Sam Detor

Lab 2

Part 1

I first updated the start-up script, linker script, and header file to the correct files for the STM32F411VE MCU—startup_stm32f411xe.s, stm32f411xe.h, STM32F411VETx_FLASH.ld respectively. I also updated the per-project makefiles and the armf4 makefile to reference the relevant updated files. After that, I used the generic header file in my .c files and uncommented the relevant section in the generic header file to include stm32f411xe.h. One of the struggles I had during this phase of Part 1 was getting the correct .s file for my compilation configuration. I was accidentally trying to use one for a custom arm configuration at first, before seeking help in office hours and then selecting the correct .s file for my gcc set up.

After doing this, I was able to make and run the "timer" example by commenting out "set_sysclk_to_168" , but the timing of the led blinks did not match what it said it should be in the code. This was because my timer was assuming that the system clock was running at 168 Mhz, when it was not. I then updated the system_stm32f4xx.h to define new PLL_M, PLL_Q, PLL_P, and PLL_N based on the recommendations found in this source to set the clock to 100 Mhz, which the board can handle. I then changed the name of the "set_sysclk_to_168" to "set_sysclk_to_100" to more accurately reflect its functioning.

After these changes were made I combined the code from the timer and external projects from the sample repo given. I enabled both the rising and falling edge trigger on the button so that the interrupt handler would be called both when it was pressed and being let go. In the actual interrupt handler, I checked if the interrupt was coming from the correct line, cleared the interrupt and waited for another trigger. This allowed the falling edge trigger to initially trigger the interrupt, and then it would hang and wait for the rising edge trigger, which paused the led blinking given the high priority level of the interrupt handler. Once the falling edge trigger was detected, the handler reset the led sequence to start at green again, and returned.

The functions of the files mentioned in the lab handout are detailed below.

startup_stm32f411xe.s:

Since C files assume that there is a call stack when they run, some code not written in C has to set up the stack before the C code can be executed. At a high level, this code, which is

written in assembly, sets up the call stack and other basics so that a C program will run. This program first defines the reset handler, which sets the stack pointer to _estack. __estack is defined in the linker script as the highest address in RAM. Then, using the section headers and footers defined in the linker script, it copies the data from flash to RAM, and fills the bss section with zeros. Then it initializes the clock, calls static constructors and then calls main. This script also sets the vector table entries for the interrupt handlers and creates aliases for these handlers that can be overwritten by user code. Also, the script sets the program counter to the reset handler, so it is the first instruction run when the program starts.

It is very important that this script and the linker script used the same variable names, as this script used the data segment addresses (like the start and end of the bss section, for example) in the startup code. This was one of the problems I ran into when updating the files.

stm32f411xe.h

This file defines nice macros that the C program can use to access and set peripheral registers without having to directly use addresses. First, this file defines structs that can be used to access registers nicely. Then all the base pointers for the major peripherals are defined. Finally, the file casts the base addresses to struct pointers and defines them as with names that correspond to the reference manual.

STM32F411VETx_FLASH.ld

The linker first sets the program entry point to be the reset handler. This means that the first thing that will be executed in the code is the reset handler. Then, it sets the location of the stack pointer, the minimum heap size, and the minimum stack size. It then also sets parameters like flash and RAM sizes, their start values, and their permissions. This is a very important section of the linker script because it specifies the memory layout that the board has. Then the script dictates the layout of the ELF file to be generated. The startup code is put first, then the standard text section and rodata sections. After that, there is a "ARM extab" section defined, which according to this stackoverflow post, contains unwiding information. Unwinding, per this source, is how to get rid of local variables and call destructors on a stack when a function call ends. Apparently this information is mostly useful for dealing with exceptions being thrown. After this section, "pre init array", "init array", and "fini array" are mentioned. From the stackoverflow post mentioned earlier, these sections are referring to the constructor and destructor C methods that are called before and after main(). After that, the standard data and bss

sections are described, and global symbols are created at the beginning and end of these sections for use in the startup script. Towards the end of the file, there is some memory reserved for the user heap/stack. The last bit deletes any sections that were not specified in the above script (source).

Makefile

At a high level, the initial per-project makefile defines some variables used in the overall makefile (armf4) and then calls the overall makefile. Some of these variables that were important to change were the TARGET and SRC variables, which are responsible for the name of the files generated and the files that need to be compiled, respectively. The linker script was also assigned to a variable as well as the processor type, for further use in the armf4 makefile.

