MC You

E155 Final Project
December 10, 2021
Santiago Rodriguez and Kariessa Schultz

Abstract

The goal of this project was to create an audio-mixing station which uses two stepper motors to play a song and plays beat sequences according to user input on a speaker. To create this system, we used the STM32F401RE microcontroller (MCU) and the MAX1000 board (FPGA). We wrote an I2C driver for the MCU to talk to a digital-to-analog converter (DAC), developed two novel encodings: one to hold the stepper motor audio data and the other for the FPGA to send the MCU commands over SPI, and wrote multiple finite state machines on the FPGA to process user input. We also created a custom box to hold our breadboards, MCU, and FPGA, to present a nicer interface for the user. The microcontroller is responsible for keeping track of the system's state including remembering two sequences of beats entered by the user, whether or not the stepper motor song is playing, and the current progress through the song. The FPGA handles user input, processing sequences of switch presses into abstract syntax which is sent to the MCU over an SPI link.

Introduction

Music production and consumption is higher than ever. We wanted to make a musical product that would allow people to play with a song in ways like a DJ. This is how we came up with functionality like putting beats over the song and jumping back to a specific point in time in the song. Inspired by the creativity of some artists on Youtube, we decided to use stepper motors to play our music.

The top-level diagram of the system is shown in Figure 1. The user presses the buttons (a matrix keypad and two recording buttons), and the resulting signals are sent to the FPGA. See Figure 2 for the available user commands.

The FPGA handles switch bounce and, if the user is recording a sequence, stores the series of button presses and the times between each press in memory. The FPGA converts this data into byte-long commands which are sent to the MCU over SPI. If the command indicates that there is a sequence to be read, then the MCU will keep running the SPI until it retrieves all of the data from the FPGA. The MCU then responds appropriately by updating its state, which is a set of global variables controlling its behavior and the saved beat sequences. In the MCU's main function, it uses time multiplexing to send audio data over an I2C link to the DAC in parallel with PWM signals to the motor drivers. This outputs sequences of beats on the speaker and determines the specific note that the motors play, respectively. Interrupts handle the inputs to the MCU from the FPGA and govern the speaker and stepper motor behavior accordingly.

Block diagram

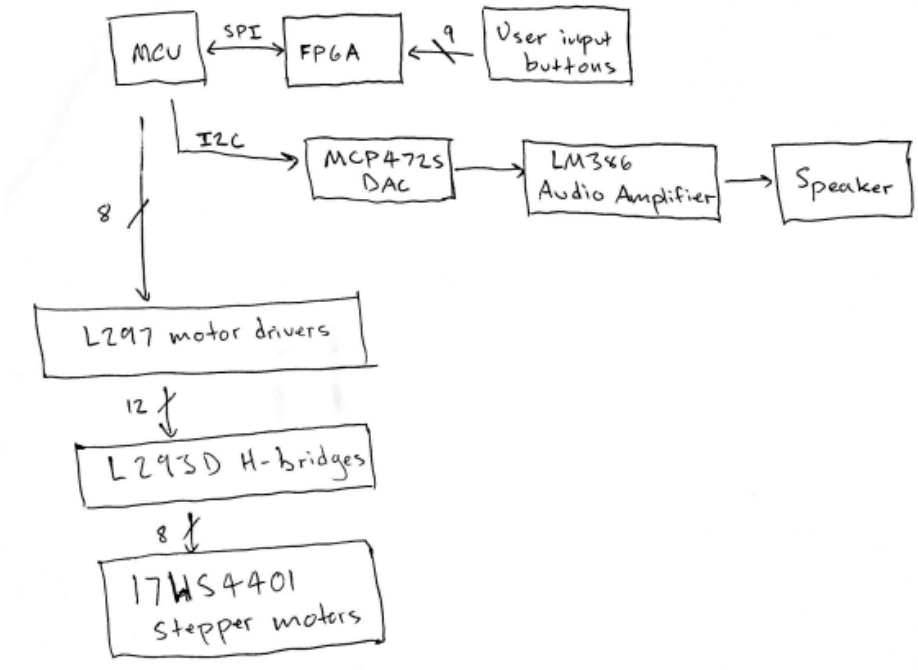

Figure 1. High level block diagram showing the flow of data through the system

User input	Desired function	Single byte encoding sent over SPI
Press left record switch, followed by a sequence of keys on the keypad	Record all presses of ‘1’, ‘2’, and ‘3’ keys on the keypad and the times between them, and save this data in the MCU in save slot 1	0x05
Press right record switch, followed by a sequence of keys on the keypad	Record all presses of ‘1’, ‘2’, and ‘3’ keys on the keypad and the times between them, and save this data in the MCU in save slot 2	0x06
Press key ‘4’	Play the sequence saved in save slot 1	0x07
Press key ‘5’	Play the sequence saved in save slot 2	0x08
Press key ‘7’	Repeatedly play the sequence saved in save slot 1, or stop playing it on repeat	0x09
Press key ‘8’	Repeatedly play the sequence saved in save slot 2, or stop playing it on repeat	0x0A
Press key ‘6’	Pause or resume stepper motor song	0x0D
Press key ‘9’	Play stepper motor song from beginning	0x04
Press key ‘*’	Mark spot in stepper motor song	0x0C
Press key ‘0’	Go back to marked spot in stepper motor song	0x0B

Figure 2. Table showing available user commands, what they do, and how they are internally encoded

Hardware Overview

New hardware

To play the sequences of beats on a speaker, we use a MCP4725 digital-to-analog converter (DAC) attached to a LM386 audio amplifier. The MCU sends the desired output voltage to the DAC over an I2C link. All devices on an I2C link use the same data wire (SDA) to communicate, with a clock wire (SCK) controlling when to read input from the data wire. In order to communicate with the DAC, the MCU first sends the DAC's address over SDA, waits for an ACK bit which indicates that the DAC has seen its address and is listening, and then sends another byte which tells the DAC what its output voltage should be. The DAC also ACKs the voltage byte.

This communication is relatively slow, at about 400 Kbps. Considering that our audio sampling rate is 8 KHz, the I2C's SDA is almost always active while the MCU is playing a beat.

The LM386 then amplifies the DAC's voltage signals into a large waveform which is played by the speaker.

To generate the sound for our song, we use stepper motors. These motors are composed of permanent magnets which are turned by electromagnets. Pulses are sent to the motor to turn the rotors by magnifying different stators across the sides. There are some teeth along the inside which the rotors click against. By controlling the frequency that the rotors turn, we can play different notes which is how we generate the music for our song. We use two stepper motors to have a bass line and a treble line, but they are easily interchangeable.

In order to get the motors to turn at the right frequency, we use L297 motor drivers and L293D H-bridges. The motor driver takes in a frequency generated by the MCU and maps it to the correct signals to send to the motor to turn the motor at that frequency. The H-bridge takes care of making sure that the right stators are powered as the driver dictates and makes sure they have enough power to turn. We attached a power supply to the circuit to power the H-bridges so that we wouldn't draw too much current from the MCU. Additionally, the motor drivers are used to set the motors in half step mode as opposed to full step mode.

Schematics

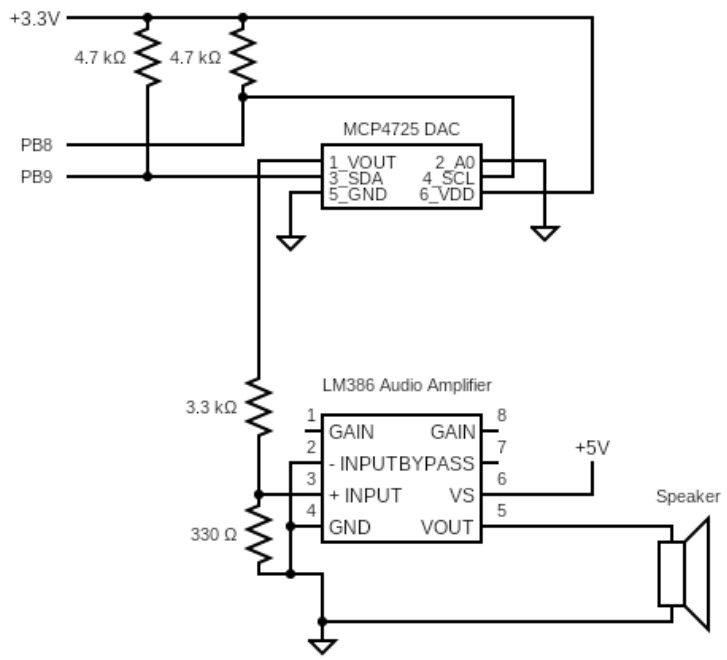

Figure 3. Speaker and DAC circuit diagram. The DAC takes input from the MCU over an I2C link and adjusts its voltage output appropriately. The audio amplifier amplifies the DAC's signal and sends it to the speaker.

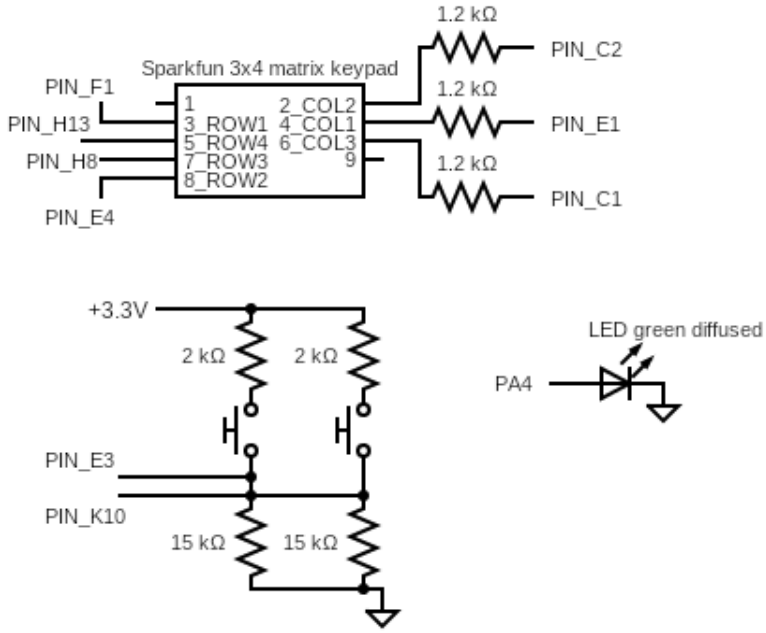

Figure 4. User interface circuit diagram. Input from the keypad and switches is sent to the FPGA for processing. The green diffused LED turns on when the MCU is finished with the initialization steps and the system is ready to use.

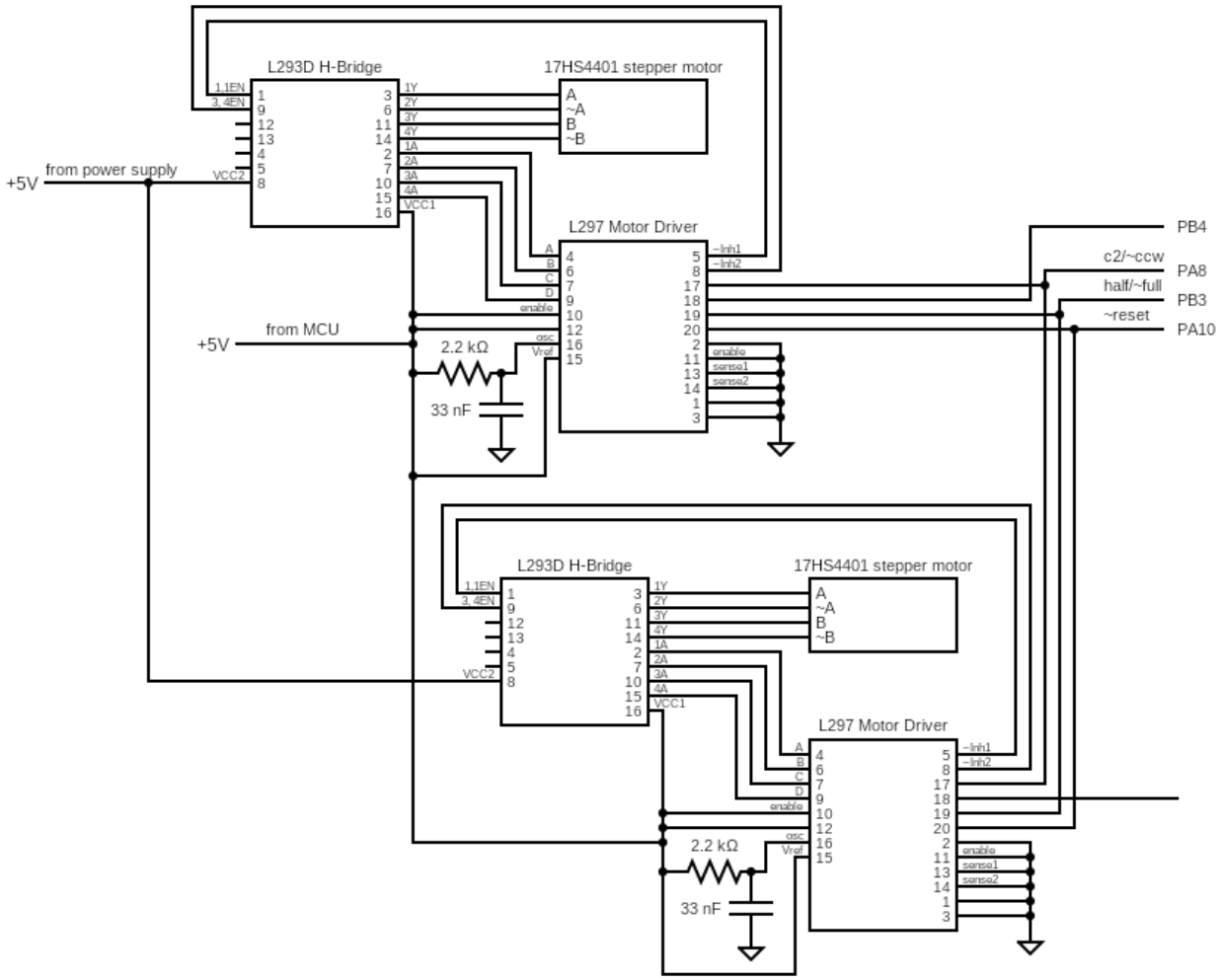

Figure 5. Stepper motor circuit diagram. The MCU controls the stepper motors by sending a clock signal to the motor drivers. These, along with the H-bridges, power the stators in the motors at the right frequency to play notes.

MCU Design

The MCU sends signals to the DAC to tell it to change its output voltage level in order to play the beats on the speaker. Because the MCU communicates with the DAC over a 400 Kbps I2C link, and our audio sampling rate is 8 Khz, when playing a beat, the MCU spends most of its time waiting for the ACK bit from the DAC telling the MCU that the DAC has received the command. Because the beat audio data is noise-tolerant (it does not matter if we send a frame a little bit late), we made this functionality our main loop.

All other functionality is implemented using interrupts.

The MCU takes care of initializing the stepper motors and changing their frequency. To do this, we use a timer (TIM2) that sends an interrupt to change the note values for the stepper motors

at 16th note intervals. 16th notes are commonly the subdivision of the beat of a song by four. This allows us to control the duration of the notes with precision. When the interrupt occurs, 2 separate timers (TIM3 and TIM4) output the correct frequency for the note they are supposed to play to the motor driver. They get the frequency from a list specific to each timer that has the frequencies for each 16th note in the song. The index of the lists is incremented every time the interrupt is sent by TIM2 to advance the lists through the song. TIM3 and TIM4 have their clocks prescaled to a smaller frequency because they don't have enough bits to count to to output the right frequencies otherwise.

The MCU also responds to user commands from the FPGA (see table 1.2 for the full list). The FPGA raises the voltage level of PA1 to tell the MCU that there is data to be read over SPI. When PA1's voltage goes high, it triggers an interrupt on the MCU. The MCU then reads a byte from the FPGA over SPI and decodes it, according to the list of functionality in Figure 2.

For the non-recording user commands, the MCU updates a set of global variables. To pause the song and play it again from the paused point, we simply turn off all the timer counters which stopped them at their current values. Turning the counters on again for each timer starts it right where it left off. To mark a point and go back to a marked point in a song, we kept track of how far TIM2 is in the process of counting a 16th note. If a user pauses somewhere in the middle, we want to make sure it starts off at the exact same point. To do this we store the timer value of TIM2 and the index of the song notes lists and return to those values when going back to that marked point.

For the recording commands, the MCU then reads in another 640 bytes from the FPGA over the SPI link, interpreting them as an array of [beat, time] data, where the beat is an eight bit encoding telling the MCU what beat to play, and the time is a 32-bit integer representing how long to wait after starting playing the beat until playing the next one. Because the time is in FPGA clock cycles (12 MHz), the MCU will convert the time to milliseconds before saving it in the data structure for the appropriate beat sequence.

FPGA Design

The FPGA design consists of four main parts: an SPI module, an input processor module, a memory module, and a commander module. See Figure 6 for the high level design.

The input processor module is responsible for processing user input into a reliable internal representation. It consists of four submodules. One debounces the user input, signals the commander when new input has been observed with the *newCommand* signal, and also signals the commander whether or not that input is one of the recording buttons (keys '1,' '2,' or '3') with the *recordCommand* signal. Another turns the matrix keypad inputs into a four bit internal representation. Two other modules debounce the pushbutton inputs, one for each pushbutton. Each debouncer module works the same way. After a change in input has been observed, they go to an intermediate state and wait to make sure that the change in input is genuine before transitioning to a different state in order to tell the rest of the system what the input is.

