The BeatBox: A Terminal-interfacing Beat Maker

Alex Herman
CPE 316-01, Fall 2024
12-12-2024

Instr: Paul Hummel

The BeatBox

Controlled by an STM32L47RG microcontroller, the BeatBox is a simple to use, terminal-interfacing device that allows a user to generate simple four measure beats in the key of C and add accompanying drum kit tracks.

The BeatBox is designed for modular usage, with the user able to select sounds for their drumkit simply by choosing from household objects like water glasses and boxes. Multiple features increase usability, like looping, screen reset, and three selectable preset drum kits. .

System Specifications

Note Output Range	C4->B4 (261Hz -> 493Hz)	
Output Tempo	86 BPM	
Note Duration Options	1-4 beats	Note: Notes may exceed measure lines without error.
Drum Configurations	3 (Preset hardcoded)	
Drum Channels	2 (Labeled as Snare and Kick Drum)	
Speaker Volume Range	~0dB -> 80dB SPL	
Servo Response Time	~100ms	To max angle (±60°)
Speaker Output Waveform	50% duty cycle square wave	

UART Terminal Specs

Baud Rate	115,200 bps
Communication Protocol	VT-100 Escape codes

System Requirements

Microcontroller	STM32L476XXX
Display	VT-100 Compatible UART Terminal
Speaker Amplifier	PAM8403
Speaker	GikFun 3Ω, 4W (EK1725)
Servo	SM-S2309S

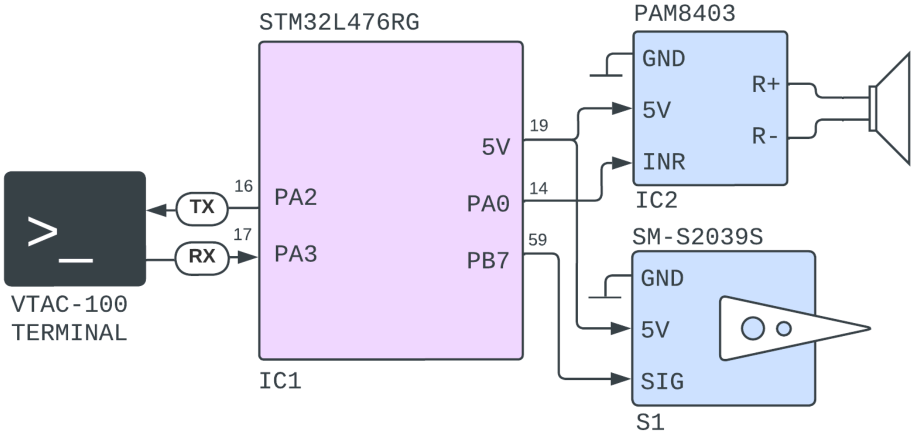


***Fig 1.1: System Schematic**. The system is composed of an STM32L47 microcontroller which controls a servo (S1) and a speaker amplifier (IC2). It communicates data over the RX and TX lines to a VT-100 Terminal.*

Software Architecture

The program is structured on a single FSM (shown at right) which manages the program flow as the user enters notes, starts playback, and configures options like looping. It also handles coordinating the timing of the playback with the servo motor, and maintaining the visual display interface.

After a simple initialization that resets the arrays and draws an empty screen, the software waits for a user to enter a keypress. Upon receiving a keypress it takes the following actions depending on the key.

User Commands	Action
[A-G][1-4][1-4][1-4]	New note A-G at measure 1-4, beat 1-4 with duration 1-4.
K[1-3]	Set Drum Kit 1-3
P	Play the current track
L	Turn on / off looping
R	Reset the display and notes

***Table 1:** User Commands.*

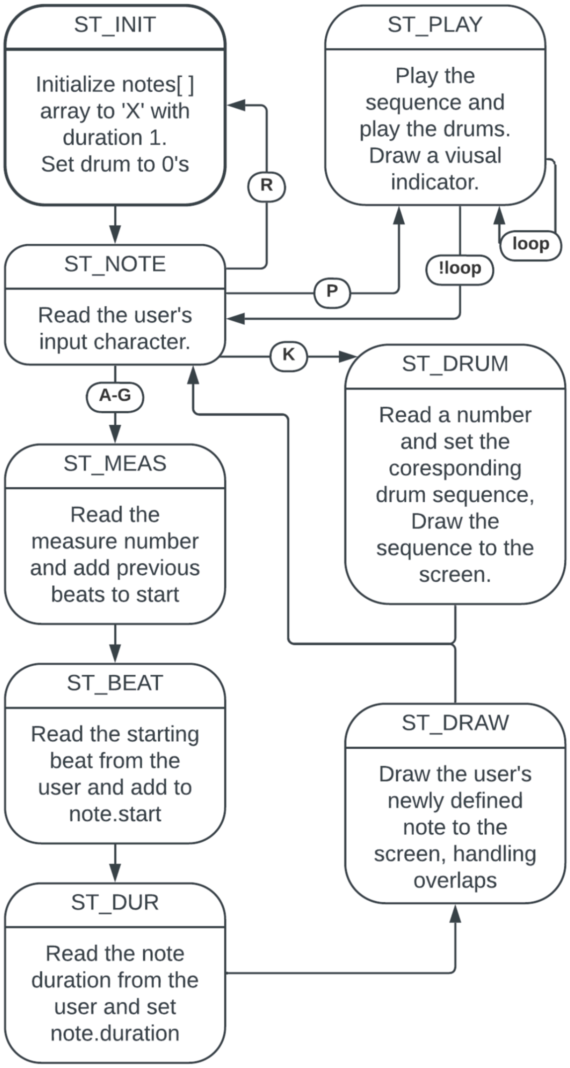
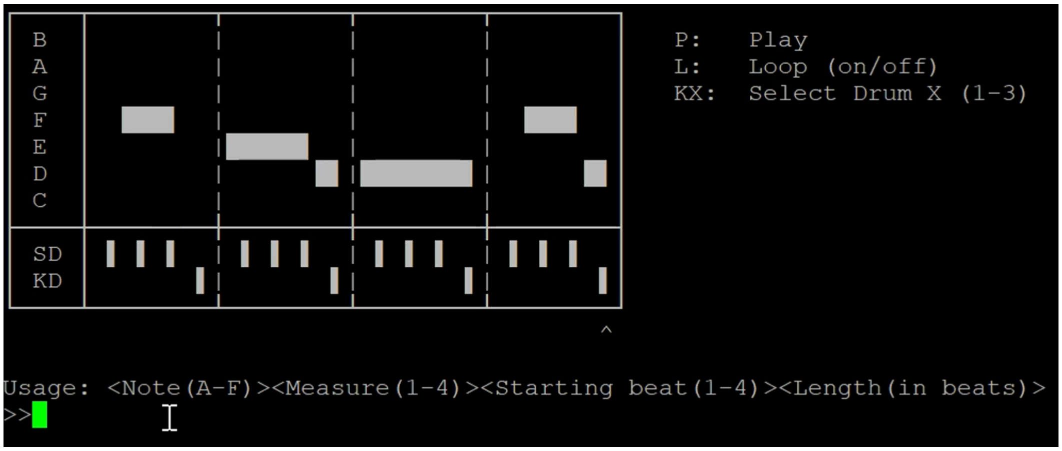