When "make" is called, it calls three sub commands, clean, build, and size. Clean is pretty self explanatory, as it removes all files/folders generated from a previous call to make. Build is more complicated, as it generates a .elf, .bin, and .lst file with the name specified by the TARGET variable, in a sub folder named by the variable OBJDIR. This command first generates the .elf file. First, all the relevant .c and .s files are compiled into .o files. All the .o file names with their corresponding file paths in OBJDIR are stored in the variable OBJS. So, the makefile compiles all the corresponding .c and .s files for the .o filenames in OBJS with the specified CFLAGS and included libraries necessary (in the case of the .c files). Then these .o files are linked together to create the .elf file based on the linker flags, one of which is the linker script mentioned earlier. Once this .elf file is created in the OBJDIR directory, then the .bin and .lst files can be created by calling "arm-none-eabi-objcopy" and "arm-none-eabi-objdump" on the .elf file created earlier with the correct flags. The "size" command then reads the elf file and prints the sizes of all the text, data, bss, sections as well as other information to the screen after you are done generating all the necessary files. "make burn" then writes the .bin file to address 0x8000000 using the st-flash command with the "write" option from the stlink-tools library. 0x8000000 looks like the standard address to write .bin files to on an STM32 board.

Source code for part1

I got my startup_stm32f411xe.s, STM32F411VETx_FLASH.ld, and stm32f411xe.h from the STM23CubeF4 repository of the STMicroelectronics github. I did not change any of these files.

From the sample repo given, I used the contents of the libs folder (the CMSIS_5 repo), and the stm32f4xx.h file without making any major changes to them.

The makefile in the Part1 folder was taken from the "blinky" project from the sample repo. I changed the name of the TARGET and SRC variables to be Part1 and Part1.c respectively. I changed the processor CDEF to -DSTM32F411xE from -DSTM32F407xx and changed the address of the linker script to point to the updated linker script: STM32F411VETx_FLASH.ld.

I also used the overall makefile (armf4). I changed the path to the startup script and to point to the updated script mentioned above. I also changed the paths to the CMSIS library and the system_stm32f4xx.c to reflect their new location in my repository.

I also used the system_stm32f4xx.c from the sample repo. I changed the name of the "set_sysclk_to_168" function to "set_sysclk_to_100" to more accurately reflect its functioning.

I also used the system_stm32f4xx.h file, but changed the PLL_M, PLL_Q, PLL_P, and PLL_N values to better fit with the STM32F411VE MCU. I also changed the name of the function declaration of the old "set_sysclk_to_168" function to "set_sysclk_to_100".

My Part1.c was a combination of the "external.c" and the "timer.c" files from the external and timer projects in the example repo.

I combined the main functions from external.c and timer.c to create my main function. I then deleted any repeated parts (like setting up the LEDs) and called set_sysclk_to_100 instead of set_sysclk_to_168. Since the board I was using had a different clock speed, I updated the prescaler (TIM2->PSC) from 8399 to 4999 based on the formula listed in the comment on lines 69-75 in timer.c. I also changed the auto reload time from 10000 to 5000 to trigger the switching of the LEDs every 0.5s instead of every 1s.

I based my EXTI0_IRQHandler on the one found in external.c. I used the function definition, if statement used to check if the interrupt was coming from EXTI0, and line 50, which was used to clear the interrupt status. I deleted everything else in that function and replaced it with my solution, which was described above.

I used the TIM2_IRQHandler from timer.c in my code. The only changes I made were declaring "i"—the variable used in timer.c's TIM2_IRQHandler—as a global variable and changing the variable name from i to ledVal.

Part 2

At first, I tried the first method for implementing a touch sensor recommended in the lab handout. However, I struggled to measure the time that the capacitor took to discharge reliably. I had a difficult time effectively prioritizing my interrupt handlers. The "time keeping" handler kept getting interrupted by other handlers, but couldn't be prioritized or it would have affected the LED blink pattern. Therefore, I switched to the second method, which proved to be easier to implement.

To implement the touch sensor, I first configured PC7 to be a "floating" pin and then periodically refreshed the charge on the pin. When my finger touches the pin, the pin's charge drops, causing a falling edge trigger, which triggers the configured interrupt handler to either pause or restart the sequence from the green LED.

