Ocean Forecast Visualizer

HMC E155 Final Project Fall 2021

Shreya Sanghai, Lauren Le

Abstract

This project inputs ocean weather data from the stormglass API and creatively visualizes it. The STM32F401RE microcontroller is used to parse the data and send the weather data to the FPGA.The Intel Max1000 FPGA detects button inputs from the user about how far into the future they wish to see the surf data for and displays the received wave height and swell period from the MCU on a LED panel. The MCU also controls a DF Player mini which plays ocean nature sound tracks based on the air conditions.

Table of Contents

Introduction

Motivation

When surfers or scuba divers want to go catch a wave or explore underwater marine life, they often check the surf or ocean forecast online to see if it is a good day for surfing or scuba diving. Sometimes reading the forecast data can be boring or inconvenient when understanding the physical implications of the numerical data. Our final project seeks to present ocean data such as wave height, swell period, wind, etc in both visual and audio form to help people envision the ocean conditions in a fun and aesthetic way. We will be visualizing data from Huntington Beach, California (see Figure 1):

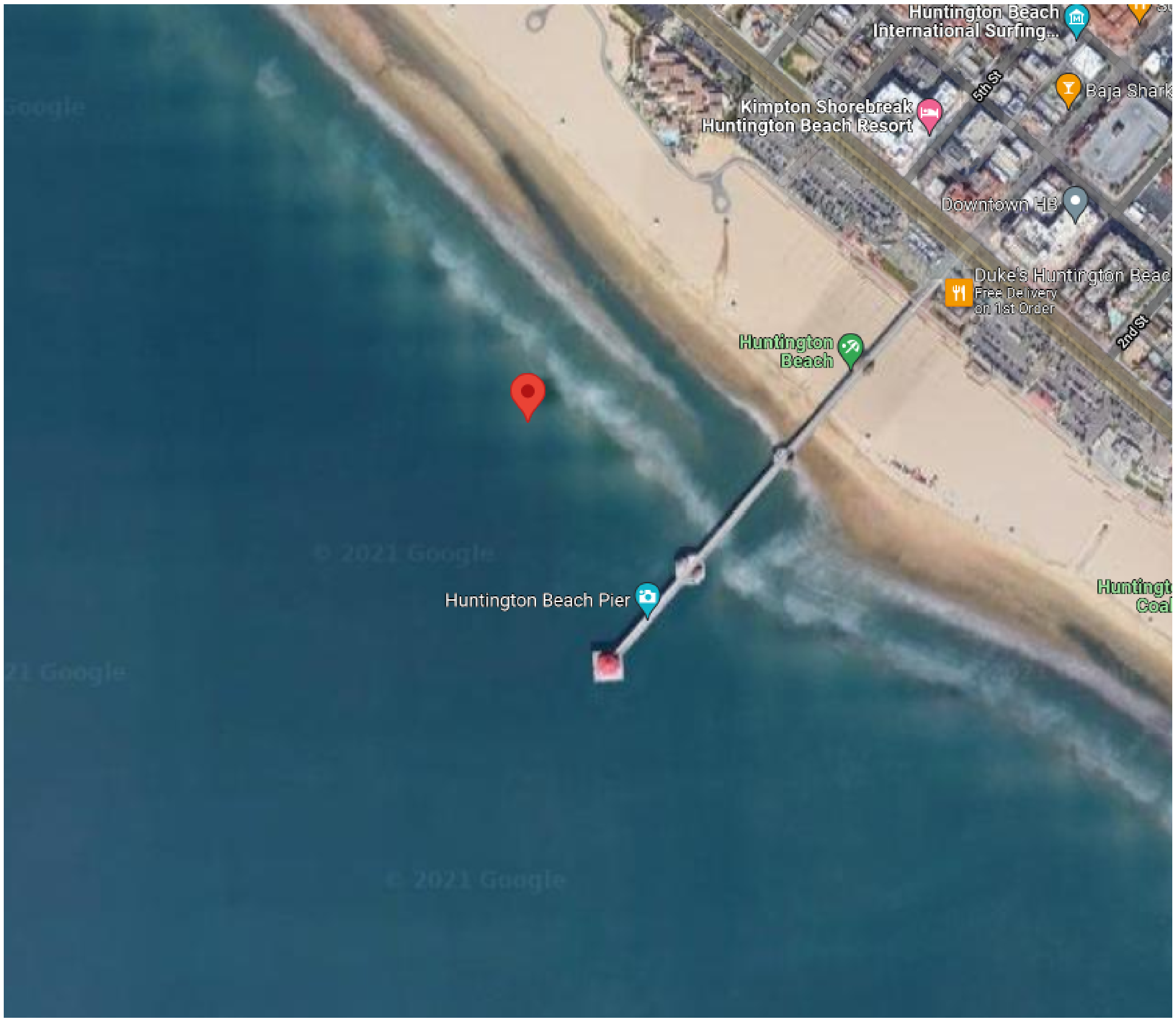

Huntington Beach International Surfing...
Baja Shark
Kimpton Shorebreak Huntington Beach Resort
5th St
Downtown HB
Duke's Huntington Beach
Free Delivery on 1st Order
Huntington Beach
2nd St
Huntington Beach Pier
© 2021 Google

Figure 1 - Location of Ocean Forecast Data - Huntington Beach, California (33.655550, -118.00710).

Our project will consist of multiple wave-shaped LED panels that stack up vertically and help disperse the light from LED strips mounted underneath the panels. Each panel represents a wave height metric (ft). Based on the ocean data retrieved by an API request, the LED strips behind the panels will sequentially light up and down along the vertical axis between the bottom panel and the wave height specified. The speed at which the LED strips light up along the vertical axis will depend on the wave swell period. Along with the LED wave display, the system will include a speaker that plays nature music to convey the calmness or chaos of ocean conditions, determined by parameters such as wind. In addition, we will provide buttons for users to press to specify how much time ahead they would like to view/hear the ocean conditions. See Figure 2:

