LCD Spectrum Visualizer

Final Project Report

Team Members: Matthew Crane, Alex Chang
Submission Date: December 13, 2019

ABSTRACT

The team attempted to construct an LCD spectrum visualizer, which would show the magnitude of the 32-point FFT of live audio data from a microphone. While the team was able to complete most aspects of the project and confirm they were working as intended (display output, SPI between ATSAM and display, SPI between ATSAM and FPGA, and the Address Generation Unit on the FPGA), due to the failure of the core butterfly unit module for the FFT module, the FFT did not output the expected values. This, in turn, prevented the display from showing the correct spectrum for the microphone input.

TABLE OF CONTENTS

1. Introduction

This goal of this project is to create a live audio spectrum visualizer, where the spectrum is the magnitude of the Fast Fourier Transform (FFT) of live audio from a microphone. The specifications of the proposed project are as follows:

- The ATSAM4S4B chip shall sample live audio voltage data off of the microphone at a rate of 2000 Hz, allowing the observation of frequencies of up to 1000 Hz;
- The Altera Cyclone EP4CE6 FPGA shall perform a hardware-accelerated 32-point FFT;
- The LCD screen shall be separated into 16 bars, each corresponding to the magnitudes of the 16 bins from the 32-point FFT, spanning 62.5 Hz each;
- The ATSAM chip shall be able to send, over SPI, the audio data to the FPGA and display data to the LCD display chip;
- The FPGA shall be able to send, over SPI, the FFT of the audio data to the ATSAM chip.

The signal flow is visualized in Figure 1.

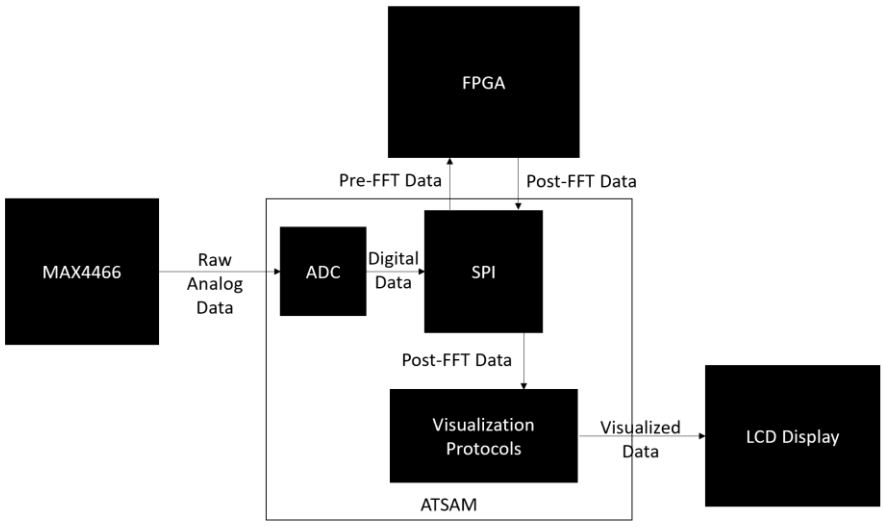

Figure 1: Block Diagram of Spectrum Visualizer

Although we did not fully complete the objectives, this report will describe in detail the design of each of the above parts and how to fit them together. Following that will be a discussion of the results we were able to achieve and what we would do next if there were more time.

2. New Hardware

This project introduces two new pieces of hardware to the system, the MAX4466 Electret Microphone Breakout and the ST7735R LCD Display Breakout.

Table 1. Parts List for the LCD Spectrum Analyzer

Company	Part Name	Cost
Adafruit	MAX4466 - Electret Microphone Amplifier Breakout	$6.95
Adafruit	ST7735R - 1.44" Color TFT LCD Display Breakout	$14.95

See Figure 2 below for the circuit schematic.