In order to do this, I first enabled the GPIOC clock and configured pin 7 on GPIOC as an input using this video, looking at the LED code from part 1, and reading the reference manual. Then I configured the SYSCFG->EXTICR2 register bits 12-15 to map EXTI7 to PC7. I had one issue with this initially. At first, I was accidentally trying to access the second EXTICR register using SYSCFG->EXTICR[2] instead of SYSCFG->EXTICR[1]. Then, I had to enable the falling edge trigger on EXTI7, and enable interrupts on line 7 as well. I also had to set the priority and enable the EXTI9_5_IRQHandler. The issue I had with this was I initially thought since I was mapping PC7 to an EXTI handler using the second EXTICR register, that I was mapping PC7 to the EXTI2 handler. Once I figured out that was wrong, things got a lot easier.

In my EXTI9_5_IRQHandler, I toggled a global variable "blinking", which dictated if the LED blinking was paused or not. If blinking was being set to 1 (not paused), then I set the global variable ledVal to 1, which restarted the blinking sequence at green.

Then I created my second timer interrupt (TIM3), which refreshed the charge on PC7 approximately every 0.3 seconds. I followed the method I used to set up TIM2 in part from (from timer.c) and enabled the TIM3 clock, set the prescaler to 4999, set the auto refill value to be 3000 (0.3 seconds), enabled interrupts from TIM3, and enabled the TIM3 module. Then I gave my TIM3 handler 3rd priority, and enabled it. This went pretty smoothly as all I had to do was copy the method used to set up TIM2.

In my TIM3 handler, I used the method from TIM2 to clear the interrupt and then added code to charge the pin.

Lastly, I slightly edited my TIM2 code from Part1, and added a line to prevent changing the LED values unless the "blinking" global variable was set to 1.

After this, I also had an issue where one touch would trigger the interrupt handler multiple times. In office hours, Zhiyao helped me with this by suggesting an decrease in the rate I was refreshing the charge on the PC7. This helped to reduce some of the instances of this error.

Source code for part 2

I used my Part1 as source code for Part2. See Source code for part 1, for details on the source code for Part1.

I used all the same files mentioned in Source code for part 1 in part 2 unchanged except the per-project makefile and Part1.c. I changed the TARGET and SRC variable in the per-project makefile (the makefile in the Part2 folder) to Part2 and Part2.c respectively.

In regards to the changes I made to Part1.c to make it Part2.c, I added and enabled the TIM3 handler, configured the PC7 pin instead of the user button, replaced the EXTI0 handler with the EXTI9_5 handler, connected the EXTI9_5 handler to PC7 instead of connecting the EXTI0 handler to the user button, and configured EXTI7 instead of EXTI0. I also changed the TIM2 handler to depend on a new global variable "blinking". My interrupt handler now also toggled "blinking" to pause the LED blinking instead of hanging and waiting for a second trigger inside the handler like my EXTI0 handler did in part 1.

Part 3

This part was greatly simplified once I found the i2s-beep sample code mentioned on Ed. I used the example code's methods for setting up the i2s and the cs43l22 audio chip, as well as their i2c write method to communicate with specific registers on the cs43l22. Then, I looked up the registers used in lines 362-371 of the example code on the audio chip datasheet to figure out how these lines played the notes in the "nem" array. Once I did that, I found the table of frequencies on page 48, which caused me to start my scale on C5 instead of C4, since the beep generator had more support for that scale. After switching the notes in "nem" for the notes of the C5 scale, I needed to fix the volume that the sounds were playing at. With the settings in the example code, the notes were playing too loud through my headphones. Using the

"CS43L22_REG_BEEP_VOL_OFF_TIME" register, I set the beeps to the minimum volume. I also lowered the headphone volume using the "CS43L22_REG_HEADPHONE_A_VOL" register. Once I fixed the volume, I used the TIM2 interrupt to play my scale on repeat by disabling the beep, changing the frequency of the beep, and then re-enabling it each time the interrupt was called.