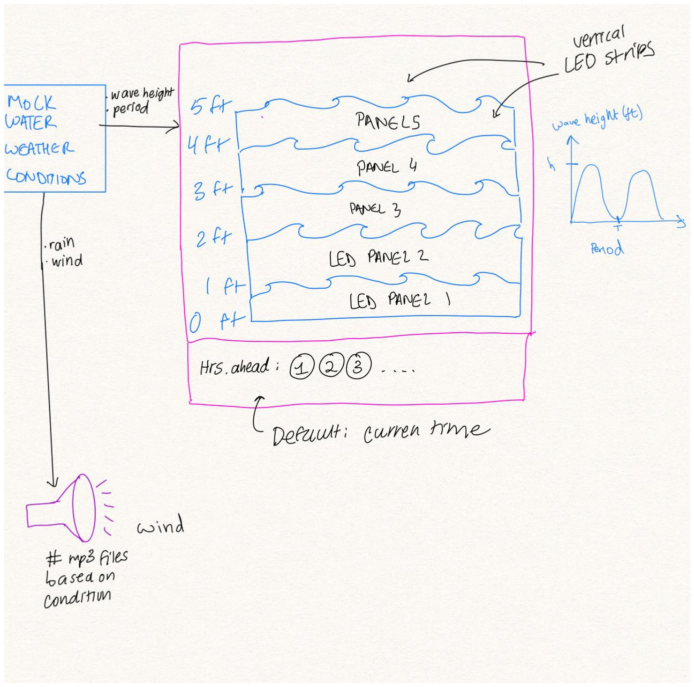


Figure 2: LED panels displaying wave height and swell period

Overview

The main components of our system are:

- Input mock ocean condition data
- Play nature music based on ocean conditions
- Visualize swell period and wave height
- Give users the option to choose how many hours ahead of current time they want to view ocean conditions (given several options of future times)

All the main components of our system need to be working individually and with each other for successful completion of the project.

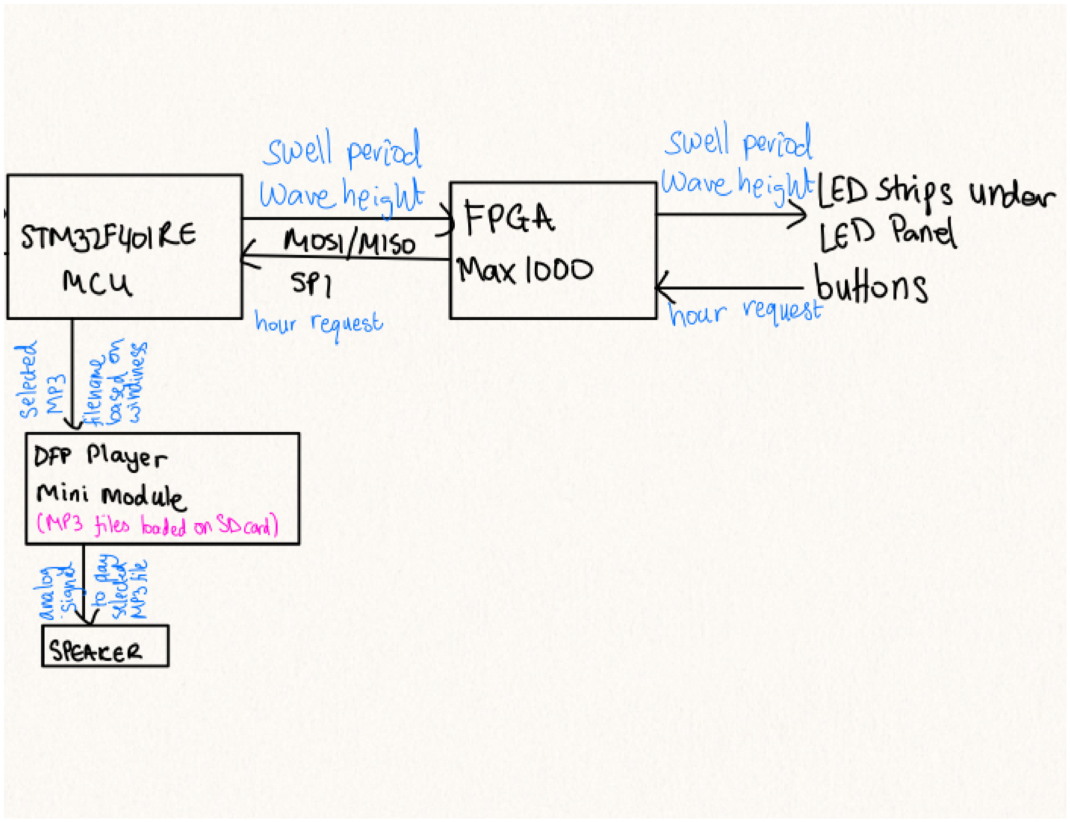


Figure 3: Block Diagram of System

New Hardware

The new hardware we will be using is:

- The DFP Player Mini Module to read MP3 files
    - The MP3 files are stored on the SD card on the DFP player
    - Based on the weather data, the MCU will determine which MP3 file must be played from a list of preloaded files on the SD card
    - The DFP Player will input the filename from the MCU and send the analog signal to the speaker to be played
- LED strips to display the wave height and the swell period
    - Each LED in the LED strip is individually controllable

- ○ We will carefully program the LED strip to display the information artistically.