The SPI module shifts out the contents of the memory when it sees the clock line from the MCU (sck) enabled. To do this, it has a shift register which resets to the contents of the memory on the edge of sck when the data input line (sdi) goes high. This design is particularly useful because it allows for code reuse. In one case, the SPI must output the single byte command. In the other case, the SPI must output the single byte command and then keep outputting the rest of the memory. By resetting whenever sdi goes high, a single SPI module is able to handle both behaviors, without worrying about how many bytes have been read by the MCU or how many bytes there are left to be read. As a consequence, when reading the SPI's memory, the MCU must always output 0x10 on the sdi line when starting a new read, and the rest of the bytes it sends must be 0x00.

The memory module is 648 bits of registers. The first byte is the first byte to be sent to the FPGA. This command is the same as the internal representation of the keypad commands, as long as a record button has not been pressed. When a record button is pressed and *saveCommand* is asserted, the memory module will instead save a different byte, to tell the MCU that it needs to read the rest of the memory in order to get the rest of the new beat sequence. The next 640 bits are arranged as a sequence of [beat, time], where the beat is a single byte encoding -- either 0x01, 0x02, or 0x03 for a real beat, or 0x0F for an empty register -- and the time is a 32 bit integer, with the FPGA's clock cycles as a unit. Each [beat, time] pair is implemented as a pair of enabled registers. The current value of *regPointer*, updated by the commander, determines which register pair is enabled when *saveCommand* is asserted. Only one register pair is enabled at a time. Also, when the *clearCommand* input is inserted, the memory module will save 0x0F in the beat register instead of the last keypad button pressed. When the MCU reads in 0x0F, it will know that it is not a valid beat and it can stop playing the sequence. This allows the FPGA to store any number of beats, up to a maximum of 16.

The commander module is a finite state machine responsible both for updating the memory and for signalling the MCU to read the memory over SPI. See Figure 7 for a detailed diagram of the states, transitions, and outputs. If neither record button is pressed, then the commander simply detects when there is a new input from the keypad, asserts *saveCommand* to tell the memory module to save the command in the memory that will be sent out to SPI, and asserts the *ready* signal to tell the MCU that there is input to be read over SPI. It will deassert *ready* once the input has been read. If a record button is pressed, then the commander asserts *saveCommand* to tell the memory module to save the record command, then waits for *recordCommand* to be asserted, meaning that a key on the keypad has been pressed. It then transitions to another state where it increments a stopwatch to measure how much time passes between beats. When *recordCommand* is asserted again, then the commander increments *regPointer*, asserts *saveCommand* to tell the memory module to save the current beat and time, and transitions back to the stopwatch state. When a record button is no longer pressed, if 16 beats have been entered, then the commander either asserts the *ready* signal and waits for the SPI transition to be over. Otherwise, it transitions to another state, asserts *clearCommand,* and increments *regPointer* until the rest of the registers have been cleared by the memory module.

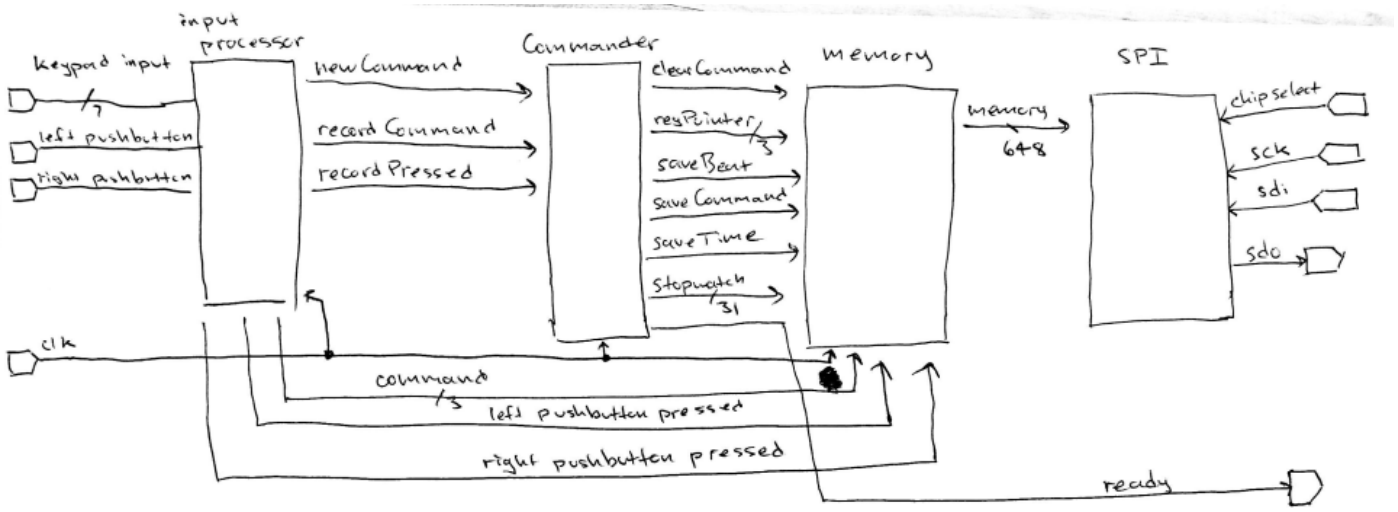

Figure 6. High level overview of FPGA modules and data flow.

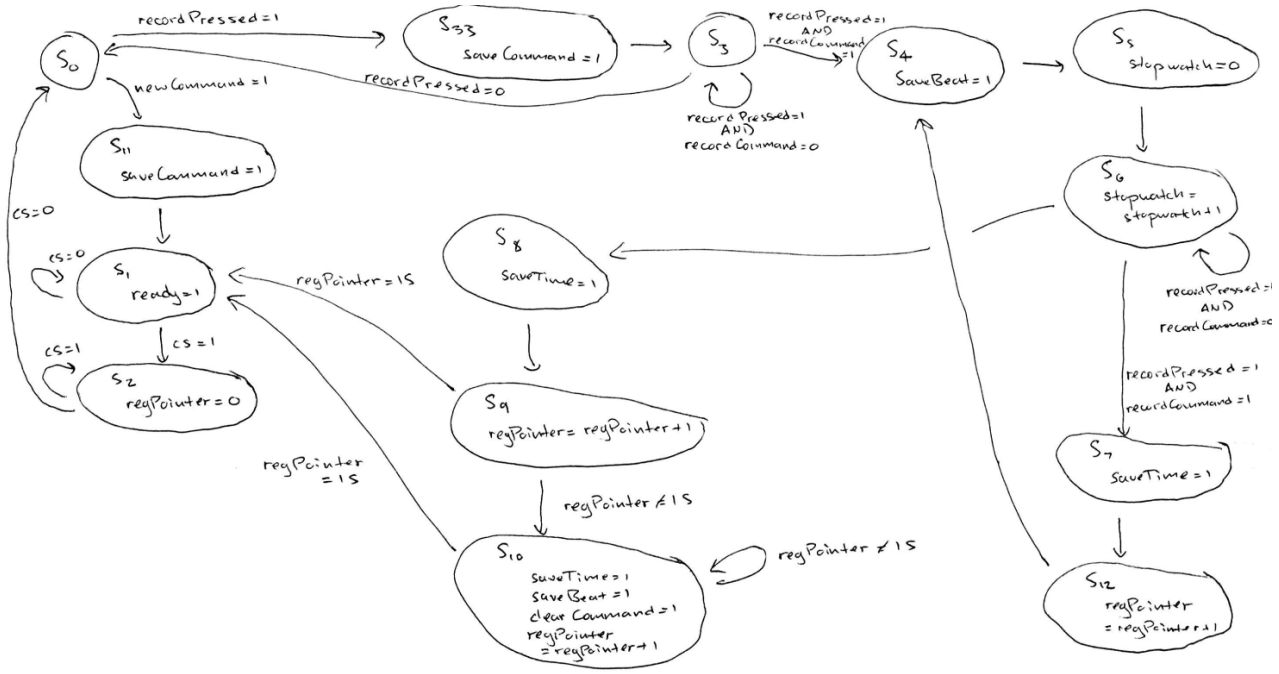

Figure 7. Commander finite state machine, with states, transitions, and outputs. Outputs are set to zero by default. For example, in the S0 state, saveCommand is equal to 0.

Results and Discussion

In the end, our system performed all of its tasks as listed in the revised project proposal. Our project was able to play a song of about 2 minutes and 15 seconds on stepper motors, start the song from the beginning at any time, pause/play the song, mark a point to jump back to, record sequences of beats to play on the speaker, play up to two saved sequences of beats on the speaker, and loop the sequences of beats on the speaker.

At the very end, we noticed some issues with the motors making a grinding sound when they weren't supposed to. We suspect that either the chips might have been heating up after being powered for too long, or the motors were drawing too much current from the power supply and needed to be regulated by lowering the voltage.

We also encountered some problems related to the wiring when assembling the final box, due to wires coming loose or coming in contact with each other. In the future, we would like to clean up the wiring so less wires cross each other, try to put all of the hardware on one breadboard, and clip the stripped sections of the wires so they are less likely to contact each other.

All in all, it is a fun system that plays a cool song and allows users to act like a DJ just as intended.

References

1. "12-Bit Digital-to-Analog Converter with EEPROM Memory in SOT-23-6 Datasheet." Microchip.
2. "2 Phase Hybrid Stepper Motor 17HS4401." MotionKing (China) Motor Industry Co.
3. "External Interrupt Using Registers " Controllerstech." *ControllersTech*, 15 July 2021, https://controllerstech.com/external-interrupt-using-registers/.
4. "L293x Quadruple Half-H Drivers Datasheet." Texas Instruments, Jan. 2016.
5. "The L297 Stepper Motor Controller User Guide." ST.
6. "L297 Stepper Motor Controllers Datasheet." ST.
7. SFUptownMaker. "I2C." *Sparkfun*, https://learn.sparkfun.com/tutorials/i2c/all.
8. "'Bare Metal' STM32 Programming (Part 4): Intro to Hardware Interrupts." *Vivonomicon's Blog*, https://vivonomicon.com/2018/04/28/bare-metal-stm32-programming-part-4-intro-to-hardware-interrupts/.

Bill of Materials

Item	Quantity	Manufacturer	Cost per unit	Total cost
MCP4725 DAC	1	Adafruit	$4.95	$4.95
MAX1000	1	Trenz Electronic	$26.66	$26.66

Nucleo-F401RE	1	STM	$13.83	$13.83
MAX1000 Shield PCB	1	OSH Park	$6.80	$6.80
Breadboard Cobbler PCB	1	OSH Park	$6.24	$6.24
Stackable headers kit	1	SparkFun Electronics	$1.50	$1.50
40-pin male breakable header	2	Sullins Connector Solutions	$0.51	$1.02
40-pin keyed ribbon cable	1	Assman WSW Components	$1.66	$1.66
40-pin male vertical keyed header	2	Sullins Connector Solutions	$0.73	$1.46
14-pin female headers	2	Sullins Connector Solutions	$0.91	$1.82
USB A to Mini B Cable White, 3 ft	1	Monoprice	$1.21	$1.21
USB A to Micro B Cable Black, 3 ft	1	Monoprice	$0.99	$0.99
L293D H-bridge	2	Adafruit	$4.50	$9.00
17HS4401 Stepper motor	2	Usongshine	$9.98	$19.96
L297 stepper motor driver	2	STMicroelectronics	$4.89	$9.78
3x4 matrix keypad	1	SparkFun Electronics	$4.50	$4.50
Pushbutton switch	2	Allied Electronics	$3.36	$6.72
LM386 audio amplifier	1	Texas Instruments	$1.50	$3.00
47000 ohm resistor	2	Jameco Electronics	$0.14	$0.28

3300 ohm resistor	1	JMar Vac Electronics	$1.79	$1.79
330 ohm resistor	1	Jameco Electronics	$0.14	$0.14
speaker	1	Philmore	$4.53	$4.53
1200 ohm resistor	3	Jameco Electronics	$0.06	$0.12
2000 ohm resistor	2	All Electronics	$0.06	$0.12
15000 ohm resistor	2	Jameco Electronics	$0.06	$0.12
Green diffused LED	1	Jameco Electronics	$0.06	$0.12
2200 ohm resistor	2	Jameco Electronics	$0.06	$0.12
33 nF capacitor	2	Mouser Electronics	$0.36	$0.72

