NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4258 Low-Level Programming
Laboratory Report

Labrapport - Exercise 2

Group 15:

Christian Odden
Haakon Jonsson
Sondre Slåttedal Havellen

October 17, 2016

1 Overview

In this exercise we wrote a program for EMF32GG that could generate and play music and sounds. Further this could be controlled by the buttons on the gamepad. The exercise asked us to make two versions of our program, using two different techniques. A baseline polling solution and a improved interrupt solution, then to measure the energy consumption of the different versions.

Our sound player system works by adding (possibly multiple concurrently playing) sounds together. If a button is pressed, a sound player is activated and played based on different properties like tempo and sound type, as well as whether or not the sound should loop. Each sound player contains it's own state with a corresponding track and track positions. Tracks are made by creating integer arrays where each integer corresponds to a given note. By doing it this way, it is easy and fast to create new songs.

Because of the CPU demanding system we use, we had to lower the sample-rate by quite much. This is not a serious problem as the sound waves is mostly squares and saws, and sounds about the same with this sample rate. A more serious side effect of this is the frequencies is not calculated with enough precision. This can most notably be heard with melodic effects and songs where each note is slightly off pitch. This was an issue in both the interrupt-based version as well as the polling-based version.

1.1 Baseline Solution

The baseline solution is implemented using polling in a busy-wait loop. The loop continually checks the state of the timer register, and pushes new samples to the DAC each timer period. It also checks the state of the buttons, and activates audio tracks based on what buttons are pressed. The buttons is checked with the same frequency as audio is pushed to the DAC (12000Hz). This may not be the best option, but makes the implementation and bookkeeping of counters much easier. Checking with a higher frequency would either way be pointless as pressing buttons this fast is impossible. Figure 1.1 show a flowchart of the polling based solution.