Schematics

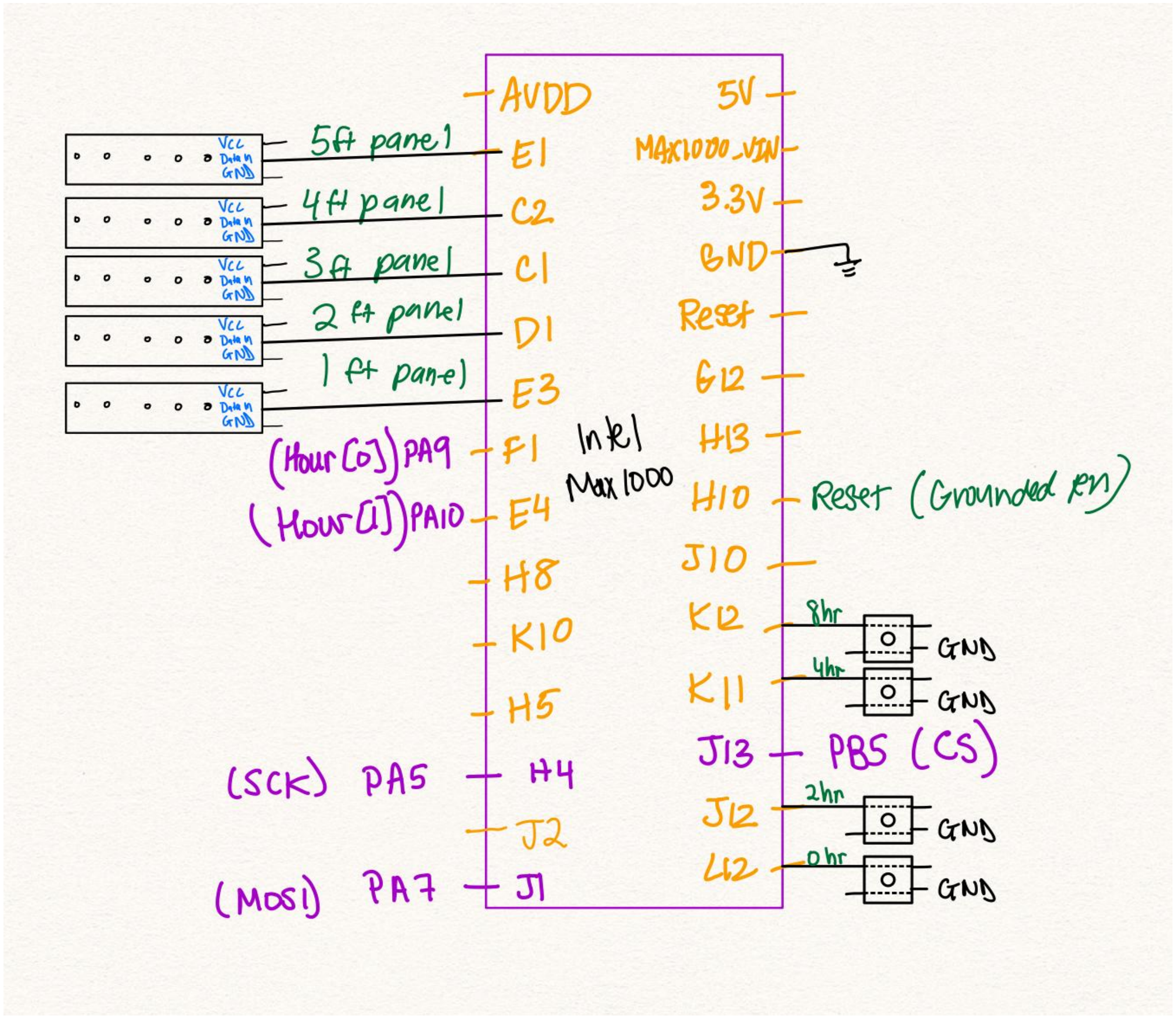


Figure 4. FPGA Schematic - LED strips, buttons and MCU

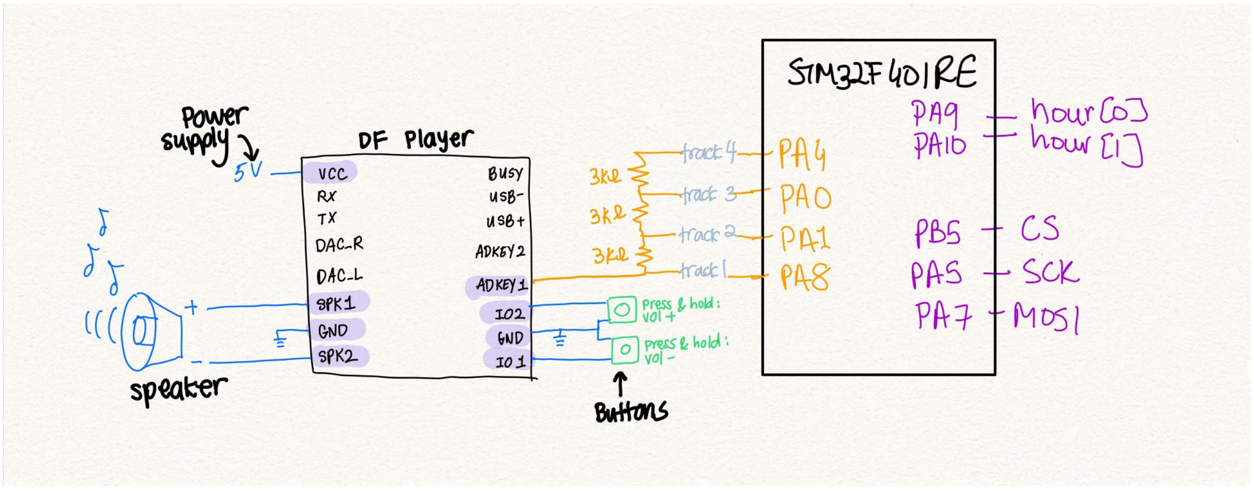


Figure 5. DF Player schematic connecting to MCU

Microcontroller(MCU) Design

We use the STM32F401RE microcontroller to:

- Parse mock ocean data to determine input to DFP player and FPGA
- Send selected mp3 filename to DFP Player which will interface with a speaker to play music
- Communicate to MAX1000 FPGA over SPI to send swell period and wave height data, and receive requested time to view ocean data

The MCU handles the logic for what data is displayed based on the time requested by the user. There are three different ocean forecast statistics: wave height, wave swell, and air temperature. The microcontroller uses digital read on two GPIO pins to decode a two bit value as four possible time requests: 0, 2, 4, or 8 hours. Next, this information is used to look up wave height and swell period from an array; the data values were obtained by a separate python script that makes a Global Weather API request to stormglass.io. Then, SPI communication (MCU is master, FPGA is slave) is used to transmit 8 bits of information where the first three least significant bits correspond to wave height and five most significant bits correspond to swell period.

The MCU also interfaces with the DF Player to control which tracks are played based on the SD card inserted on the DF Player. Based on the hour (0 corresponds to current time, 2 corresponds to 2 hours into the future, etc) requested from the FPGA, we use digital write low to a pin wired to a resistor ladder to complete the desired circuit. Based on the DF Player datasheet, the ADKEY1 pin interprets a specific resistance for each numbered track loaded on the SD card (001.mp3, 002.mp3, etc). In order to control multiple tracks played from the same ADKEY1 pin without changing the resistor value manually, we used the resistor ladder so the total resistance adds up linearly for every next track on the SD card.

FPGA Design

We use the MAX1000 FPGA to:

- Interface with LED strips and synchronize timing of lighting based on wave height and swell period
- Interface with buttons which provide user with multiple options to select how many hours ahead of time they want to view ocean data and send input to MCU and receive back the swell period and wave hieght

Lighting up LEDs

The LEDs need 5V of power to run. We used a power supply to generate the power and connected it to the Vcc and ground pins of the LED strip. We used a FPGA pin to send the data out to the LED strip. We determined that although the FPGA pin is 3.3V, it is within the correct logic levels as long as the strip has 5V of power.

