Drumroll-Bot

ECE 3140 Final Project Report
Claire Chen (clc288) and Yuqi (Mark) Zhao (yz424)

1) Introduction

Frequently, while playing or composing music, a musician desires rhythmic accompaniment that more closely resembles a drum beat than the simple ticking of a metronome. The ability for a musician to instantly add a consistent and customizable beat can easily allow him or her to hear what the music will sound like in a larger band. Furthermore, a physical instrument provides a realistic timbre as compared to an electronically-produced sound. Thus, we present the *Drumroll-Bot*, a robot that functions as an easily-customizable metronome by employing two servos to beat a pair of drums corresponding to an rhythm input by the user.

The Drumroll-Bot has servo motors, each acting like an 'arm' that controls one drum stick each (e.g. snare drum and cymbals). On startup/reset, the user enters a sequence of beats using two buttons on a four-button keypad, which will move the two servos and generate the desired rhythm, allowing the user to hear his or her rhythm input. The Drumroll-Bot will remember this sequence of beats which will be played back once the user pushes the stop button. It then functions as a metronome, continuously playing back the input on loop until the user resets the robot. This allows the user to fully customize not only the BPM of the metronome, but to customize the drum rhythm to fit the genre of music, e.g. jazz, rock, latin, etc., as well as the time signature of the beats. Compared to other commercially-available systems, which can cost upwards of $1000 (https://www.youtube.com/watch?v=5rwhbBKfXnY), our Drumroll-bot provides a cheap means for a musician to significantly broaden his or her artistic range.

2) System Diagram

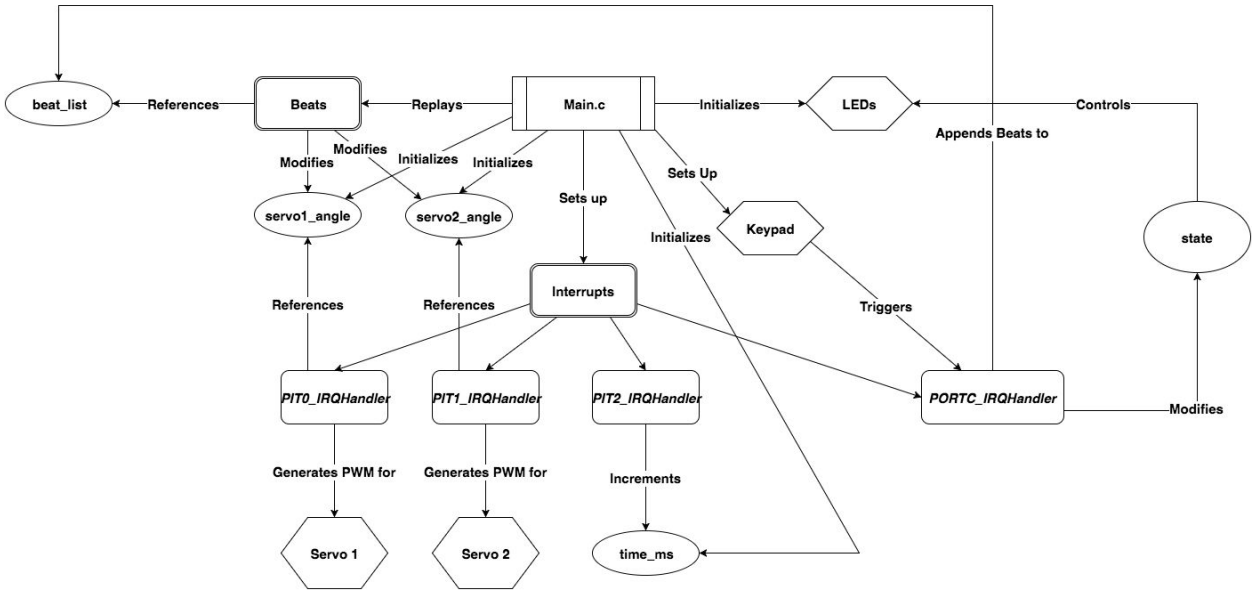

Figure 01: System Diagram

2.1 Overview

In order to improve the modularity and organization of our code, we created separate .c/.h files for each of the major components of our project (i.e. Servos, Beats, Buttons, and LEDs). This section provides a brief overview of each component and its interactions, with more detail being provided in Section 4. As shown in the figure above, our main function is located in *main.c*. The library files for each of the four components are included in *main.c*, which utilizes functions from each respective header file, as shown in the system diagram.