After I was able to successfully play the scale starting at C5 on a loop at a reasonable volume, I moved on to crafting the state machine to differentiate between a single click or double click. I started by configuring the EXTI0 handler to be connected to PA0, as was done in Part1, and have a rising edge trigger. My first attempt at this was similar to how the touch sensor worked in PC7. I had a global variable "click" which I set to 0 every 0.3 seconds, and every time EXTI0 was triggered, it incremented the click variable. Then, if "click" was 0, i.e. it was the first click that had been registered by the handler, the scale was paused. If "click" was 1, the scale was restarted. This had the unfortunate side effect of causing the scale to pause for a second before the next click of a double click triggered an interrupt. Therefore, after visiting office hours, I decided to create a state machine and use timers to differentiate between a single click and a double click. Now, when the first click triggers an interrupt, I start 2 timers, one for debouncing and one to differentiate between a single and double click. The debouncing timer makes sure that the state machine does not take into account clicks that happen less than 10 ms after the first one. When the second timer triggers (after 0.3 seconds), it checks the state of the program. If the program state is FIRST_PRESS_DEBOUNCE, i.e. a second rising edge trigger has not been detected, then it assumes that it was a single click and pauses the scale. If a second click triggers the interrupt before 0.3s have elapsed, the state of the program will be changed back to its initial state (NO_PRESS) and the scale will restart.

Rule of Representation:

There were two major places in my code where I utilized the rule of representation: the state machine and the "scale" array.

In the state machine, I chose to create an enum that detailed all possible program states and use one "myState" variable to store the current program state. I did this instead of having separate variables that stored different pieces of information. For example, I could have had one variable to determine how many times the button interrupt had been triggered and another to

determine the state of the debouncing timer. This could have allowed me to have fewer states, but would have made the procedural logic that switches between states more complicated. Right now, my "if" statements changing the state of the system consist of " if (myState == State_x), then myState = State_y". This allows for readable code and easier debugging than if each state depended on many different variables.

For the scale array, I choose to use #define to store each note using its letter and number code instead of its actual value (ie C6 instead of 0x80). This way, when I was creating the scale array, it looks more readable and is more easily interpreted. Additionally, since each note in the scale array differs from the previous one by 0x10, I could have simply incremented a variable by 0x10 to get the next note and just reset the value back to 0x10 (the value of C5) once the variable had gotten to the last note on the scale. This however, would have been harder to follow and less simple than traversing an array.

Source code for part 3

I used the files in the "include", "lib", and "flash" folders as well as the armf4 makefile during part 3. See Source code for part 1 for details on the files in these three folders and the armf4 makefile.

During this part, I did add one file to the "include" folder, cs43l22.h, which mainly assigned names to the different registers on the cs43l22 audio chip. I got this from the example repo's "include" folder and did not change it.

The makefile in my Part3 folder was copied from the one in my Part2 folder, and is identical except I changed the TARGET and SRC variables to Part3 and Part3.c respectively. Please see Source code for part 2 for more details on the source of the makefile.

I used i2s-beep.c from the example repo given to create my Part3.c. I used the init_i2s_pll, init_i2s3, init_cs43l22, start_cs43l22, I2C1_ER_IRQHandler, i2c_start, i2c_stop, i2c_read, and i2c_write methods created in i2s-beep.c in my Part3.c without making any changes. I also used lines 281-356 of the i2s-beep.c main function in my Part3.c main function with no changes. I used lines 359 and 360 of the i2s-beep.c main function with small changes. I set my volume to 7, while the i2s-beep.c main function volume was set to 16. I did not use lines 363-371 in my main function, but they did help me create the contents of my TIM2_IRQHandler.

To the source code I got from i2s-beep.c, I added code to initialize the user button, tie PA0 to EXTI0, and initialize timer 2, 3, and 4. I used source code from part 1 to set up the user button and tie PA0 to EXTI0. Specifically, I used lines 82-84 (PA0 configuration), 114-127 (EXTI0 set up). I also used source code from part 2 to help me set up the timers (lines 128-130). I modified the timer code from part 2 based on the different settings I wanted my timer to have. For example, I changed the timer 3 mode from recurring to one pulse, and the timer refill value from 0.5s to 0.3 seconds. I also added code for the EXTI0, TIM2, TIM3, and TIM4 interrupt handlers, which used the same method for clearing the interrupts as was used in Part 2, but perform different functions. Lastly, I added #defines for each of the notes I was using, the array of notes the TIM2 handler steps through, an enum "processState" that lists all potential states for the system, and various global variables used to pause and play the scale.

