Musical LED's

E155 Final Project

Ethan Falicov & Shriya Nadgauda

December 1, 2020

Abstract:

Our final project was inspired by LED's that respond to music playing in the environment. Therefore, for our final project, we programmed our microcontroller to control the lights on an LED strip based on pressure readings from a microphone. SPI and I2S were used to interface with the LEDs and microphone respectively. The color of each LED represents the volume at a time stamp. The entire strip then displays volume over time. We attempted to implement a frequency mode, where the color of each LED represents the magnitude of that frequency. The Cooley-Tukey algorithm was used to perform FFT analysis. We were able to successfully implement a volume waveform. However, we did have some issues getting consistently plausible microphone readings, therefore, we were unable to successfully implement frequency analysis.

Introduction

For our final project, we built a music controlled LED strip controlled by the STM32F401RE microcontroller. The microcontroller listened to the music using an external microphone and I2S protocol. The color and pattern of the lights on the LED strip then changed based on the music and the LED strip was controlled using a SPI protocol. A block diagram of our project can be seen in FIgure 1. The color of the LEDs was used to display a volume waveform. Our microcontroller operated at 84MHz, SPI at 28MHz and I2S at 3MHz. Because the microcontroller operated at a much higher frequency then both the SPI and I2S we had plenty of time for analysis and conversion of pressure readings into LED values.

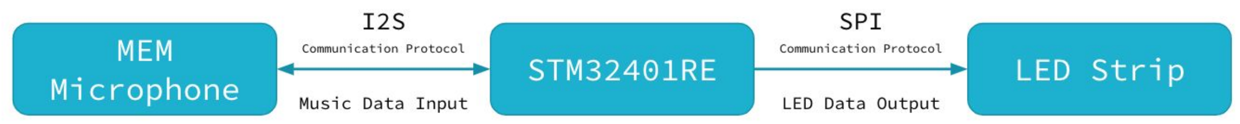

Figure 1. Block Diagram

Our microphone often returned invalid pressure readings. Therefore, when displaying a volume waveform, we took the maximum pressure difference across valid 64 samples as the volume for that timestamp in order to ensure an accurate value. The pressure was then converted into a color output on the strip using a HSL algorithm. This algorithm uses a specified hue, saturation and lightness value to determine the red, green and blue components of the color. The color of the LED's went from red, representing loud sounds, to blue, representing quieter sounds. The color values for the LED's were stored in an array that was updated each loop. We considered switching to a ring-buffer, however there was no noticeable delay caused by updating the array.

During FFT mode, we could not use the same method of obtaining microphone readings as we needed a consistent sample rate. We tried multiple methods of obtaining valid samples at a consistent sampling rate using timers and delays. However, there was too much noise in the microphone data and we were unable to get meaningful data from it. 128 samples were taken for each round of FFT analysis. The Cooley-Tukey algorithm was then applied to these samples to determine the frequency components of the music. Due to feedback from our classmates, we knew that this algorithm was the fastest DTFT algorithm. Our FFT returned the magnitude of the frequency component for 2^N bins. We wanted to have N = 7 for 128 frequency bins. Each bin would then be represented by an LED on the strip. Based on the magnitude of the frequency component, the HSL algorithm was again used to convert the magnitude into a color. Low magnitudes would be represented by blues and higher magnitudes by reds.

New Hardware

There were three new pieces of hardware we used in this project: the LED strip, microphone and a power supply. The power supply was necessary to provide the LED's with enough current (10A) as the microcontroller would not be able to do so. The LED strip contained 144 LED's. Figure 2 shows the LED dataframe that would need to be transmitted over SPI in order to correctly interface with the LED strip.