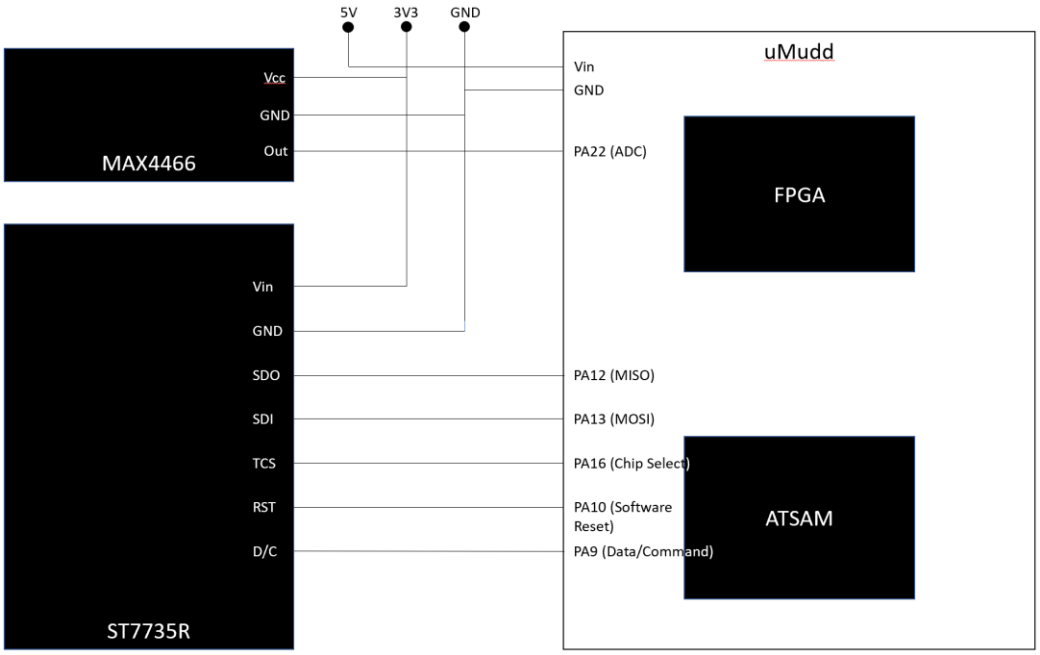

Figure 2: Breadboard Schematic

The MAX4466 is a simple electret microphone connected to a breakout board that applies a gain to it (adjustable via a screw on the back). For this project, the gain was set as high as possible

(125x). We selected Vcc to be 3.3V (offsetting the output voltage by Vcc/2) and connected the output pin to PA22/ADC9 of the ATSAM chip.

The ST7735R is a display chip connected to a 128x128 TFT LCD screen, capable of being written to over SPI. For SPI, the display chip must take an 8-bit command input, and then as many subsequent 8-bit data parameters as necessary for that specific command. The screen is powered with 3.3V, with the standard SPI pins (SI, SCK) being connected to the corresponding pins on the ATSAM (PA13/MOSI, PA14/SCK, respectively). We did not connect SO to the ATSAM since we did not need to read data from the display chip.

Pin TCS is the display chip select, which is enabled when pulled low. Pin RST performs a software reset on the screen when pulled low. Pin D/C indicates to the chip whether or not the incoming serial information is data (high) or a command (low). The other pins are unconnected.

3. FPGA Design

The FPGA was to receive audio voltage data, normalized to 1 and expressed in 16-bit Q5.10 format (MSB is the sign bit, and following bits range from 2^4 to 2^{-10}). Then, it will perform a radix-2 FFT on the data, and then send it back to the ATSAM for processing and display. This output will be 32 bits, with bits 31:16 as real and 15:0 as imaginary, both in Q5.10 format. There are two main modules: SPI and FFT. Figure 3 shows the block diagram of the implementation.

3.1. SPI Module [module RAM_SPI]

First of all, the FFT requires that the data be loaded into the RAM (we chose to have data initially loaded into both RAMs) in a bit-reversed order – for example, data meant to be stored in address 1 (00001) is instead stored in address 16 (10000). This step essentially allows the final

processed FFT data to be read normally. Thus, there is a control signal to force address bit-reversal when data is being loaded into the FPGA, but not when data is being sent out from it.