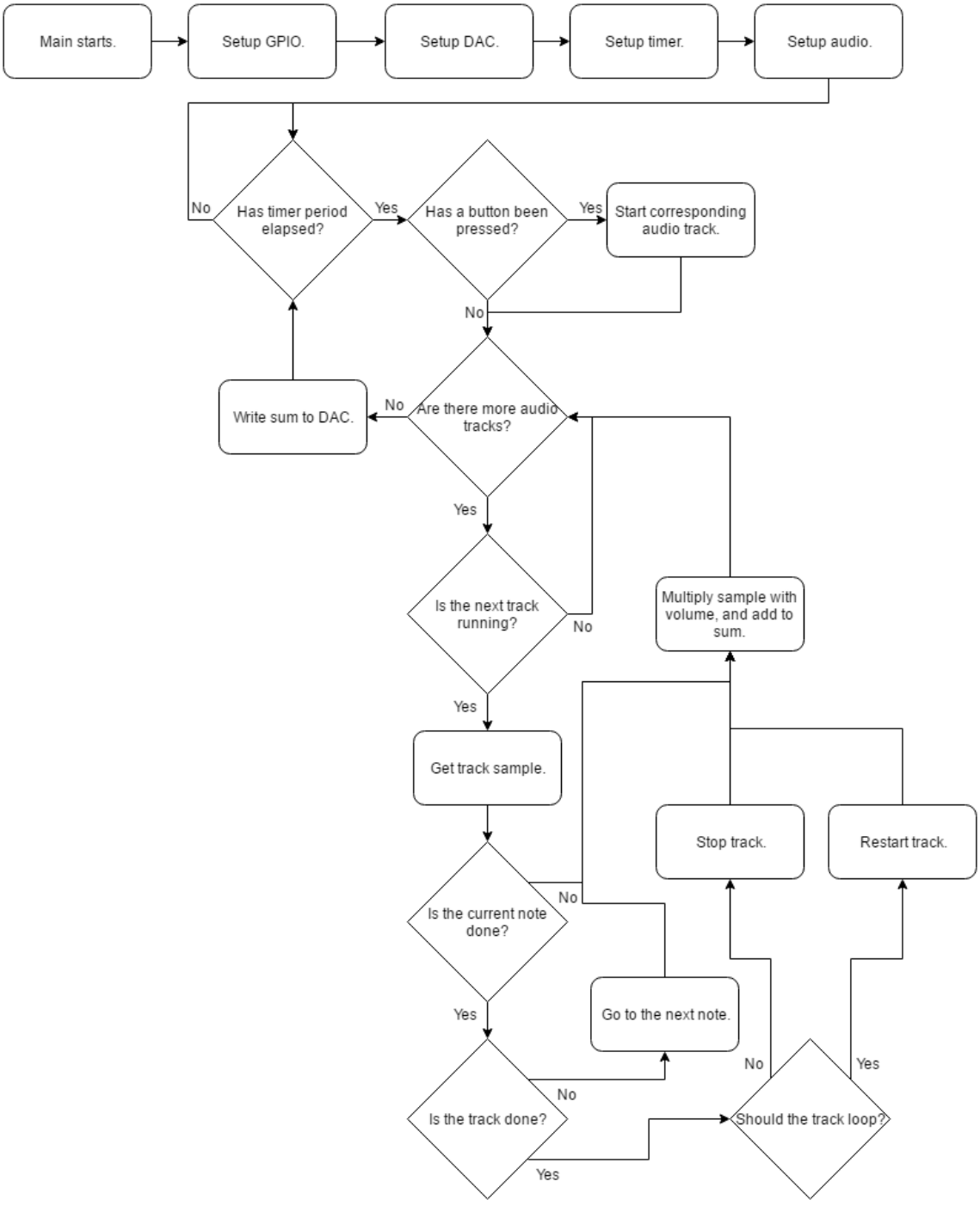

Figure 1.1: Flowchart of the Polling based solution.

1.2 Improved Solution

The improved solution is implemented using interrupts for both the timer and the buttons. Instead of continually checking whether or not the timer value has reached a given value, the timer generates an interrupt when this value is reached. Similarly, instead of always checking the state of the buttons, we only check which button is pressed when there is a button interrupt. Apart from this, the improved solution uses mostly the same logic as the baseline solution. The interrupt based structure is shown in the flowchart in figure 1.2.

The system works by entering deep sleep mode, and then waiting for a button event. When a button event happens a song or effect is triggered. For the duration of given song or effect, samples is generated and pushed to the DAC. When no more sound players is active, the system enters sleep mode again. This saves a lot of power consumption when no sound is active.