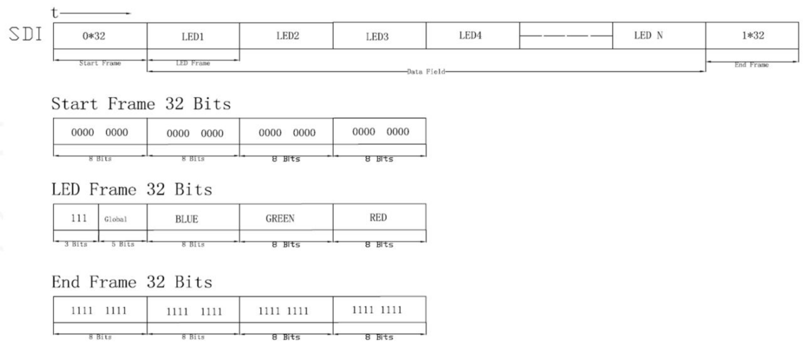

Figure 2. Instruction format LED Strip

The LED instruction sequence begins with a start frame of 32 0's. This is then followed by 144 frames to control the color of each LED in the strip. Each frame is 32-bits made up of 3 bits of 1's followed by 5 global bits to control current through the LED's (therefore brightness) and then 3 sets of 8 bits to control blue, green and red components of the color. These 24 bits were essentially like specifying an RGB hex code for color selection. The instruction sequence ends with a 32-bit frame of 1's.

The LED's connected to the microcontroller as shown in Figure 3. We did not have to connect VDD as the strip drew power separately from the power supply. The LED's took a clock input that operated at up to 30MHz as well as data in as described above. CKI was connected to the SPI SCK, and SDI was connected to SPI MOSI. The strip was also capable of being connected to another strip and therefore had clock out and data out pins that we did not use at the end.

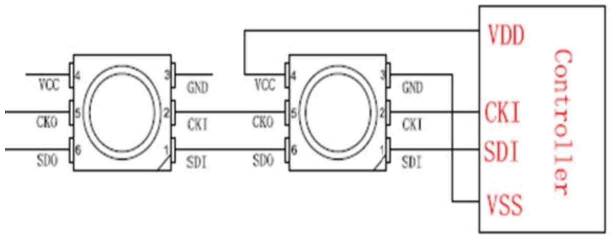

Figure 3. LED strip to Microcontroller

The microphone relayed pressure values back to the microcontroller using the I2S protocol. The microphone connected to clock, word select and MISO pins on the microcontroller. The I2S clock operated at 3MHz which allowed for sampling at 48KHz. We used the microcontroller I2S word select which toggled the word select pin after each data reading. This is traditionally used to gather microphone data from a left and right microphone on a single data line. We found that using the microcontroller controlled world select resulted in better microphone readings than bit banging a GPIO port manually due to timing constraints. The microphone also had a select pin that determined whether data was sent when word select when high or low. We left this pin floating which sets word select as active low. The microphone sends 32 bits of data each cycle. The first 18 bits are data bits, while the 14 remaining bits are 0's and can be discarded. The format of the microphone data can be seen in Figure 4.