Appendix A: C code

Main.c

```
// main.c
// Kariessa Schultz & Santiago Rodriguez
// kschultz@hmc.edu & sdrodriguez@hmc.edu
// 12/9/21

#include <stdint.h>
#include "audio_data.h"
#include "STM32F401RE_TIM2_5.h"
#include "STM32F401RE_TIM10_11.h"
#include "STM32F401RE_GPIO.h"
#include "STM32F401RE_RCC.h"
#include "STM32F401RE_FLASH.h"
#include "STM32F401RE_I2C.h"
#include "STM32F401RE_EXTI.h"
#include "STM32F401RE_SPI.h"
#include "main.h"
#include "STM32F401RE_SYSCFG.h"
#include "songs.h"

volatile uint8_t stepper_song_is_paused = 1; // song is paused by default

// set marked points to the beginning of the song by default
volatile uint32_t stepper_marked_index = 0;

// for pause points
volatile uint16_t stepper_song_index = 0;
volatile uint32_t stepper_tim_position = 0;  // value of the timer
register when paused

// do not start playing the sequences by default
volatile uint8_t play_seq_one_once = 0;
volatile uint8_t play_seq_two_once = 0;
volatile uint8_t repeat_seq_one = 0;
volatile uint8_t repeat_seq_two = 0;

// the dac's 7 bit address is 1100010, last bit is to stay as master
```

```
static uint8_t DAC_ADDRESS = 0b11000100;
static uint8_t MAX_NUMBER_OF_BEATS = 16;

// represents choice of audio array
enum BeatChoice {ZERO, ONE, TWO, NONE};

// structure to hold sequence data
typedef struct {
   uint16_t rest;         // in milliseconds; has enough digits to
represent up to just over a minute
   enum BeatChoice beat;
} SeqElement;

SeqElement seq_one[16];
SeqElement seq_two[16];

#define NULL 0

/*
 Configures PA1 (used for SPI) and TIM2 (used for the stepper motor song)
interrupts
*/
void configureInterrupts(void) {
   // Enable SYSCFG clock domain in RCC
   RCC->APB2ENR |= (1<<14);

   // enable interrupt from PA1
   SYSCFG->EXTICR1 &= ~(0xf<<4);
   // configure mask bit for PA1
   EXTI->IMR |= (1<<1);
   // enable rising edge trigger for PA1
   EXTI->RTSR |= (1<<1);
   // disable falling edge trigger for PA1
   EXTI->FTSR &= ~(1<<1);

   // turn on EXTI1 interrupt
   NVIC_SetPriority (EXTI1_IRQn, 0);
   NVIC_EnableIRQ (EXTI1_IRQn);

   // turn on TIM2 interrupt
```

```
   NVIC_SetPriority(TIM2_IRQn, 2);
   NVIC_EnableIRQ(TIM2_IRQn);
}

/*
 Read beats and times between them from the SPI link with the FPGA
*/
void readBeatsAndTimes(SeqElement sequence[]) {
   for (int i = 0; i < MAX_NUMBER_OF_BEATS; ++i){
       uint8_t beat = spiSendReceive(0x00);
       if (beat == 1) {
           sequence[i].beat = ZERO;
       } else if (beat == 2) {
           sequence[i].beat = ONE;
       } else if (beat == 3) {
           sequence[i].beat = TWO;
       } else {
           sequence[i].beat = NONE;
       }

       uint32_t time = spiSendReceive(0x00);
       time = time << 8;
       time = time + spiSendReceive(0x00);
       time = time << 8;
       time = time + spiSendReceive(0x00);
       time = time << 8;
       time = time + spiSendReceive(0x00);
       sequence[i].rest = (time) / (12000);  // convert 12 Mhz clock
cycles to milliseconds
   }
}

/*
 Interrupt handler for PA1

 When PA1 goes high, it means there is something in the FPGA to be read
out
 over SPI
*/
void EXTI1_IRQHandler(void)
```

```
{
   if (EXTI->PR & (1 << 1)) {
       // clear the interrupt
       EXTI->PR |= (1 << 1);

       // read the command from the FPGA and respond appropriately
       uint8_t command = spiSendReceive(0x80);
       if (command == 0x0D) {
           // pause or resume song
           if (!stepper_song_is_paused) {
               TIM2->CR1.CEN = 0;
               TIM3->CR1.CEN = 0;
               TIM4->CR1.CEN = 0;
           } else {
               TIM2->CR1.CEN = 1;
               TIM3->CR1.CEN = 1;
               TIM4->CR1.CEN = 1;
           }
           stepper_song_is_paused = !stepper_song_is_paused;
       } else if (command == 0x0C) {
           // mark spot in song
           stepper_tim_position = TIM2->CNT;
           stepper_marked_index = stepper_song_index;
       } else if (command == 0x0B) {
           // go back to marked spot in song
           TIM2->CR1.CEN = 0;
           TIM2->CNT = stepper_tim_position;
           TIM2->CR1.CEN = 1;
           changeMotorPWM(TIM3, song[stepper_marked_index]);
           changeMotorPWM(TIM4, song2[stepper_marked_index]);
           stepper_song_index = stepper_marked_index;
       } else if (command == 0x04) {
           // play song from the beginning (reset song pointer)
           TIM2->CR1.CEN = 0;
           TIM2->CNT = 0;
           TIM2->CR1.CEN = 1;
           changeMotorPWM(TIM3, song[0]);
           changeMotorPWM(TIM4, song2[0]);
           TIM3->CR1.CEN = 1;
           TIM4->CR1.CEN = 1;
```

```
            stepper_song_index = 0;
            stepper_song_is_paused = 0;
        } else if (command == 0x05) {
            // read in the rest of the transmission from SPI, then save it
as sequence 1
            readBeatsAndTimes(seq_one);
        } else if (command == 0x06) {
            // read in the rest of the transmission from SPI, then save it
as sequence 2
            readBeatsAndTimes(seq_two);
        } else if (command == 0x07) {
            // play sequence 1 once
            play_seq_one_once = 1;
        } else if (command == 0x08) {
            // play sequence 2 once
            play_seq_two_once = 1;
        } else if (command == 0x09) {
            // either play sequence 1 on repeat, or stop playing it on
repeat
            repeat_seq_one = !repeat_seq_one;
        } else if (command == 0x0A) {
            // either play sequence 2 on repeat, or stop playing it on
repeat
            repeat_seq_two = !repeat_seq_two;
        } else {
            // command is invalid, do nothing
        }
    }
}

/*
 Interrupt handler for TIM2

 When TIM2 goes off, it means that we have reached the end of a note in
 our stepper motor song

 Each part is written as a series of 16th notes, so we can use one timer
 to control the note timing for both parts
*/
void TIM2_IRQHandler(void) {
```

```
    if (TIM2->SR.CC1IF == 1) {
        TIM2->SR.CC1IF = 0;
    }
    if (!stepper_song_is_paused) {
        changeMotorPWM(TIM3, 2*song[stepper_song_index]);
        changeMotorPWM(TIM4, 2*song2[stepper_song_index]);
        stepper_song_index++;
        if (stepper_song_index > sizeof(song)/sizeof(song[0])) {
            // stop playing when we reach the end of the song
            stepper_song_index = 0;
            stepper_song_is_paused = 1;
            changeMotorPWM(TIM3, 0);
            changeMotorPWM(TIM4, 0);
        }
    }
 }

/*
 Overwrite the beat of each seqElement in the sequence with NONE,
 starting from and including index
*/
void resetRestOfSequence(SeqElement sequence[], uint8_t index) {
    for(int i = index; index < 16; ++index) {
        sequence[i].beat = NONE;
    }
}

/*
 Return the array of audio data with the number corresponding to choice

 Returns NULL if number is NONE
*/
uint8_t* selectAudioData(enum BeatChoice number, int* arraySize) {
    if (number == ZERO) {
        (*arraySize) = sizeof(notes1)/sizeof(notes1[0]);
        return notes1;
    } else if (number == ONE) {
        *arraySize = sizeof(notes2)/sizeof(notes2[0]);
        return notes2;
    } else if (number == TWO) {
```

```
        *arraySize = sizeof(notes3)/sizeof(notes3[0]);
        return notes3;
    } else {
        return NULL;  // nothing to return
    }
}

/*
 Does alll initialization necessary for the main loop
*/
void initializeAll(void) {
    // configure flash to run at a higher speed
    configureFlash();

    // configure clock to run at 84 MHz
    configureClock();

    // turn on clock to peripherals
    RCC->AHB1ENR.GPIOAEN = 1;   // GPIOA
    RCC->AHB1ENR.GPIOBEN = 1;   // GPIOB
    RCC->APB1ENR |= 1;          // TIM2
    RCC->APB1ENR |= (1 << 3);   // TIM5
    RCC->APB1ENR |= (1<<21);    // I2C
    RCC->APB2ENR |= (1<<17);    // TIM10
    RCC->APB2ENR |= (1<<14);    // SYSCF

    // enable all needed GPIO pins

    pinMode(GPIOA, 10, GPIO_OUTPUT); // ~reset
    pinMode(GPIOB, 3, GPIO_OUTPUT);  // half/~full
    pinMode(GPIOA, 8, GPIO_OUTPUT);  // CW/~CCW USE PA8 for GPIO instead,
USE PB4 for TIM3
    pinMode(GPIOB, 6, GPIO_ALT);     // TIM4
    GPIOB->AFRL |= (2 << 24);

    pinMode(GPIOA, 15, GPIO_ALT);    // set pin 15 as alternate function
    GPIOA->AFRH |= (1 << 28);        // set pin 15 alternate function as
timer 2 channel 1
    pinMode(GPIOB, 4, GPIO_ALT);
    GPIOB->AFRL |= (2 << 16);
```

```
pinMode(GPIOA, 3, GPIO_INPUT);   // ready pin
pinMode(GPIOA, 4, GPIO_OUTPUT);  // LED

// I2C pins
pinMode(GPIOB, 8, GPIO_ALT);    // SCL
pinMode(GPIOB, 9, GPIO_ALT);    // SDA
GPIOB->AFRH |= 0b0100;
GPIOB->AFRH |= 0b0100 << 4;
GPIOB->OSPEEDR |= (3 << 16) | (3 << 18);  // select high speed

// I2C specification requires pins to be open-drain
setPinToOpenDrain(GPIOB, 8);
setPinToOpenDrain(GPIOB, 9);

configureInterrupts();

// "clock divide" = master clock frequency / desired baud rate
// the phase for the SPI clock is 0 and the polarity is 0
spiInit(1, 0, 0);

// configure timers
configureTimer1011(TIM10);
configureLongARRTimer(TIM5);
configureMotorTimer(TIM3);
configureMotorTimer(TIM4);
configureDurationTimer(TIM2);

// note timers should be off by default
TIM3->CR1.CEN = 0;
TIM4->CR1.CEN = 0;

// configure I2C
configureI2C();

// send general call reset as the DAC's datasheet recommends
enable_I2C();
I2C_general_call_reset();
disable_I2C();
```

```
    // reset sequences to be empty
    resetRestOfSequence(seq_one, 0);
    resetRestOfSequence(seq_two, 0);

    // set motors to half turn mode
    digitalWrite(GPIOB, 3, 1);
    digitalWrite(GPIOB, 4, 1);

    // reset motor drivers
    digitalWrite(GPIOA, 10, 1);
    digitalWrite(GPIOA, 10, 0);
    digitalWrite(GPIOA, 10, 1);
}

/*
 Plays the given beat sequence by sending the frames from the audio data
 arrays to the DAC over I2C
*/
void playSequence(SeqElement sequence[]) {
    for (int i = 0; i < MAX_NUMBER_OF_BEATS; ++i) {
        if (sequence[i].beat != NONE) {
            // play the beat encoded in the ith element
            setTimerFromFreq1011(TIM10, 8000);
            setTimerFromTime(TIM5, sequence[i].rest);
            TIM5->SR.CC1IF = 0;

            int numFrames;
            uint8_t* frameArray = selectAudioData(sequence[i].beat,
&numFrames);
            int frameIndex = 0;

            enable_I2C();

            while (1) {
                // check if we're done
                if (frameIndex > numFrames - 1) {
                    break;
                }

                //check if we've reached the end of a frame
```

```
            if (TIM10->SR.CC1IF == 1) {
                TIM10->SR.CC1IF = 0;
                // assume notes are 8 bits
                // the DAC has 12 bit precision, so to take full
advantage of it
                // we multiply the note value by 16
                uint16_t frame = frameArray[frameIndex] << 4;
                I2C_fast_write_to_DAC(DAC_ADDRESS, frame);
                frameIndex += 1;
            }
        }
        disable_I2C();
        // wait for the rest between beats
        while (1) {
            if (TIM5->SR.CC1IF == 1) {
                TIM5->SR.CC1IF = 0;
                break;
            }
        }
    }
  }
}

int main(void) {
  initializeAll();

  digitalWrite(GPIOA, 4, 1);  // turn LED green

  while(1) {
      if (play_seq_one_once) {
          playSequence(seq_one);
          play_seq_one_once = 0;
      } else if (play_seq_two_once) {
          playSequence(seq_two);
          play_seq_two_once = 0;
      }
      while (repeat_seq_one) {
          playSequence(seq_one);
      }
      while (repeat_seq_two) {
```

```
            playSequence(seq_two);
        }
    }
}
```

Main.h

```
// main.h
// Josh Brake
// jbrake@hmc.edu
// 9/30/21

#ifndef MAIN_H
#define MAIN_H

////////////////////////////////////////////////////////////////////////////////
/////
// Custom defines
////////////////////////////////////////////////////////////////////////////////
/////

#define LED_PIN 5
#define BUTTON_PIN 13 // PC13
#define DELAY_TIM TIM2

#define NVIC_ISER0 ((uint32_t *) 0xE000E100UL)
#define NVIC_ISER1 ((uint32_t *) 0xE000E104UL)
#define SYSCFG_EXTICR4 ((uint32_t *) (0x40013800UL + 0x14UL))

////////////////////////////////////////////////////////////////////////////////
/////
// IRQn_Type and __NVIC_PRIO_BITS from stm32f401xe.h
////////////////////////////////////////////////////////////////////////////////
/////

/**
* @brief STM32F4XX Interrupt Number Definition, according to the selected
device
*        in @ref Library_configuration_section
*/
typedef enum
```

```
{
/******  Cortex-M4 Processor Exceptions Numbers
***********************************************************/
 NonMaskableInt_IRQn         = -14,    /*!< 2 Non Maskable Interrupt
*/
 MemoryManagement_IRQn       = -12,    /*!< 4 Cortex-M4 Memory Management
Interrupt                              */
 BusFault_IRQn               = -11,    /*!< 5 Cortex-M4 Bus Fault
Interrupt                                   */
 UsageFault_IRQn             = -10,    /*!< 6 Cortex-M4 Usage Fault
Interrupt                                  */
 SVCall_IRQn                 = -5,     /*!< 11 Cortex-M4 SV Call Interrupt
*/
 DebugMonitor_IRQn           = -4,     /*!< 12 Cortex-M4 Debug Monitor
Interrupt                                */
 PendSV_IRQn                 = -2,     /*!< 14 Cortex-M4 Pend SV Interrupt
*/
 SysTick_IRQn                = -1,     /*!< 15 Cortex-M4 System Tick
Interrupt                                 */
/******  STM32 specific Interrupt Numbers
***********************************************************************/
 WWDG_IRQn                   = 0,      /*!< Window WatchDog Interrupt
*/
 PVD_IRQn                    = 1,      /*!< PVD through EXTI Line
detection Interrupt                         */
 TAMP_STAMP_IRQn             = 2,      /*!< Tamper and TimeStamp
interrupts through the EXTI line             */
 RTC_WKUP_IRQn               = 3,      /*!< RTC Wakeup interrupt through
the EXTI line                         */
 FLASH_IRQn                  = 4,      /*!< FLASH global Interrupt
*/
 RCC_IRQn                    = 5,      /*!< RCC global Interrupt
*/
 EXTI0_IRQn                  = 6,      /*!< EXTI Line0 Interrupt
*/
 EXTI1_IRQn                  = 7,      /*!< EXTI Line1 Interrupt
*/
 EXTI2_IRQn                  = 8,      /*!< EXTI Line2 Interrupt
*/
```

```
 EXTI3_IRQn                  = 9,      /*!< EXTI Line3 Interrupt
*/
 EXTI4_IRQn                  = 10,     /*!< EXTI Line4 Interrupt
*/
 DMA1_Stream0_IRQn           = 11,     /*!< DMA1 Stream 0 global Interrupt
*/
 DMA1_Stream1_IRQn           = 12,     /*!< DMA1 Stream 1 global Interrupt
*/
 DMA1_Stream2_IRQn           = 13,     /*!< DMA1 Stream 2 global Interrupt
*/
 DMA1_Stream3_IRQn           = 14,     /*!< DMA1 Stream 3 global Interrupt
*/
 DMA1_Stream4_IRQn           = 15,     /*!< DMA1 Stream 4 global Interrupt
*/
 DMA1_Stream5_IRQn           = 16,     /*!< DMA1 Stream 5 global Interrupt
*/
 DMA1_Stream6_IRQn           = 17,     /*!< DMA1 Stream 6 global Interrupt
*/
 ADC_IRQn                    = 18,     /*!< ADC1, ADC2 and ADC3 global
Interrupts                            */
 EXTI9_5_IRQn                = 23,     /*!< External Line[9:5] Interrupts
*/
 TIM1_BRK_TIM9_IRQn          = 24,     /*!< TIM1 Break interrupt and TIM9
global interrupt                    */
 TIM1_UP_TIM10_IRQn          = 25,     /*!< TIM1 Update Interrupt and
TIM10 global interrupt                */
 TIM1_TRG_COM_TIM11_IRQn     = 26,     /*!< TIM1 Trigger and Commutation
Interrupt and TIM11 global interrupt */
 TIM1_CC_IRQn                = 27,     /*!< TIM1 Capture Compare Interrupt
*/
 TIM2_IRQn                   = 28,     /*!< TIM2 global Interrupt
*/
 TIM3_IRQn                   = 29,     /*!< TIM3 global Interrupt
*/
 TIM4_IRQn                   = 30,     /*!< TIM4 global Interrupt
*/
 I2C1_EV_IRQn                = 31,     /*!< I2C1 Event Interrupt
*/
 I2C1_ER_IRQn                = 32,     /*!< I2C1 Error Interrupt
*/
```

```
 I2C2_EV_IRQn                = 33,     /*!< I2C2 Event Interrupt
*/
 I2C2_ER_IRQn                = 34,     /*!< I2C2 Error Interrupt
*/
 SPI1_IRQn                   = 35,     /*!< SPI1 global Interrupt
*/
 SPI2_IRQn                   = 36,     /*!< SPI2 global Interrupt
*/
 USART1_IRQn                 = 37,     /*!< USART1 global Interrupt
*/
 USART2_IRQn                 = 38,     /*!< USART2 global Interrupt
*/
 EXTI15_10_IRQn              = 40,     /*!< External Line[15:10]
Interrupts                                */
 RTC_Alarm_IRQn              = 41,     /*!< RTC Alarm (A and B) through
EXTI Line Interrupt                     */
 OTG_FS_WKUP_IRQn            = 42,     /*!< USB OTG FS Wakeup through EXTI
line interrupt                       */
 DMA1_Stream7_IRQn           = 47,     /*!< DMA1 Stream7 Interrupt
*/
 SDIO_IRQn                   = 49,     /*!< SDIO global Interrupt
*/
 TIM5_IRQn                   = 50,     /*!< TIM5 global Interrupt
*/
 SPI3_IRQn                   = 51,     /*!< SPI3 global Interrupt
*/
 DMA2_Stream0_IRQn           = 56,     /*!< DMA2 Stream 0 global Interrupt
*/
 DMA2_Stream1_IRQn           = 57,     /*!< DMA2 Stream 1 global Interrupt
*/
 DMA2_Stream2_IRQn           = 58,     /*!< DMA2 Stream 2 global Interrupt
*/
 DMA2_Stream3_IRQn           = 59,     /*!< DMA2 Stream 3 global Interrupt
*/
 DMA2_Stream4_IRQn           = 60,     /*!< DMA2 Stream 4 global Interrupt
*/
 OTG_FS_IRQn                 = 67,     /*!< USB OTG FS global Interrupt
*/
 DMA2_Stream5_IRQn           = 68,     /*!< DMA2 Stream 5 global interrupt
*/
```

```
 DMA2_Stream6_IRQn           = 69,     /*!< DMA2 Stream 6 global interrupt
*/
 DMA2_Stream7_IRQn           = 70,     /*!< DMA2 Stream 7 global interrupt
*/
 USART6_IRQn                 = 71,     /*!< USART6 global interrupt
*/
 I2C3_EV_IRQn                = 72,     /*!< I2C3 event interrupt
*/
 I2C3_ER_IRQn                = 73,     /*!< I2C3 error interrupt
*/
 FPU_IRQn                    = 81,     /*!< FPU global interrupt
*/
 SPI4_IRQn                   = 84      /*!< SPI4 global Interrupt
*/
} IRQn_Type;

#define __NVIC_PRIO_BITS          4U       /*!< STM32F4XX uses 4 Bits for
the Priority Levels */

#include "cmsis_gcc.h"
#include "core_cm4.h"


#endif // MAIN_H
```

STM32F401RE_EXTI.h

```
#ifndef STM32F4_EXTI_H
#define STM32F4_EXTI_H

#include <stdint.h>
#define __IO volatile

// Base address of EXTI
#define EXTI_BASE (0x40013C00UL)

////////////////////////////////////////////////////////////////////////////
/////
// Bitfield struct for EXTI
////////////////////////////////////////////////////////////////////////////
/////
```

```
typedef struct {
   __IO uint32_t IMR       :32;
   __IO uint32_t EMR       :32;
   __IO uint32_t RTSR      :32;
   __IO uint32_t FTSR      :32;
   __IO uint32_t SWIER     :32;
   __IO uint32_t PR        :32;
} EXTI_TypeDef;

#define EXTI ((EXTI_TypeDef *) EXTI_BASE)

////////////////////////////////////////////////////////////////////////////
/////
// Function prototypes
////////////////////////////////////////////////////////////////////////////
/////

#endif
```

STM32F401RE_FLASH.h

```
// STM32F401RE_FLASH.h
// Header for FLASH functions

#ifndef STM32F4_FLASH_H
#define STM32F4_FLASH_H

#include <stdint.h>

////////////////////////////////////////////////////////////////////////////
/////
// Definitions
////////////////////////////////////////////////////////////////////////////
/////

#define __IO volatile

// Base addresses for GPIO ports
#define FLASH_BASE (0x40023C00UL) // base address of RCC
```

```
////////////////////////////////////////////////////////////////////////////
/////
// Bitfield struct for GPIO
////////////////////////////////////////////////////////////////////////////
/////

typedef struct {
 __IO uint32_t LATENCY :4;
 __IO uint32_t         :4;
 __IO uint32_t PRFTEN  :1;
 __IO uint32_t ICEN    :1;
 __IO uint32_t DCEN    :1;
 __IO uint32_t ICRST   :1;
 __IO uint32_t DCRST   :1;
 __IO uint32_t         :19;
} ACR_bits;

typedef struct {
 __IO ACR_bits ACR;      /*!< FLASH access control register,   Address
offset: 0x00 */
 __IO uint32_t KEYR;     /*!< FLASH key register,              Address
offset: 0x04 */
 __IO uint32_t OPTKEYR;  /*!< FLASH option key register,       Address
offset: 0x08 */
 __IO uint32_t SR;       /*!< FLASH status register,           Address
offset: 0x0C */
 __IO uint32_t CR;       /*!< FLASH control register,          Address
offset: 0x10 */
 __IO uint32_t OPTCR;    /*!< FLASH option control register ,  Address
offset: 0x14 */
 __IO uint32_t OPTCR1;   /*!< FLASH option control register 1, Address
offset: 0x18 */
} FLASH_TypeDef;

#define FLASH ((FLASH_TypeDef *) FLASH_BASE)

////////////////////////////////////////////////////////////////////////////
/////
// Function prototypes
```

```
////////////////////////////////////////////////////////////////////////////
/////

void configureFlash();

#endif
```