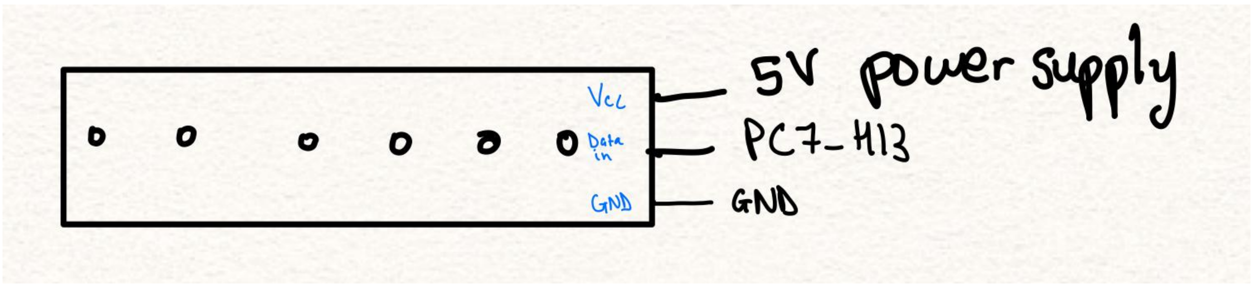


Figure 6. LED lights connected to 5V power supply, ground, and pin from STM32F401RE microcontroller.

Below is the block diagram for our LED strip. The WS2812b strip takes in a single data line. Each LED on the strip takes 24 bits representing the colour with which it should light up and then passes the remaining bits onto the next LED which does the same. Each bit, 1 or 0 is sent in 1.1μs based on the datasheet. A ‘1’ is high for 0.75μs and low for 0.32μs, and a ‘0’ is high for 0.32μs and low for 0.75μs. Since the FPGA clock is 12Mhz, 13 clock cycles is 1.1μs. So we either send a high for 4 cycles and low for 9 cycles for a ‘0’ or a high for 9 cycles and a low for 4 cycles for a ‘1’.

We had 10 led strips of 24 leds each divided into 5 panels (2 strips per panel). The number of panels we light up depends on the wave height. The time it takes to turn on and turn off all 24 leds in the strip depends on the swell period. We have 5 FPGA pins driving the 5 panels. Depending on the wave height, we have an enable signal which determines whether the panel is on or not.

FSM design

We have an FSM that drives each strip. The FSM has 3 states: waiting, writing and delay. The FSM stays in the writing stage for 13 clock cycles and sends either a 9 highs and 4 lows or 4 highs and 9 lows depending on the current bit being sent. Once 13 cycles have passed and the counter reaches 0, it goes back to the waiting state. The waiting state checks if all the leds in the strip have been written to and if so goes into the delay state. If not, we go back and write the next bit of information. The delay state is used to reset the led strip and give it a new string of bits. The delay is also used to interpret the swell period by having a delay between each led lighting up in the desired color on the strip. The steps that the FSM follows are detailed below.

1) Chose desired color (shade of blue if led_num is less than wave_num, otherwise black)
2) Write bit based on chosen color (13 clock cycles)
3) Wait until 24 bits have been written (one led)
4) Increment led_num
5) Wait until all 24 leds have been written
6) Delay desired amount of time based on swell period (and to reset the led strip)
7) Increment wave num
8) Repeat steps 1-7 until wave_num is 24
9) Decrement wave_num
10) Repeat steps 1-6 and 9 until wave_num is 0
11) Repeat steps 1-12

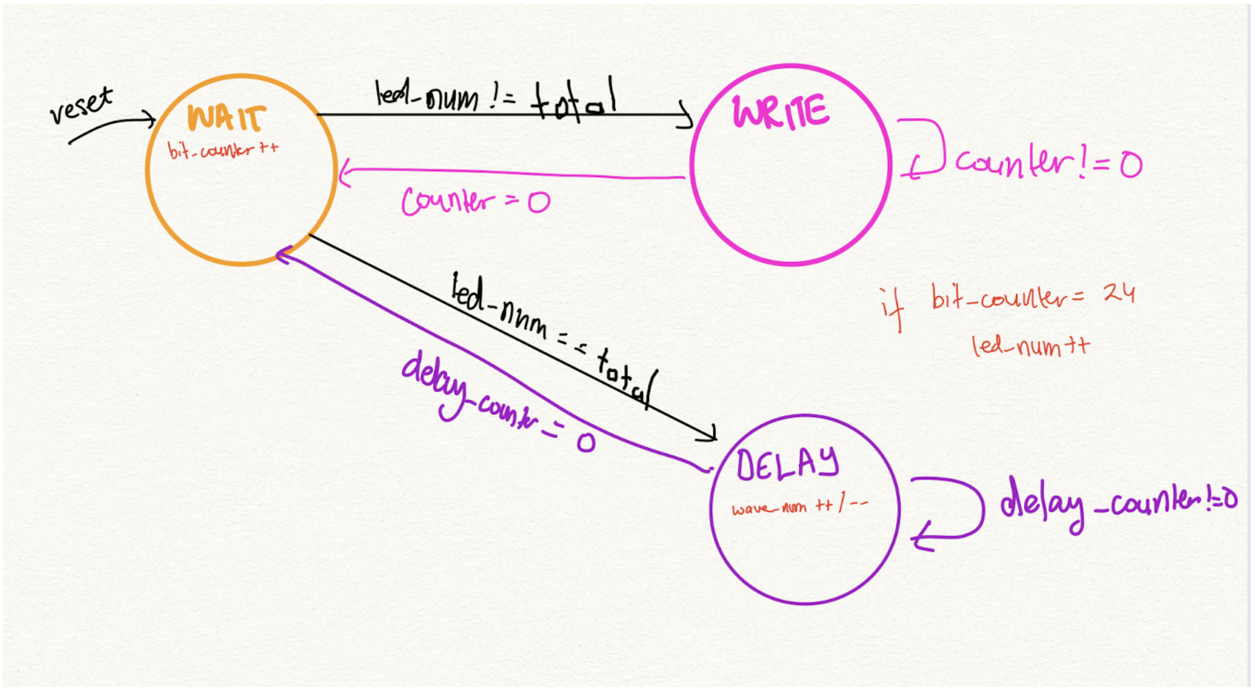


Figure 7: FSM for LED strips

User Inputs

We have 4 buttons wired to 4 FPGA pins. The button decoder module waits to detect a button press and assigns the hour based on which button was pressed. The module has an enabled flop that stores the last button press. The enable signal goes high when a new button is pressed. The module then sends the output of the last hour pressed to the MCU over two pins to represent the four possible button options.

Once the MCU detects a change in the data requested by the user, it sends over the swell period and the wave height over SPI and the FPGA is configured as an SPI slave.