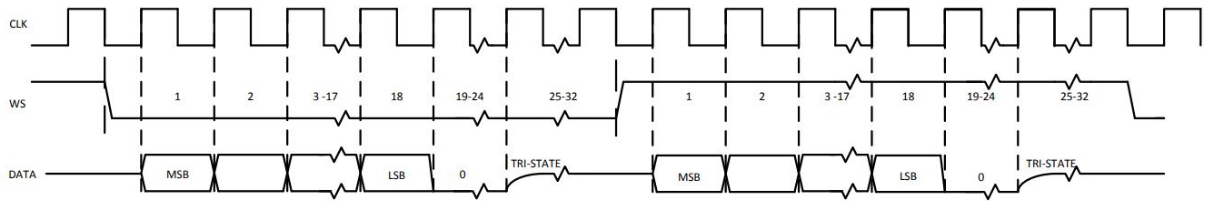

Figure 4. Microphone data signal

Schematics

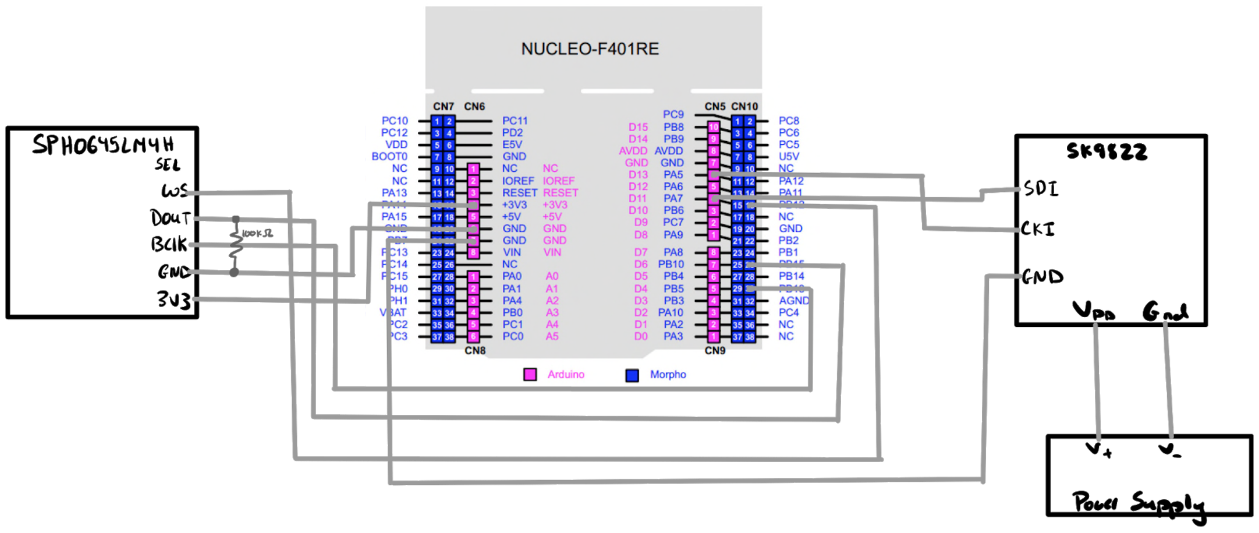

Figure 5. Schematic Diagram

Microcontroller Design

The project was separated into four main areas: LED Strip SPI, microphone I2S, LED color control, and signal processing. LED SPI and microphone I2S library functionalities are outlined in the new hardware section. These libraries included only the functions to interface with each peripheral. This section will go into how the peripherals and their interfaces were integrated in the microcontroller.

LED Color Control

Since the configuration of color for the LED strip required knowing the desired RGB hex code and sending it in the dataframe, an alternative method for selecting color and brightness of LED's was necessary before we would be able to implement our music based control in an intuitive way. The method we decided to use to accomplish this was to convert the RGB color space to HSL (Hue Saturation Lightness) space. The specific method was adapted from the wikipedia page on HSL and HSV [1]. This new color space broke colors down into 3 components. Colors could be represented as an angular quantity from 0 to 360 with red corresponding to 0 (and 360) degrees, green being 120, and blue being 240 (and all other colors in between). The intensity of the colors was controlled with the saturation parameter, which went from 0 (white) to 1 (pure color). Lightness was used to change the overall brightness. The output of the HSL conversion was equivalent to an RGB hex code to specify the color of the LED in RGB space. One important thing to note is that the LED's have an additional 5 bit global current consumption parameter that was not manipulated, and kept constant regardless of color for simplicity (though it could also be used to change brightness).

Signal processing

The microphone readings were much less than ideal. The values tended to be either 0, -1, the maximum reading, or minimum reading. The example code for the microphone from [2] even included a loop to discard raw measurements that were either 0 or -1. This same example code was used as a reference for obtaining the volume waveform. We empirically found that taking 64 (non 0 and non-1) samples and finding the maximum difference between readings gave the best result. This difference was then normalized to be in the range of -1 to 1 and converted to decibel scale. This conversion would allow for the volume amplitudes to be accurately displayed since human perception of loudness also follows a logarithmic scale. The pressure readings were then thresholded to be in the range of -30 to 0 dB. Readings of -30 dB were assigned the color purple (color angle 280) while 0 dB corresponded to red (color angle 0). Amplitudes (in dB) in between were linearly mapped to color angles. The brightness of the LED's was also adjusted based on the same color angle such that low amplitude readings (purple) were much dimmer than high amplitude readings (red). These color angles and brightnesses were then sent through the HSL to RGB conversion method and sent to the LED strip for display.