STM32F401RE_FLASH.c

```
// STM32F401RE_FLASH.c
// Source code for FLASH functions

#include "STM32F401RE_FLASH.h"

void configureFlash() {
   FLASH->ACR.LATENCY = 2; // Set to 0 waitstates
   FLASH->ACR.PRFTEN = 1; // Turn on the ART
}
```

STM32F401RE_GPIO.h

```
// STM32F401RE_GPIO.h
// Header for GPIO functions

#ifndef STM32F4_GPIO_H
#define STM32F4_GPIO_H

#include <stdint.h> // Include stdint header

////////////////////////////////////////////////////////////////////////////
/////
// Definitions
////////////////////////////////////////////////////////////////////////////
/////

// Values for GPIO pins ("val" arguments)
#define GPIO_LOW    0
#define GPIO_HIGH   1

// Base addresses for GPIO ports
#define GPIOA_BASE  (0x40020000UL)
#define GPIOB_BASE  (0x40020400UL)
```

```
#define GPIOC_BASE  (0x40020800UL)

// Arbitrary GPIO functions for pinMode()
#define GPIO_INPUT  0
#define GPIO_OUTPUT 1
#define GPIO_ALT    2
#define GPIO_ANALOG 3

////////////////////////////////////////////////////////////////////////////
/////
// Bitfield structs
////////////////////////////////////////////////////////////////////////////
/////

typedef struct {
   volatile uint32_t AFRL0  : 4;
   volatile uint32_t AFRL1  : 4;
   volatile uint32_t AFRL2  : 4;
   volatile uint32_t AFRL3  : 4;
   volatile uint32_t AFRL4  : 4;
   volatile uint32_t AFRL5  : 4;
   volatile uint32_t AFRL6  : 4;
   volatile uint32_t AFRL7  : 4;
} AFRL_bits;

typedef struct {
   volatile uint32_t AFRH8  : 4;
   volatile uint32_t AFRH9  : 4;
   volatile uint32_t AFRH10 : 4;
   volatile uint32_t AFRH11 : 4;
   volatile uint32_t AFRH12 : 4;
   volatile uint32_t AFRH13 : 4;
   volatile uint32_t AFRH14 : 4;
   volatile uint32_t AFRH15 : 4;
} AFRH_bits;

// GPIO register structs here
typedef struct {
   volatile uint32_t MODER;   // GPIO Offset 0x00 GPIO port mode register
   volatile uint32_t OTYPER;  // GPIO Offset 0x04
```

```
   volatile uint32_t OSPEEDR; // GPIO Offset 0x08
   volatile uint32_t PURPDR;  // GPIO Offset 0x0C
   volatile uint32_t IDR;     // GPIO Offset 0x10
   volatile uint32_t ODR;     // GPIO Offset 0x14
   volatile uint32_t BSRR;    // GPIO Offset 0x18
   volatile uint32_t LCKR;    // GPIO Offset 0x1C
   volatile uint32_t AFRL;    // GPIO Offset 0x20
   volatile uint32_t AFRH;    // GPIO Offset 0x24
} GPIO_TypeDef;

// Pointers to GPIO-sized chunks of memory for each peripheral
#define GPIOA ((GPIO_TypeDef *) GPIOA_BASE)
#define GPIOB ((GPIO_TypeDef *) GPIOB_BASE)
#define GPIOC ((GPIO_TypeDef *) GPIOC_BASE)

////////////////////////////////////////////////////////////////////////////
/////
// Function prototypes
////////////////////////////////////////////////////////////////////////////
/////

void pinMode(GPIO_TypeDef *, int pin, int function);

int digitalRead(GPIO_TypeDef *, int pin);

void digitalWrite(GPIO_TypeDef *, int pin, int val);

void setPinToOpenDrain(GPIO_TypeDef* GPIO_PORT_PTR, int pin);

void togglePin(GPIO_TypeDef *, int pin);

#endif
```

STM32F401RE_GPIO.C

```
// STM32F401RE_GPIO.c
// Source code for GPIO functions

#include "STM32F401RE_GPIO.h"

void pinMode(GPIO_TypeDef* GPIO_PORT_PTR, int pin, int function) {
```

```
    switch(function) {
        case GPIO_INPUT:
            GPIO_PORT_PTR->MODER &= ~(0b11 << 2*pin);
            break;
        case GPIO_OUTPUT:
            GPIO_PORT_PTR->MODER |= (0b1 << 2*pin);
            GPIO_PORT_PTR->MODER &= ~(0b1 << (2*pin+1));
            break;
        case GPIO_ALT:
            GPIO_PORT_PTR->MODER &= ~(0b1 << 2*pin);
            GPIO_PORT_PTR->MODER |= (0b1 << (2*pin+1));
            break;
        case GPIO_ANALOG:
            GPIO_PORT_PTR->MODER |= (0b11 << 2*pin);
            break;
    }
}

void setPinToOpenDrain(GPIO_TypeDef* GPIO_PORT_PTR, int pin) {
    GPIO_PORT_PTR->OTYPER |= (1 << pin);
}

int digitalRead(GPIO_TypeDef* GPIO_PORT_PTR, int pin) {
    return ((GPIO_PORT_PTR->IDR) >> pin) & 1;
}

void digitalWrite(GPIO_TypeDef* GPIO_PORT_PTR, int pin, int val) {
    if(val == 1) {
        GPIO_PORT_PTR->ODR |= (1 << pin);
    }
    else if(val == 0) {
        GPIO_PORT_PTR->ODR &= ~(1 << pin);
    }

}

void togglePin(GPIO_TypeDef* GPIO_PORT_PTR,int pin) {
    // Use XOR to toggle
    GPIO_PORT_PTR->ODR ^= (1 << pin);
}
```

STM32F401RE_I2C.h

```
// STM32F401RE_S2I.h
// Header for S2I functions

#ifndef STM32F4_I2C_H
#define STM32F4_I2C_H

#include <stdint.h>

////////////////////////////////////////////////////////////////////////////
/////
// Definitions
////////////////////////////////////////////////////////////////////////////
/////

#define I2C1_BASE (0x40005400UL)

////////////////////////////////////////////////////////////////////////////
/////
// Bitfield structs
////////////////////////////////////////////////////////////////////////////
/////

typedef struct {
 volatile uint32_t PE          : 1;
 volatile uint32_t SMBUS       : 1;
 volatile uint32_t             : 1;
 volatile uint32_t SMBTYPE     : 1;
 volatile uint32_t ENARP       : 1;
 volatile uint32_t ENPEC       : 1;
 volatile uint32_t ENGC        : 1;
 volatile uint32_t NO_STRETCH  : 1;
 volatile uint32_t START       : 1;
 volatile uint32_t STOP        : 1;
 volatile uint32_t ACK         : 1;
 volatile uint32_t POS         : 1;
 volatile uint32_t PEC         : 1;
 volatile uint32_t ALERT       : 1;
 volatile uint32_t             : 1;
```

```
 volatile uint32_t SWRST       : 1;
 volatile uint32_t             : 16;
} I2C_CR1_bits;

typedef struct {
   volatile uint32_t FREQ      : 6;
   volatile uint32_t           : 2;
   volatile uint32_t ITERREN   : 1;
   volatile uint32_t ITEVTEN   : 1;
   volatile uint32_t ITBUFEN   : 1;
   volatile uint32_t DMAEN     : 1;
   volatile uint32_t LAST      : 1;
   volatile uint32_t           : 3;
   volatile uint32_t           : 16;
} I2C_CR2_bits;

typedef struct {
   volatile uint32_t ADD0      : 1;
   volatile uint32_t ADD7      : 7;
   volatile uint32_t ADD9      : 2;
   volatile uint32_t reserved1 : 3;
   volatile uint32_t reserved2 : 1;
   volatile uint32_t ADDMODE   : 1;
   volatile uint32_t           : 16;
} I2C_OAR1_bits;

typedef struct {
   volatile uint32_t DR : 8;
   volatile uint32_t    : 16;
} I2C_DR_bits;

typedef struct {
   volatile uint32_t SB        : 1;
   volatile uint32_t ADDR      : 1;
   volatile uint32_t BTF       : 1;
   volatile uint32_t ADD10     : 1;
   volatile uint32_t STOPF     : 1;
   volatile uint32_t           : 1;
   volatile uint32_t RxNE      : 1;
   volatile uint32_t TxE       : 1;
```

```
   volatile uint32_t BERR      : 1;
   volatile uint32_t ARLO      : 1;
   volatile uint32_t AF        : 1;
   volatile uint32_t OVR       : 1;
   volatile uint32_t PEC_ERR   : 1;
   volatile uint32_t           : 1;
   volatile uint32_t TIME_OUT  : 1;
   volatile uint32_t SMB_ALERT : 1;
   volatile uint32_t           : 16;
} I2C_SR1_bits;

typedef struct {
   volatile uint32_t MSL        : 1;
   volatile uint32_t BUSY       : 1;
   volatile uint32_t TRA        : 1;
   volatile uint32_t            : 1;
   volatile uint32_t GENCALL    : 1;
   volatile uint32_t SMBDEFAULT : 1;
   volatile uint32_t SMBHOST    : 1;
   volatile uint32_t DUALF      : 1;
   volatile uint32_t PEC        : 8;
   volatile uint32_t            : 16;
} I2C_SR2_bits;

typedef struct {
   volatile uint32_t CCR        : 12;
   volatile uint32_t            : 2;
   volatile uint32_t DUTY       : 1;
   volatile uint32_t FS         : 1;
   volatile uint32_t            : 16;
} I2C_CCR_bits;

typedef struct {
 volatile I2C_CR1_bits   CR1;       /*!< I2C control register 1,
Address offset: 0x00 */
 volatile I2C_CR2_bits   CR2;       /*!< I2C control register 2,
Address offset: 0x04 */
 volatile I2C_OAR1_bits  OAR1;      /*!< I2C own address register 1,
Address offset: 0x08 */
```

```
 volatile uint32_t       OAR2;       /*!< I2C own address register 2,
Address offset: 0x0C */
 volatile I2C_DR_bits    DR;         /*!< I2C data register,
Address offset: 0x10 */
 volatile I2C_SR1_bits   SR1;        /*!< I2C status register 1,
Address offset: 0x14 */
 volatile I2C_SR2_bits   SR2;        /*!< I2C status register 2,
Address offset: 0x18 */
 volatile I2C_CCR_bits   CCR;        /*!< I2C clock control register,
Address offset: 0x1C */
 volatile uint32_t       TRISE;      /*!< I2C TRISE register,
Address offset: 0x20 */
 volatile uint32_t       FLTR;       /*!< I2C FLTR register,
Address offset: 0x24 */
} I2C_TypeDef;

#define I2C1 ((I2C_TypeDef *) I2C1_BASE)

////////////////////////////////////////////////////////////////////////////
/////
// Function prototypes
////////////////////////////////////////////////////////////////////////////
/////

void configureI2C(void);
void enable_I2C(void);
void disable_I2C(void);
void I2C_write_to_DAC(uint8_t address, uint16_t msg);
void I2C_fast_write_to_DAC(uint8_t address, uint16_t msg);
void I2C_general_call_reset(void);
void I2C_start_and_send_address(uint8_t address);

#endif
```

STM32F401RE_I2C.c

```
// STM32F401RE_GPIO.c
// Source code for I2C functions

#include "STM32F401RE_I2C.h"
#include "STM32F401RE_RCC.h"
```

```
// write to DAC register incoming; don't enter power down mode
uint8_t DAC_WRITE_COMMAND = 0b01000000;

void configureI2C(void) {
   // turn on clock to I2C
   RCC->APB1ENR |= (1<<21);

   // disable I2C peripheral during configuration
   I2C1->CR1.PE = 0;

   // set I2C mode
   I2C1->CR1.SMBUS = 0;

   // 7 bit addressing mode
   I2C1->OAR1.ADDMODE = 0;

   // this bit must be kept to 1 by software
   I2C1->OAR1.reserved2 = 1;

   // enable standard mode; want 400 KHz
   I2C1->CCR.FS = 1;
   I2C1->CCR.DUTY = 1;

   // speed of APB1
   I2C1->CR2.FREQ = 42;

   // TPCLK1 is about 24 nanoseconds: setting this gives us about 400 KHz
I2C speed
    I2C1->CCR.CCR = 4;

   // maximum duration of the SCL feedback loop
   // using Fm mode, with maximum allowed SCL rise time 300 ns
   // 300 / 24 = 12 then add 1
   I2C1->TRISE = 13;
}

void enable_I2C(void) {
   I2C1->CR1.PE = 1;
}
```

```
void disable_I2C(void) {
   I2C1->CR1.PE = 0;
}

void I2C_start_and_send_address(uint8_t address) {
   // send START byte
   I2C1->CR1.START = 1;

   // wait for START to be sent
   while(I2C1->SR1.SB != 1);

   // set last byte of address to 0 to indicate we are in master mode
   address |= 0;

   // send address
   I2C1->DR.DR = address;

   // wait for transmission to be successful
   while (I2C1->SR1.ADDR != 1);

   // wait to make sure we are in master mode
   // also clears ADDR bit, since we do that by reading SR1 and then SR2
   while (I2C1->SR2.MSL != 1);
}

void I2C_general_call_reset(void) {
   I2C_start_and_send_address(0x00);  // general call
   I2C1->DR.DR = 0x06;   // general call reset code

   // wait for transmission to be successful
   while (I2C1->SR1.TxE != 1);

   // send STOP byte
   I2C1->CR1.STOP = 1;
}

// must enable I2C before calling this function!
// sends first byte, then 2nd byte, starting from left hand side
void I2C_fast_write_to_DAC(uint8_t address, uint16_t msg) {
```

```
    I2C_start_and_send_address(address);

   // send first eight bits of the message
   I2C1->DR.DR = (uint8_t) (msg >> 8);

   // wait for transmission to be successful
   while (I2C1->SR1.TxE != 1);

   // send last eight bits of the message
   I2C1->DR.DR = (uint8_t) (msg);

   // wait for transmission to be successful
   while (I2C1->SR1.TxE != 1);

   // send STOP byte
   I2C1->CR1.STOP = 1;
}
```