Results

In summary, we were able to use the MCU, FPGA, buttons, and LED strips to visualize the ocean condition data for wave height (ranges from 2 to 5 ft), swell period (ranges from from 5 to 31 seconds), and air conditions. The FPGA handles user pressed buttons by keeping track of the most recent button pressed corresponding to the desired hour to visualize ocean conditions. The desired hour information is sent to the MCU by digital read, and the MCU extracts the correct stormglass API ocean data and transmits the swell period and wave height values back to the FPGA over SPI. Then, the FPGA uses the wave height number to control how many horizontal panels are lit and to accurately display the speed at which the LED strips light up.

While the MCU was also interfaced with the DF Player to play different ocean nature soundtracks, the hardware was corrupted during final review check in and was therefore not able to play any mp3 files to the speakers. However, the circuit and code for the DF Player was previously tested and fully functional. Overall, there were multiple iterations of brainstorming, testing, failing, debugging, and modifying of the logic implementation and hardware design components to achieve a system that achieves the majority of the goals of the project delineated earlier in this paper.

References

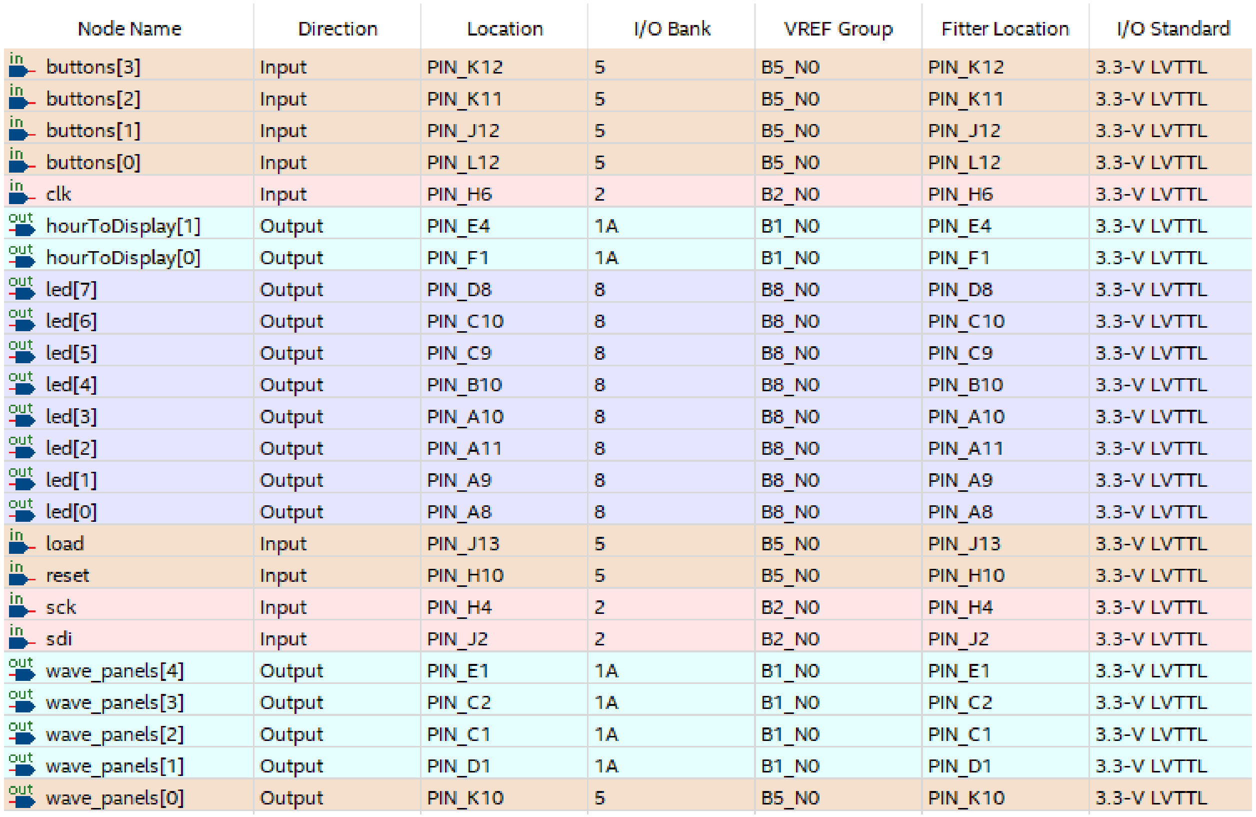
1) E85 Textbook
   S. Harris and D. Harris, *Digital Design and Computer Architecture: Arm edition*, Elsevier Science, 2015.
2) DFP Player Manual
   https://picaxe.com/docs/spe033.pdf
3) STM32401RE Datasheet
   https://pages.hmc.edu/brake/class/e155/fa21/assets/doc/STM32F401RE_Datasheet.pdf
4) STM32 Reference Manual
   https://pages.hmc.edu/brake/class/e155/fa21/assets/doc/STM32F401RE_Reference_Manual_RM0368.pdf
5) Intex Max 1000 user guide
   https://pages.hmc.edu/brake/class/e155/fa21/assets/doc/MAX1000UserGuide.pdf
6) LED strips datasheet
   https://voltiq.ru/datasheets/WS2812B_datasheet_EN.pdf
7) API for Marine Data
   https://stormglass.io/

Bill of Materials

Component	Description	Location	Unit Price	Quantity	Price	Link
DFP Player Mini Module	Plays MP3 files saved on SD Card	DF Robot	$5.99	1	$5.99	DF Robot
Speaker	Plays MP3 files audibly	Amazon	$7.99	1	$7.99	Amazon
Micro SD Card	Store MP3 Files	Amazon	$7.99	1	$7.99	Amazon
LED Strips	Display Wave data	Amazon	$18.99	2	$37.98	Amazon
TOTAL					$59	