Part 4

To start, I had to decide on which songs I wanted to play. "Twinkle twinkle little star" and "Mary had a little lamb" seemed like the best options as they could be played with the 8 notes used in part 3, and most of the notes are played at the same tempo. I stored the songs as arrays of the notes that I defined and used in part 4. I also defined "Rest" to be zero and added "Rest" as an array element when pauses were needed in the song. The first problem I ran into when playing these songs is that they both have 2 of the same notes in a row. If I were to use the same method of playing notes that I did in part 3, two of the same notes played back to back would sound like one really long note. In order to fix this, I configured the beeps to be in "single" mode instead of continuous mode like I did previously. This meant that the beep generator would play a single beep for the "on time" specified in the lower 4 bits of the CS43L22_REG_BEEP_FREQ_ON_TIME register as opposed to generating a beep at the specified frequency until the beep generator was disabled. At first, I had a difficult time getting the beep to play because I was not disabling the beep before changing the frequency and on time settings. After reading the data sheet more closely, I realized that this was not allowed. Once I disabled the beep before I changed the frequency and ontime settings and then re-enabled the beep with the correct settings, the song played as expected.

Since both twinkle twinkle little star and mary had a little lamb both have some notes that are longer than others, I created a new "tempo" array which was as big as its corresponding song

array to keep track of whether the relative on-time of that note was short or long. Then when the song was played, the value in the tempo array that corresponded to a given note was used to index one of the three tempo speed arrays—tempo1x, tempo2x, and tempo4x—based on the current playback speed to determine what binary on-time code to return for that note. Then, the value sent to the audio chip to configure each note before it was played was the note code and the on-time code "or'ed" together, as these both are set in the same register.

Due to the large amount of timers that I anticipated being used in the state machine, and the need to change the speed of the song being played, I choose to play the song notes in the while loop of the main method and use systick to create a consistent time delay between the notes. I felt that it would be easier to change the input to the "delay" function than to change the value of the timer interrupt value on the fly.

The next problem I needed to tackle was differentiating between the single, double, and long clicks. This was a large increase in difficulty from the state machine necessary in part 3. This is because in order to differentiate between a long press and a single click, the EXTI0 handler had to have both falling and rising edge triggers enabled so that you could tell when the button was "unpressed" as well as pressed. This meant that there were many more states that had to be dealt with. Also, debouncing became much more important because now there were double the opportunities for the button to register the wrong amount of clicks, and many more ways to mess up the states as there were more potential states to begin with. So, I spent a lot of time building my state machine using just LEDs.

One of the issues I had initially was that I was not disabling my long press timer once the first "unpress" was detected. Since the long press timer takes a little while to finish, sometimes if you tried to do two single clicks quite close together, the long timer triggered by the first click would finish just as the second first click triggered the EXTI0 interrupt handler. This caused the program to think that the two single clicks were actually a long press.

Most of my other issues came down to debouncing. Without debouncing, the button registered way too many clicks and this would wreak havoc on the program state. With too much debouncing, the program would fail to pick up the "unpress" of the first click and could only register double clicks and long clicks. Most of the work came down to optimizing the right times and the right length of time to not let the program state change after the interrupt handler was triggered. I ended up with a state machine that didn't count clicks that came in until 12.5 ms after

a click changed a program state. The only exception to this is when the program state changes to "LONG_PRESS" after a long button press has been registered. There is no debouncing in between that state change and the "LONG_PRESS_DONE" state change which is triggered by the EXTI0 handler. I found that even putting a very small amount of debouncing in between that state change would cause the interrupt to not "count" the falling edge trigger that was triggered by "un-pressing" the button after a long press.

After getting the state machine to work with LEDs, I decided to keep the LEDs in the code as indicator lights of the program state. Toggling the green, red, and blue LEDs mean that a single, double, or long click has been registered, respectively.

Next, I implemented the various changes—pausing/playing the song, switching the song, and changing the speed of the song—in the correct location in the state machine.

For ease of switching between songs, I created a struct "songInfo" which was a datatype to store all the relevant information about a song. I created two of these structs as global variables, one for each song, and one struct songInfo pointer called "currentInfo" which pointed to the songInfo struct that was currently being used. This made switching between songs and accessing the right data when playing the song simple.

For pausing and playing the song, I used a global variable "playing" and toggled it when a long press was registered. In the while loop that played the song, notes were only sent to the cs43l22 chip when playing equaled 1. I choose to make the pause or play global, meaning switching to a different song did not affect the paused/playing state of the program.