STM32F401RE_RCC.h

```
// STM32F401RE_RCC.h
// Header for RCC functions

#ifndef STM32F4_RCC_H
#define STM32F4_RCC_H

#include <stdint.h>

////////////////////////////////////////////////////////////////////////////////
/////
// Definitions
////////////////////////////////////////////////////////////////////////////////
/////

#define __IO volatile

// Base addresses
#define RCC_BASE (0x40023800UL) // base address of RCC

// PLL
#define PLLSRC_HSI 0
```

```
#define PLLSRC_HSE 1

// Clock configuration
#define SW_HSI  0
#define SW_HSE  1
#define SW_PLL  2

// fastest clock speed
uint32_t SystemCoreClock;  // Updated by configureClock()
////////////////////////////////////////////////////////////////////////////
/////
// Bitfield structs
////////////////////////////////////////////////////////////////////////////
/////
typedef struct {
   volatile uint32_t HSION     : 1;
   volatile uint32_t HSIRDY    : 1;
   volatile uint32_t           : 1;
   volatile uint32_t HSITRIM   : 5;
   volatile uint32_t HSICAL    : 8;
   volatile uint32_t HSEON     : 1;
   volatile uint32_t HSERDY    : 1;
   volatile uint32_t HSEBYP    : 1;
   volatile uint32_t CSSON     : 1;
   volatile uint32_t           : 4;
   volatile uint32_t PLLON     : 1;
   volatile uint32_t PLLRDY    : 1;
   volatile uint32_t PLLI2SON  : 1;
   volatile uint32_t PLLI2SRDY : 1;
   volatile uint32_t           : 4;
} CR_bits;

typedef struct {
   volatile uint32_t PLLM      : 6;
   volatile uint32_t PLLN      : 9;
   volatile uint32_t           : 1;
   volatile uint32_t PLLP      : 2;
   volatile uint32_t           : 4;
   volatile uint32_t PLLSRC    : 1;
   volatile uint32_t           : 1;
```

```
   volatile uint32_t PLLQ      : 4;
   volatile uint32_t           : 4;
} PLLCFGR_bits;

typedef struct {
   volatile uint32_t SW        : 2;
   volatile uint32_t SWS       : 2;
   volatile uint32_t HPRE      : 4;
   volatile uint32_t           : 2;
   volatile uint32_t PPRE1     : 3;
   volatile uint32_t PPRE2     : 3;
   volatile uint32_t RTCPRE    : 5;
   volatile uint32_t MCO1      : 2;
   volatile uint32_t I2SSCR    : 1;
   volatile uint32_t MCO1PRE   : 3;
   volatile uint32_t MCO2PRE   : 3;
   volatile uint32_t MCO2      : 2;
} CFGR_bits;

typedef struct {
   volatile uint32_t GPIOAEN   : 1;
   volatile uint32_t GPIOBEN   : 1;
   volatile uint32_t GPIOCEN   : 1;
   volatile uint32_t GPIODEN   : 1;
   volatile uint32_t GPIOEEN   : 1;
   volatile uint32_t           : 2;
   volatile uint32_t GPIOHEN   : 1;
   volatile uint32_t           : 4;
   volatile uint32_t CRCEN     : 1;
   volatile uint32_t           : 3;
   volatile uint32_t           : 5;
   volatile uint32_t DMA1EN    : 1;
   volatile uint32_t DMA2EN    : 1;
   volatile uint32_t           : 9;
} AHB1ENR_bits;

typedef struct {
 __IO CR_bits      CR;           /*!< RCC clock control register,
Address offset: 0x00 */
```

```
 __IO PLLCFGR_bits PLLCFGR;       /*!< RCC PLL configuration register,
Address offset: 0x04 */
 __IO CFGR_bits    CFGR;          /*!< RCC clock configuration register,
Address offset: 0x08 */
 __IO uint32_t     CIR;           /*!< RCC clock interrupt register,
Address offset: 0x0C */
 __IO uint32_t     AHB1RSTR;      /*!< RCC AHB1 peripheral reset register,
Address offset: 0x10 */
 __IO uint32_t     AHB2RSTR;      /*!< RCC AHB2 peripheral reset register,
Address offset: 0x14 */
 __IO uint32_t     AHB3RSTR;      /*!< RCC AHB3 peripheral reset register,
Address offset: 0x18 */
 uint32_t          RESERVED0;     /*!< Reserved, 0x1C
*/
 __IO uint32_t     APB1RSTR;      /*!< RCC APB1 peripheral reset register,
Address offset: 0x20 */
 __IO uint32_t     APB2RSTR;      /*!< RCC APB2 peripheral reset register,
Address offset: 0x24 */
 uint32_t          RESERVED1[2];  /*!< Reserved, 0x28-0x2C
*/
 __IO AHB1ENR_bits AHB1ENR;       /*!< RCC AHB1 peripheral clock register,
Address offset: 0x30 */
 __IO uint32_t     AHB2ENR;       /*!< RCC AHB2 peripheral clock register,
Address offset: 0x34 */
 __IO uint32_t     AHB3ENR;       /*!< RCC AHB3 peripheral clock register,
Address offset: 0x38 */
 uint32_t          RESERVED2;     /*!< Reserved, 0x3C
*/
 __IO uint32_t     APB1ENR;       /*!< RCC APB1 peripheral clock enable
register,                   Address offset: 0x40 */
 __IO uint32_t     APB2ENR;       /*!< RCC APB2 peripheral clock enable
register,                   Address offset: 0x44 */
 uint32_t          RESERVED3[2];  /*!< Reserved, 0x48-0x4C
*/
 __IO uint32_t     AHB1LPENR;     /*!< RCC AHB1 peripheral clock enable in
low power mode register, Address offset: 0x50 */
 __IO uint32_t     AHB2LPENR;     /*!< RCC AHB2 peripheral clock enable in
low power mode register, Address offset: 0x54 */
 __IO uint32_t     AHB3LPENR;     /*!< RCC AHB3 peripheral clock enable in
low power mode register, Address offset: 0x58 */
```

```
 uint32_t          RESERVED4;     /*!< Reserved, 0x5C
*/
 __IO uint32_t     APB1LPENR;     /*!< RCC APB1 peripheral clock enable in
low power mode register, Address offset: 0x60 */
 __IO uint32_t     APB2LPENR;     /*!< RCC APB2 peripheral clock enable in
low power mode register, Address offset: 0x64 */
 uint32_t          RESERVED5[2];  /*!< Reserved, 0x68-0x6C
*/
 __IO uint32_t     BDCR;          /*!< RCC Backup domain control register,
Address offset: 0x70 */
 __IO uint32_t     CSR;           /*!< RCC clock control & status
register,                        Address offset: 0x74 */
 uint32_t          RESERVED6[2];  /*!< Reserved, 0x78-0x7C
*/
 __IO uint32_t     SSCGR;         /*!< RCC spread spectrum clock
generation register,              Address offset: 0x80 */
 __IO uint32_t     PLLI2SCFGR;    /*!< RCC PLLI2S configuration register,
Address offset: 0x84 */
 uint32_t          RESERVED7[1];  /*!< Reserved, 0x88
*/
 __IO uint32_t     DCKCFGR;       /*!< RCC Dedicated Clocks configuration
register,                 Address offset: 0x8C */
} RCC_TypeDef;

#define RCC ((RCC_TypeDef *) RCC_BASE)

////////////////////////////////////////////////////////////////////////////
/////
// Function prototypes
////////////////////////////////////////////////////////////////////////////
/////

void configurePLL();
void configureClock();

#endif
```

STM32F401RE_RCC.c

```
// STM32F401RE_RCC.c
// Source code for RCC functions
```

```c
#include "STM32F401RE_RCC.h"

void configurePLL() {
   // Set clock to 84 MHz
   // Output freq = (src_clk) * (N/M) / P
   // (8 MHz) * (336/16) / 4 = 42 MHz
   // M:16, N:336, P:4, Q:7
   // Use HSE as PLLSRC

   RCC->CR.PLLON = 0; // Turn off PLL
   while (RCC->CR.PLLRDY != 0); // Wait till PLL is unlocked (e.g., off)

   // Load configuration
   RCC->PLLCFGR.PLLSRC = PLLSRC_HSE;
   RCC->PLLCFGR.PLLM = 8;
   RCC->PLLCFGR.PLLN = 336;
   RCC->PLLCFGR.PLLP = 0b01;
   RCC->PLLCFGR.PLLQ = 4;

   // Enable PLL and wait until it's locked
   RCC->CR.PLLON = 1;
   while(RCC->CR.PLLRDY == 0);
}

void configureClock(){
   /* Configure APB prescalers
   1. Set APB2 (high-speed bus) prescaler to no division
   2. Set APB1 (low-speed bus) to divide by 2.
   */

   RCC->CFGR.PPRE2 = 0b000;
   RCC->CFGR.PPRE1 = 0b100;

   // Turn on and bypass for HSE from ST-LINK
   RCC->CR.HSEBYP = 1;
   RCC->CR.HSEON = 1;
   while(!RCC->CR.HSERDY);

   // Configure and turn on PLL for 84 MHz
```

```
    configurePLL();

    // Select PLL as clock source
    RCC->CFGR.SW = SW_PLL;
    while(RCC->CFGR.SWS != 0b10);

    SystemCoreClock = 84000000;
}
```

STM32F401RE_SPI.h

```
// STM32F401RE_SPI.h
// Header for SPI functions

#ifndef STM32F4_SPI_H
#define STM32F4_SPI_H

#include <stdint.h> // Include stdint header

//////////////////////////////////////////////////////////////////////////////
/////
// Definitions
//////////////////////////////////////////////////////////////////////////////
/////

#define SPI1_BASE (0x40013000UL)
#define __IO volatile

//////////////////////////////////////////////////////////////////////////////
/////
// Bitfield structs
//////////////////////////////////////////////////////////////////////////////
/////

typedef struct {
  __IO uint32_t CPHA        : 1;
  __IO uint32_t CPOL        : 1;
  __IO uint32_t MSTR        : 1;
  __IO uint32_t BR          : 3;
  __IO uint32_t SPE         : 1;
  __IO uint32_t LSBFIRST    : 1;
```

```
  __IO uint32_t SSI          : 1;
  __IO uint32_t SSM          : 1;
  __IO uint32_t RXONLY       : 1;
  __IO uint32_t DFF          : 1;
  __IO uint32_t CRCNEXT      : 1;
  __IO uint32_t CRCEN        : 1;
  __IO uint32_t BIDIOE       : 1;
  __IO uint32_t BIDIMODE     : 1;
  __IO uint32_t              : 16;
} SPI_CR1_bits;

typedef struct {
  __IO uint32_t RXDMAEN      : 1;
  __IO uint32_t TXDMAEN      : 1;
  __IO uint32_t SSOE         : 1;
  __IO uint32_t              : 1;
  __IO uint32_t FRF          : 1;
  __IO uint32_t ERRIE        : 1;
  __IO uint32_t RXNEIE       : 1;
  __IO uint32_t TXEIE        : 1;
  __IO uint32_t              : 24;
} SPI_CR2_bits;

typedef struct {
  __IO uint32_t RXNE         : 1;
  __IO uint32_t TXE          : 1;
  __IO uint32_t CHSIDE       : 1;
  __IO uint32_t UDR          : 1;
  __IO uint32_t CRCERR       : 1;
  __IO uint32_t MODF         : 1;
  __IO uint32_t OVR          : 1;
  __IO uint32_t BSY          : 1;
  __IO uint32_t FRE          : 1;
  __IO uint32_t DFF          : 1;
  __IO uint32_t CRCNEXT      : 1;
  __IO uint32_t CRCEN        : 1;
  __IO uint32_t BIDIOE       : 1;
  __IO uint32_t BIDIMODE     : 1;
  __IO uint32_t              : 16;
} SPI_SR_bits;
```

```
typedef struct {
 __IO uint32_t DR  : 16;
 __IO uint32_t     : 16;
} SPI_DR_bits;


typedef struct {
 __IO SPI_CR1_bits CR1;        /*!< SPI control register 1 (not used in
I2S mode),      Address offset: 0x00 */
 __IO SPI_CR2_bits CR2;        /*!< SPI control register 2,
Address offset: 0x04 */
 __IO SPI_SR_bits SR;         /*!< SPI status register,
Address offset: 0x08 */
 __IO SPI_DR_bits DR;         /*!< SPI data register,
Address offset: 0x0C */
 __IO uint32_t CRCPR;      /*!< SPI CRC polynomial register (not used in
I2S mode), Address offset: 0x10 */
 __IO uint32_t RXCRCR;     /*!< SPI RX CRC register (not used in I2S
mode),         Address offset: 0x14 */
 __IO uint32_t TXCRCR;     /*!< SPI TX CRC register (not used in I2S
mode),         Address offset: 0x18 */
 __IO uint32_t I2SCFGR;    /*!< SPI_I2S configuration register,
Address offset: 0x1C */
 __IO uint32_t I2SPR;      /*!< SPI_I2S prescaler register,
Address offset: 0x20 */
} SPI_TypeDef;

// Pointers to GPIO-sized chunks of memory for each peripheral
#define SPI1 ((SPI_TypeDef *) SPI1_BASE)

////////////////////////////////////////////////////////////////////////////
/////
// Function prototypes
////////////////////////////////////////////////////////////////////////////
/////

/* Enables the SPI peripheral and intializes its clock speed (baud rate),
polarity, and phase.
```

```
*    -- clkdivide: (0x01 to 0xFF). The SPI clk will be the master clock /
clkdivide.
*    -- cpol: clock polarity (0: inactive state is logical 0, 1: inactive
state is logical 1).
*    -- cpha: clock phase (1: data changed on leading edge of clk and
captured on next edge,
*          0: data captured on leading edge of clk and changed on next
edge)
* Note: the SPI mode register is set with the following unadjustable
settings:
*    -- Master mode
*    -- Fixed peripheral select
*    -- Chip select lines directly connected to peripheral device
*    -- Mode fault detection enabled
*    -- WDRBT disabled
*    -- LLB disabled
*    -- PCS = 0000 (Peripheral 0 selected), means NPCS[3:0] = 1110
* Refer to the datasheet for more low-level details. */
void spiInit(uint32_t clkdivide, uint32_t cpol, uint32_t ncpha);

/* Transmits a character (1 byte) over SPI and returns the received
character.
*    -- send: the character to send over SPI
*    -- return: the character received over SPI */
uint8_t spiSendReceive(uint8_t send);

/* Transmits a short (2 bytes) over SPI and returns the received short.
*    -- send: the short to send over SPI
*    -- return: the short received over SPI */
uint16_t spiSendReceive16(uint16_t send);

#endif
```

STM32F401RE_SPI.c

```
// STM32F401RE_SPI.c
// SPI function declarations

#include "STM32F401RE_SPI.h"
#include "STM32F401RE_RCC.h"
#include "STM32F401RE_GPIO.h"
```

```
/* Enables the SPI peripheral and intializes its clock speed (baud rate),
polarity, and phase.
*    -- br: (0b000 - 0b111). The SPI clk will be the master clock /
2^(BR+1).
*    -- cpol: clock polarity (0: inactive state is logical 0, 1: inactive
state is logical 1).
*    -- cpha: clock phase (0: data captured on leading edge of clk and
changed on next edge,
*          1: data changed on leading edge of clk and captured on next
edge)
* Refer to the datasheet for more low-level details. */

void spiInit(uint32_t br, uint32_t cpol, uint32_t cpha) {
   // Turn on GPIOA and GPIOB clock domains (GPIOAEN and GPIOBEN bits in
AHB1ENR)
   RCC->AHB1ENR.GPIOAEN = 1;
   RCC->AHB1ENR.GPIOBEN = 1;

   RCC->APB2ENR |= (1 << 12); // Turn on SPI1 clock domain (SPI1EN bit in
APB2ENR)

   // Initially assigning SPI pins
   pinMode(GPIOA, 5, GPIO_ALT); // PA5, Arduino D13, SPI1_SCK
   pinMode(GPIOA, 6, GPIO_ALT); // PA6, Arduino D12, SPI1_MISO
   pinMode(GPIOA, 7, GPIO_ALT); // PA7, Arduino D11, SPI1_MOSI
   pinMode(GPIOB, 5, GPIO_OUTPUT); // PB5, Arduino D10, Manual CS

   // Set output speed type to high for SCK
   GPIOA->OSPEEDR |= (0b11 << 2*5);

   // Set to AF05 for SPI alternate functions
   GPIOA->AFRL |= (1 << 18) | (1 << 16);
   GPIOA->AFRL |= (1 << 22) | (1 << 20);
   GPIOA->AFRL |= (1 << 26) | (1 << 24);
   GPIOA->AFRL |= (1 << 30) | (1 << 28);


   SPI1->CR1.BR = br;      // Set the clock divisor
   SPI1->CR1.CPOL = cpol;  // Set the polarity
```

```
   SPI1->CR1.CPHA = cpha;  // Set the phase
   SPI1->CR1.LSBFIRST = 0; // Set least significant bit first
   SPI1->CR1.DFF = 0;      // Set data format to 8 bits
   SPI1->CR1.SSM = 0;      // Turn off software slave management
   SPI1->CR2.SSOE = 1;     // Set the NSS pin to output mode
   SPI1->CR1.MSTR = 1;     // Put SPI in master mode
   SPI1->CR1.SPE = 1;      // Enable SPI
}

/* Transmits a character (1 byte) over SPI and returns the received
character.
*    -- send: the character to send over SPI
*    -- return: the character received over SPI */
uint8_t spiSendReceive(uint8_t send) {
   digitalWrite(GPIOB, 5, 1);
   SPI1->CR1.SPE = 1;

   while(!(SPI1->SR.TXE)); // Wait until the transmit buffer is empty
   SPI1->DR.DR = send; // Transmit the character over SPI
   while(!(SPI1->SR.RXNE)); // Wait until data has been received
   uint8_t rec = SPI1->DR.DR;
   SPI1->CR1.SPE = 0;
   digitalWrite(GPIOB, 5, 0);
   return rec; // Return received character
}

/* Transmits a short (2 bytes) over SPI and returns the received short.
*    -- send: the short to send over SPI
*    -- return: the short received over SPI */
uint16_t spiSendReceive16(uint16_t send) {
   digitalWrite(GPIOB, 6, 0);
   SPI1->CR1.SPE = 1;
   SPI1->DR.DR = send;

   while(!(SPI1->SR.RXNE));
   uint16_t rec = SPI1->DR.DR;

   SPI1->CR1.SPE = 0;
   digitalWrite(GPIOB, 6, 1);
```

```
    return rec;
}
```

STM32F401RE_SYSCFG.h

```
#ifndef STM32F4_SYSCFG_H
#define STM32F4_SYSCFG_H

#include <stdint.h>

#define __IO volatile

// Base address of EXTI
#define SYSCFG_BASE (0x40013800UL)

////////////////////////////////////////////////////////////////////////////
/////
// Bitfield struct for SYSCFG
////////////////////////////////////////////////////////////////////////////
/////

typedef struct {
    __IO uint32_t MEMRMP        :32;
    __IO uint32_t PMC           :32;
    __IO uint32_t EXTICR1       :32;
    __IO uint32_t EXTICR2       :32;
    __IO uint32_t EXTICR3       :32;
    __IO uint32_t EXTICR4       :32;
    __IO uint32_t CMPCR         :32;
} SYSCFG_TypeDef;

#define SYSCFG ((SYSCFG_TypeDef *) SYSCFG_BASE)

////////////////////////////////////////////////////////////////////////////
/////
// Function prototypes
////////////////////////////////////////////////////////////////////////////
/////

void configureFlash();
```

```
#endif
```