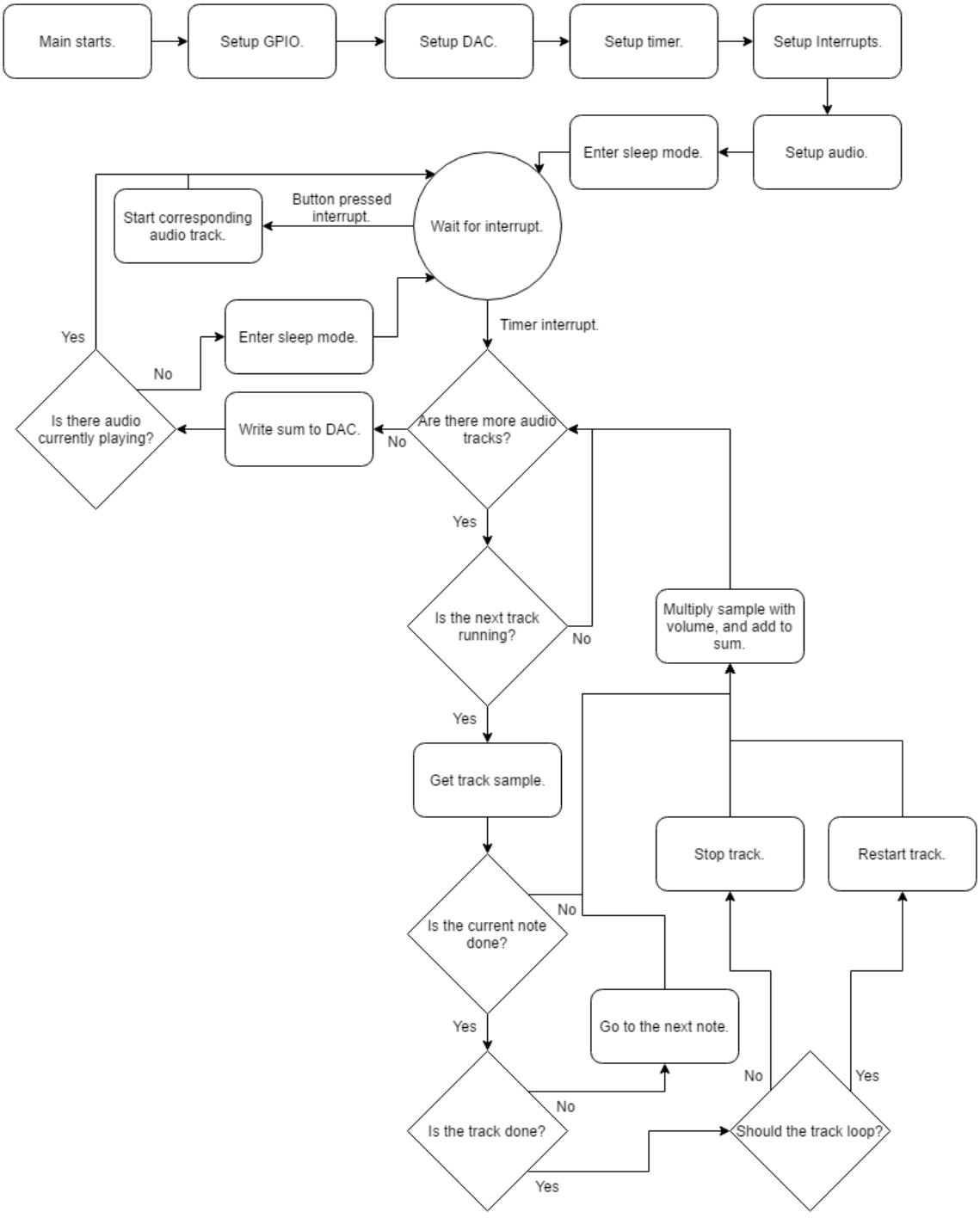

Figure 1.2: Flow chart of the Interrupt based solution.

2 Energy Measurements

Our solution with interrupts employs Energy Mode 2 when no sound is playing. This is far more efficient than using polling, since power consumption is over an order of magnitude lower while asleep. When no sound is playing, the system enters deep sleep mode, and is awakened when a button is pressed. For the duration of the song or effect the consumption is, as expected, higher. It is even higher than the polling version, which can possibly be explained by overhead with the interrupt mechanisms.

There is several improvements we could have done with our solution. We did not utilize the DMA. This is because our solution generates the samples real-time. If we were to use the DMA, we would have to generate samples for a given period, store them in memory and then push to DAC while the CPU were in some sleep mode. This would have made the solution much more complicated as we would have to generate the sound upfront and have the CPU handle interrupts in a much more complicated way.

As can be see in Figure 2.1, power consumption increases when playing sounds. This is clearly visible on the figure between sample 10 and 55. The spikes on the end of the plot corresponds to the sound effects, which is much shorter.

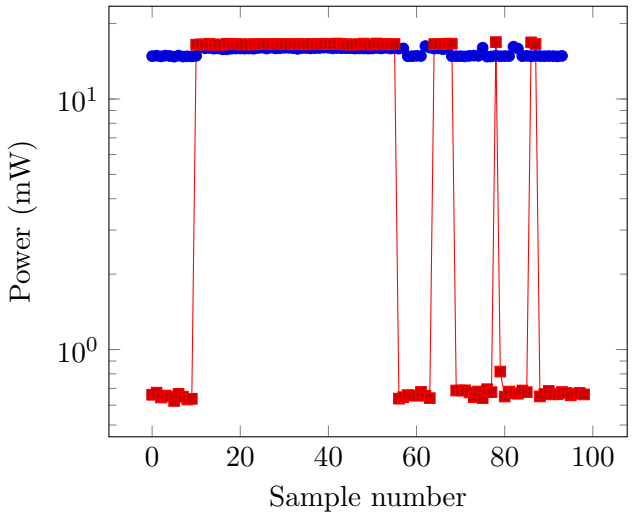

Figure 2.1: Log plot of energy consumption, not including the lights. Interrupt solution in red, and polling solution in blue. 100 ms per sample.