Appendix

Verilog Pin Assignments



Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard
buttons[3]	Input	PIN_K12	5	B5_N0	PIN_K12	3.3-V LVTTL
buttons[2]	Input	PIN_K11	5	B5_N0	PIN_K11	3.3-V LVTTL
buttons[1]	Input	PIN_J12	5	B5_N0	PIN_J12	3.3-V LVTTL
buttons[0]	Input	PIN_L12	5	B5_N0	PIN_L12	3.3-V LVTTL
clk	Input	PIN_H6	2	B2_N0	PIN_H6	3.3-V LVTTL
hourToDisplay[1]	Output	PIN_E4	1A	B1_N0	PIN_E4	3.3-V LVTTL
hourToDisplay[0]	Output	PIN_F1	1A	B1_N0	PIN_F1	3.3-V LVTTL
led[7]	Output	PIN_D8	8	B8_N0	PIN_D8	3.3-V LVTTL
led[6]	Output	PIN_C10	8	B8_N0	PIN_C10	3.3-V LVTTL
led[5]	Output	PIN_C9	8	B8_N0	PIN_C9	3.3-V LVTTL
led[4]	Output	PIN_B10	8	B8_N0	PIN_B10	3.3-V LVTTL
led[3]	Output	PIN_A10	8	B8_N0	PIN_A10	3.3-V LVTTL
led[2]	Output	PIN_A11	8	B8_N0	PIN_A11	3.3-V LVTTL
led[1]	Output	PIN_A9	8	B8_N0	PIN_A9	3.3-V LVTTL
led[0]	Output	PIN_A8	8	B8_N0	PIN_A8	3.3-V LVTTL
load	Input	PIN_J13	5	B5_N0	PIN_J13	3.3-V LVTTL
reset	Input	PIN_H10	5	B5_N0	PIN_H10	3.3-V LVTTL
sck	Input	PIN_H4	2	B2_N0	PIN_H4	3.3-V LVTTL
sdi	Input	PIN_J2	2	B2_N0	PIN_J2	3.3-V LVTTL
wave_panels[4]	Output	PIN_E1	1A	B1_N0	PIN_E1	3.3-V LVTTL
wave_panels[3]	Output	PIN_C2	1A	B1_N0	PIN_C2	3.3-V LVTTL
wave_panels[2]	Output	PIN_C1	1A	B1_N0	PIN_C1	3.3-V LVTTL
wave_panels[1]	Output	PIN_D1	1A	B1_N0	PIN_D1	3.3-V LVTTL
wave_panels[0]	Output	PIN_K10	5	B5_N0	PIN_K10	3.3-V LVTTL

Verilog

```
module Ocean_Data_SPI(input  logic clk,
                    input logic reset,
           input  logic sck, // Serial Clock Output - FPGA PA5_H4 ...  MCU
D13
           input  logic sdi, // MOSI - FPGA PA7_J2 ... MCU D11
//           output logic sdo, // MISO - FPGA PA6_J1  ... MCU D12
           input  logic load, // load - FPGA PB5_J13 ... MCU D4

                    input logic [3:0] buttons,
           output logic [1:0] hourToDisplay,
                    output logic [4:0] wave_panels,

                    output logic [7:0] led); // done PB3_J12 ... D3
```

```
       logic [7:0] q;
       logic [2:0] wave_height;
       logic [4:0] wave_enable;
       logic [4:0] swell_period;
       logic [7:0] counter;

       assign wave_height = q[2:0];
       assign swell_period = q[7:3];

       assign led[2:0] = hourToDisplay;
       assign led[7:3] = buttons;

       spi_slave_receive_only spi(clk, sck, sdi,load, q, counter);
       flop flop1(clk, reset, buttons, hourToDisplay);
       waves wave1(clk, reset, wave_height, swell_period,  wave_panels,
wave_enable);



endmodule

// slave (FPGA) only needs to receive data from master (MCu)
module spi_slave_receive_only(input logic      clk, sck,
                                                           input logic
      sdi,
                                                           input logic
load,
                                                           output logic
      [7:0] q,
                                                           output logic
[6:0] counter);
      always_ff @(posedge sck) begin
            q <= {q[6:0], sdi};
      end
endmodule


// CODE FOR BUTTONS
module flop(input logic clk, reset,
            input logic [3:0] buttons,
            output logic [1:0] hourToDisplay); // ADD EN
      logic [1:0] hours;
      logic en;
      decoder decoder1(buttons,hours,en);

        always_ff @ (posedge clk) begin
        if (en)
            begin
                hourToDisplay <= hours[1:0];
            end
        end

endmodule

module decoder(input logic [3:0] buttons,
                              output logic [2:0] hours,
                              output logic en);

      always_comb
      // hours
      begin
            if (buttons == ~4'b0001) hours = 3'd0;
            else if (buttons == ~4'b0010) hours = 3'd1;
            else if (buttons == ~4'b0100) hours = 3'd2;
            else if (buttons == ~4'b1000) hours = 3'd3;
            else hours = 3'd4;
      end

      always_comb
      // enable
      begin
            if (hours == 3'd4) en = 1'b0;
            else en = 1'b1;
      end

endmodule

module waves ( input  logic clk,
               input logic reset,
                              input logic [2:0] wave_height,
                              input logic [4:0] swell_period,
               output logic [4:0] leds,
                              output logic [4:0] wave_enable);
      panel_decoder panel_decoder1( wave_height, wave_enable);

      bit_transmitter bit_transmitter1(clk, reset, wave_enable[0],
swell_period, leds[0]);
      bit_transmitter bit_transmitter2(clk, reset, wave_enable[1],
swell_period, leds[1]);
      bit_transmitter bit_transmitter3(clk, reset, wave_enable[2],
swell_period, leds[2]);
      bit_transmitter bit_transmitter4(clk, reset, wave_enable[3],
swell_period, leds[3]);
      bit_transmitter bit_transmitter5(clk, reset, wave_enable[4],
swell_period, leds[4]);


endmodule

module panel_decoder(input logic [2:0] wave_height,
                              output logic [4:0] wave_enable);

      always_comb
      if (wave_height < 3'd2) wave_enable = 5'b00001;
      else if (wave_height == 3'd2) wave_enable = 5'b00011;
      else if (wave_height == 3'd3) wave_enable = 5'b00111;
      else if (wave_height == 3'd4) wave_enable = 5'b01111;
      else wave_enable = 5'b11111;
endmodule


module bit_transmitter (input logic clk, reset, wave_enable,
                                                input logic [4:0]
swell_period,
                                    output logic out);


      typedef enum logic [1:0] {WAITING, WRITING, DELAY ,DONE} statetype;
      statetype state, next_state;


      logic [5:0] bit_counter,led_num, wave_num, counter, next_counter,
low_level;
      logic [6:0] bit_index;
      logic [7:0] total_leds;
      logic [22:0] delay_counter;
      logic [23:0] curr_color;
      logic [24:0] color [23:0]; // 25 leds with 24 bits of color for each
one
      logic [25:0] delay_timer;
      logic increment;
      logic [23:0] dark, medium_dark, teal, medium_light, light,
very_light;


      assign dark = 24'h1f001f;
      assign medium_dark = 24'h921156;
      assign teal = 24'hab0047;
      assign medium_light = 24'he9449c;
      assign light = 24'hffddef;
      assign very_light = 24'hffffff;

      assign color[0] = dark;
      assign color[1] = dark;
      assign color[2] = dark;
      assign color[3] = dark;
      assign color[4] = medium_dark;
      assign color[5] = medium_dark;
      assign color[6] = medium_dark;
      assign color[7] = medium_dark;
      assign color[8] = teal;
      assign color[9] = teal;
      assign color[10] = teal;
      assign color[11] = teal;
      assign color[12] = medium_light;
      assign color[13] = medium_light;
      assign color[14] = medium_light;
      assign color[15] = medium_light;
      assign color[16] = light;
      assign color[17] = light;
      assign color[18] = light;
      assign color[19] = light;
      assign color[20] = very_light;
      assign color[21] = very_light;
      assign color[22] = very_light;
      assign color[23] = very_light;

      // assign curr_color = increment ? 24'b000111110000000000011111 :
```

```
24'b111111111111111111111111;
      assign curr_color = wave_enable ? color[led_num] :
24'b000000000000000000000000;
// assign curr_color = wave_enable ? teal : 24'b000000000000000000000000;
//    assign color = wave_enable ? ((increment ?
24'b000111110000000000011111 : 24'b111111111111111111111111)) :
24'b000000000000000000000000 ;
      assign total_leds = 8'd24;
      assign curr_bit = (led_num < wave_num) ? curr_color[bit_counter] :0 ;
//color[bit_index];
//    assign curr_bit = (led_num < wave_num) ? color[bit_counter] :0 ;
//color[bit_index];
      assign low_level = curr_bit ? 6'd4 : 6'd9;
      assign out = ((state == WRITING)) ? (counter > low_level) : 1'b0;
      assign delay_timer =  swell_period*12000000/48 -7488;


      always_ff @(posedge clk, posedge reset)

            if (reset) begin
                  state <= WAITING;
                  led_num <=0;
                  wave_num=1;
                  bit_counter <=0;
                  bit_index <= 0;
                  delay_counter <=0;
                  increment <=1;
            end

            else
            begin

                  state <= next_state;
                  counter <= next_counter;

                  if (bit_counter==6'd24)  led_num = led_num +1'b1;
                  else if (state == DELAY) led_num = 0;
                  else led_num = led_num;

                  if (state == DELAY & next_state == WAITING) begin
                  if (increment) wave_num = wave_num +1'b1;
                  else wave_num = wave_num - 1'b1;
                  end
                  else wave_num = wave_num;

                  if (bit_counter == 6'd24 | state == DELAY) bit_counter =
6'd0;
                  else if (state === WAITING) bit_counter = bit_counter +
1'b1;
                  else bit_counter = bit_counter;

                  if (state == DELAY) bit_index = 0;
                  else if (state == WAITING) bit_index = bit_index +1;
                  else bit_index = bit_index;

                  if (state == DELAY) delay_counter = delay_counter +1;
                  else if (state == WAITING) delay_counter = 0;
                  else delay_counter = delay_counter;

                  if(wave_num == total_leds) increment = 0;
                  else if (wave_num == 'd0) increment = 1;
                  else increment = increment;
            end


      always_comb
            case (state)
                  WAITING:
                  begin
                  if (led_num==total_leds) begin
                  next_state = DELAY;
                  next_counter = 6'd0;
                  end
                  else begin
                  next_state = WRITING;
                  next_counter = 6'd13;
                  end
                  end

                  WRITING:
                  if (counter == 0) begin
                  next_state = WAITING;
                  next_counter = 6'd0;
                  end else begin
                  next_state = WRITING;
                  next_counter = counter - 6'd1;
```

```
                    end

                    DELAY:
                    if (delay_counter > delay_timer) begin // 7m is 17.4s for 
30 leds
                    next_state = WAITING;
                    next_counter = 6'd0;
                    end
                    else begin
                    next_state = DELAY;
                    next_counter = 6'd0;
                    end

                    DONE: begin
                    next_state = DONE;
                    next_counter = 6'd0;
                    end
            endcase

endmodule
```

