NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4258 Low-Level Programming
Laboratory Report

Exercise 2

Group 7:

Vegard Seim Karstang
Johannes Andersen
Daniel Fedai Larsen

October 14, 2016

1 Overview

In exercise 2 we were tasked with writing a program in C. The program should be able to play a start-up sound and three sound effects. We were to create two solutions, one baseline solution using busy-waiting and one improved solution using interrupts. In addition, the task said to use a consistent coding style which we did to the best of our efforts. Lastly, we were to utilize multiple makefile targets, allowing for easy building and uploading of the two different versions of our solution.

1.1 Baseline Solution

No matter which solution is employed, the following functions are always run at the start of the program: *setup_gpio(), setup_dac(), setup_timer()*. We also turn off all lights on the game pad to make sure that no power is wasted. In both solutions the buttons 1-4 are used to play the different sounds. Button 1 plays the hit effect, button 2 plays the metal effect, button 3 plays the win effect and button 4 plays the melody.

For our baseline solution we used busy-waiting in order to detect buttons being pressed. This is done in our *main()* function in **ex2.c**. By writing *make baseline* when wanting to make our solution, we enter the *ifdef BASELINE* section in *main()*.

Here we start by creating two variables that will help us keep track of which buttons were pressed (*old_buttons*) and the value of the timer last time we entered the while loop below (*old_time_value*). We then proceed to start our while loop which keeps our program running. We then read the value of the timer from *TIMER_CNT* and store it in a temporary value *time_value*. We proceed by checking whether or not our *time_value* is less then *old_time_value*. If it is, this means that the timer has counted up to our set *SAMPLE_PERIOD*, specified in the top of **ex2.c**, and the next value for the current sound effect should be played by calling *play_effects()*. It is worth noting that playing the start-up melody has precedence over playing the hit effect, which again has precedence over the metal hit effect, which again has precedence over the win effect (accomplished by the if-else statements in *play_effects()*).

We finish each iteration by updating *old_time_value* to the current *time_value*, and detecting which buttons are pressed. The result of which buttons are pressed, along with the bit-wise negation of *GPIO_PC_DIN*, is passed as parameters to *reset_counters* which allows us determine which sound effect to play according to which button(s) are pressed.

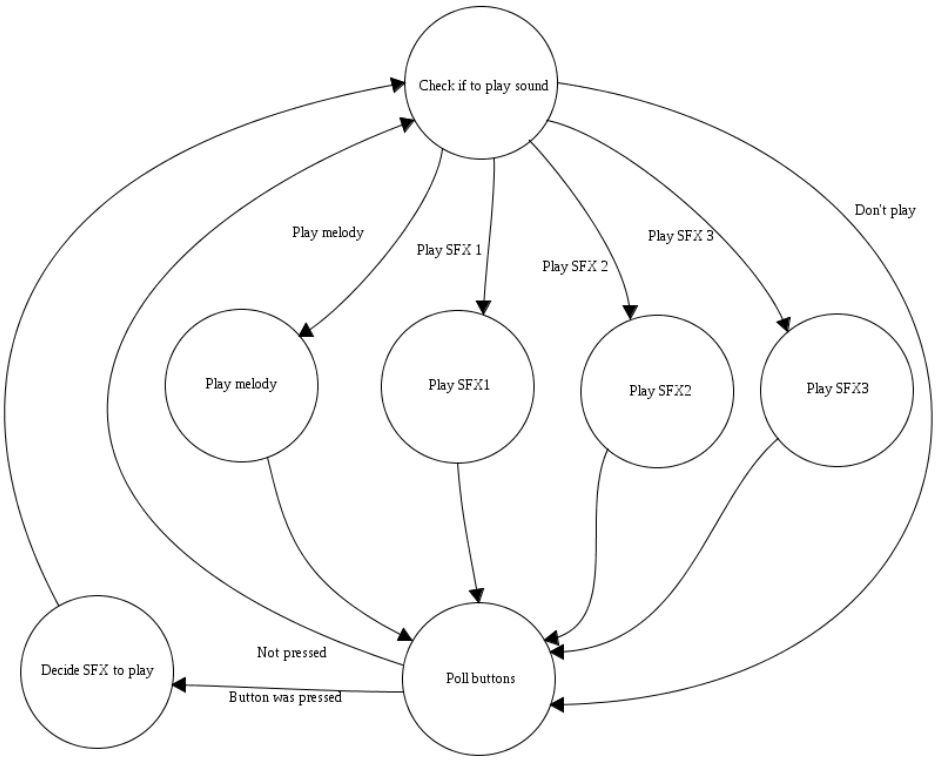

Figure 1.1: Baseline solution FSM.