In addition to volume waveform detection, we attempted to get an FFT display working. The FFT algorithm used was from [3], and was essentially an implementation of the Cooley-Tukey algorithm. This algorithm worked in place, and we used a 128 bin implementation for the strip. Pressure readings were read in at a 20 KHz sample rate, and since most songs did not use notes that high, we believed it would be acceptable for this application. The first 128 LED's of the strip displayed the magnitude of each frequency bin from the FFT, with color being adjusted based on the magnitude. This is the main area where the microphone limitations were noticeable. The entire FFT function and LED output was tested using software-generated pressure readings of square waves, and the expected spectrum of odd harmonics was observed on the strip. When the pressure readings were instead grabbed from the microphone, a square wave input would not register above the noise floor. Even when blowing directly on the microphone, the FFT generated appeared random and would not change in amplitude. We believe that this is a direct result of the microphone mainly sampling 0 or -1. We tried a similar method as we had adopted for the volume waveform, where 0 and -1 values were discarded and this did not work either. We attempted taking the mean of 64 samples, the maximum difference in 64 samples and many other combinations, but the FFT still would not function properly, likely due to the fact that the microphone did not measure many intermediate values in its range. We believe that with a microphone with better measurement fidelity, the FFT implementation would work properly.

Results

We were able to successfully implement the volume waveform. There was no noticeable delay between the music playing and the LED's changing. However, we found that the microphone returned mostly very high or very low values, with few intermediate ones. As a result, there wasn't much color variation in our volume waveform as the majority of the values were clustered around the max pressure or 0. An example of the LED display can be seen in Figure 6.

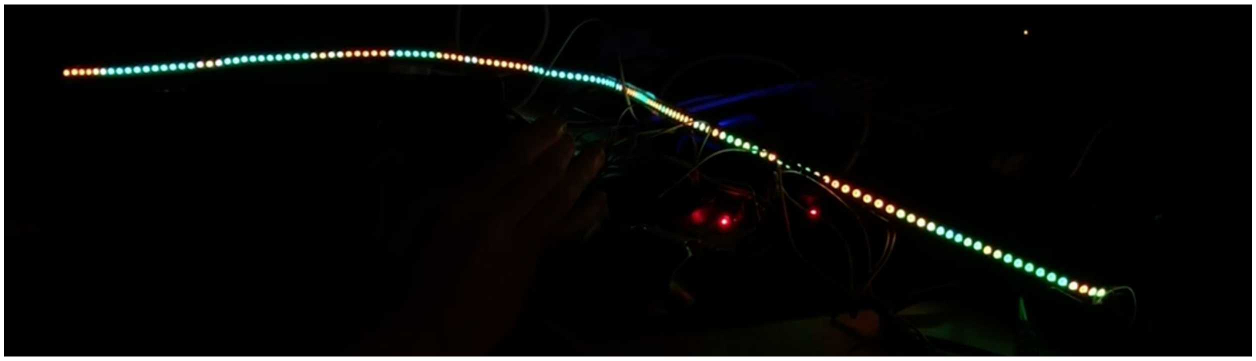

Figure 6. LED Volume Waveform

For FFT analysis, we did not have a successful implementation due to the lack of precise microphone measurements. We were able to test the FFT algorithm using pre-set pressure values to pass in a square wave. Using these values, we were able to successfully take the FFT and program the LED's to change color based on the magnitude of its corresponding frequency bin. However, lack of clean microphone data prevented us from fully implementing FFT.

References

[1] “HSL and HSV.” *Wikipedia*, Wikimedia Foundation, 25 Nov. 2020, en.wikipedia.org/wiki/HSL_and_HSV.