2.2 Architectural Choices and Interrupts

We approached the design of our project in 3 phases: Servo, Buttons, and Beats, testing the functionality of component before moving on to the next component, and assembling the components together in *main.c* at the end. First, we started by implementing our own servo controller. Since the hardware does not provide a dedicated servo controller, we decided to implement our own servo controller in software, as we will describe later. However, to do so, we used a PIT timer for each servo, and writing HIGH or LOW to a GPIO pin after each interrupt. Since we are effectively bit banging the waveform for servo control through the PIT interrupt handlers, the handlers determine the pulse duration, and thus position, based on two global variables *servo1_angle* and *servo2_angle*. Thus, we created functions that changed the position of each servo based on input parameters by modifying these parameters.

After we determined that the servo control worked properly by verifying the waveforms on the oscilloscope and measuring the pulse widths, we moved on and implemented the logic for our button presses. The buttons we received are effectively switches. Thus, we configure a GPIO pin corresponding to each button to be pulled down to a logic LOW. When the switch is pressed, the pin is connected to +5V, and we trigger an interrupt on the rising edge of the waveform. We handle all of the logic of prompting for user input, recording user input, and managing the beat linked list in the IRQ handler. In the figure below, we show a state machine of how we prompt for user input and then replay the beats.

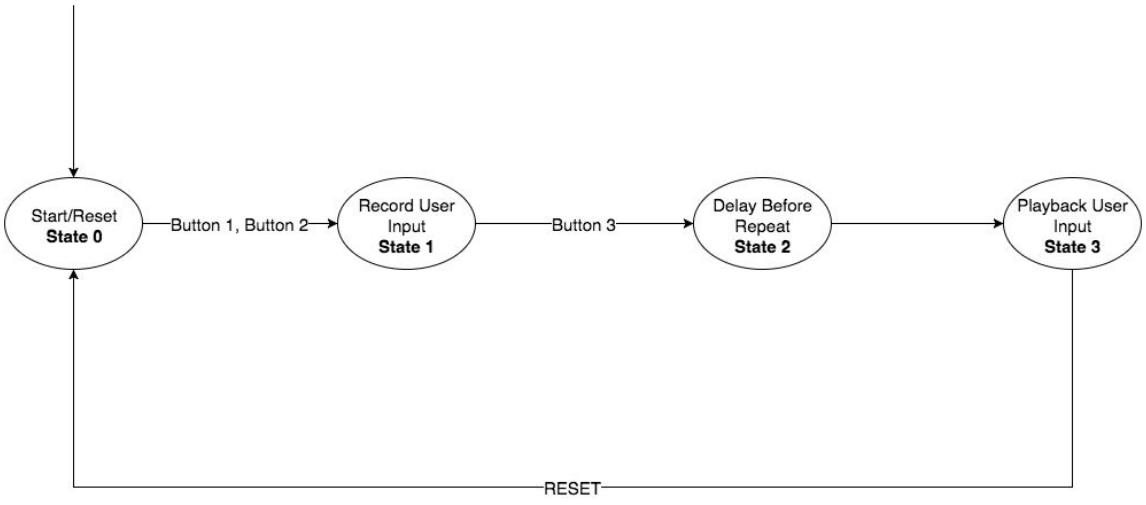

Figure 02: Button Press State Machine

Once we verified that the state machine registered button presses in the PORTC_IRQHandler, we implemented the last component of our project: beats. We first decided that in order to save an effectively limitless number of beats (depending on the heap size), we would create a linked list to store our "beats". Secondly, since the beats needed to be replayed back at a specific time, we decided to have a global variable, *time_ms*, that stores the time elapsed, in milliseconds, since the beginning of each period. To increment the timer, we used another PIT (PIT2), which triggers an interrupt every millisecond and increments the value of *time_ms* by 1 in the IRQHandler.

We reference the head of our linked list via the global variable *beat_list.* For each node of our list, we create a struct *beat_t*, as we will describe in Section 4, that stores the servo number, time of the beat, and a pointer to the next element in the queue.