1.2 Improved Solution

The improved solution was implemented using interrupts for both the timer events and the GPIO. In our *main()* function we set up the NVIC to send interrupts from the GPIO and TIMER1 to the CPU. We then write 0 to *EMU_CTRL* and 2 to the *System Control Register*. This enables the CPU to enter EM1 on a *wfi* or *wfe* instruction. A *wfi* instruction is then added via inline assembly. This causes the CPU to enter EM1 while it waits for either a timer or GPIO interrupt.

We chose to use EM1 because this is the only sleep mode that allows the high frequency oscillator to run while the CPU is sleeping. We need the high frequency oscillator because it runs the timer.

When a GPIO interrupt happens one of the GPIO interrupt handlers are run. Both interrupt handlers call the *handle_gpio_interrupt* function. This function reads the *GPIO_IF* register and a bitwise negation of *GPIO_PC_DIN*. These values are then passed to the *reset_counters* function. The *reset_counters* function determines which sound should play by resetting the counter that keeps track of how much of a sound has already been played. At the end of the interrupt handler the interrupt is cleared to prevent the same interrupt from firing multiple times.

When a timer interrupt happens the TIMER1 interrupt handler is run. This function calls the *play_effects* function which plays the correct sound effect. The timer interrupt is then cleared by writing 1 to *TIMER1_IFC*.