STM32F401RE_TIM2_5.h

```
// STM32F401RE_TIM.h
// Header for timer functions, for timers 2 through 5

#ifndef STM32F4_TIM_H
#define STM32F4_TIM_H

#include <stdint.h>

////////////////////////////////////////////////////////////////////////////////
/////
// Definitions
////////////////////////////////////////////////////////////////////////////////
/////

#define __IO volatile
#define TIM2_BASE (0x40000000UL)
#define TIM3_BASE (0x40000400UL)
#define TIM4_BASE (0x40000800UL)
#define TIM5_BASE (0x40000C00UL)

typedef struct {
 volatile uint32_t CEN       : 1;
 volatile uint32_t UDIS      : 1;
 volatile uint32_t URS       : 1;
 volatile uint32_t OPM       : 1;
 volatile uint32_t DIR       : 1;
 volatile uint32_t CMS       : 2;
 volatile uint32_t ARPE      : 1;
 volatile uint32_t CKD       : 2;
 volatile uint32_t           : 22;
} CR1_bits;

typedef struct {
 volatile uint32_t SMS       : 3;
 volatile uint32_t           : 1;
 volatile uint32_t TS        : 3;
 volatile uint32_t MSM       : 1;
```

```
 volatile uint32_t ETF       : 4;
 volatile uint32_t ETPS      : 2;
 volatile uint32_t ECE       : 1;
 volatile uint32_t ETP       : 1;
 volatile uint32_t           : 16;
} SMCR_bits;

typedef struct {
 volatile uint32_t CC1S      : 2;
 volatile uint32_t OC1FE     : 1;
 volatile uint32_t OC1PE     : 1;
 volatile uint32_t OC1M      : 3;
 volatile uint32_t OC1CE     : 1;
 volatile uint32_t CC2S      : 2;
 volatile uint32_t OC2FE     : 1;
 volatile uint32_t OC2PE     : 1;
 volatile uint32_t OC2M      : 3;
 volatile uint32_t OC2CE     : 1;
 volatile uint32_t           : 16;
} CCMR1_bits;

typedef struct {
 volatile uint32_t UG        : 1;
 volatile uint32_t CC1G      : 1;
 volatile uint32_t CC2G      : 1;
 volatile uint32_t CC3G      : 1;
 volatile uint32_t CC4G      : 1;
 volatile uint32_t           : 1;
 volatile uint32_t TG        : 1;
 volatile uint32_t           : 25;
} EGR_bits;

typedef struct {
 volatile uint32_t UIF       : 1;
 volatile uint32_t CC1IF     : 1;
 volatile uint32_t CC2IF     : 1;
 volatile uint32_t CC3IF     : 1;
 volatile uint32_t CC4IF     : 1;
 volatile uint32_t           : 1;
 volatile uint32_t TIF       : 1;
```

```
 volatile uint32_t           : 2;
 volatile uint32_t CC1OF     : 1;
 volatile uint32_t CC2OF     : 1;
 volatile uint32_t CC3OF     : 1;
 volatile uint32_t CC4OF     : 1;
 volatile uint32_t           : 19;
} SR_bits;

typedef struct {
 __IO CR1_bits      CR1;            /*!< TIMx control register 1,
Address offset: 0x00 */
 __IO uint32_t      CR2;            /*< TIMx control register 2,
Address offset: 0x04 */
 __IO SMCR_bits     SMCR;           /*< TIMx slave mode control register,
Address offset: 0x08 */
 __IO uint32_t      DIER;           /*< TIMx DMA/Interrupt enable
register,                          Address offset: 0x0C */
 __IO SR_bits       SR;             /*< TIMx status register,
Address offset: 0x10 */
 __IO EGR_bits      EGR;            /*< TIMx event generation register,
Address offset: 0x14 */
 __IO CCMR1_bits    CCMR1;          /*< TIMx capture/compare mode register
1,                      Address offset: 0x18 */
 __IO uint32_t      CCMR2;          /*< TIMx capture/compare mode register
2,                      Address offset: 0x1C */
 __IO uint32_t      CCER;           /*< TIMx capture/compare enable
register,                         Address offset: 0x20 */
 __IO uint32_t      CNT;            /*< TIMx counter,
Address offset: 0x24 */
 __IO uint32_t      PSC;            /*< TIMx prescaler,
Address offset: 0x28 */
 __IO uint32_t      ARR;            /*< TIMx auto-reload register,
Address offset: 0x2C */
 __IO uint32_t      reserved1;
 __IO uint32_t      CCR1;           /*< TIMx capture/compare register 1,
Address offset: 0x34 */
 __IO uint32_t      CCR2;           /*< TIMx capture/compare register 2,
Address offset: 0x38 */
 __IO uint32_t      CCR3;           /*< TIMx capture/compare register 3,
Address offset: 0x3C */
```

```
 __IO uint32_t      CCR4;           /*< TIMx capture/compare register 4,
Address offset: 0x40 */
 __IO uint32_t      reserved2;
 __IO uint32_t      DCR;            /*< TIMx DMA control register,
Address offset: 0x48 */
 __IO uint32_t      DMAR;           /*< TIMx DMA address fr full transfer,
Address offset: 0x4C */
 __IO uint32_t      OR;             /*< TIM2 or TIM5 option register,
Address offset: 0x40 */
} TIM_TypeDef;

#define TIM2 ((TIM_TypeDef *) TIM2_BASE)
#define TIM3 ((TIM_TypeDef *) TIM3_BASE)
#define TIM4 ((TIM_TypeDef *) TIM4_BASE)
#define TIM5 ((TIM_TypeDef *) TIM5_BASE)

// function prototypes

void configureTimer(TIM_TypeDef* TIM);
void configureShortARRTimer(TIM_TypeDef* TIM);
void configureLongARRTimer(TIM_TypeDef* TIM);
void setPWMTimer(TIM_TypeDef* TIM, float freq);
void configureDurationTimer(TIM_TypeDef* TIM);
void configureMotorTimer(TIM_TypeDef* TIM);
void changePWM(TIM_TypeDef* TIM, int frequency);
void changeMotorPWM(TIM_TypeDef* TIM, int frequency);
void setTimerFromTime(TIM_TypeDef* TIM, int time);

#endif
```

STM32F401RE_TIM2_5.c

```
// STM32F401RE_TIM2_5.c
// Function declarations for timers 2 through 5

#include "STM32F401RE_TIM2_5.h"
#include "STM32F401RE_RCC.h"
#include <stdio.h>
#include <math.h>

#define FAST 0
```

```
#define SLOW 1

void configureTimer(TIM_TypeDef* TIM) {
   // set counter clock to CK_INT
   // set SMS = 000 in the TIMx_SMCR register
   TIM->SMCR.SMS = 0b000;
   TIM->CR1.CEN = 1; // enable clock
   TIM->CCMR1.CC1S = 0b00;  // configure OC1 as output
   TIM->CCMR1.OC1M = 0b110; // select PWM mode
   TIM->CCMR1.OC1PE = 1;    // enable preload register
   TIM->CR1.ARPE = 1;       // enable auto-reload preload register
   TIM->CR1.DIR = 0;        // put timer in upcounting configuration
}

void configureDurationTimer(TIM_TypeDef* TIM) {
   RCC->APB1ENR |= 0b1111;     // enable APB1 peripheral clock (timer
location)
   RCC->CFGR.MCO1 = 3;         // Set output of MCO1 to be PLL
   TIM->CCR1 = 0x006ACFC0;
   TIM->ARR = 0x00D59F80;

   TIM->CCMR1.OC1PE = 1;      // enable preload register
   TIM->CR1.ARPE = 1;         // enable auto preload
   TIM->CCMR1.OC1M = 6;       // set capture/compare channel 1 to PWM mode
   TIM->CCER |= 1;            // enable capture/compare channel 1
   TIM->CR1.URS = 1;          // Only allow Timer overflow to send update
interrupts
   TIM->DIER |= (1 << 1);     // Allow capture compare channel 1 to give
interrupts
   TIM->EGR.UG = 1;           // set UG to update all registers
   TIM->CR1.CEN = 1;          // enable counter
   TIM->EGR.UG = 1;           // set UG to update all registers
}

void configureMotorTimer(TIM_TypeDef* TIM) {
   RCC->APB1ENR |= 0b1111;     // enable APB1 peripheral clock (timer
location)
   RCC->CFGR.MCO1 = 3;         // Set output of MCO1 to be PLL
   TIM->CCR1 = 2;
   TIM->ARR = 3;
```

```
    TIM->CCMR1.OC1PE = 1;       // enable preload register
    TIM->CR1.ARPE = 1;          // enable auto preload
    TIM->CCMR1.OC1M = 6;        // set capture/compare channel 1 to PWM mode
    TIM->CCER |= 1;             // enable capture/compare channel 1
    TIM->EGR.UG = 1;            // set UG to update all registers
    TIM->CR1.CEN = 1;           // enable counter
    TIM->EGR.UG = 1;            // set UG to update all registers
    TIM->PSC = 1282;
}

void setTimerFromTime(TIM_TypeDef* TIM, int time) {
    float slowTimerHz = (float) 1/( (float) 4 * (float) time / (float)
1000);
    setPWMTimer(TIM, slowTimerHz);
}

void changePWM(TIM_TypeDef* TIM, int frequency) {
    if (frequency == 0) {
        TIM->ARR = (int) (2);
        TIM->CCR1 = (int) (3);
    } else {
        TIM->ARR = (long) (SystemCoreClock/frequency);
        TIM->CCR1 = (long) (SystemCoreClock/(2*frequency));
    }
}

void changeMotorPWM(TIM_TypeDef* TIM, int frequency) {
    if (frequency == 0) {
        TIM->ARR = (int) (2);
        TIM->CCR1 = (int) (3);
    } else {
        TIM->ARR = (long) (SystemCoreClock/1282/frequency);
        TIM->CCR1 = (long) (SystemCoreClock/1282/(2*frequency));
    }
}

void configureLongARRTimer(TIM_TypeDef* TIM) {
    // configure a timer that has a 32 bit ARR register
```

```
    // in this case we can set the prescale factor to 0 for maximum
accuracy

    configureTimer(TIM);
    TIM->PSC = 0;
}

void setPWMTimer(TIM_TypeDef* TIM, float freq) {
    // based on an input frequency, sets the duty cycle and frequency in a
timer set up for PWM,
    // then resets it

    // calculate clock frequency from 84MHz external clock and prescale
factor
    // note we assume here that 84/TIM->PSC will be an integer, so we don't
have any rounding
    int clockFreq = (int) (84 * pow(10, 6))/(TIM->PSC + 1) ;

    // value we should count up to, in order to generate the right
frequency
    int valueToCountTo = (int) ((float)(clockFreq)) / (float) (freq * 2.0);

    // set TIMx_ARR register to value we should count up to, in order to
generate the right frequency
    TIM->ARR = valueToCountTo;

    // set duty cycle in TIMx_CCR1 register, by setting that value to half
of the TIMx_ARR
    // again, we are assuming that there is only one PWM timer, and that it
is set to use OC1
    TIM->CCR1 = valueToCountTo/2;

    // initialize all registers by setting the UG bit in the TIMx_EGR
register
    TIM->EGR.UG = 1;
}
```

STM32F401RE_TIM10_11.h

```
// STM32F401RE_TIM10_11.h
// Header for timer functions, for timers 10 through 11
```

```
#ifndef STM32F4_TIM10_11_H
#define STM32F4_TIM10_11_H

#include <stdint.h>

////////////////////////////////////////////////////////////////////////////
/////
// Definitions
////////////////////////////////////////////////////////////////////////////
/////

#define __IO volatile
#define TIM10_BASE (0x40014400UL)
#define TIM11_BASE (0x40014800UL)

typedef struct {
 volatile uint32_t CEN       : 1;
 volatile uint32_t UDIS      : 1;
 volatile uint32_t URS       : 1;
 volatile uint32_t OPM       : 1;
 volatile uint32_t           : 3;
 volatile uint32_t ARPE      : 1;
 volatile uint32_t CKD       : 2;
 volatile uint32_t           : 22;
} CR1;

typedef struct {
 volatile uint32_t CC1S      : 2;
 volatile uint32_t OC1FE     : 1;
 volatile uint32_t OC1PE     : 1;
 volatile uint32_t OC1M      : 3;
 volatile uint32_t OC1CE     : 1;
 volatile uint32_t CC2S      : 2;
 volatile uint32_t OC2FE     : 1;
 volatile uint32_t OC2PE     : 1;
 volatile uint32_t OC2M      : 3;
 volatile uint32_t OC2CE     : 1;
 volatile uint32_t           : 16;
} CCMR1;
```

```
typedef struct {
 volatile uint32_t UG         : 1;
 volatile uint32_t CC1G       : 1;
 volatile uint32_t            : 30;
} EGR;

typedef struct {
 volatile uint32_t UIF        : 1;
 volatile uint32_t CC1IF      : 1;
 volatile uint32_t            : 7;
 volatile uint32_t CC1OF      : 1;
 volatile uint32_t            : 22;
} SR;

typedef struct {
 __IO CR1      CR1;           /*!< TIMx control register 1,
Address offset: 0x00 */
 __IO uint32_t reserved0;
 __IO uint32_t      SMCR;     /*< TIMx slave mode control register,
Address offset: 0x08 */
 __IO uint32_t      DIER;     /*< TIMx DMA/Interrupt enable register,
Address offset: 0x0C */
 __IO SR       SR;            /*< TIMx status register,
Address offset: 0x10 */
 __IO EGR      EGR;           /*< TIMx event generation register,
Address offset: 0x14 */
 __IO CCMR1    CCMR1;         /*< TIMx capture/compare mode register 1,
Address offset: 0x18 */
 __IO uint32_t reserved00;
 __IO uint32_t      CCER;     /*< TIMx capture/compare enable register,
Address offset: 0x20 */
 __IO uint32_t      CNT;      /*< TIMx counter,
Address offset: 0x24 */
 __IO uint32_t      PSC;      /*< TIMx prescaler,
Address offset: 0x28 */
 __IO uint32_t      ARR;      /*< TIMx auto-reload register,
Address offset: 0x2C */
 __IO uint32_t      reserved1;
```

```
 __IO uint32_t      CCR1;      /*< TIMx capture/compare register 1,
Address offset: 0x34 */
 __IO uint32_t      reserved2;
 __IO uint32_t      reserved3;
 __IO uint32_t      reserved4;
 __IO uint32_t      reserved5;
 __IO uint32_t      reserved6;


 __IO uint32_t      OR;             /*< TIMx option register,
Address offset: 0x5C */
} TIM1011_TypeDef;

#define TIM10 ((TIM1011_TypeDef *) TIM10_BASE)
#define TIM11 ((TIM1011_TypeDef *) TIM11_BASE)

// function prototypes
void configureTimer1011(TIM1011_TypeDef* TIM);
void setTimerFromFreq1011(TIM1011_TypeDef* TIM, float freq);

#endif
```

STM32F401RE_TIM10_11.c

```
// STM32F401RE_TIM10_11.c
// Function declarations for timers 10 through 11

#include "STM32F401RE_TIM10_11.h"
#include <stdio.h>
#include <math.h>

void configureTimer1011(TIM1011_TypeDef* TIM) {
   // the clock will be sent to CK_INT by default

   // since we only have one PWM timer, we can assume that it just uses
OC1
   // configure OC1 as output
   TIM->CCMR1.CC1S = 0b00;
```

```
   // select PWM mode by writting 110 (PWM Mode 1) in the OC1M bits in the
TIMx_CCMRx register
   TIM->CCMR1.OC1M = 0b110;

   // enable the preload register by setting the OCxPE bit in the
TIMx_CCMRx register
   TIM->CCMR1.OC1PE = 1;

   // enable the auto-reload preload register by setting the ARPE bit in
the TIMx_CR1 register
   TIM->CR1.ARPE = 1;
}

void setTimerFromFreq1011(TIM1011_TypeDef* TIM, float freq) {
   // based on an input frequency, sets the duty cycle and frequency in a
timer set up for PWM,
   // then initializes it

   // calculate clock frequency from 84MHz external clock and prescale
factor
   // note we assume here that 84/TIM->PSC will be an integer, so we don't
have any rounding
   int clockFreq = (int) (84 * pow(10, 6))/(TIM->PSC + 1) ;

   // value we should count up to, in order to generate the right
frequency
   int valueToCountTo = (int) ((float)(clockFreq)) / (float) (freq * 2.0);

   // set TIMx_ARR register to value we should count up to, in order to
generate the right frequency
   TIM->ARR = valueToCountTo;

   // set duty cycle in TIMx_CCR1 register, by setting that value to half
of the TIMx_ARR
   TIM->CCR1 = valueToCountTo/2;

   // initialize all registers by setting the UG bit in the TIMx_EGR
register
   TIM->EGR.UG = 1;
```

```
    // set CEN bit in TIMx_CR1 to 1 to enable the clock
    TIM->CR1.CEN = 1;
}
```