[2] Ada, Lady. “Adafruit I2S MEMS Microphone Breakout.” *Adafruit Learning System*, learn.adafruit.com/adafruit-i2s-mems-microphone-breakout/.

[3] “Fast Fourier Transform.” *Fast Fourier Transform - Rosetta Code*, rosettacode.org/wiki/Fast_Fourier_transform.

Parts List

Part	Source	Part Number	Cost
STM401RE Microcontroller	HMC	----	----
LED Strip	www.amazon.com	B07BPX2KFD	$36.88
MEM Microphone	www.adafruit.com	3421	$6.95
Power Supply	www.amazon.com	B0852HL336	$21.99

Appendix A: main.c

```c
/**
   Main: Contains main function
   @file main.c
   @author
   @version
*/

#include "STM32F401RE.h"
#include "math.h"
#include <time.h>

#define NUM_LEDS 144 //144
#define MAX_PRESSURE (1 << 17)

// Hue from 0 - 360, saturation from 0 - 1, lightness from 0 - 1
// Algorithm based on https://en.wikipedia.org/wiki/HSL_and_HSV
void hsvConvert(double hue, double saturation, double lightness, uint8_t* red,  uint8_t*
green, uint8_t* blue) {
   double C = (1.0 - fabs(2.0*lightness -1.0)) * saturation;
   double X = C * (1.0 - (fabs(fmod((hue / 60.0), 2.0) -1.0)));
   double m = lightness - C/2.0;

   double Rp;
   double Gp;
   double Bp;

   switch ((int) floor(hue / 60)) {
       case (0):
           Rp = C;
           Gp = X;
           Bp = 0;
           break;
       case (1):
           Rp = X;
           Gp = C;
           Bp = 0;
           break;
       case (2):
           Rp = 0;
           Gp = C;
           Bp = X;
           break;
       case (3):
           Rp = 0;
           Gp = X;
           Bp = C;
           break;
       case (4):
           Rp = X;
```

```c
            Gp = 0;
            Bp = C;
            break;
        case (5):
            Rp = C;
            Gp = 0;
            Bp = X;
            break;
        case (6): //only 360 gets here
            Rp = C;
            Gp = 0;
            Bp = X;
            break;
    }
    *red = (uint8_t) ((Rp + m) * 255);
    *green = (uint8_t) ((Gp + m) * 255);
    *blue = (uint8_t) ((Bp + m) * 255);
}

int main(void) {

    configureFlash();
    configureClock(); //8 MHz
    RCC->APB1ENR.TIM2EN = 1; // TIM2_EN
    initTIM(TIM2);

// ========================== Initiate THINGS ==========================
    i2sInit();
    spiInit(2, 0, 0);
    uint32_t ledValues[NUM_LEDS + 2][4];

    // set everything dark to begin with
    ledValues[0][0] = 0x00;
    ledValues[0][1] = 0x00;
    ledValues[0][2] = 0x00;
    ledValues[0][3] = 0x00;

    uint8_t blue = 0x04;
    uint8_t green = 0x00;
    uint8_t red = 0x00;

    for (uint8_t i = 1; i < NUM_LEDS + 1; i++) {
        ledValues[i][0] = 0b11100000;
        ledValues[i][1] = 0x00;
        ledValues[i][2] = 0x00;
        ledValues[i][3] = 0x00;
    }
    ledValues[NUM_LEDS + 1][0] = 0xff;
    ledValues[NUM_LEDS + 1][1] = 0xff;
    ledValues[NUM_LEDS + 1][2] = 0xff;
    ledValues[NUM_LEDS + 1][3] = 0xff;
```

```
   uint8_t global = 0b11100111;
   volatile int32_t pressure = 0;
   int numSamples = 64;
   int samples[numSamples];
   volatile float pressureAmplitude = 0;
   float logResolution = 30;

   int numSamplesFFT = 128;
   complex fftBuffer[numSamplesFFT];

   pinMode(GPIOA, 10, GPIO_OUTPUT);

   volatile float bufferAmplitudeLog;

   volatile float maxamp = 0;
   volatile float minamp = 0;
   while(1){
       // *************    FFT Waveform  ************

       // for (int j = 0; j < numSamplesFFT; j++) {