ST_INIT
Initialize notes[ ] array to 'X' with duration 1. Set drum to 0's
ST_PLAY
Play the sequence and play the drums. Draw a visual indicator.
R
loop
!loop
P
ST_NOTE
Read the user's input character.
K
ST_DRUM
Read a number and set the coresponding drum sequence, Draw the sequence to the screen.
A-G
ST_MEAS
Read the measure number and add previous beats to start
ST_BEAT
Read the starting beat from the user and add to note.start
ST_DRAW
Draw the user's newly defined note to the screen, handling overlaps
ST_DUR
Read the note duration from the user and set note.duration

***Fig 1.2: Program flow FSM.** FSM handles user input, the display, and playback.*



```
 B                                    P:   Play
 A                                    L:   Loop (on/off)
 G                                    KX:  Select Drum X (1-3)
 F
 E
 D
 C

 SD
 KD

                            ^
Usage: <Note(A-F)><Measure(1-4)><Starting beat(1-4)><Length(in beats)>
>>
```

***Fig. 1.3:** UART Terminal interface featuring note and drum display, with usage and command tips. The display also features a "^" to denote current beat during playback.*

Adding Notes

As the user types into the terminal, they begin by specifying the note they wish to play (A-G), followed by the measure, beat, and duration. A Note is a struct that has a duration, note (char), and start. As the user enters numbers after a note letter [A-G], the note is constructed. The start can be easily calculated by adding beats from previous measures to the starting beat.

After the note is constructed, it gets added to the notes[32] array at the index of its starting position and then printed to the screen. The printing function shown in **fig. 1.4.** handles any potential note overlaps, and adds or deletes notes as necessary to maintain the display.

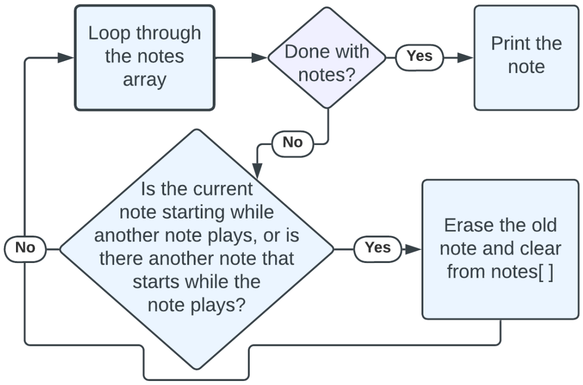

Loop through the notes array
Done with notes?
Yes
Print the note
No
Is the current note starting while another note plays, or is there another note that starts while the note plays?
Yes
Erase the old note and clear from notes[ ]
No

***Fig 1.4: Overlap Checking.** Ensures that no two notes would overlap, it calls the note printing function shown in **Fig 1.5** to print and delete.*

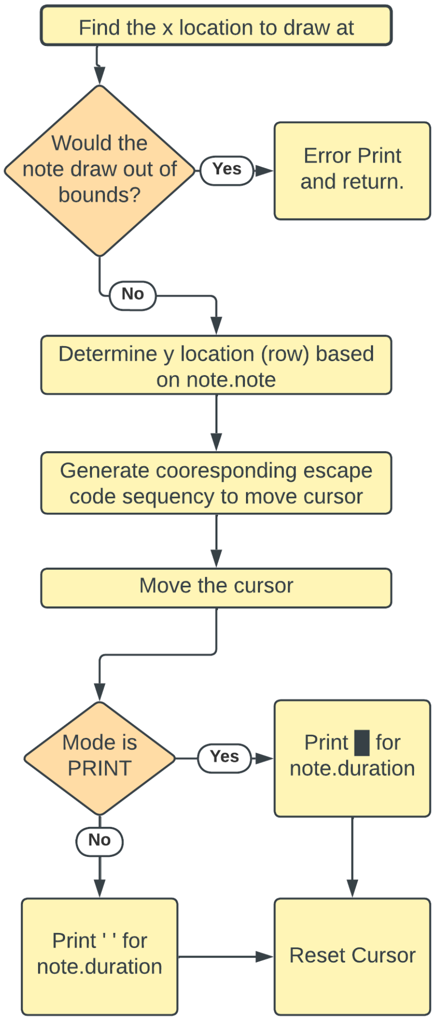

Find the x location to draw at
Would the note draw out of bounds?
Yes
Error Print and return.
No
Determine y location (row) based on note.note
Generate cooresponding escape code sequency to move cursor
Move the cursor
Mode is PRINT
Yes
Print █ for note.duration
No
Print ' ' for note.duration
Reset Cursor

***Fig 1.5. Note Printing.** The notes are printed according to their starting position relative to the measure. Notes can also be deleted in the same manner simply by printing a space.*

In addition to regular notes of the scale, the user may also select from three drum kit tracks. These are preprogrammed and upon selection, are immediately printed to the screen by following the procedure defined in **Fig. 1.6**.

Note: Additional care is taken to avoid overwriting existing measure dividers by using #defines for the three deliniators.