Appendix B: Verilog code

```
/*
  System Verilog code for MC You

  Takes input from a 4x3 matric keypad and two pushbutton switches
  Interprets them as user commands, records and saves timing information,
  and sends it over an SPI link

  Author: Kariessa Schultz
  Email:  kschultz@g.hmc.edu
  Created 12/8/2021
*/

/*
Pin assignments
  PIN_H6: clk, 12MHz clock
  PIN_H4: sck, for SPI
  PIN_H1: MISO/sdo, for SPI
  PIN_J1: MOSI/sdi, for SPI
  PIN_J13: CS, for SPI
  PIN_H5: ready, signals MCU when there is data to be read from the FPGA
  PIN_K10: record_l, input from left record pushbutton
  PIN_E3:  record_r, input from right record pushbutton
  PIN_F1: row[0], controls 0th row of the keypad
  PIN_E4: row[1], controls 1st row of the keypad
  PIN_H8: row[2], controls 2nd row of the keypad
  PIN_H13: row[3], controls the 3rd row of the keypad
  PIN_E1: col[0], 0th column of the keypad
  PIN_C2: col[1], 1st column of the keypad
  PIN_C1: col[2], 2nd column of the keypad
*/

/*
  mcyou

  Top level module with SPI interface and input processor core

  Author: Kariessa Schultz
  Email:  kschultz@g.hmc.edu
  Created 12/4/2021
*/

module mcyou(
```

```
  input  logic clk,
  input  logic sck, sdi, cs,
  input logic recordl, recordr,
    input logic [2:0] col,
    output logic [3:0] row,
  output logic sdo,
  output logic ready,
    output logic [7:0] led
);
  logic [3:0] clean_input;
  logic [647: 0] memory = 0;
  spi spi(sck, sdi, sdo, cs, memory);
  processorCore core(clk, cs, row, col, clean_input, ready, memory, recordl, recordr, led);
endmodule

/*
  spi

  SPI interface. Shifts out contents of memory module.
  Assumes that the data on sdi will always take the form 0x8000...0

  Author: Kariessa Schultz
  Email:  kschultz@g.hmc.edu
  Created 12/6/2021
*/

module spi(
  input logic sck, sdi,
  output logic sdo,
  input logic cs,
  input [647:0] memory
);

  logic sdodelayed = 0;
  logic [647:0] memorycaptured = 0;   // shift register

  // shift the register one bit at a time on the positive edge of sck
  always_ff @(posedge sck) begin
          if (sdi == 1) memorycaptured = memory;                  // initialize shift register with the input
      else {memorycaptured} = {memorycaptured[646:0], sdi};
  end

  // sdo should change on the negative edge of sck
  always_ff @(negedge sck) begin
```

```
            sdodelayed = memorycaptured[646];
  end

  // when the MCU sends the first sdi signal, shift out msb before clock edge
  // by definition, the msb is always 0
  assign sdo = (sdi == 1) ? 0 : sdodelayed;
endmodule

/*
  processorCore

  High level module: wires together other modules for reading and storing user input

  Author: Kariessa Schultz
  Email:  kschultz@g.hmc.edu
  Created 12/8/2021
*/
module processorCore(
  input logic clk, cs,
  output logic [3:0] row,
  input logic [2:0] col,
  output logic [3:0] clean_key_input,
  output logic ready,
  output logic [647:0] memory,
  input logic recordl, recordr,
);

  logic [11:0] one_hot;
  logic [3:0] raw_key_input;
  logic en, newCommand, recordPressed, clean_record_r_input, clean_record_l_input;
  logic saveBeat, saveTime, saveCommand, clearCommand, recordCommand;
  logic [31:0] stopwatch;
  logic [3:0] regPointer;

  assign recordPressed = clean_record_r_input | clean_record_l_input;

  // read and debounce user input and save it in a register for later processing
  keypadScanner scanner(clk, col, row, one_hot);
  switchDebouncer keypadBouncer (clk, raw_key_input, en, newCommand, recordCommand);
  recordDebouncer recordBouncerr (clk, recordr, clean_record_r_input);
  recordDebouncer recordBouncerl (clk, recordl, clean_record_l_input);

  // translate input into a four bit internal representation
  decoder decoder(one_hot, raw_key_input);
```

```
  commandRegister commandr(clk, en, raw_key_input, clean_key_input);

  commander com(
        clk,
        newCommand,
        recordCommand,
        cs,
        recordPressed,
        ready,
        saveBeat,
        saveTime,
        saveCommand,
        clearCommand,
        stopwatch,
        regPointer
  );

  recordMemory mem(
        clk,
        saveBeat,
        saveTime,
        saveCommand,
        clearCommand,
        clean_key_input,
        clean_record_l_input,
        clean_record_r_input,
        stopwatch,
        regPointer,
        memory
  );
endmodule

/*
  keypadScanner

  This module reads the input from the 4x3 matrix keypad by sending voltage
  to each row in turn, and recording the result as a one-hot encoding.
  The one-hot encoding is twelve bits; each three bit segment represents the
  result of sending the voltage to a different row.

  Adapted from code written by Kariessa Schultz for a 4x4 matrix keypad
  in lab 4 (9/26/2021)
```

```
  Author: Kariessa Schultz
  Email:  kschultz@g.hmc.edu
  Created 11/28/2021
*/

module keypadScanner(
  input logic clk,
  input logic [2:0] col,
  output logic [3:0] row,
  output logic [11:0] one_hot
);

  logic [7:0] clk_counter = 8'h00;
  always_ff@(posedge clk) begin
          clk_counter = clk_counter > 8'hF0 ? 8'h00 : clk_counter + 1;
          if   ((clk_counter >= 0)
          && (clk_counter < 8'h3C)) begin        // row 0 (keys 1, 2, 3)
          row = 4'b0001;
          one_hot[11:9] = col;
          end else if   ((clk_counter >= 8'h3C)
                          && (clk_counter < 8'h7A)) begin  // row 1 (keys 4, 5, 6)
           row = 4'b0010;
           one_hot[8:6] = col;
          end else if   ((clk_counter >= 8'h7A)
                    && (clk_counter < 8'hB4)) begin  // row 2 (keys 7, 8, 9)
           row = 4'b0100;
           one_hot[5:3] = col;
          end else begin                                // row 3 (keys *, 0, #)
           row = 4'b1000;
           one_hot[2:0] = col;
          end
  end
endmodule

/*
    decoder

    This module decodes clean user input into the commands to be sent to the FPGA

    It assumes that user input is a one-hot encoding, so if multiple keys are pressed,
    it will only register the 'first' one as the input

    It is not responsible for handling the recording buttons, but the encoded commands from
    the beat keys can be sent to the MCU as they are, and will be interpreted as
```

```
   commands to play the correct beats

  Author: Kariessa Schultz
  Email:  kschultz@g.hmc.edu
  Created 12/7/2021
*/

module decoder(
  input logic [11:0] one_hot,
  output logic [3:0] command
);

  always_comb
          casez(one_hot)
          12'b1???????????: command = 4'h3;  // key 3: record beat 3
          12'b01??????????: command = 4'h2;  // key 2: record beat 2
          12'b001?????????: command = 4'h1;  // key 1: record beat 1
          12'b0001????????: command = 4'hD;  // key 6: pause or resume motor song
          12'b00001???????: command = 4'h8;  // key 5: play sequence 2 once
          12'b000001??????: command = 4'h7;  // key 4: play sequence 1 once
            12'b0000001?????: command = 4'h4;  // key 9: play motor song from beginning
          12'b00000001????: command = 4'hA;  // key 8: play sequence 2 on repeat
          12'b000000001???: command = 4'h9;  // key 7: play sequence 1 on repeat
          12'b0000000001??: command = 4'h0;  // key #: no meaning assigned...
          12'b00000000001?: command = 4'hB;  // key 0: go back to marked spot in motor song
          12'b000000000001: command = 4'hC;  // key *: mark spot in motor song
          12'b000000000000: command = 4'h0;  // no meaning assigned...
          default                           : command = 4'h0;  // no meaning assigned...
          endcase
endmodule

/*
  switchDebouncer

  This module deals with switch bounce for the keypad input.

  It waits between 10 and 20 ms after first detecting a change in input to
  do anything about it. The value is between 10 and 20 ms, because it waits
  for a clock counter to equal zero, and then for it to equal its max value,
  which takes at least 10 ms and up to 20 ms.

  After waiting between 10 and 20 ms, if the change in input persists,
  then the module saves the input in a register for other modules to handle
```

```
  It then asserts newCommand to tell the commander it saw new input,
  and asserts recordCommand if the key that was pressed was '1', '2', or '3'

  Once the user is no longer pressing a key, it resets in preparation
  to handle the next input

  Author: Kariessa Schultz
  Email:  kschultz@g.hmc.edu
  Created 12/7/2021
*/

module switchDebouncer(
  input logic clk,
  input logic [3:0] raw_input,
        output logic en, newCommand, recordCommand
);
  typedef enum logic [2:0] {S0, S1, S2, S3, S4, S5, S6, S7} statetype;

  statetype state = S0;
  statetype nextState = S0;
  logic [16:0] clk_counter = 0;

  always_ff@(posedge clk) begin
        state <= nextState;
        clk_counter <= clk_counter > 17'b11101010011000000 ? 0 : clk_counter + 1;
  end

  always_comb
        case(state)
        S0: if (raw_input != 0) nextState = S1;         // we saw something nonzero
        else nextState = S0;
        S1: if (clk_counter == 8'h00) nextState = S2;  // wait for the counter edge
        else nextState = S1;
        S2: if (clk_counter > 17'b11101010011000000)
        if (raw_input != 0) nextState = S3;     // definitely got some input
        else nextState = S0;                    // false alarm, no input
        else nextState = S2;                    // keep waiting
        S3: nextState = S7;                     // clobber register with new input
        S7: nextState = S4;                             // signal commander
        S4: if (raw_input == 0) nextState = S5;         // we saw something zero
        else nextState = S4;
        S5: if (clk_counter == 0) nextState = S6;       // wait for the counter edge
        else nextState = S5;
        S6: if (clk_counter > 17'b11101010011000000)
```

```
                        if (raw_input == 0) nextState = S0;    // no input
                        else nextState = S4;                   // still have input
                    else nextState = S6;                   // keep waiting
            default: nextState = S0;
            endcase

  // output logic
  assign en = (state == S3);
  assign newCommand = (state == S7);
  assign recordCommand = (state == S7) & ((raw_input == 1) | (raw_input == 2) | (raw_input ==
3));
endmodule

/*
  recordDebouncer

  This module deals with switch bounce for the record button inputs.

  Based on observation from the oscilloscope, these buttons are not
  as prone to experiencing switch bounce. Therefore, we can switch states
  more quickly.

  This module waits after detecting a change in input to do anything about
  it. After waiting, if the change in input persists, then the module
  assumes that the change in input is real, and heads to a different
  state to wait for the next change.

  Because the system is only interested in whether or not the record button
  is currently pressed, this module does not control any registers.

  Adapter from similar code written by Santiago Rodriguez for a 4x4 matrix
  keypad in lab 4 (9/26/2021)

  Author: Kariessa Schultz & Santiago Rodriguez
  Email:  kschultz@g.hmc.edu & sdrodriguez@g.hmc.edu
  Created 12/6/2021
*/

module recordDebouncer(
  input logic clk,
  input logic record_raw_input,
  output logic record_clean_input
);
```

```
  typedef enum logic [2:0] {S0, S1, S2, S3, S4, S5} statetype;

  statetype state = S0;
  statetype nextState = S0;
  logic [4:0] clk_counter = 0;

  always_ff@(posedge clk) begin
          if (clk_counter > 5'b11100) begin
          clk_counter <= 0;
           state <= nextState;
     end else clk_counter <= clk_counter + 1;
  end

  always_comb
          case(state)
          S0: nextState = record_raw_input ? S1 : S0;  // wait until we see something nonzero
           S1: nextState = S2;                        // wait for counter edge
           S2: nextState = record_raw_input ? S3 : S0;          // confirm we saw something
nonzero
           S3: nextState = record_raw_input ? S3 : S4;          // wait until we see something zero
           S4: nextState = S5;                        // wait for counter edge
           S5: nextState = record_raw_input ? S3 : S0;          // confirm we saw something zero
          default: nextState = S0;
          endcase

  // output logic
  // assume we have input once we pass the debouncing test,
  // then keep assuming we have input until proven otherwise
  // this prevents momentary errors from messing with the commander
  assign record_clean_input = ((state == S3) | (state == S4) | (state == S5));
endmodule

/*
  commander

  This module detects when there is a new command, processes it, and asserts
  the ready signal when there is something new for the MCU to read

  To record beats, the module stays in a recording state until the next button
  is pressed, incrementing a stopwatch counter.

  To save user input when recording, this module increments a pointer after
  each beat is pressed. The pointer's value is used by the recordMemory
  module to save user input in the appropriate registers. We need 16 total
```

```
  register pointer values, so we can use a four bit counter and let it overflow.

  For any recording states where the commander might stay there a long time,
  there is a check to make sure that a recording button is still pressed. This
  ensures that the stopwatch times are accurate

  cs is the chip select signal, which tells us when the command has been shifted
  out and we can stop asserting ready.

  Author: Kariessa Schultz
  Email:  kschultz@g.hmc.edu
  Created 12/7/2021
*/

module commander(
  input logic clk, newCommand, recordCommand, cs, recordPressed,
  output logic ready, saveBeat, saveTime, saveCommand, clearCommand,
  output logic [31:0] stopwatch,
  output logic [3:0] regPointer
);
  typedef enum logic [4:0] {S0, S1, S2, S3, S33, S4, S5, S6, S7, S8, S9, S10, S11, S12}
statetype;

  statetype state = S0;
  statetype nextState = S0;

  // go to the next state and, if needed, update stopwatch and register pointer
  always_ff@(posedge clk) begin
        state <= nextState;
    if (state == S5)
        stopwatch <= 0;
    else if (state == S6)
        stopwatch <= stopwatch + 1;
    if (state == S2)
        regPointer <= 0;
    else if ((state == S12) | (state == S9) | (state == S10))
        regPointer <= regPointer + 1;
  end

  always_comb
        case(state)
        // states for handling keypad user commands
        S0: if (recordPressed) nextState = S33;
        else nextState = newCommand ? S11 : S0;
```

```
            S11: nextState = S1;                                    // save command in memory
            S1: nextState = cs ? S2 : S1;  // wait for chip select to be asserted
            S2: nextState = cs ? S2 : S0;  // wait for chip select to be deasserted

            // states for handling recording
          S33: nextState = S3;            // save record command
            S3: if (recordPressed) nextState = recordCommand ? S4 : S3;
          else nextState = S0;
          S4: nextState = S5;             // save beat
          S5: nextState = S6;                                                  // initialize stopwatch
to 0
          S6: if (recordPressed)          // count time until next button pressed
          nextState = recordCommand ? S7 : S6;
          else nextState = S8;                              // done recording
          S7: nextState = S12;            // save stopwatch value
           S12: nextState = S4;           // increment register pointer
           S8: nextState = S9;                    // save current time
           S9: nextState = (regPointer == 15) ? S1 : S10;

          // recording state to clear the rest of the registers, if necessary
           S10: nextState = (regPointer == 15) ? S1 : S10;

          default: nextState = S0;
  endcase

  // output logic
  assign ready = (state == S1);
  assign saveBeat = ((state == S4) | (state == S10));
  assign saveTime = ((state == S7) | (state == S8) | (state == S10));
  assign saveCommand = ((state == S33) | (state == S11));
  assign clearCommand = (state == S10);
endmodule

/*
  commandRegister

  A 4 bit enabled register, for storing keypad input

  Author: Kariessa Schultz
  Email:  kschultz@g.hmc.edu
  Created 12/4/2021
*/

module commandRegister(
```

```
  input logic clk, en,
  input logic [3:0] d,
  output logic [3:0] q
);

  always_ff@(posedge clk) begin
          if (en) q <= d;
  end
endmodule

/*
  commandRegister

  A 32 bit enabled register, for storing stopwatch values

  Author: Kariessa Schultz
  Email:  kschultz@g.hmc.edu
  Created 12/6/2021
*/

module stopwatchRegister(
  input logic clk, en,
  input logic [31:0] d,
  output logic [31:0] q
);

  always_ff@(posedge clk) begin
          if (en) q <= d;
  end
endmodule

/*
  recordMemory

  This module is 648 bits of registers.

  The first eight bits contains the command to be sent to the FPGA, which
  may not be the same as the four bit internal representation of which key
  on the keypad has been pressed.

  The next 640 bits are grouped into enabled register pairs. Each pair consists
  of a commandRegister and a stopwatchRegister. The command register saves
  {4'b0000, beat}, where beat is four bits representing which beat has been pressed.
  If the register does not hold a value, then beat is set to 0xF. This allows us to
```

```
  record beat sequences of any length, up to the maximum of 16 beats. The
  stopwatch register is four bytes, and saves the time to rest after playing that
  beat, measured in FPGA clock cycles (12 MHz)

  The register pairs are enabled and disabled based on the value of regPointer,
  such that only one register pair is enabled at a time.

  Author: Kariessa Schultz
  Email:  kschultz@g.hmc.edu
  Created 12/6/2021
*/

module recordMemory(
  input logic clk, saveBeat, saveTime, saveCommand, clearCommand,
  input logic [3:0] command,
  input logic record_l_input, record_r_input,
  input logic [31:0] stopwatch,
  input logic [3:0] regPointer,
  output logic [647:0] memory
);

  logic [15:0] enBeat = 0;
  logic [15:0] enTime = 0;
  logic [3:0] commandToSave;

  assign commandToSave = clearCommand ? 4'hF : command;

  // write the command to the first register if the record button is
  // not pressed. Otherwise, use the command that means "record left"
  // or "record right" to the MCU
  always_ff @(posedge clk) begin
          if (saveCommand)
          if (record_r_input) memory[647:640] <= 8'h05;
          else begin
            if (record_l_input) memory[647:640] <= 8'h06;
            else memory[647:640] <= {4'b0000, command};
           end
  end

  assign enBeat = saveBeat ? (16'h0001 << regPointer) : 0;
  assign enTime = saveTime ? (16'h0001 << regPointer) : 0;

  oneBeatTime obt0(clk, enBeat[0], enTime[0], commandToSave, stopwatch, memory[635:632],
memory[631:600]);
```

```
  oneBeatTime obt1(clk, enBeat[1], enTime[1], commandToSave, stopwatch, memory[595:592],
memory[591: 560]);
  oneBeatTime obt2(clk, enBeat[2], enTime[2], commandToSave, stopwatch, memory[555:552],
memory[551: 520]);
  oneBeatTime obt3(clk, enBeat[3], enTime[3], commandToSave, stopwatch, memory[515:512],
memory[511: 480]);
  oneBeatTime obt4(clk, enBeat[4], enTime[4], commandToSave, stopwatch, memory[475:472],
memory[471: 440]);
  oneBeatTime obt5(clk, enBeat[5], enTime[5], commandToSave, stopwatch, memory[435:432],
memory[431: 400]);
  oneBeatTime obt6(clk, enBeat[6], enTime[6], commandToSave, stopwatch, memory[395:392],
memory[391: 360]);
  oneBeatTime obt7(clk, enBeat[7], enTime[7], commandToSave, stopwatch, memory[355:352],
memory[351: 320]);
  oneBeatTime obt8(clk, enBeat[8], enTime[8], commandToSave, stopwatch, memory[315:312],
memory[311: 280]);
  oneBeatTime obt9(clk, enBeat[9], enTime[9], commandToSave, stopwatch, memory[275:272],
memory[271: 240]);
  oneBeatTime obt10(clk, enBeat[10], enTime[10], commandToSave, stopwatch,
memory[235:232], memory[231: 200]);
  oneBeatTime obt11(clk, enBeat[11], enTime[11], commandToSave, stopwatch,
memory[195:192], memory[191: 160]);
  oneBeatTime obt12(clk, enBeat[12], enTime[12], commandToSave, stopwatch,
memory[155:152], memory[151: 120]);
  oneBeatTime obt13(clk, enBeat[13], enTime[13], commandToSave, stopwatch,
memory[115:112], memory[111: 80]);
  oneBeatTime obt14(clk, enBeat[14], enTime[14], commandToSave, stopwatch, memory[75:72],
memory[71: 40]);
  oneBeatTime obt15(clk, enBeat[15], enTime[15], commandToSave, stopwatch, memory[35:32],
memory[31: 0]);

endmodule

/*
  oneBeatTime

  This module is an enabled register pair saving a single beat and its
  associated time value.

  Author: Kariessa Schultz
  Email:  kschultz@g.hmc.edu
  Created 12/6/2021
*/
```

```
module oneBeatTime(
  input logic clk, enBeat, enTime,
  input logic [3:0] command_in,
  input logic [31:0] stopwatch_in,
  output logic [3:0] command_out,
  output logic [31:0] time_out
);
  commandRegister com(clk, enBeat, command_in, command_out);
  stopwatchRegister tim(clk, enTime, stopwatch_in, time_out);
endmodule
```