       //     pressure = 0;
       //     while (pressure == 0 || pressure == -1) {
       //         pressure = (int32_t) i2sReceive();
       //     }
       //     // if (j < (numSamplesFFT/2)) { // Uncomment for square wave testing
       //     //     pressure = 1 << 14;
       //     // } else {
       //     //     pressure = 0;
       //     // }

       //     pressure = (int32_t) i2sReceive();
       //     pressureAmplitude = ((float)((pressure >> 14) +
MAX_PRESSURE)/(1.7*MAX_PRESSURE));
       //     double multiplier = 0.5 * (1 - cos(2*M_PI*(j)/(numSamplesFFT-1))); //Hanning
window
       //     fftBuffer[j] = (complex) (pressureAmplitude * multiplier);

       //     togglePin(GPIOA, 10);
       //     delay_micros(TIM2, 50);
       // }

       // fft(fftBuffer, numSamplesFFT);

       // for (int i = 1; i < numSamplesFFT+1; i++){
       //     float fftAmp = cabs(fftBuffer[i-1]);
       //     bufferAmplitudeLog = fmin(15, fmax(0.0, 20.0*log10(fftAmp)));
       //     minamp = fmin(minamp, bufferAmplitudeLog);
       //     maxamp = fmax(maxamp, bufferAmplitudeLog);
       //     hsvConvert((280.0/15.0) * (15-(bufferAmplitudeLog)), 1, 0.01, &red, &green,
&blue);

       //     ledValues[i][0] = global;
```

```
        //     ledValues[i][1] = blue;
        //     ledValues[i][2] = green;
        //     ledValues[i][3] = red;
        // }

        // for (uint8_t i = 0; i <= NUM_LEDS + 1; i++) {
        //     spiSendReceive(ledValues[i][0]);
        //     spiSendReceive(ledValues[i][1]);
        //     spiSendReceive(ledValues[i][2]);
        //     spiSendReceive(ledValues[i][3]);
        // }


        // ************** Volume Waveform *************

        for (int i = 0; i < numSamples; i++) {
            pressure = 0;
            while (pressure == 0 || pressure == -1){
                pressure = (int32_t) i2sReceive();
            }
            pressure = pressure >> 14;
            samples[i] = pressure;
        }
        int maxSamp = 0;
        int minSamp = 0;

        for (int i = 0; i < numSamples; i++) {
            maxSamp = fmax(maxSamp, samples[i]);
            minSamp = fmin(minSamp, samples[i]);
        }

        // convert pressure to log
        pressureAmplitude = 20*log10((float)(maxSamp - minSamp) / (1.65*MAX_PRESSURE));
        pressureAmplitude = fmin(0.0, fmax(-logResolution, pressureAmplitude));

        int colorAngle = round((280.0 - (280.0/logResolution * (logResolution +
pressureAmplitude))));

        hsvConvert(colorAngle, 1, 0.005 + 0.5*(280.0 - (float)colorAngle)/280.0, &red,
&green, &blue);
        ledValues[1][0] = global;
        ledValues[1][1] = blue;
        ledValues[1][2] = green;
        ledValues[1][3] = red;

        for (uint8_t i = NUM_LEDS; i >= 1; i--) {
            if (i != 1) {
                ledValues[i][0] = ledValues[i-1][0];
                ledValues[i][1] = ledValues[i-1][1];
                ledValues[i][2] = ledValues[i-1][2];
                ledValues[i][3] = ledValues[i-1][3];
            }
        }
        // colorAngle = (int) fmod((colorAngle + 1), 360);

        for (uint8_t i = 0; i <= NUM_LEDS + 1; i++) {
            spiSendReceive(ledValues[i][0]);
            spiSendReceive(ledValues[i][1]);
            spiSendReceive(ledValues[i][2]);
            spiSendReceive(ledValues[i][3]);
        }

        delay_micros(TIM2, 7500);
    }
}
```

