Lab 2 Report | Hao Li | cs425 | 3-17-21

The code is at the Github repository that I shared to you last time.
https://github.com/hl723/cs425

Part 1 LED and Button

(1) Files

include/startup_stm32f407vgtx.s

- This file must be executed each time a reset event occurs. Once the setup is complete from this code, is when our source code will run. The setup includes setting the stack pointer, copy data from flash memory to SRAM. The second part of this file also defines the various interrupt handlers and sets them to the default handler.

include/stm32f407xx.h

- This file defines the necessary data structures and macro definitions for all the address mapping of all the peripherals. It defines the registers and bits to access and modify the peripherals.

flash/STM32F407VGTX_FLASH.ld

- This linker script sets the heap and stack size along with the stack location. This file is used by the startup file above to initialize data from the data segment.

(2) Make and make burn

- make: When we run make, we compile our code with the arm gcc compiler and we also compile the include/system_stm32f4xx.c file that does the reset of the necessary RCC registers. It then builds the above linker file which ultimately generates the executable linkable format (elf) file and the .bin binary that will be ready to write to the chip.
- make burn: we use st-flash to write the binary generated above directly into the SRAM of the chip at the address 0x8000000.

For part 1, it was really simple to get the basic functionality to work. It just required referencing the GPIO and timer chapters of the manual alongside with blinky.c and timer.c. To implement the functionality of the user button, I had to understand what button.c was doing and imitate that in my code. Although I figured out a better way to do the pausing/resuming in the later parts as I understood more of what was happening, my solution that I have in part 1 was basically setting a global “stopped” variable to let my code in the timer interrupt to not proceed to the next light and do nothing.

Part 2 LED Display and Touch

Having part 1 working, the difficulty here was setting up PC7. With the listed instructions, I tried to follow them while imitating what I did for the user button that was PA0. It took some time to understand how the interrupt lines worked and to realize that EXTI7 did not have its own interrupt handler but rather it was grouped with the lines 5-9. The rest of the configuration was just simply following the instructions and finding the right registers to modify from the datasheet. I was sort of glad that this part had the most instructions on what we are supposed to do, it really helps when I try to figure out how to even start and what approach to take.

Part 3 Speaker and Touch

The rule of representation was used when storing the sample outputs and generated prescalers to match the desired frequencies of the 8 tones. Instead of computing all of those numbers on the fly in the interrupt handler, I created two global static arrays that carried the 256 samples that I generated to create a sine wave. The other array was an array of prescalers that stored the prescaler needed to set timer 2 to get the frequency of the note specified by the same index of the notes array (which is just an array of the 8 tone frequencies). By making such arrays, I was able to easily loop through the notes and samples in the two timer interrupts making the code much cleaner to read while having the logic be in the data structure.

This part took about 4 full days for me to complete. The first 3 was spent in massive confusion on how to even make the first sound in the speaker without using the given beep function. After quite some time I realized that the workflow was basically:

Our digital value => DAC => Audio DAC => headphone speakers

I was really confused on how each of the arrows connected themselves together. After being puzzling for a long time, it turns out that we can write values to the DAC according to the manual, analog passthrough means that the two DACs were connected (in some sense), and that the code from i2s-beep already utilizes i2s to send the output to the headphone speakers. Once I got the output of the first sound, I got the 8 tones very quickly. The button carried over as well. The touch PC7 pin was somewhat tricky. However, after a series of discussions on Slack, that was also resolved after a few hours.

Part 4

The functionality of this part was pretty straightforward to carry out given that part 3 was working. The tempo could be changed by changing the ARR value (or by doing some math changing the prescaler) for the timer that changes the tone from one to another (that was an observation found when trying to compute the prescaler from the previous part). The single click user button was not a problem, the double-click was what was interesting. The solution that I used was disabling the first interrupt, and having a very short stalling period while continuously polling for another trigger interrupt from the button. If so, we detected a double click, if not, we got a single click. The pause/resume was done by turning off the clocks for producing the tones and writing audio. To switch the songs, it was just simply setting curr_song to the other song.

For this part, the rule of representation was used to represent the two different melodies. I have two different arrays that store the sequence of notes (as macros of indices to my prescaler array). I also keep track of the two speeds of each melody and the tone that each melody last played. By representing the data like such, I was able to quickly know what tone and what speed to change in the timer interrupts very easily by checking which melody it is currently playing. Alternatively, I could have made a struct for each melody which would better enhance the rule of representation instead of just having two integer arrays of macros to denote the sequence of notes for each melody.