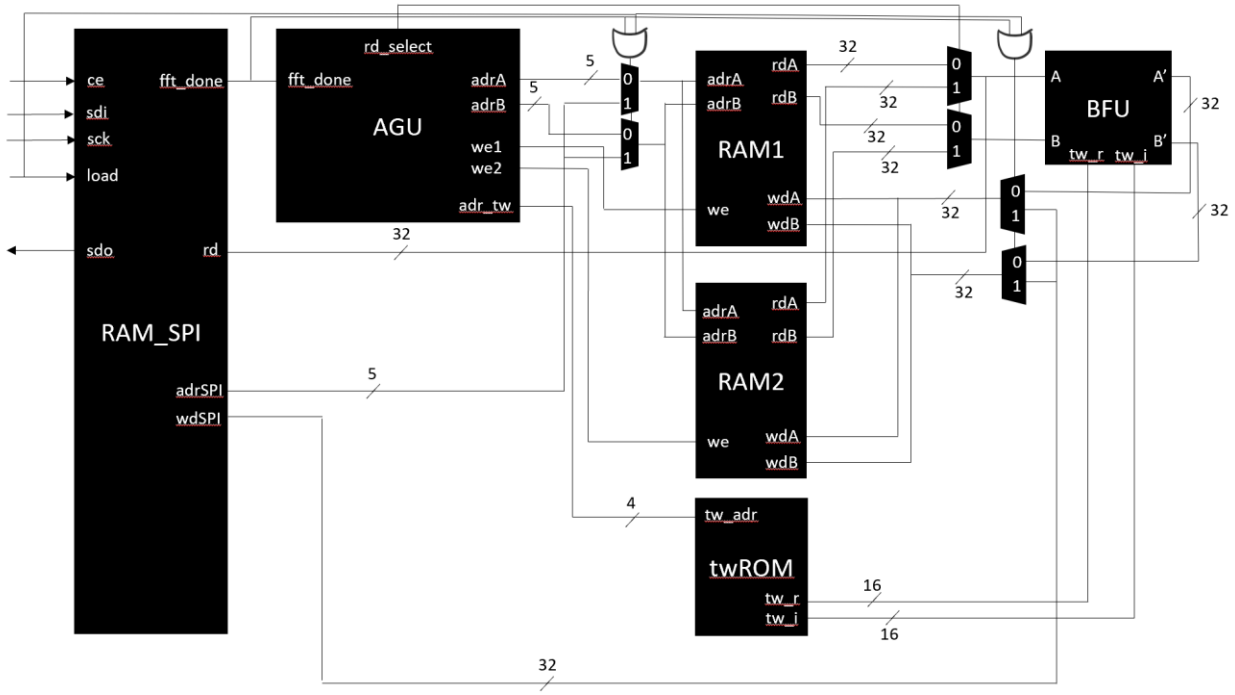

Figure 3: Full Diagram of FFT Implementation with SPI

There also need to be control signals to prevent the FFT module from running during SPI, as well as to override any assertions of addresses or write data from the FFT module (using muxes). We thus have the LOAD and FFT_DONE signals. On LOAD, the modules reset and the SPI module starts shifting data into the RAM. Once done, it deasserts LOAD, and waits for the FFT_DONE signal from the FFT module. When FFT_DONE is asserted, the SPI module begins to shift data out (once again overriding any signals to the RAM from the FFT module).

The shift register used is 48-bits long, for the 32-bit output and 16-bit input. The SPI module automatically generates an address to the RAM, such that, every 16 bits (during read-in) or 32 bits (during read-out), the address increments, writing to or reading from the next word in RAM.

We also have a signal CE, which must be pulled high for the FPGA to register input data.

3.2. FFT Module [module fftTop]

See Appendix A for the SystemVerilog code loaded onto the FPGA.

We decided to implement the FFT described by George Slade [1], leaving out the pipelining present in the paper for simplicity (and general lack of need of such efficiency). The FFT is made up of two two-port RAM units, a twiddle ROM, the Butterfly Unit (BFU), and the Address Generation Unit (AGU).

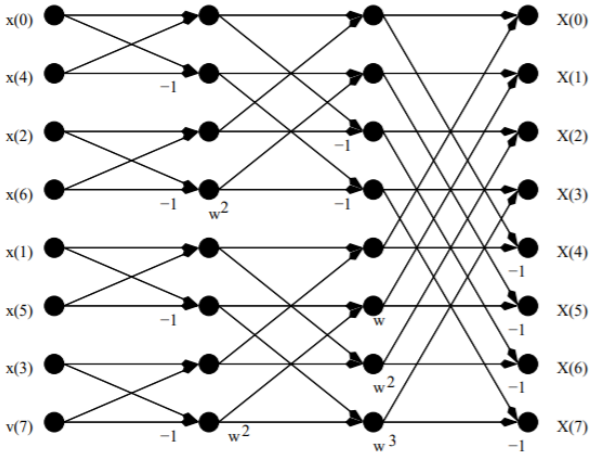

Figure 4: An 8-point Radix-2 FFT as described by George Slade. Each point is a BFU. [1]

3.2.1. Two-Port RAM Units [module RAM2]

The two-port RAM units allow for simultaneous read/writes into two different addresses. They already exist on this FPGA as-is, so no further synthesis was required in terms of logic elements. Each address of the RAM stores a 32-bit word, for 16 bits of real data (MSB) and 16 bits of imaginary data (LSB).