Appendix C: Python code

For convenience, we wrote a Python script which converts three inputted .wav files into a C header file, with audio data in the correct format for our program. This makes it easy to generate arrays of audio data for our speaker to play. See below.

wav_converter.py

```
# script for reading a wav file and printing it to a text file as an array of integers
# and constants, in a format consistent with C syntax

import argparse
import sys
import wave

FILE1 = "./wav_files_for_testing/taps.wav"
FILE2 = "./wav_files_for_testing/sine.wav"
FILE3 = "./wav_files_for_testing/woop.wav"
OUTPUT_FILE = "audio_data.h"

def print_to_file(audio_file, output_fp, file_number):
        fp1 = wave.open(audio_file, 'rb')
        n_frames = fp1.getnframes()
        waves = fp1.readframes(n_frames)   # bytes object
        sampwidth = fp1.getsampwidth()
        framerate = fp1.getframerate()
        fp1.close()

        output_fp.write("\n")
        output_fp.write(f"int sampwidth{file_number} = {sampwidth};\n")
        output_fp.write(f"int framerate{file_number} = {framerate};\n")

        if sampwidth == 1:
        output_fp.write(f"uint8_t notes{file_number}[] =")
        output_fp.write("{")
        else:
        print("Wave file must have a bit depth of 8 (1 byte). Exiting without writing audio data.")
        return

        for i in range(0, len(waves) - sampwidth, sampwidth):
        output_fp.write(f"{int.from_bytes(waves[i:i+sampwidth], sys.byteorder, signed=False)},")

        output_fp.write(f"{int.from_bytes(waves[-sampwidth:], sys.byteorder, signed=False)}")
        output_fp.write("};\n")
```

```
        output_fp.write("\n")

def main():
        with open(OUTPUT_FILE, 'w') as fp:
        fp.write("// C file generated by Python script to hold audio data\n\n")
        fp.write("#include <stdint.h>\n")
        fp.write("#ifndef AUDIO_DATA\n")
        fp.write("#define AUDIO_DATA\n")
        print_to_file(FILE1, fp, 1)
        print_to_file(FILE2, fp, 2)
        print_to_file(FILE3, fp, 3)

        fp.write("\n")
        fp.write("#endif\n")

if __name__ == "__main__":
        main()
```

Appendix D: Audio data

Here is the stepper song audio data. The speaker audio data is very long, so we have not included it here. We have included the Python script that we used to generate the header file above, so that way you may make your own arrays of audio data for the speaker if you wish.

```
// songs.h
// Motor song data for both parts
// Santiago Rodriguez
// sdrodriguez@hmc.edu
// 12/9/21

const int song[] = {0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
```

```
0,
0,
0,
0,
440,
440,
440,
440,
440,
440,
349.2,
349.2,
392,
392,
392,
392,
392,
392,
392,
392,
392,
392,
392,
392,
392,
329.6,
329.6,
349.2,
349.2,
349.2,
349.2,
392,
392,
440,
440,
349.2,
349.2,
349.2,
349.2,
```

```
349.2,
349.2,
293.7,
293.7,
329.6,
329.6,
392,
392,
349.2,
349.2,
329.6,
329.6,
440,
440,
349.2,
349.2,
329.6,
329.6,
293.7,
293.7,
329.6,
329.6,
329.6,
329.6,
277.2,
277.2,
277.2,
277.2,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
392,
392,
293.7,
293.7,
```

```
293.7,
293.7,
293.7,
293.7,
440,
440,
349.2,
349.2,
329.6,
329.6,
293.7,
293.7,
329.6,
329.6,
392,
392,
349.2,
349.2,
329.6,
329.6,
293.7,
293.7,
293.7,
293.7,
0,
0,
293.7,
293.7,
392,
392,
392,
392,
392,
349.2,
329.6,
329.6,
349.2,
349.2,
349.2,
349.2,
```

```
349.2,
349.2,
349.2,
349.2,
0,
0,
392,
392,
349.2,
349.2,
329.6,
329.6,
293.7,
293.7,
293.7,
293.7,
0,
0,
293.7,
293.7,
392,
392,
392,
392,
392,
349.2,
329.6,
329.6,
440,
440,
440,
440,
293.7,
293.7,
293.7,
293.7,
0,
0,
329.6,
329.6,
```

```
293.7,
293.7,
277.2,
277.2,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
349.2,
349.2,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
0,
0,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
440,
440,
293.7,
293.7,
392,
392,
329.6,
329.6,
440,
440,
440,
440,
```

```
440,
440,
349.2,
349.2,
392,
392,
392,
392,
392,
392,
392,
392,
392,
392,
392,
392,
392,
329.6,
329.6,
349.2,
349.2,
349.2,
349.2,
392,
392,
440,
440,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
293.7,
293.7,
329.6,
329.6,
392,
392,
```

```
349.2,
349.2,
329.6,
329.6,
440,
440,
349.2,
349.2,
329.6,
329.6,
293.7,
293.7,
329.6,
329.6,
329.6,
329.6,
329.6,
277.2,
277.2,
277.2,
277.2,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
392,
392,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
440,
440,
349.2,
349.2,
```

```
329.6,
329.6,
293.7,
293.7,
329.6,
329.6,
392,
392,
349.2,
349.2,
329.6,
329.6,
293.7,
293.7,
293.7,
293.7,
0,
0,
293.7,
293.7,
392,
392,
392,
392,
392,
349.2,
329.6,
329.6,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
0,
0,
392,
392,
```

```
349.2,
349.2,
329.6,
329.6,
293.7,
293.7,
293.7,
293.7,
0,
0,
293.7,
293.7,
392,
392,
392,
392,
392,
349.2,
329.6,
329.6,
440,
440,
440,
440,
293.7,
293.7,
293.7,
293.7,
0,
0,
329.6,
329.6,
293.7,
293.7,
277.2,
277.2,
293.7,
293.7,
293.7,
293.7,
```

```
293.7,
293.7,
293.7,
293.7,
349.2,
349.2,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
0,
0,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
440,
440,
293.7,
293.7,
392,
392,
329.6,
329.6,
0,
0,
293.7,
0,
293.7,
0,
293.7,
0,
0,
0,
311.1,
0,
```

```
311.1,
0,
311.1,
0,
0,
0,
293.7,
0,
293.7,
0,
440,
0,
0,
0,
138.6,
0,
138.6,
0,
138.6,
0,
0,
0,
293.7,
0,
293.7,
0,
293.7,
0,
0,
0,
392,
0,
392,
0,
392,
0,
0,
0,
392,
0,
```

```
392,
0,
392,
0,
349.2,
349.2,
349.2,
349.2,
329.6,
329.6,
329.6,
329.6,
0,
0,
293.7,
0,
293.7,
0,
293.7,
0,
0,
0,
311.1,
0,
311.1,
0,
311.1,
0,
0,
0,
293.7,
0,
293.7,
0,
440,
0,
0,
0,
138.6,
0,
```

```
138.6,
0,
138.6,
0,
0,
0,
293.7,
0,
293.7,
0,
293.7,
0,
0,
0,
392,
0,
392,
0,
392,
0,
0,
0,
392,
0,
392,
0,
392,
0,
349.2,
349.2,
349.2,
349.2,
329.6,
329.6,
329.6,
329.6,
293.7,
293.7,
293.7,
293.7,
```

```
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
440,
440,
440,
440,
440,
440,
440,
349.2,
349.2,
392,
392,
392,
392,
```

```
392,
392,
392,
392,
392,
392,
392,
392,
392,
392,
329.6,
329.6,
349.2,
349.2,
349.2,
349.2,
392,
392,
440,
440,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
293.7,
293.7,
329.6,
329.6,
392,
392,
349.2,
349.2,
329.6,
329.6,
440,
440,
349.2,
349.2,
```

```
329.6,
329.6,
293.7,
293.7,
329.6,
329.6,
329.6,
329.6,
277.2,
277.2,
277.2,
277.2,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
392,
392,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
440,
440,
349.2,
349.2,
329.6,
329.6,
293.7,
293.7,
329.6,
329.6,
392,
392,
```

```
349.2,
349.2,
329.6,
329.6,
293.7,
293.7,
293.7,
293.7,
0,
0,
293.7,
293.7,
392,
392,
392,
392,
392,
349.2,
329.6,
329.6,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
0,
0,
392,
392,
349.2,
349.2,
329.6,
329.6,
293.7,
293.7,
293.7,
293.7,
```

```
0,
0,
293.7,
293.7,
392,
392,
392,
392,
392,
349.2,
329.6,
329.6,
440,
440,
440,
440,
293.7,
293.7,
293.7,
293.7,
0,
0,
329.6,
329.6,
293.7,
293.7,
277.2,
277.2,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
392,
392,
392,
392,
```

```
392,
349.2,
329.6,
329.6,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
0,
0,
392,
392,
349.2,
349.2,
329.6,
329.6,
293.7,
293.7,
293.7,
293.7,
0,
0,
293.7,
293.7,
392,
392,
392,
392,
392,
349.2,
329.6,
329.6,
440,
440,
440,
440,
```

```
293.7,
293.7,
293.7,
293.7,
0,
0,
329.6,
329.6,
293.7,
293.7,
277.2,
277.2,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7};

const int song2[] = {293.7,
293.7,
293.7,
293.7,
0,
0,
0,
0,
146.85,
146.85,
146.85,
```

```
146.85,
0,
0,
0,
0,
196.0,
196.0,
196.0,
196.0,
0,
0,
0,
0,
220.0,
220.0,
220.0,
220.0,
0,
0,
220.0,
220.0,
293.7,
293.7,
293.7,
293.7,
0,
0,
0,
0,
146.85,
146.85,
146.85,
146.85,
0,
0,
0,
0,
196.0,
196.0,
196.0,
```

```
196.0,
0,
0,
0,
0,
220.0,
220.0,
220.0,
220.0,
0,
0,
220.0,
220.0,
146.85,
146.85,
146.85,
146.85,
0,
0,
0,
0,
164.8,
164.8,
164.8,
164.8,
0,
0,
0,
0,
174.6,
174.6,
174.6,
174.6,
0,
0,
0,
0,
196.0,
196.0,
196.0,
```

```
196.0,
0,
0,
0,
0,
293.7,
293.7,
293.7,
293.7,
0,
0,
0,
0,
146.85,
146.85,
146.85,
146.85,
0,
0,
0,
0,
146.85,
146.85,
146.85,
146.85,
0,
0,
0,
0,
220.0,
220.0,
196.0,
196.0,
174.6,
174.6,
164.8,
164.8,
349.2,
349.2,
349.2,
```

```
349.2,
349.2,
349.2,
349.2,
349.2,
233.1,
233.1,
233.1,
233.1,
233.1,
233.1,
233.1,
233.1,
293.7,
293.7,
293.7,
293.7,
349.2,
349.2,
349.2,
349.2,
329.6,
329.6,
329.6,
329.6,
329.6,
329.6,
329.6,
329.6,
349.2,
349.2,
349.2,
349.2,
440,
440,
440,
440,
293.7,
293.7,
293.7,
```

```
293.7,
293.7,
293.7,
293.7,
293.7,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
329.6,
329.6,
329.6,
329.6,
329.6,
329.6,
329.6,
329.6,
293.7,
293.7,
293.7,
293.7,
0,
0,
0,
0,
146.85,
146.85,
146.85,
146.85,
0,
0,
0,
0,
196.0,
196.0,
196.0,
```

```
196.0,
0,
0,
0,
0,
220.0,
220.0,
220.0,
220.0,
0,
0,
220.0,
220.0,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
146.85,
146.85,
146.85,
146.85,
0,
0,
0,
0,
196.0,
196.0,
196.0,
196.0,
0,
0,
0,
0,
220.0,
220.0,
220.0,
```

```
220.0,
0,
0,
220.0,
220.0,
146.85,
146.85,
146.85,
146.85,
0,
0,
0,
0,
164.8,
164.8,
164.8,
164.8,
0,
0,
0,
0,
174.6,
174.6,
174.6,
174.6,
0,
0,
0,
0,
196.0,
196.0,
196.0,
196.0,
0,
0,
0,
0,
146.85,
146.85,
146.85,
```

```
146.85,
0,
0,
0,
0,
146.85,
146.85,
146.85,
146.85,
0,
0,
0,
0,
146.85,
146.85,
146.85,
146.85,
0,
0,
0,
0,
220.0,
220.0,
196.0,
196.0,
174.6,
174.6,
164.8,
164.8,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
233.1,
233.1,
233.1,
```

```
233.1,
233.1,
233.1,
233.1,
233.1,
293.7,
293.7,
293.7,
293.7,
349.2,
349.2,
349.2,
349.2,
329.6,
329.6,
329.6,
329.6,
329.6,
329.6,
329.6,
329.6,
349.2,
349.2,
349.2,
349.2,
440,
440,
440,
440,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
349.2,
349.2,
349.2,
```

```
349.2,
349.2,
349.2,
349.2,
349.2,
329.6,
329.6,
329.6,
329.6,
329.6,
329.6,
329.6,
329.6,
293.7,
293.7,
293.7,
293.7,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
164.8,
164.8,
196.0,
```

```
196.0,
174.6,
174.6,
164.8,
164.8,
293.7,
293.7,
233.1,
0,
233.1,
0,
233.1,
0,
0,
0,
233.1,
0,
233.1,
0,
233.1,
0,
0,
0,
220.0,
0,
220.0,
0,
349.2,
0,
0,
0,
220.0,
0,
196.0,
0,
220.0,
0,
0,
0,
220.0,
```

```
0,
220.0,
0,
220.0,
0,
0,
0,
329.6,
0,
329.6,
0,
329.6,
0,
0,
0,
329.6,
0,
329.6,
0,
329.6,
0,
277.2,
277.2,
277.2,
277.2,
277.2,
277.2,
277.2,
277.2,
0,
0,
233.1,
0,
233.1,
0,
233.1,
0,
0,
0,
233.1,
```

```
0,
233.1,
0,
233.1,
0,
0,
0,
220.0,
0,
220.0,
0,
349.2,
0,
0,
0,
220.0,
0,
196.0,
0,
220.0,
0,
0,
0,
220.0,
0,
220.0,
0,
220.0,
0,
0,
0,
329.6,
0,
329.6,
0,
329.6,
0,
0,
0,
329.6,
```

```
0,
329.6,
0,
329.6,
0,
277.2,
277.2,
277.2,
277.2,
277.2,
277.2,
277.2,
277.2,
293.7,
293.7,
293.7,
293.7,
0,
0,
0,
0,
146.85,
146.85,
146.85,
146.85,
0,
0,
0,
0,
196.0,
196.0,
196.0,
196.0,
0,
0,
0,
0,
220.0,
220.0,
220.0,
```

```
220.0,
0,
0,
220.0,
220.0,
293.7,
293.7,
293.7,
293.7,
0,
0,
0,
0,
146.85,
146.85,
146.85,
146.85,
0,
0,
0,
0,
196.0,
196.0,
196.0,
196.0,
0,
0,
0,
0,
220.0,
220.0,
220.0,
220.0,
0,
0,
220.0,
220.0,
146.85,
146.85,
146.85,
```

```
146.85,
0,
0,
0,
0,
164.8,
164.8,
164.8,
164.8,
0,
0,
0,
0,
174.6,
174.6,
174.6,
174.6,
0,
0,
0,
0,
196.0,
196.0,
196.0,
196.0,
0,
0,
0,
0,
146.85,
146.85,
146.85,
146.85,
0,
0,
0,
0,
146.85,
146.85,
146.85,
```

```
146.85,
0,
0,
0,
0,
146.85,
146.85,
146.85,
146.85,
0,
0,
0,
0,
220.0,
220.0,
196.0,
196.0,
174.6,
174.6,
164.8,
164.8,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
233.1,
233.1,
233.1,
233.1,
233.1,
233.1,
233.1,
233.1,
293.7,
293.7,
293.7,
```

```
293.7,
349.2,
349.2,
349.2,
349.2,
329.6,
329.6,
329.6,
329.6,
329.6,
329.6,
329.6,
329.6,
349.2,
349.2,
349.2,
349.2,
440,
440,
440,
440,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
349.2,
349.2,
349.2,
349.2,
440,
440,
440,
440,
293.7,
293.7,
293.7,
```

```
293.7,
293.7,
293.7,
293.7,
293.7,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
233.1,
233.1,
233.1,
233.1,
233.1,
233.1,
233.1,
233.1,
293.7,
293.7,
293.7,
293.7,
349.2,
349.2,
349.2,
349.2,
329.6,
329.6,
329.6,
329.6,
329.6,
329.6,
329.6,
329.6,
349.2,
349.2,
349.2,
```

```
349.2,
440,
440,
440,
440,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
293.7,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
329.6,
329.6,
329.6,
329.6,
329.6,
329.6,
329.6,
329.6,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
349.2,
```

```
349.2,
349.2,
349.2,
349.2,
349.2};
```