Software

main.c

```
/**
   Main
   @file
   @author Shreya Sanghai and Lauren Le
   @version 1.0 12/10/2021
*/
#include <stdio.h>
#include "STM32F401RE.h"

////////////////////////////////////////////////
// Constants
////////////////////////////////////////////////

#define LOAD_PIN    5 //PB
#define DONE_PIN    3 //PB
#define BUTTON_0    9 //PA
```

```
#define BUTTON_1    10 //PA
#define MCK_FREQ 100000
#define TRACK1 4 //PA
#define TRACK2 0 // PA
#define TRACK3 1 // PA
#define TRACK4 8 // PA

char wave_height[9] = {0x04, 0x03, 0x02, 0x00, 0x05,0x04, 0x03, 0x02, 0x03};
char swell_period[9] = {0x0A,0x14, 0x05, 0x10, 0x01, 0x03, 0x05, 0x02, 0x1F};
char weather[9] = {0x01, 0x02, 0x02, 0x03, 0x03, 0x04, 0x04, 0x04, 0x04};

int getHour(){
 int hours;
 volatile int bit_zero = digitalRead(GPIOA, BUTTON_0);
 volatile int bit_one = digitalRead(GPIOA, BUTTON_1);
 if (bit_zero && bit_one) hours = 8;
 else if (bit_one) hours = 4;
 else if (bit_zero) hours = 2;
 else hours = 0;

 return hours;


}

void playTrack1(){

 pinMode(GPIOA, TRACK1, GPIO_OUTPUT);
 pinMode(GPIOA, TRACK2, GPIO_OUTPUT);
 pinMode(GPIOA, TRACK3, GPIO_OUTPUT);
 pinMode(GPIOA, TRACK4, GPIO_OUTPUT);

 digitalWrite(GPIOA,TRACK1, 0);
 digitalWrite(GPIOA,TRACK2, 0);
 digitalWrite(GPIOA,TRACK3, 0);
 digitalWrite(GPIOA,TRACK4, 0);

}

void playTrack2(){
 pinMode(GPIOA, TRACK1, GPIO_INPUT);
 pinMode(GPIOA, TRACK2, GPIO_OUTPUT);
 pinMode(GPIOA, TRACK3, GPIO_OUTPUT);
 pinMode(GPIOA, TRACK4, GPIO_OUTPUT);

 digitalWrite(GPIOA,TRACK2, 0);
 digitalWrite(GPIOA,TRACK3, 0);
 digitalWrite(GPIOA,TRACK4, 0);
}

void playTrack3(){
 pinMode(GPIOA, TRACK1, GPIO_INPUT);
 pinMode(GPIOA, TRACK2, GPIO_INPUT);
 pinMode(GPIOA, TRACK3, GPIO_OUTPUT);
 pinMode(GPIOA, TRACK4, GPIO_OUTPUT);

 digitalWrite(GPIOA,TRACK3, 0);
 digitalWrite(GPIOA,TRACK4, 0);

}

void playTrack4(){
 pinMode(GPIOA, TRACK1, GPIO_INPUT);
 pinMode(GPIOA, TRACK2, GPIO_INPUT);
 pinMode(GPIOA, TRACK3, GPIO_INPUT);
 pinMode(GPIOA, TRACK4, GPIO_OUTPUT);
 digitalWrite(GPIOA,TRACK4, 0);
}



////////////////////////////////////////////////
// Main
////////////////////////////////////////////////

int main(void) {

 // Configure flash latency and set clock to run at 84 MHz
 configureFlash();
 configureClock();

 // Enable GPIOA clock
 RCC->AHB1ENR.GPIOAEN = 1;
 RCC->AHB1ENR.GPIOBEN = 1;
 // RCC->AHB1ENR.GPIOCEN = 1;
 // "clock divide" = master clock frequency / desired baud rate
 // the phase for the SPI clock is 1 and the polarity is 0
 spiInit(1, 0, 0);


 // // Load and done pins
 pinMode(GPIOB, LOAD_PIN, GPIO_OUTPUT);
 pinMode(GPIOB, DONE_PIN, GPIO_INPUT);
 pinMode(GPIOA, BUTTON_0, GPIO_INPUT);
 pinMode(GPIOA, BUTTON_1, GPIO_INPUT);

 volatile int hour, curr_hour;
 hour = 0;

 while(1) {

 curr_hour = getHour();

 if (hour != curr_hour) {

   uint8_t dataToSend;
   dataToSend = (swell_period[curr_hour] << 3) + wave_height[curr_hour];

   digitalWrite(GPIOB, LOAD_PIN, 1);
   spiSendReceive(dataToSend);
   while(SPI1->SR.BSY); // Confirm all SPI transactions are completed
   digitalWrite(GPIOB, LOAD_PIN, 0);

   if (weather[curr_hour] == 0x01) { playTrack1();}
   else if (weather[curr_hour] == 0x02) { playTrack2();}
   else if (weather[curr_hour] == 0x03) { playTrack3();}
   else {playTrack4();}
   }


 hour = curr_hour;
 }
}
```