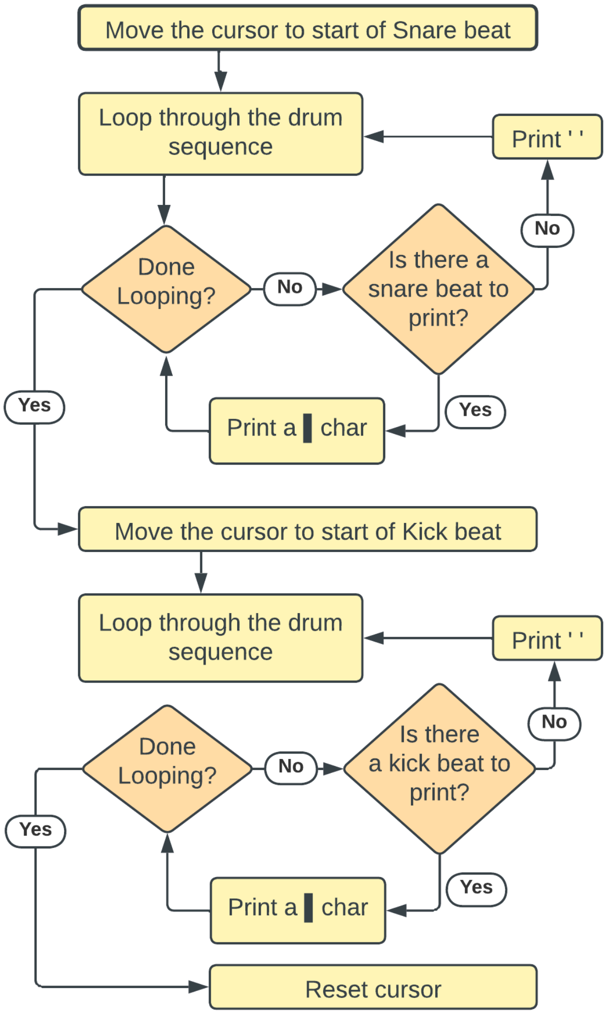

Move the cursor to start of Snare beat
Loop through the drum sequence
Print ' '
No
Done Looping?
No
Is there a snare beat to print?
Yes
Yes
Print a ▌ char
Move the cursor to start of Kick beat
Loop through the drum sequence
Print ' '
No
Done Looping?
No
Is there a kick beat to print?
Yes
Yes
Print a ▌ char
Reset cursor

Fig 1.6: Beat Printing. *Similar to the note print, beat print determines appropriate cursor placement and prints block characters to the screen. A notable difference is that with Beat Print all of the notes are added at once, and it loops through the drum_beat array twice to print snares, then kicks.*

UART

Communicating over UART is achieved at a 115200 baud connection, with VT-100 ESC Codes for cursor movement. A list of common escape codes is shown below in **table 2.** Escape codes must be printed by first printing the ESC character, 0x1B.

Escape Code	Function
"[H"	Resets the cursor to upper left c the screen
"[a;bH"	Sets the curson to col a, row b
"[0K"	Clears character in front of cursor on the current line

Table 2: *Common UART Escape codes. [1]*

In order to display the characters as the user enters them, a simple echo script is set up in the UART interrupt handler that sends the character back to the screen. Additionally characters are stored in a buffer so that they can be accessed by other functions.

Playback

After the user has selected their notes and drum kit, the notes must be played back by the speaker and servo motor. The speaker is powered by an amplifier (PAM8403) which takes the relatively weak signal from the GPIO pin A0 and turns it into a positive and negative voltage wave for the speaker.

The PWM wave is generated off the CR1 output of TIM2, which is set to have a frequency that corresponds to the note being played (note frequencies are found with a lookup table).

Note: In this setup the duty cycle is constant 50%, however PWM allows for easily modifiable duty cycle, which could allow for dynamic speaker volume, like attack and decay effects

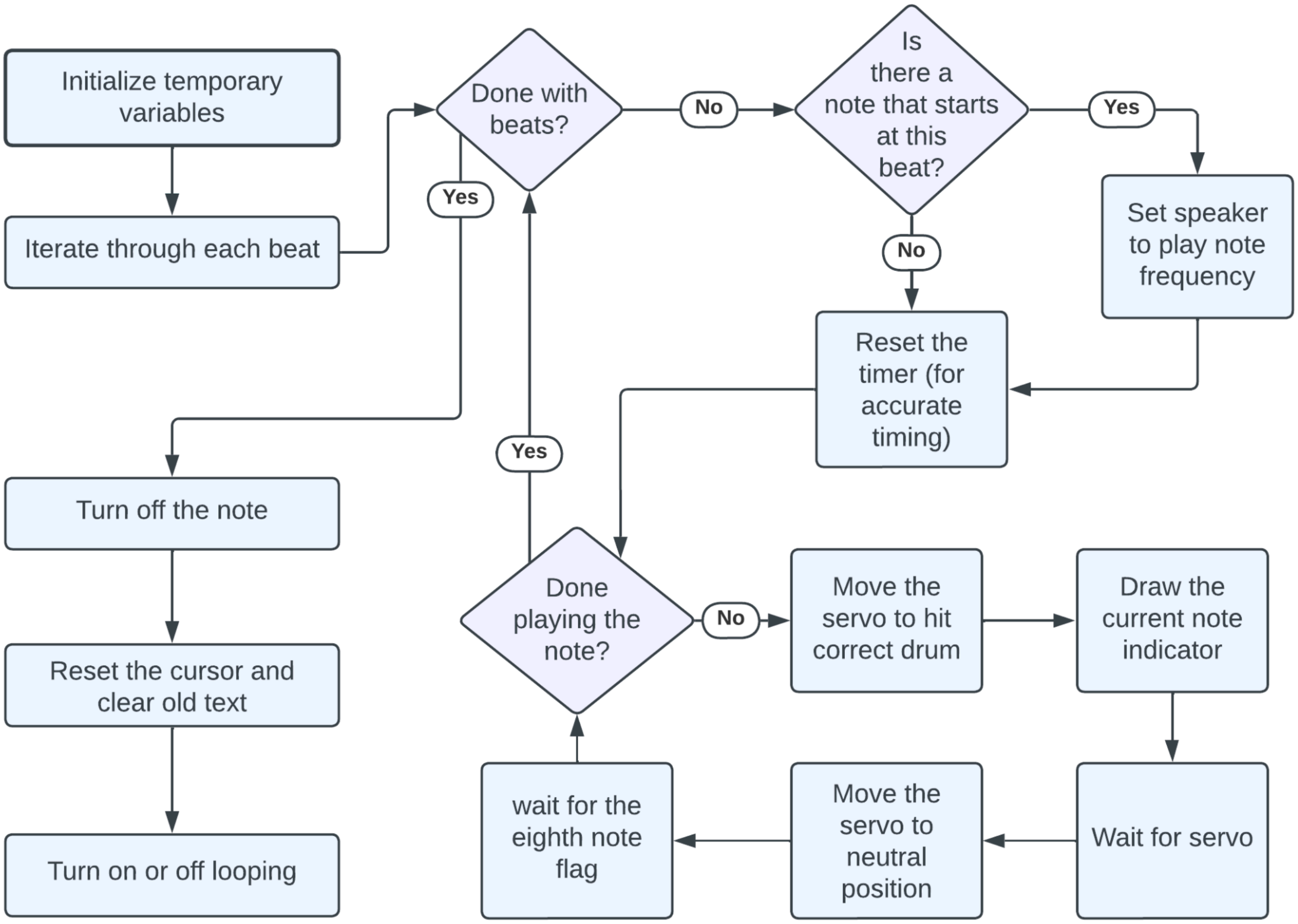

Initialize temporary variables
Iterate through each beat
Done with beats?
Yes
No
Is there a note that starts at this beat?
Yes
No
Set speaker to play note frequency
Reset the timer (for accurate timing)
Yes
Turn off the note
Reset the cursor and clear old text
Turn on or off looping
Done playing the note?
No
Move the servo to hit correct drum
Draw the current note indicator
Wait for servo
Move the servo to neutral position
wait for the eighth note flag

Fig. 1.7: Playback State. *The program iterates through the beats array, turning on and off the speaker as it encounters each note, at the same time, it also iterates through the selected drum track and moves the servo motor to the corresponding location.*

The speaker is turned on and off, with the timing being controlled by a flag set in the TIM5 ISR. The servo motor is also controlled via PWM, however it has more specific requirements. The servo can be driven anywhere from 50-200Hz [2], and through higher frequencies decrease latency, it does increase the strain on the gears.