2.3 System Interaction

After completing and testing our three separate components, we integrated them in our main function. We begin by initializing the appropriate power and clocks to each of the PIT timers, as well as setting up each GPIO pin for button presses as well as the servo signals. As described earlier, most of our logic is handled in the interrupt handlers. Thus, we simply wait until the state is ready for user playback (state = 3). In the meantime, user inputs (i.e. button presses) are handled by the PORTC_IRQHandler. Every time a servo button is pressed, a new *beat_t* is malloced and initialized with the servo to be hit and the time that it was pressed, and then the *beat_t* is placed into the aforementioned linked list. Additionally in the PORTC_IRQHandler, every time a button is pressed, we also call the appropriate servo functions to hit the servo. These functions modify the global variables *servo1_angle* and *servo2_angle*. Accordingly, in the PIT interrupt handlers for each respective servo, the timer values loaded for the pulse widths, and thus servo position, correspond to the values stored in *servo1_angle* and *servo2_angle.*

Once the user has finished inputting commands, the main function simply loops infinitely while traversing through the loop. At the beginning of every traversal, the value of *time_ms* is reset. We then start incrementing the timer. Since our *beat_list* is ordered as a FIFO queue, depending on which buttons were pressed first, we simply wait until the current time is greater than the hit time of the next beat in the queue. At that point, we again call the servo functions to hit the appropriate servo. As a result, we keep traversing through the queue and replaying the sequence of user-input beats.

3) Hardware Description

3.1 Bill of Materials:

Part	Quantity	Cost	Distributor	Link
MG995R High-Torque Servo	2	$19.95	Adafruit	https://www.adafruit.com/product/1142
4-Button Keypad	1	$2.95	Adafruit	https://www.adafruit.com/products/1332

3.2 Hardware Schematic

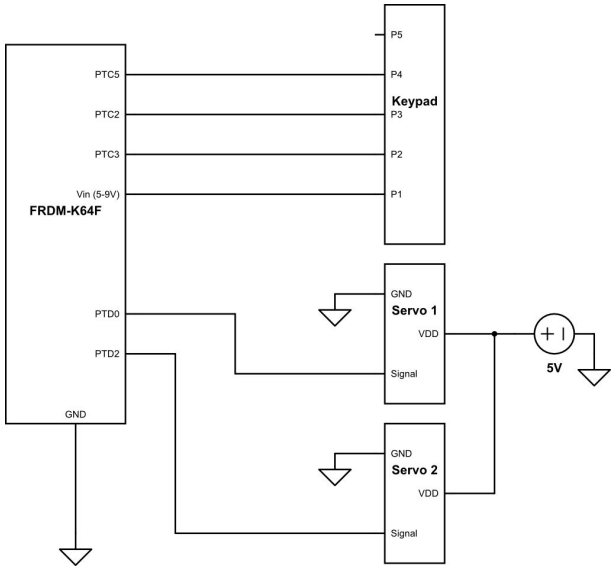

Figure 03: Hardware Schematic

**Note that to provide enough current to power the servos, we used an external 5V power supply from a laboratory-bench power supply, as shown in the schematic above.

4) Software Description

4.1 Beats

Beat Data Structure:

Each beat is stored as a beat_t data structure that contains the following fields:

int hit_time: integer that stores the time the hit occurred in a period

int servo_num: number representing which servo has been tapped

struct beat next*: pointer to the next beat_t structure in the beat list

Beat List:

The sequence of beats entered by the user is kept track of in a linked list of *beat_t* structures. A pointer to a *beat_t* structure maintains the head of the list. At every instance the user pushes buttons 1 or 2, a new *beat_t* structure is created by setting each field in the structure to the appropriate values. Then it is appended to the tail of the beat list. To signify when they have finished entering beats, the user presses button number 3. This final button press is also stored as a beat with servo number 3 and the time at which it occurred.

Beat Times:

A timer that increments every millisecond is used to keep track of when a beat occurs in time. This timer is implemented using periodic interrupt timer (PIT) 2 that triggers an interrupt every millisecond (20970 clock cycles on a 20.97 MHz clock). Within PIT2's IRQ handler, the global variable *time_ms*, signifying the elapsed time in milliseconds every period, is incremented by 1. A period is defined as the length of time from when user enters the first beat to when the user pushes button 3 to signify that they have finished entering beats.