SPI.c

```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, 6, GPIO_OUTPUT); // PB6, 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) {
   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
   volatile uint8_t rec = SPI1->DR.DR;
   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;
}
```

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);

#endif
```

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;
   }
}

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);
}
```

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

// Pin definitions for every GPIO pin
#define GPIO_PA0    0
#define GPIO_PA1    1
#define GPIO_PA2    2
#define GPIO_PA3    3
#define GPIO_PA4    4
```

```c
#define GPIO_PA5    5
#define GPIO_PA6    6
#define GPIO_PA7    7
#define GPIO_PA8    8
#define GPIO_PA9    9
#define GPIO_PA10   10
#define GPIO_PA11   11
#define GPIO_PA12   12
#define GPIO_PA13   13
#define GPIO_PA14   14
#define GPIO_PA15   15

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

// 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 togglePin(GPIO_TypeDef *, int pin);

#endif
```

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
}
```

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 structs
///////////////////////////////////////////////////////////////////////////////
```

```
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
```

RCC.c

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

#include "STM32F401RE_RCC.h"

void configurePLL() {
   // Set clock to 84 MHz
   // Output freq = (src_clk) * (N/M) / P
   // (8 MHz) * (336/8) / 4 = 84 MHz
   // M:8, N:336, P:4
   // 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; // divide by 4

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

void configureClock(){
   /* Configure APB prescalers to be ready for 84 MHz operation
       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;
}
```

RCC.h

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

```
#ifndef STM32F4_RCC_H
#define STM32F4_RCC_H

#include <stdint.h>

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

// Global defines related to clock
uint32_t SystemCoreClock;    // Updated by configureClock()
#define HSE_VALUE 8000000    // Value of external input to OSC from ST-LINK

#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

///////////////////////////////////////////////////////////////////////////////
// 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.h

```
// STM32F401RE.h
// Header to include all other STM32F401RE libraries.

#ifndef STM32F4_H
#define STM32F4_H

#include <stdint.h>

// Include other peripheral libraries

#include "STM32F401RE_GPIO.h"
#include "STM32F401RE_FLASH.h"
#include "STM32F401RE_RCC.h"
#include "STM32F401RE_SPI.h"


#endif
```

Python Script for Ocean Data

queryOceanData-New-Coordinate.py

```
# Website for Marine Data: https://stormglass.io/
# API key:
e5275c9c-4ce7-11ec-ba81-0242ac130002-e5275d5a-4ce7-11ec-ba81-0242ac130002

import requests
import arrow
import json
import numpy as np

LATITUDE = 33.65550
LONGTITUDE = -118.00710

response = requests.get(
  'https://api.stormglass.io/v2/weather/point',
  params={
    'lat': LATITUDE,
    'lng': LONGTITUDE,
    'params': ','.join([
        'waveHeight',
        'wavePeriod',
        'waveDirection',
        'airTemperature',
        'waterTemperature',
        'swellPeriod',
        'gust',
        'cloudCover'
    ]),
  },
  headers={
    'Authorization':
'e5275c9c-4ce7-11ec-ba81-0242ac130002-e5275d5a-4ce7-11ec-ba81-0242ac130002'
  }
)

# Do something with response data.
json_data = response.json()
print(json.dumps(json_data, indent=1))
print("There are two keys from the JSON object. 'hours' = actual marine
data 'meta' = info about your actual API request: ",json_data.keys())
print("Here is the meta data: \n", json.dumps(json_data["meta"], indent=1))
print("Example of data getting data first from: ",
json_data['hours'][0]['time'])

def printData(param):
    dataSource = 'meteo'
    metersToFeet = 3.28084
    for thing in json_data['hours']:
        print("Time: " + str(thing["time"]) + str(": "), param + str(": ")
+ str(int(thing[param][dataSource]*metersToFeet)))

def printTemperature(param):
    dataArray = []
    dataSource = 'noaa'
    for thing in json_data['hours']:
        dataArray = dataArray + [thing[param][dataSource]*(9.0/5.0)+32]
        print("Time: " + str(thing["time"]) + str(": "), param + str(": ")
+ str((thing[param][dataSource]*(9.0/5.0)+32)))
```

```
    return dataArray

temperature = printTemperature('waterTemperature')
temperature = np.array(temperature)
print(np.ceil(temperature).astype(int))
printData('swellPeriod')
printData('waveHeight')
```