This device uses a 100Hz signal to send pulses to the servo. The pulse width defines the position of the servo, with 0.9ms being -60°, 1.5ms being neutral, and 2.1 being 60° [2]. The pulse width is changed dynamically by changing the TIM4 CCR2 value according to equation (1) (which works for any angle n, with -60 < n < 60).

$$(1) \quad \mathrm{CCR2} = \frac{\mathrm{SYS\_CLK}}{400} \cdot \left((0.0006 \cdot \tfrac{\mathrm{DEG}}{\mathrm{MAX\_ANGLE}}) + 0.0015\right)$$

User Manual

Getting Started

Begin by plugging the device into a USB power source, and connecting a VT-100 equipped terminal. Immediately you should see the display showing up as seen in figure 3 (without the notes). If not, see **Troubleshooting**.

If you wish to use the servo motor to generate drum tracks, you will also want to locate two objects you think will make good drum blocks. To experiment, you can walk around your house with a pencil and tap on different objects until you find something that sounds fun (we recommend items that have distinctive tones like glasses and boxes).

Adding Notes

Adding notes is easy as 123. Really! Say we want to add the note A in measure 1, beat 2, with a duration of 3. We would type “A123”, simple right?

You can select between any of the available notes A-G, putting them on one of four measures, four starting beats, and four durations leading to practically infinite melodic masterpieces. One slight exception is that no two notes are allowed to play at the same time. If you make a mistake and add a note in the wrong spot, you can always just replace that note with a different one, and the software will handle the switch.

Note: Astute musicians may have recognized that in our example, A123 will play into the next measure. This is totally OK! Our software handles this so you can play whatever your heart desires.

Adding Drums

Now that you’ve selected your drum blocks, its time to make some music! Position your items such that the servo motor will swing and hit them. As the servo swings to the right, it will hit the “kick drum” and the “snare” to the left. Once you are happy with their positioning, enter “K1” to select the first drum kit. There are three drum kits to choose from, each with their own unique sound and pattern.

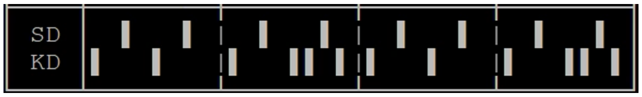


Fig 2.1 Drum Kit 1

Playing the Song

Now that you’ve got your notes and drums it's time to play! Simply type ‘P’ and you should hear the lovely melody of square waves and servo taps. If either of those isn’t happening refer to **Troubleshooting.**

By default the song will only play through once, if you wanted to listen to your beat on repeat (because it’s so good you just can get enough, obviously.) press ‘L’ and the current track will play until you press ‘L’ again.

If you wish to clear everything and start over, you can always press ‘R’.

Troubleshooting

Issue: My screen isn’t showing anything.
Sol: Make sure the cable is securely connected and the terminal is set for 115200 baud.
Issue: My speaker isn’t playing anything.
Sol: Ensure the speaker is securely connected; make sure the volume knob is turned up.
Issue: My servo arm keeps falling off.
Sol: Give the servo arm more room to swing