Replaying Beats:

When state 3 is reached, we traverse the beat list to playback the user-entered beat sequence. At the beginning of every iteration of the traverse, the *time_ms* is reset to 0 and the elapsed time timer is reset to 0, signifying that a new period has begun. As we traverse through the beat list, as we arrive at each beat node, we wait until the time of that beat matches the current time in the period. When the times match, we tap the appropriate servo. The final beat, stored with servo number 3, allows for the final pause between the last beat and the first beat upon replaying the sequence. Once we reach the end of the list, the traverse repeats from the head of the list until the user resets the board.

4.2 Keypad

Button Pin Configuration: *Buttons.c > void configure_buttons(void)*

As shown in figure XX, buttons 1, 2, and 3 (4 is unused) on the keypad are each connected to a port C pin on the board. Each of these pins is configured as a GPIO pin by writing the pin mux control bits in their corresponding pin control registers (PCRs) to 001. Each pin is then configured for general-purpose input by writing 0s to appropriate bits in port C's port data direction register. To pull down each pin to a logic 0, the pull enable (PE) field in each pin's PCR is set to 1 to enable the internal pulldown or pullup resistor. Once the PE field has been set, the internal pulldown resistor is enabled by writing a 1 to each PCRs pull select bit. An interrupt is set to trigger on the rising edge of each pin by setting the interrupt configuration bits in the PCRs to 1001 and writing a 1 to each PCRs interrupt status flag bit. Finally, port C's interrupt request is enabled.

Button Interrupt Handler: *Main.c > void PORTC_IRQHandler(void)*

Port C's IRQ handler is called every time a button on the keypad is pressed. If the state is 0 or 1 and either button 1 or 2 is pressed, a beat is created with the appropriate time value and servo number. It is then added to the beat list. If the state is 0, the timer is started, the state is changed to 1, and the red LED is turned on along with the green LED to create a yellow light, signifying to the user that the program is currently in state 1 and that they should continue to enter beats. If button 3 is pressed while in state 1, a final beat with servo number 3 and current time is added to the beat list, the state changes to 2, and the global time variable and PIT2 timer (maintains elapsed time) are reset. The red LED also turns on the signify to the user that the beat-entering period has terminated. If button 1 is pressed while in states 0 or 1, servo 1 taps one beat. If button 2 is pressed, servo 2 taps one beat.

4.3 Servo Control

Servos are controlled via a signal wire that receives a 20ms period pulse with a pulse-high width of between 1 and 2 ms. For the servos we used, a minimum pulse width of 1ms corresponded to the minimum angle of 0 degrees and a maximum pulse width of 2ms corresponded to the maximum angle of 180 degrees, as shown in the figure below.

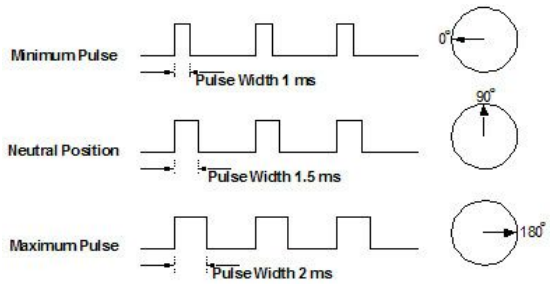

Figure 04: Servo Control Pulses

Each servo is controlled by a pulse signal generated by its own PIT. The signal pin of each servo is connected to a GPIO output pin, which sends the generated pulse to the servo. To turn the servo to a desired angle, both the length of the high pulse and the length of the low pulse (remainder of the 20ms period) in clock cycles must be loaded to the timer. Every time the timer generates an interrupt, within the IRQ handler, either a 1 or 0 is written to the GPIO pin based on the previous level of the pin. A new load value is also loaded into the timer to control the length of the high or low pulse. For example, if the desired pulse width is 1ms, the high-pulse load value will be 1ms and the low-pulse load value will be 19ms (20ms - 1ms). Every time the corresponding IRQ handler is called, if the GPIO pin is 1, it will be written to 0 and the low-pulse load value will be loaded to the timer. The timer will trigger an interrupt in 19ms, at which point, the IRQ handler will write the GPIO pin to 1 and load the high-pulse load value to the timer. The logic within the IRQ handler creates a high pulse the length of the high-pulse load value and a low pulse that completes the 20ms period.