Appendix B: SPI Library

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).
*    -- ncpha: clock phase (0: data changed on leading edge of clk and captured on next
edge,
*          1: 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 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, 7, GPIO_ALT); // PA7, Arduino D11, SPI1_MOSI
   pinMode(GPIOA, 4, GPIO_ALT); // PA4, Arduino A2, SPI1_NSS
   pinMode(GPIOB, 6, GPIO_OUTPUT); // PB6, Arduino D10, Manual CS

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

   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 16 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) {
//     SPI1->DR.DR = send; // Transmit the character over SPI
//     while (!(SPI->SPI_SR.RDRF)); // Wait until data has been received
//     return (char) (SPI->SPI_RDR.RD); // 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;
}

uint8_t spiSendReceive(uint8_t send) {
   SPI1->CR1.SPE = 1;
   SPI1->DR.DR = send;

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

   SPI1->CR1.SPE = 0;
   return rec;
}

void spiSend16(uint16_t message) {
   SPI1->DR.DR = message;                      // Send the message to the buffer
   while (!SPI1->SR.TXE || SPI1->SR.BSY);  // Wait for the transmit buffer to be empty
}
```

spi.h

```c
// 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);


void spiSend16(uint16_t message);
#endif
```

Appendix C: I2S Library

I2S.c

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

#include "STM32F401RE_I2S.h"
#include "STM32F401RE_RCC.h"
#include "STM32F401RE_GPIO.h"

void i2sInit() {
   configurePLLI2S();
   RCC->AHB1ENR.GPIOBEN = 1;
   RCC->APB1ENR.SPI2EN = 1;

   pinMode(GPIOB, 12, GPIO_ALT); // PB12, SPI2_WS
   pinMode(GPIOB, 13, GPIO_ALT); // PB13, SPI2_CLK
   pinMode(GPIOB, 15, GPIO_ALT); // PB15, SPI2_MISO


   // Set to AF05 for SPI alternate functions
   GPIOB->AFRH |= (1 << 18) | (1 << 16);
   GPIOB->AFRH |= (1 << 22) | (1 << 20);
   GPIOB->AFRH |= (1 << 30) | (1 << 28);

   // Serial Clock baud rate configuration
   I2S2->I2SPR.I2SDIV = 3;
   I2S2->I2SPR.ODD = 1;

   // Steady state clock level and output the master clock
   I2S2->I2SCFGR.CKPOL = 1;
   I2S2->I2SPR.MCKOE = 1;

   // Activate I2S
   I2S2->I2SCFGR.I2SMOD = 1;

   // Select I2S standard
   I2S2->I2SCFGR.I2SSTD = 0b00;

   // Data frame specific sizes
   I2S2->I2SCFGR.DATLEN = 0b01;
   I2S2->I2SCFGR.CHLEN = 0b1;

   // Master mode and direction
   I2S2->I2SCFGR.I2SCFG = 0b11;
   I2S2->I2SCFGR.I2SE = 1;
}

int32_t i2sReceive() {
   while(!(I2S2->SR.RXNE));
   uint16_t temp1 = I2S2->DR.DR;

   while(!(I2S2->SR.RXNE));
   uint16_t temp2 = I2S2->DR.DR;

   uint32_t rec = (temp1 << 16) + temp2;
   rec = (int32_t) rec;
   return rec;
}
```