Code

main.c

```
/*********************************************************************
  * Project: BeatBox, a GarageBand style interface that runs
  *  in the terminal.
  * Features:
  * - Configurable notes and pattern
  * - Playing on a speaker
  * - Drum configuration with servo control
  * - Looping
  * - Reset
  ********************************************************************/

#include "main.h"
#include "uart.h"
#include "note.h"
#include "servo.h"
#include <stdio.h>
#include <stdlib.h>

#define KICK_ANGLE 60
#define SNARE_ANGLE -60

#define PRINT 0
#define DEL 1

#define C4 261.63
#define D4 293.66
#define E4 329.63
#define F4 349.23
#define G4 392.00
#define A4 440.00
#define B4 493.88

#define CARROT_START 6
#define CARROT_Y     13
#define FORMAT_BUF   20

#define NUM_BEATS 32
#define DRUM_DELAY 0.1 // s
volatile char user_char;
volatile uint8_t char_ready = 0;
volatile uint8_t eighth_note;
volatile uint8_t drum_delay = 0;
int8_t drum_pattern[NUM_BEATS];
uint8_t input_indx = 0;
uint16_t frequency = 440;
uint16_t duty = 50;
Note notes[NUM_BEATS];
uint8_t looping = 0;
```

```
uint8_t error_flag = 0;

/* Private function prototypes -----------------------------------------------*/
void safe_Note_Print(Note note);
void SystemClock_Config(void);
void TIM2_Init(void);
void TIM5_Init(void);
void delay_us(const uint32_t time_us);

int main(void)
{
  HAL_Init();
  SystemClock_Config();

  UART_init();
  TIM2_Init();
  TIM5_Init();
  servo_Init();
  __enable_irq();

  typedef enum {
           ST_INIT,
      ST_NOTE,
      ST_MEAS,
      ST_BEAT,
      ST_DUR,
      ST_DRUM,
      ST_PLAY,
      ST_DRAW,
           ST_DRAW_DRUM,
      ST_LOOP
  } State;

  State PS = ST_INIT;
  State NS;

  while (1)
  {
      Note new;
      switch (PS) {
           case ST_INIT:
             // Init notes array and drum_pattern to empty values;
             for (int i = 0; i < NUM_BEATS; i++) {
                 notes[i].start = i; // For sorting purposes
                 notes[i].duration = 1;
                 notes[i].note = 'X';

                 drum_pattern[i] = 0;
             }

             UART_SCREEN_Print();
             UART_ESC_Code("7"); // store original cursor location
             NS = ST_NOTE;
             break;
```

```
case ST_NOTE:
    // wait for the user to enter a character
    char_ready = 0;
    while(!(char_ready));
    char_ready = 0;

    // [A-F]: Note, K: Drum-Kit, P: Play, L: Loop, R: Reset
    switch (user_char) {
        case('A'):
            new.note = 'A';
            NS = ST_MEAS;
            break;
        case('B'):
            new.note = 'B';
            NS = ST_MEAS;
            break;
        case('C'):
            new.note = 'C';
            NS = ST_MEAS;
            break;
        case('D'):
            new.note = 'D';
            NS = ST_MEAS;
            break;
        case('E'):
            new.note = 'E';
            NS = ST_MEAS;
            break;
        case('F'):
            new.note = 'F';
            NS = ST_MEAS;
            break;
        case('G'):
            new.note = 'G';
            NS = ST_MEAS;
            break;
        case('K'):
            NS = ST_DRUM;
            break;
        case('P'):
            NS = ST_PLAY;
            break;
        case('L'):
            looping ^= 1;
            user_char = ' ';
            NS = ST_PLAY;
            break;
        case('R'):
            NS = ST_INIT;
            break;
        default:
            UART_Print_And_Reset("invalid character.");
            NS = PS;
```

```
                        break;
                  }
                  break;

            case ST_MEAS:
                  while(!(char_ready));
                  char_ready = 0;
                  // add beats from previous measures to start.
                  switch (user_char) {
                        case('1'):
                              new.start = (1 - 1)*8;
                              break;
                        case('2'):
                              new.start = (2 - 1)*8;
                              break;
                        case('3'):
                              new.start = (3 - 1)*8;
                              break;
                        case('4'):
                              new.start = (4 - 1)*8;
                              break;
                        default:
                              UART_Print_And_Reset("invalid measure.");
                              error_flag = 1;
                              break;
                  }
                  if (error_flag) {
                        NS = PS;
                        error_flag = 0;
                  } else {
                        NS = ST_BEAT;
                  }
                  break;

            case ST_BEAT:
                  while(!(char_ready));
                  char_ready = 0;
                  // Add the beat to the start (total beats)
                  switch (user_char) {
                        case('1'):
                              new.start += 0;
                              break;
                        case('2'):
                              new.start += 2;
                              break;
                        case('3'):
                              new.start += 4;
                              break;
                        case('4'):
                              new.start += 6;
                              break;
                        default:
                              UART_Print_And_Reset("invalid starting beat.");
                              error_flag = 1;
```

```
                    break;
            }
            if (error_flag) {
                NS = PS;
                error_flag = 0;
            } else {
                NS = ST_DUR;
            }
            break;

        case ST_DUR:
            while(!(char_ready));
            char_ready = 0;
            // Set the duration
            switch (user_char) {
                case('1'):
                    new.duration = 2;
                    break;
                case('2'):
                    new.duration = 4;
                    break;
                case('3'):
                    new.duration = 6;
                    break;
                case('4'):
                    new.duration = 8;
                    break;
                default:
                    UART_Print_And_Reset("invalid duration.");
                    error_flag = 1;
                    break;
            }
            if (error_flag) {
                NS = PS;
                error_flag = 0;
            } else {
                NS = ST_DRAW;
            }
            break;

        case ST_DRUM:
            while(!(char_ready));
            char_ready = 0;
            switch (user_char) {
                case('1'):
                    // beat 1: alternating kick and snare
                    int8_t pattern1[NUM_BEATS] = {1,0,-1,0,1,0,-1,0,1,0,-1
                                                  ,0,1,1,-1,1,1,0,-1,0,1

,0,-1,0,1,0,-1,0,1,1,-1,1};
                    for (int i = 0; i < NUM_BEATS; i++) {
                        drum_pattern[i] = pattern1[i];
                    }
                    break;
```

```
                      case('2'): // Snare with some kick
                           int8_t pattern2[NUM_BEATS] =
{0,-1,0,-1,0,-1,0,1,0,-1,0,

                                                      -1,0,-1,0,1,0,-1,0,-1,0,-1,
                                                         0,1,0,-1,0,-1,0,-1,0,1};
                           for (int i = 0; i < NUM_BEATS; i++) {
                                 drum_pattern[i] = pattern2[i];
                           }
                           break;
                      case('3'): // Less drums