3.2.2. Twiddle ROM [modules twROM_Re, twROM_Im]

The twiddle factors (also known as "roots-of-unity") for a FFT are defined by Slade [1] as:

$$w^n = e^{-\frac{j2\pi n}{N}} \quad (1)$$

where N is the number of points in the FFT. The FFT requires twiddle factors for $n \in [0, N/2)$, since, on the final BFU level (described in Section 3.2.4), one twiddle factor is used per pair of data. As seen on Figure 4, twiddle factors labeled up to w^3) were used on the final level, a total of 4 twiddle factors used for an 8-point FFT. We implemented the twiddle factors using a ROM, where we hard-coded the twiddle factors in Q5.10 format (for consistency with the input data). This is the fastest way for the FFT to retrieve twiddle factors, since it involves no calculation, unlike the CORDIC method. For a 32-point FFT, we simply took the Q15 values from Table 2 of Slade's paper [1]. For other values of N, a twiddle factor generator (and translator to Q5.10 format) would be necessary.

3.2.3. Butterfly Unit [module BFU]

The butterfly unit takes in two complex numbers and performs a calculation as shown below:

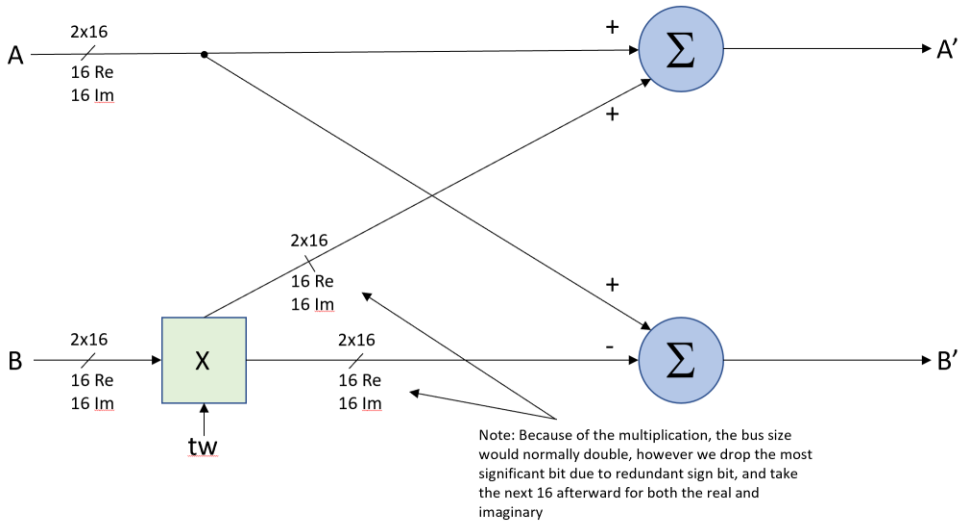

Figure 5: Butterfly Unit Diagram

Mathematically described, this operation would be:

$$\begin{cases} A' = A + B * (tw) \\ B' = A - B * (tw) \end{cases} \quad (2)$$

where A and B are the current pair of complex data fed into the BFU, and tw is the corresponding twiddle factor for this pair, obtained from the twiddle ROMs. Thus, this complex arithmetic expands to the following (where the subscripts r means real and i means imaginary):

$$\begin{cases} A'_r = A_r + T_r \\ A'_i = A_i + T_i \\ B'_r = A_r - T_r \\ B'_i = A_i - T_i \end{cases} \quad \begin{array}{l} \\ \\ T_r = B_r * (tw)_r - B_i * (tw)_i \\ T_i = B_r * (tw)_i + B_i * (tw)_r \end{array} \quad (3)$$

Note that additional processing is required for T_r and T_i. Multiplication between two 16-bit numbers results in a 32-bit number – thus, T_r and T_i each need to be truncated to 16 bits. Binary arithmetic between two Q15 numbers causes an extra sign bit to appear in the MSB of the final 32-bit number – thus, the slice should be from [30:15] for Q15 multiplication. However, since we are using Q5.10, the slice should instead be [25:10] – moved down 5 bits due to the 5 integer bits present in the Q5.10 format.

3.2.4. Address Generation Unit [module AGU]

The AGU serves as a control module for the entire FFT process, generating the addresses for the RAMs and the twiddle ROM, as well as asserting enable and selection signals.