For switching between speeds, I updated the speed and the speed_tempo_vals variables using "currentInfo". In order to get the right delay between notes based on the speed the song was playing at, I use the speed variable to index the delayVals array. Notes are played 1s, 0.5s, and 0.25s apart when the song is played at 1x, 2x, and 4x speed respectively. In order for the songs to sound more natural, the on-time of the short and long notes needed to vary with the speed as well. So, the speed_tempo_vals array is used to get the right binary code for the on-time of each note given the current speed and relative tempo of that note. The speed_tempo_vals array is indexed with the current note's tempo value, which is obtained from the tempo array in the currentInfo variable. There is one caveat to varying note on-times based on speed. When the songs are played at 1x or 2x speed, there are enough beep on-time options to vary the on-time of the short and long notes. At 4x speed, however, there is only one beep on-time option that is

under 250ms (the 4x speed delay between notes), so all the notes are played for the same amount of time (~86 ms). However, since the notes are being played quickly, the lack of differentiation between the short and long notes is not noticeable.

Rule of Representation:

There are many more examples of the use of the rule of representation in part 4 than part 3 because the task was more complex. Some of the examples covered in the part 3 "Rule of Representation" still apply here, however, since they were explored in the part 3 section, in this section I will only focus on new uses of the rule of representation that were not applicable in part 3.

One of the first examples of the application of the rule of representation is how I store the relevant information about each song. I created a songInfo struct, with two instances, one for Mary Had a Little Lamb and one for Twinkle Twinkle Little Star. I also have a struct pointer "currentInfo" that points to the currently playing song. This makes switching between songs very programmatically easy, as all you have to do is change what struct "currentInfo" is pointing to. This removes the issues that come with having multiple global variables to describe each song. For example, you could forget to switch one of the global variables to the new song value or forget to save information about the current song when switching songs. Additionally, the code is much more readable and easily updatable if I ever wanted to add more parameters to describe each song, or add the capability to switch between more than two songs.

Another example of an application of the rule of representation in my part 4 code is that I used two arrays to describe each song: the tempo array and the song array. The song array contains the frequency values and the tempo array contains the relative duration of the note. Since the on-time and the frequency of the beep are set in the same register (the on-time is the lower 4 bits and the frequency is the upper 4 bits), I could have combined these arrays. However, I choose not to do that because it would be more difficult to change the on-time of the notes based on the speed the song was being played at. I would have needed to access and change the lower 4 bits of the given array value with difficult logic to do this. Therefore, it was much simpler to follow the rule of representation and separate the two values.

I also applied the rule of representation when choosing how to adjust the delay and on-time of the beep based on the speed that the song is being played at. I could have created

methods with state machines that provide this info based on the speed variable from the currentInfo struct pointer and the tempo value of the current note. However, I choose to store the different delay values and potential tempo values in arrays and index them based on the speed and the current note's tempo value instead. As the rule of representation states, this was more readable, less prone to error, and easily updatable. This was especially important when I was still figuring out the correct on-times and relative tempos that each note should have to sound natural, and was changing these values frequently.

Source code for part 4

My largest source of previously written code for part 4 was my part 3 code. Please see Source code for part 3 for more information on the sources I used to write my part 3 code.

One of the biggest changes I made to my part 3 code was moving the note playing from the TIM2 interrupt to the while loop using systick. In order to implement this, I used code from systick.c, which is in the systick folder of the example repository provided. I used the SysTick_Handler, init_systick, and delay_ms functions written in systik.c. I called init_systick in my main method. I also called the delay_ms function at the end of the while loop in my main function to create a consistent delay between notes.

Changes I made to the part 3 source code included adding global variables to describe each song and the relevant information required to play them at different speeds. I also added new states to my processState enum to help support the differentiation between the long, short, and double clicks. Additionally, I updated the initialization/configuration of my timers and EXTI0 to reflect their new role in the part 4 code. My TIM2, TIM3, TIM4, and EXTI0 handlers and the overall state machine system were also updated to support the long click. The function of each of the click types (short, long, and double) was also updated to support the new functions needed in part 4 (song and speed switching). I also, as previously mentioned, moved the "playing" of the song to the while loop in my main method, and called "delay_ms" to ensure a consistent delay between notes.