                           int8_t pattern3[NUM_BEATS] = {1,0,0,1,-1,0,0,0,0,0,0,
                                                         0,1,-1,0,0,1,0,0,1,-1,0
                                                         ,0,0,0,0,0,0,1,-1,0,0};
                           for (int i = 0; i < NUM_BEATS; i++) {
                                 drum_pattern[i] = pattern3[i];
                           }
                           break;
                      default:
                           UART_Print_And_Reset("invalid kick seq, select 1-3");
                           error_flag = 1;
                           break;
                 }
                 // Print the drum beat and reset the cursor
                 UART_BEAT_Print(drum_pattern);
                 UART_ESC_Code("8");
                 UART_ESC_Code("[0K");

                 if (error_flag) {
                      NS = PS;
                      error_flag = 0;
                 } else {
                      NS = ST_NOTE;
                 }
                 break;

           case ST_PLAY:
                 Note temp;
                 uint8_t carrot_x = CARROT_START;
                 char formatted_str[FORMAT_BUF];

                 // go through each beat
                 for (int i = 0; i < NUM_BEATS; i++) {

                      temp = notes[i];
                      // Check if there is a note that starts at this beat
                      if (temp.note != 'X') {
                           // Get the notes frequency
                           frequency = get_Freq(temp);
                           // Turn on the timer with correct freq and duty cycle
                           TIM2->ARR = SYS_CLK / frequency;
                           TIM2->CCR1 = (SYS_CLK / frequency) / 2; // 50% duty
                           TIM2->EGR |= TIM_EGR_UG;    // Update the registers
                      }
```

```
            // Reset the timer for accurate timing.
            TIM5->EGR = TIM_EGR_UG;
            // Play note for note.duration and do the drum
            for (int j = 0; j < temp.duration; j++) {
                // Play the drum beat
                if (drum_pattern[i] == -1) {
                    servo_Move(SNARE_ANGLE);
                } else if (drum_pattern[i] == 1) {
                    servo_Move(KICK_ANGLE);
                }

                // Draw the carrot
                carrot_x++;
                // Generate the ESC Code string below table (y = 13)
                sprintf(formatted_str, "[%d;%dH", CARROT_Y, carrot_x);
                UART_ESC_Code(formatted_str); // Move the cursor
                UART_ESC_Code("[2K"); // Clear the line
                UART_CHAR_Print('^');
                // Skip measure lines
                if ((carrot_x == DELIN_1) |
                    (carrot_x == DELIN_2) |
                    (carrot_x == DELIN_3)) {
                    carrot_x++;
                }

                delay_us(100000); // Adding a 100ms delay for servo
                servo_Move(NEUTRAL);

                eighth_note = 0;
                while(!(eighth_note));
                i++;
            }
            i--;
            TIM2->CCR1 = 0; // Turn off the sound.
            delay_us(8000); // Adding a 8ms delay for note spacing.
        }

        UART_ESC_Code("8"); // Reset the cursor
        UART_ESC_Code("[0K"); // Clear old text


        // Check to see if we need to start/stop looping.
        if (user_char == 'L') {
            user_char = 'P'; // prevent edge case where it only loops
once
            looping ^= 1;    // toggle looping
            char_ready = 0;
        }

        if (looping == 1) {
            NS = ST_PLAY;
            break;
```

```
                 }
                 NS = ST_NOTE;
                 break;

           case ST_DRAW:
                 // Print the note (checking to see if it overlaps)
                 safe_Note_Print(new);
                 notes[new.start] = new;

                 UART_ESC_Code("8");   // Reset the cursor
                 UART_ESC_Code("[0K");

                 NS = ST_NOTE;
                 break;

           case ST_LOOP:
                 // Handle ST_LOOP state
                 NS = ST_LOOP;
                 break;

           default:
                 // Handle invalid state
                 UART_print("whoop.");
                 break;
      }

      // Update the present state (PS) to the next state (NS)
      PS = NS;

  }
}

void safe_Note_Print(Note note) {
    Note temp;
    for (int i = 0; i < NUM_BEATS; i++) {
        temp = notes[i];
        // Look to see if the new note would be overlapping any existing notes
        if (((note.start >= temp.start) && (note.start < (temp.start +
            temp.duration))) | ((temp.start > note.start) && (temp.start <
            (note.start + note.duration)))) {
            // Replace the note in the notes list.
            notes[temp.start].duration = 1;
            notes[temp.start].note = 'X';
            UART_NOTE_Print(temp, DEL);
        }
    }
    UART_NOTE_Print(note, PRINT);
}

void USART2_IRQHandler(void) {
 if (USART2->ISR & USART_ISR_RXNE) { // Check if data is received
      user_char = USART2->RDR; // Read the received data
      char_ready = 1;
      // Echo back the received character
      while (!(USART2->ISR & USART_ISR_TXE)); // Wait until Tx register is empty
      USART2->TDR = user_char;           // Transmit the data
 }
}

void TIM2_Init(void) {

  RCC->APB1ENR1 |= (RCC_APB1ENR1_TIM2EN);

  // Generate wave at frequency
  TIM2->ARR = SYS_CLK / frequency;
  // output sound wave initially off
  TIM2->CCR1 = 0;

  // Edge aligned mode, upcounting
  TIM2->CR1 &= ~(TIM_CR1_CMS);
  TIM2->CR1 &= ~(TIM_CR1_DIR);

  // PWM 2 (active above ccr1) for up/down counting
  TIM2->CCMR1 &= ~(TIM_CCMR1_OC1M);
  TIM2->CCMR1 |= (6 << TIM_CCMR1_OC1M_Pos);

  // Preload enable
  TIM2->CR1 |= TIM_CR1_ARPE;        // Enable Auto-Reload Preload
  TIM2->CCMR1 |= TIM_CCMR1_OC1PE;   // Enable CCR1 Preload

  // Enable the OC1 channel output. (Pin A0)
  TIM2->CCMR1 &= ~(TIM_CCMR1_CC1S);
  TIM2->CCER |= (TIM_CCER_CC1E);

  // GPIO for PA0, AF1:
  // Output, High Speed, No push pull
  RCC->AHB2ENR  |= (RCC_AHB2ENR_GPIOAEN);
  GPIOA->MODER   &= ~(GPIO_MODER_MODE0);
  GPIOA->MODER   |=  (GPIO_MODER_MODE0_1);
  GPIOA->OTYPER  &= ~(GPIO_OTYPER_OT0);
  GPIOA->PUPDR   &= ~(GPIO_PUPDR_PUPD0);
  GPIOA->OSPEEDR |=  (3 << GPIO_OSPEEDR_OSPEED0_Pos);

  // AF1
  GPIOA->AFR[0]  &=  ~(GPIO_AFRL_AFSEL0);
  GPIOA->AFR[0]  |=   (1 << GPIO_AFRL_AFSEL0_Pos);

  //start timer
  TIM2->CR1  |= TIM_CR1_CEN;
}

void TIM5_Init(void) {
 RCC->APB1ENR1 |= (RCC_APB1ENR1_TIM5EN);

 TIM5->ARR = SYS_CLK* 2 / 8 + 1;
 // CCR1 for 1/8th note delay
 TIM5->CCR1 = SYS_CLK*2 / 8;
 // TIM5 Channel 1 (eighth note delay) as output compare mode
 TIM5->CCMR1 &= ~TIM_CCMR1_OC1M;
 TIM5->CCMR1 |= (1 << TIM_CCMR1_OC1M_Pos); // goes high on match
 TIM5->CCER |= TIM_CCER_CC1E;  // Enable capture/compare channel 1
 TIM5->DIER |= TIM_DIER_CC1IE; // Interrupts enabled

 NVIC -> ISER[1] = (1 << (TIM5_IRQn % 32)); // NVIC interrupt enable

 TIM5->CR1  |= TIM_CR1_CEN;
}

void TIM5_IRQHandler(void) {
 if (TIM5->SR & TIM_SR_CC1IF) { // Check if CCR1 triggered the interrupt
      TIM5->SR &= ~TIM_SR_CC1IF; // Clear the interrupt flag

      // Set the flag
      eighth_note = 1;
 }
}

void delay_us(const uint32_t time_us) {
 // set the counts for the specified delay
 SysTick->LOAD = (uint32_t)((time_us * (SystemCoreClock / 1000000)) - 1);
 SysTick->VAL = 0;                                      // clear the timer count
 SysTick->CTRL &= ~(SysTick_CTRL_COUNTFLAG_Msk);        // clear the count flag
 while (!(SysTick->CTRL & SysTick_CTRL_COUNTFLAG_Msk)); // wait for the flag
}
```

