras and Tim Hartrick. Fixed point math library for c. `https://sourceforge.net/projects/fixedptc/`, 2010. [Online; accessed 13-Oktober-2016].