I2S.h

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

#ifndef STM32F4_I2S_H
#define STM32F4_I2S_H

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

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

#define I2S2_BASE (0x40003800UL)
#define __IO volatile

///////////////////////////////////////////////////////////////////////////////
// Bitfield structs
///////////////////////////////////////////////////////////////////////////////
typedef struct {
 __IO uint32_t RXDMAEN      : 1;
 __IO uint32_t TXDMAEN      : 1;
 __IO uint32_t              : 1;
 __IO uint32_t              : 1;
 __IO uint32_t              : 1;
 __IO uint32_t ERRIE        : 1;
 __IO uint32_t RXNEIE       : 1;
 __IO uint32_t TXEIE        : 1;
 __IO uint32_t              : 24;
} I2S_CR2_bits;

typedef struct {
 __IO uint32_t RXNE         : 1;
 __IO uint32_t TXE          : 1;
 __IO uint32_t              : 1;
 __IO uint32_t              : 1;
 __IO uint32_t              : 1;
 __IO uint32_t             : 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;
} I2S_SR_bits;

typedef struct {
 __IO uint32_t DR  : 16;
 __IO uint32_t     : 16;
} I2S_DR_bits;

typedef struct {
 __IO uint32_t CHLEN     : 1;
 __IO uint32_t DATLEN    : 2;
 __IO uint32_t CKPOL     : 1;
 __IO uint32_t I2SSTD    : 2;
 __IO uint32_t           : 1;
 __IO uint32_t PCMSYNC   : 1;
 __IO uint32_t I2SCFG    : 2;
 __IO uint32_t I2SE      : 1;
 __IO uint32_t I2SMOD    : 1;
 __IO uint32_t           : 4;
 __IO uint32_t           : 16;
} I2S_I2SCFGR;

typedef struct {
 __IO uint32_t I2SDIV    : 8;
 __IO uint32_t ODD       : 1;
 __IO uint32_t MCKOE     : 1;
 __IO uint32_t           : 6;
 __IO uint32_t           : 16;
} I2S_I2SPR;


typedef struct {
 __IO uint32_t CR1;            /*!< SPI control register 1 (not used in I2S mode),
Address offset: 0x00 */
 __IO I2S_CR2_bits CR2;        /*!< SPI control register 2,
Address offset: 0x04 */
 __IO I2S_SR_bits SR;         /*!< SPI status register,
Address offset: 0x08 */
 __IO I2S_DR_bits DR;         /*!< SPI data register,
Address offset: 0x0C */
 __IO uint32_t CRC;      /*!< SPI CRC polynomial register (not used in I2S mode), Address
offset: 0x10 */
 __IO uint32_t RXCRC;     /*!< SPI RX CRC register (not used in I2S mode),         Address
```

```
offset: 0x14 */
 __IO uint32_t TXCRC;     /*!< SPI TX CRC register (not used in I2S mode),         Address
offset: 0x18 */
 __IO I2S_I2SCFGR I2SCFGR;    /*!< I2S_I2S configuration register,
Address offset: 0x1C */
 __IO I2S_I2SPR I2SPR;      /*!< SPI_I2S prescaler register,                         Address
offset: 0x20 */
} I2S_TypeDef;

// Pointers to GPIO-sized chunks of memory for each peripheral
#define I2S2 ((I2S_TypeDef *) I2S2_BASE)

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

void i2sInit();
int32_t i2sReceive();

#endif
```

Appendix D: FFT Library

FFT.c

```c
#include <stdio.h>
#include <math.h>
#include <complex.h>

// From https://rosettacode.org/wiki/Fast_Fourier_transform#C
double PI;
typedef double complex cplx;
void _fft(cplx buf[], cplx out[], int n, int step) {
   if (step < n) {
       _fft(out, buf, n, step * 2);
       _fft(out + step, buf + step, n, step * 2);
       for (int i = 0; i < n; i += 2 * step) {
           cplx t = cexp(-I * PI * i / n) * out[i + step];
           buf[i / 2]     = out[i] + t;
           buf[(i + n)/2] = out[i] - t;
       }
   }
}
void fft(cplx buf[], int n) {
   PI = atan2(1,1) * 4;

   cplx out[n];
   for (int i = 0; i < n; i++) out[i] = buf[i];
   _fft(buf, out, n, 1);
}
```

FFT.h

```c
#ifndef FFT_H
#define FFT_H

#include <stdio.h>
#include <math.h>
#include <complex.h>

void _fft();
void fft();

#endif
```