uart.c

```
 /*********************************************************************
   * fie: uart.c
   * Purpose: Configures and allows writing to a terminal via UART.
   * Features:
   * - Adjustable Baud Rate
   * - ESC Codes for cursor movement
   * - Functions for INT, VOLT, and String printing
   * - ISR with echo and character read
   * Porject Specific Functionality:
   * - Screen Printing
   * - Note Printing and Deleting
   * - Beat / Pattern printing
   * - Error code printing with cursor reset
   ********************************************************************/

 #include "uart.h"
 #include <stdio.h>

 void UART_ESC_Code(char* code) {
   // wait before escape code
      while (!(USART2->ISR & USART_ISR_TXE));
      // print the esc code, 0x1B
```

```
     USART2->TDR = 0x1B;
     // Print the rest of the code characters
     for (int i = 0; i < strlen(code) + 1; i++) {
               // Wait for there to be space in the transfer register
               while (!(USART2->ISR & USART_ISR_TXE));
               // Put the current character in the transfer register
               USART2->TDR = code[i];
          }
}

void UART_SCREEN_Print(void) {
     UART_ESC_Code("[H"); // Reset the cursor
     UART_ESC_Code("[2J");
     // Print the Table Display
     UART_print("┌────┬───────┬───────┬───────┬───────┐ ");
     UART_ESC_Code("[2;0H");
     UART_print("│ B  │       |       |       |       │   P:   Play");
     UART_ESC_Code("[3;0H");
     UART_print("│ A  │       |       |       |       │   L:   Loop
(on/off)");
     UART_ESC_Code("[4;0H");
     UART_print("│ G  │       |       |       |       │   KX:  Select Drum X
(1-3)");
     UART_ESC_Code("[5;0H");
     UART_print("│ F  │       |       |       |       │   R:   Reset");
     UART_ESC_Code("[6;0H");
     UART_print("│ E  │       |       |       |       │");
     UART_ESC_Code("[7;0H");
     UART_print("│ D  │       |       |       |       │");
     UART_ESC_Code("[8;0H");
     UART_print("│ C  │       |       |       |       │");
     UART_ESC_Code("[9;0H");
     UART_print("├────┼───────┼───────┼───────┼───────┤");
     UART_ESC_Code("[10;0H");
     UART_print("│ SD │       |       |       |       │");
     UART_ESC_Code("[11;0H");
     UART_print("│ KD │       |       |       |       │");
     UART_ESC_Code("[12;0H");
     UART_print("└────┴───────┴───────┴───────┴───────┘");
     UART_ESC_Code("[15;0H");

     // Print user directives and usage notes
     UART_print("Usage: <Note(A-F)><Measure(1-4)><Starting beat(1-4)><Length(in
beats)>");
     UART_ESC_Code("[16;0H");
     UART_print(">>");

}

// Print a note (or delete if mode == 1);
void UART_NOTE_Print(Note note, uint8_t mode) {
     // Determine the note's position relative to the display
     char formatted_str[BUF_LEN];
```

```c
// Find the x location
// x location = start + visual offset + measure lines
// start = meas + beat in meas
uint16_t note_x = note.start + START_OF_TABLE + (note.start) / 8;

// Check if x_location will go out of bounds.
if (note_x + note.duration > END_OF_TABLE) {
     UART_Print_And_Reset("Notes cannot exceed the measure boundaries");
     return;
}

// Find the y location
uint16_t note_y;
switch (note.note) {
     case 'C':
          note_y = 8;
          break;
     case 'D':
          note_y = 7;
          break;
     case 'E':
          note_y = 6;
          break;
     case 'F':
          note_y = 5;
          break;
     case 'G':
          note_y = 4;
          break;
     case 'A':
          note_y = 3;
          break;
     case 'B':
          note_y = 2;
          break;
     default:
          note_y = -1; // invalid input
          break;
  }

// Generate the ESC Code string
sprintf(formatted_str, "[%d;%dH", note_y, note_x);
// Move the cursor to the correct location for the note
UART_ESC_Code(formatted_str);

// Print the correct number of notes (duration - 1 = number of squares)
for (int i = 0; i < (note.duration) - 1; i++) {
     if (mode == PRINT) {
          UART_print("█");
     } else {
       UART_print(" ");
     }
     if ((note_x == DELIN_1) | (note_x == DELIN_2) | (note_x == DELIN_3)) {
          UART_print("|");
```

```
        }
        note_x++;
    }
    // Print a half square
    if (mode == PRINT) {
        UART_print("█ ");
    } else {
      UART_print("  ");
    }
    UART_ESC_Code("8"); // Reset cursor

}

void UART_BEAT_Print(int8_t sequence[32]) {

    // Move the cursor to the start of the SNARE beat.
    UART_ESC_Code("[10;7H");
    // Loop through the beats of the seq.
    for (int i = 0; i < NUM_BEATS; i++) {
        // Print measure markers as necessary
        if ((i == 8) || (i == 16) || (i == 24)) {
            UART_print("|");
        }
        // Check if there is a drum beat to print
        if (sequence[i] == -1) {
            UART_print("█ ");
        } else {
            UART_print("  ");
        }
    }

    // Move to start of the KICK beat.
    UART_ESC_Code("[11;7H");
    // Loop through the beats of the seq.
    for (int i = 0; i < NUM_BEATS; i++) {
        // Print measure markers as necessary
        if ((i == 8) || (i == 16) || (i == 24)) {
            UART_print("|");
        }
        // Check if there is a drum beat to print
        if (sequence[i] == 1) {
            UART_print("█ ");
        } else {
            UART_print("  ");
        }
    }

    // Reset the cursor
    UART_ESC_Code("8");
}

// Delayed print and clear printing errors etc.
void UART_Print_And_Reset(char* error) {
    UART_ESC_Code("8"); // Reset cursor and clear line before printing.
```

```
     UART_ESC_Code("[0K");
     UART_print(error);  // print error
     HAL_Delay(1000);    // wait 1s
     UART_ESC_Code("8"); // restore cursor
     UART_ESC_Code("[0K"); // clear line
}

// Sends a string through UART
void UART_print(char* string) {
     for (int i = 0; i < strlen(string) + 1; i++) {
                // Wait for there to be space in the transfer register
                while (!(USART2->ISR & USART_ISR_TXE));
                // Put the current character in the transfer register
                USART2->TDR = string[i];
           }
}

void UART_CHAR_Print(uint8_t ch) {
     // Wait for there to be space in the transfer register
     while (!(USART2->ISR & USART_ISR_TXE));
     // Put the char in the TDR
     USART2->TDR = ch;
}

void UART_init(void) {
     // Clock configurations
     RCC->APB1ENR1  |= RCC_APB1ENR1_USART2EN; // turn on the USART clock
     RCC->AHB2ENR   |=  (RCC_AHB2ENR_GPIOAEN); // turn on GPIOA clock
     RCC->CCIPR         |= (RCC_CCIPR_ADCSEL); // Select system clock USART
source

     // Register Initializations
     USART2->CR1 &= ~(USART_CR1_M1 | USART_CR1_M0); // M1 and M0 set data size 8

     USART2->BRR = SYS_CLK / BAUD; // Setting for 115.2 kbs

     USART2->CR2 &= ~(USART_CR2_STOP); // 1 stop bit

     USART2->CR1 |= (USART_CR1_UE); // Turn on the USART
     USART2->CR1 |= (USART_CR1_TE); // Transmit enable
     USART2->CR1 |= (USART_CR1_RE); // Recieve enable

     //Set up interrupts for echo
     USART2->CR1 |= USART_CR1_RXNEIE; // Enable RXNE interrupt
     NVIC->ISER[1] = (1 << (USART2_IRQn % 32));

     // Set up PA2 (TX) and PA3 (RX)
     // Alternate Function
     GPIOA->MODER   &= ~(GPIO_MODER_MODE2 | GPIO_MODER_MODE3);
  GPIOA->MODER   |=  (GPIO_MODER_MODE2_1 | GPIO_MODER_MODE3_1);

  // Push Pull
  GPIOA->OTYPER  &= ~(GPIO_OTYPER_OT2 | GPIO_OTYPER_OT3);
  // No pull up / pull down
  GPIOA->PUPDR   &= ~(GPIO_PUPDR_PUPD2 | GPIO_PUPDR_PUPD3);
  // Very fast
  GPIOA->OSPEEDR |=  ((3 << GPIO_OSPEEDR_OSPEED2_Pos)
                         | (3 << GPIO_OSPEEDR_OSPEED3_Pos));

  // Put pins in AF7
  GPIOA->AFR[0]  &=  ~(GPIO_AFRL_AFSEL2 |
                                                      GPIO_AFRL_AFSEL3);
  GPIOA->AFR[0]  |=   ((7 << GPIO_AFRL_AFSEL2_Pos)
                                                 | (7 << GPIO_AFRL_AFSEL3_Pos));
}
```