There are two methods within *Servo.c* to determine the appropriate high-pulse and low-pulse timer load values based on the desired angle. To determine the high-pulse load value, we take an angle between 0˚ and 180˚ and scale it to a load value between 1ms and 2ms (20970 clock cycles and 41940 cycles). To determine the low-pulse load value, we subtract the high-pulse load value from the total period of 20ms to find the remainder of the period.

The timers for each servo are configured in the method *servo_setup_timers()*. Within this method, each servo is initialized to its neutral position (defined globally in *Definitions.h*). For each servo, there are two variables that help construct a hit. *tap_dat* is set to 1 when a servo is executing a hit, and *tap_counter* keeps the time a servo has been in its hit position. To hit a servo, we call method *servo_hit(int servo_num)*, which sets the desired servo angle for the appropriate servo to the hit angle (defined globally) and sets that servo's *tap_dat* variable 1.

Within each servo's PIT IRQ handler, along with generating a pulse, if tap_dat is 1, meaning a servo is mid-tap and in its hit position, *tap_counter* is incremented by 1 each pulse period (20ms). If *tap_counter* has reached the globally defined hit duration *TAP_LIMIT*, we set the servo to its neutral angle, set *tap_dat* to 0 to signify that the servo is no longer in its hit position, and reset *tap_counter*.

5) Testing

We implemented the keyboard interrupts and servo control sequentially, which allowed us to test each part individually as they were implemented. Testing the keyboard interrupts involved probing each port C pin with an oscilloscope to see rising and falling edges. We used the red and green built-in LEDs to test the button-differentiating logic in port C's IRQ handler. While testing the buttons, we noticed a

bouncing problem where multiple rising edges were being detected upon one button press. Our solution to this problem is detailed below in section 6. We tested the servo control by writing different angles and observing the generated pulse from the GPIO pins with an oscilloscope to see if the pulse width matched what we expected. We tested the functionality of the beat linked list by stepping through program using the debugger and watching the appropriate variables to see if they changed as expected.

6) Challenges and Results

6.1 Challenges

The major challenge we faced was switch bouncing, since we trigger an interrupt on the rising edge of a pulse. We began by viewing the switch press on an oscilloscope and saw that we sporadically noticed multiple rising edges on a button press. We first tried to debounce the switch by using a low-pass filter built from an RC circuit. However, we could still not get reliable button press that was always consistent. Thus, we decided to debounce in software. Initially we considered using a delay loop in the button IRQ Handler, but decided not to use a long delay in an interrupt. Thus, we devised a means to re-appropriate the PIT timers used to control the servos in order to "debounce" the switches.

We know that each PIT used for the servos has a period of 20ms, regardless of position. In order to make sure that multiple close button "presses" due to bouncing do not affect the movement of the servo, we used a counter that incremented every period to serve as a secondary "timer". When a servo moves due a rising edge-triggered interrupt, the servo movement will ignore any further inputs for 160ms (or when the secondary "timer" is 8) and continue moving towards the drum. Thus, even if there are additional bounces, the servo effectively ignores them and only reacts to the first button press within that time-frame.

6.2 Results

Our project behaves as expected. Upon reset or startup, we wait for the user to input. Once the user begins playing his or her beat, the device begins recording the user-input rhythm, with the servos beating the drums as the buttons are pressed in order to allow the user to hear the beats. After the user presses the "stop" button, there is a short pause, and the servos continuously play the user's inputs on a loop.

In order to actually hit the drums, we created a servo controller through software by generating the waveforms using GPIO output pins and a PIT timer in order to time the pulse widths in the waveform. We created a function that modifies a global variable corresponding to a desired position for the servo. In the PIT interrupt handler for each servo timer, we effectively handle the logic in order to output the correct waveform to hold the servo at that position, based on the global variable. In effect, we are able to easily add additional servos to our system (and thus additional drums), up to the limit of PIT timers that the hardware provides by simply loading different values to the timer.

7) References, Code Reuse

Other than the MK64F12.h library and stdlib.h, we did not use any external libraries in order to implement our project. We reused our queue-append code from lab 5, which we referenced from the provided solutions to lab 3, in order to append our *beat_t* elements to our *beat_list* FIFO queue.
We referred to the K64F Reference manual (http://cache.freescale.com/files/microcontrollers/doc/ref_manual/K64P144M120SF5RM.pdf) for the hardware-specific implementation of our project.

Figure 04 is taken from https://www.servocity.com/html/how_do_servos_work_.html#.Vz6rNZMrJE4