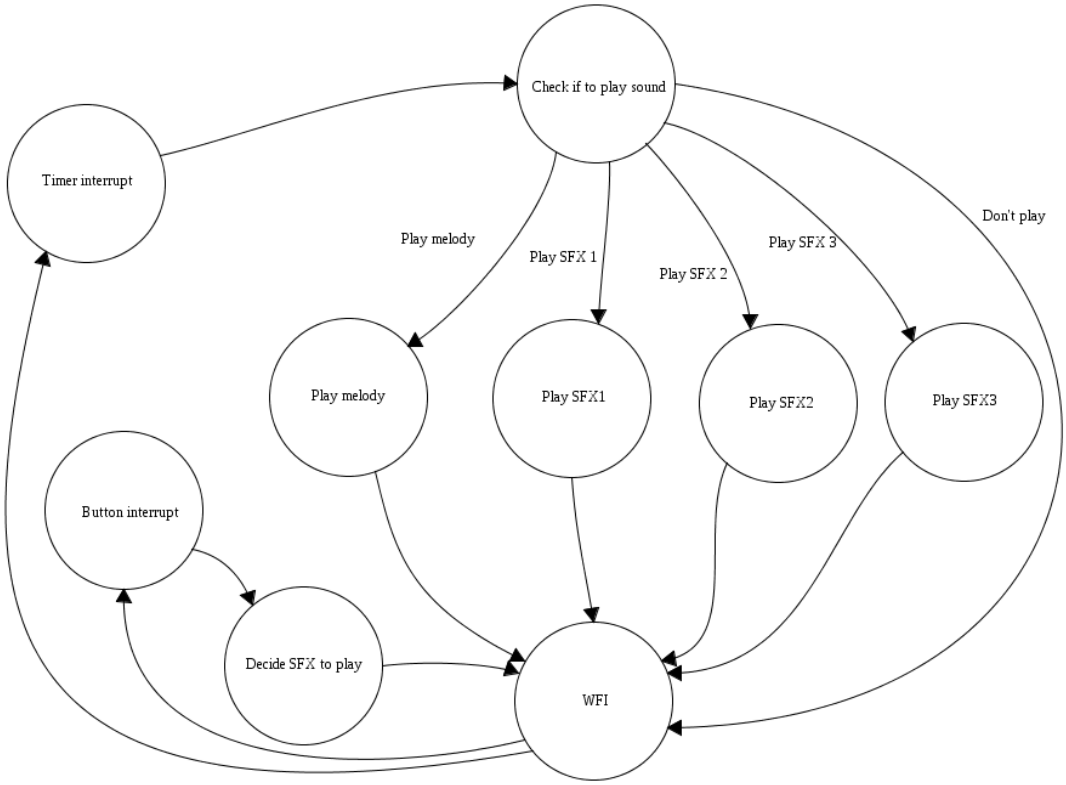

Figure 1.2: Improved solution FSM.

2 Energy Measurements

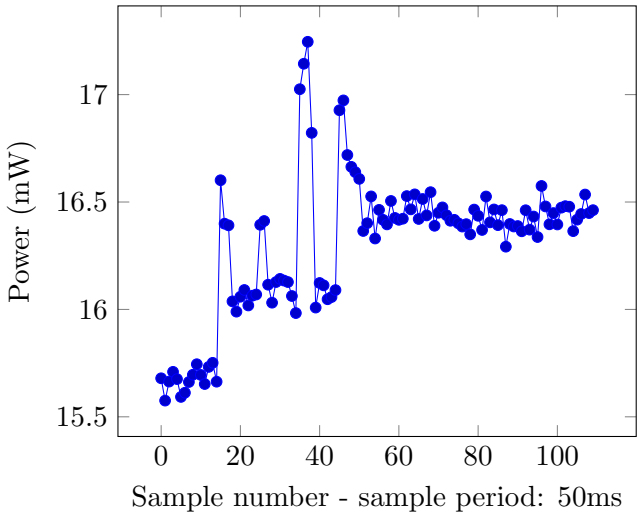

Figure 2.1: Power consumption plot generated from eAProfiler data using busy-wait (baseline solution). The 1st spike (sample number 18) occurred when playing the hit effect, the 2nd spike (sample number 27) occurred when playing the metal hit effect, the 3rd spike (sample number 38) occurred when playing the win effect and the last spike (sample number 45) occurred when playing the start-up melody.

The first thing we notice by looking at the two figures is that the power consumption during the playing of sound effects are generally higher in Figure 2.1, although for our start-up melody, the two solutions are almost at an equal power consumption level. However, there are some noticeable differences. First of all, Figure 2.2 has a much more stable power consumption and most importantly; less power consumption when no sound effect is playing. This is due to the micro controller entering *sleep mode* when waiting for the next sound effect to play, while in Figure 2.1, the micro controller is constantly polling for events that will play sound effects.

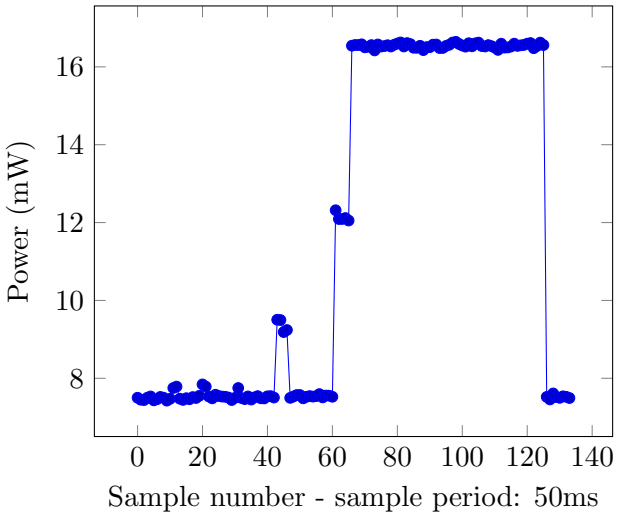

Figure 2.2: Power consumption plot generated from eAProfiler data using interrupts (improved solution). The two first sound effects played (hit effect and metal hit effect, sample number 15 and 20 respectively) are barely noticeable. The first noticeable spike (sample number 43) occurred when playing the win effect and the last spike (sample number 60) occurred when playing the start-up melody.