uart.h

```
/*********************************************************
  * file: uart.h
  * UART function definitions and variable initializations.
  ********************************************************/

#ifndef UART_H
#define UART_H

#include <string.h>
#include "main.h"
#include "note.h"

/* Macro Definitions */
#define SYS_CLK 40000000
#define BAUD    115200
#define DC      1
#define AC      0

#define VREF_MV 3300        // 3.3V
#define ADC_MAX_COUNT 4095  // Maximum ADC count for a 12-bit ADC
#define DELIN_1 14
#define DELIN_2 23
#define DELIN_3 32
#define END_OF_TABLE 42
#define START_OF_TABLE 7
#define BUF_LEN

#define KICK 1
#define SNARE -1

#define PRINT 0
#define DEL 1

#define NUM_BEATS

/* Function Prototypes */
void UART_init(void);
void UART_print(char *string);
void UART_ESC_Code(char *code);
void UART_CHAR_Print(uint8_t ch);
void UART_BEAT_Print(int8_t sequence[32]);
void UART_SCREEN_Print(void);
void UART_NOTE_Print(Note note, uint8_t mode);
void UART_Print_And_Reset(char* error);
#endif
```

note.c

```
/*
 * note.c
 * The note struct and methods
 */

#include "note.h"

uint16_t get_Freq(Note note) {

     char note_letter = note.note;
     switch (note_letter) {
          case 'C':
               return 262;
               break;
          case 'D':
               return 294;
               break;
          case 'E':
               return 330;
               break;
          case 'F':
               return 349;
               break;
          case 'G':
               return 392;
               break;
          case 'A':
               return 440;
               break;
          case 'B':
               return 494;
               break;
          default:
               return -1; // invalid input
               break;
       }
}
```

note.h

```
/*
 * note.h
 * Header file for note.c
 */
#ifndef NOTE_H
#define NOTE_H
```

```
#include "main.h"

typedef struct {
           char     note;
    uint16_t duration; // number of beats, range(0.5-4)
    uint16_t start; // starting beat (includes measure)
} Note;

uint16_t get_Freq(Note note);
uint8_t check_Note(Note note);

#endif // NOTE_H
```

servo.c

```
/*
 * servo.c
 * Code to Control a simple servo motor
 * Servo is controlled by PWM with 0.9 - 2.1ms pulses
 * servo_Move allows for precise positioning of the servo
 */
#include "servo.h"

// Move servo to position degrees (-60 -> 60)
void servo_Move(int16_t degrees) {
     // Set the duty cycle to a scaled value depending on frequency
     TIM4->CCR2 = (SYS_CLK / 400) * ((POS_PERIOD - NEUTRAL_PERIOD) *
(degrees/POS) + NEUTRAL_PERIOD);
}

void servo_Init(void) {
     // Set up PWM on a TIM4 channe2
  RCC->APB1ENR1 |= (RCC_APB1ENR1_TIM4EN);

  // 100Hz signal for fast servo response.
  TIM4->PSC = 399;   // Prescaler
  TIM4->ARR = 999;   // Auto-reload for 10ms period = 100Hz

  // CCR2 set to start at 1.5ms (neutral)
  TIM4->CCR2 = (SYS_CLK / 400) * NEUTRAL_PERIOD;

  // Edge aligned mode, upcounting
  TIM4->CR1 &= ~(TIM_CR1_CMS);
  TIM4->CR1 &= ~(TIM_CR1_DIR);

  // PWM 1 (active below ccr2) for up/down counting
  TIM4->CCMR1 &= ~(TIM_CCMR1_OC2M);
  TIM4->CCMR1 |= (6 << TIM_CCMR1_OC2M_Pos);

  // Preload enable
  TIM4->CR1 |= TIM_CR1_ARPE;         // Enable Auto-Reload Preload
  TIM4->CCMR1 |= TIM_CCMR1_OC2PE;    // Enable CCR2 Preload
  // Enable the OC2 channel output. (Pin B7)
  TIM4->CCMR1 &= ~(TIM_CCMR1_CC2S);
  TIM4->CCER |= (TIM_CCER_CC2E);

  // GPIO for PB7, AF2:
  // Output, High Speed, No push pull
  RCC->AHB2ENR  |= (RCC_AHB2ENR_GPIOBEN);
  GPIOB->MODER   &= ~(GPIO_MODER_MODE7);
  GPIOB->MODER   |=  (GPIO_MODER_MODE7_1);
  GPIOB->OTYPER  &= ~(GPIO_OTYPER_OT7);
  GPIOB->PUPDR   &= ~(GPIO_PUPDR_PUPD7);
  GPIOB->OSPEEDR |=  (3 << GPIO_OSPEEDR_OSPEED7_Pos);

  // AF1
  GPIOB->AFR[0]  &=  ~(GPIO_AFRL_AFSEL7);
  GPIOB->AFR[0]  |=   (2 << GPIO_AFRL_AFSEL7_Pos);

  //start timer
  TIM4->CR1  |= TIM_CR1_CEN;
}
```

servo.h

```c
/*
 * servo.h
 * Header File for servo.c
 */

#ifndef SRC_SERVO_H_
#define SRC_SERVO_H_

#include "main.h"
#include "uart.h"

#define NEG -60
#define POS 60
#define NEUTRAL 0

// seconds
#define NEUTRAL_PERIOD 0.0015
#define POS_PERIOD 0.0021
#define NEG_PERIOD 0.0009

void servo_Move(int16_t degrees); //degrees -60->60
void servo_Init(void);

#endif /* SRC_SERVO_H_ */
```

References

[1] P. Hummel and J. Gerfen, *Microcontrollers and Embedded Applications Laboratory Manual: STM32L476RG / STM32L4A6ZG / STM32L496ZG* [Class Notes], with contributions by T. Houalla, Version 2.1a, © Paul Hummel and Jeff Gerfen.

[2] SpringRC, "SM-S2039S Servo Motor Datasheet," [Online]. Available: https://www.servodatabase.com/servo/springrc/sm-s2039s. [Accessed: Dec. 10, 2024].