First, the AGU utilizes two counters: one to keep track of the current FFT level (represented as $l = [0, \log_2 N)$) and one to keep track of the current pair number (represented as $p = [0, N/2)$). The FFT level is, essentially, the number of times pairs of data must be processed until the output is the proper FFT output. Figure 4, for an 8-point FFT, shows 3 levels of calculations. The pair number simply keeps track of how many BFU calculations have occurred in the current level. For example, pair number 15 would be the final BFU calculation for a 32-bit FFT level.

These two values – l and p – are necessary to obtain the correct RAM addresses for each BFU calculation, since the data that needs to be used varies by level and pair. The equations for the address of A and B for the BFU are as follows:

$$\begin{cases} \text{adrA} = p + 2^l \left\lfloor \frac{p}{2^l} \right\rfloor \\ \text{adrB} = A + 2^l \end{cases} \tag{4}$$

The floor function can functionally be ignored; binary division will automatically truncate the output as necessary.

The twiddle address has a similar equation, as follows:

$$\text{twAdr} = 2^{L-l} \left\lfloor \frac{p}{2^{L-l}} \right\rfloor \tag{5}$$

where L is the maximum level possible for the N-point FFT ($L = 4$ for our 32-point FFT).

The AGU also asserts the rd_select signal, which selects which RAM is being read for the current level. The reason two RAMs are required is that the FPGA only supports two-port RAM – so, one RAM will hold A and B (to be read into the BFU), and the other will have A' and B' written to it. The rd_select signal simply alternates between levels, starting on the RAM1.

Finally, the AGU also asserts the write-enables for each RAM. While the FFT is not done processing, this is essentially !rd_select – whichever RAM is not being read. When the FFT is done processing, both write-enables are disabled to prevent data corruption.

4. Microcontroller Design

See Appendices B and C for the C code loaded onto the ATSAM.

We utilized the given libraries for the ATSAM in Lab 7 – for this project, the relevant libraries (aside from initialization functions) would be the SPI (data transfer), ADC (microphone read),

and TC (delay) libraries. We set the ADC to 10-bit resolution to ensure enough room for bit growth from the FFT. We also needed to translate the Arduino ST7735R library into C, as shown in Appendix C.

The ATSAM routine is very simple. The screen initializes with a black background. Then, it enters a loop of:

- 'Erasing' (i.e: painting black) the previously displayed spectrum from the display;
- Reading microphone if the last read was at least 1/2000 of a second ago (where the new point of the microphone data pushes out the oldest point, i.e: FIFO);
- Requesting an FFT of the current microphone data;
- Converting the FFT data into bar heights to send to the display chip.
- 'Writing' (i.e: painting white) the spectrum to display.

5. Results

Ultimately, the project was not successful. In terms of successes, the LCD screen was separated into 16 bars corresponding to the output of the FPGA (confirmed visually), and the ATSAM chip was able to correctly communicate over SPI with the display chip (confirmed visually) and FPGA (confirmed for MOSI by checking the ATSAM memory, the SPI on the logic analyzer, and the subsequent real portion of RAM (which was extracted out to 16 LEDs), and vice versa for the MISO). The microphone appeared to be sampling properly, being able to show tones of up to 800 Hz from the microphone, though we did not experiment further due to more pressing issues. We are suspicious that the microphone is not sampling at a consistent rate due to the various delays caused by multiple SPI transactions between the ATSAM and FPGA/ST7735 chip, and that a 2000 Hz sampling rate only permits 500 microseconds between samples. That

being said, the microphone does work, as does the display and SPI. This inability to delay the loop also prevents us from controlling the refresh rate on the display, causing it to flicker.

The most significant issue was the FFT module on the FPGA. While we were able to confirm that the AGU was generating the correct addresses for every pair and level in the simulation, we were not able to get the correct outputs as dictated in Table 5 of Slade's paper [1]. We narrowed down the issue to the BFU and twiddle factor ROMs, since they were the only modules that could have caused a calculation issue aside from the AGU and SPI modules, which we confirmed to be working. We heavily suspect that our use of the Q5.10 format (which we selected since Slade mentioned "5 extra bits" [1]) made BFU calculation incorrect, due to potential complications of having both integer and fractional bits.

In short, if the BFU worked as intended, it is very likely the project would have succeeded.

For the future, it would be best to use Q15 format over Q5.10, since there is no need to worry about bit growth for numbers less than or equal to 1 in decimal, and we already went through the trouble of normalizing the input data to 1. Furthermore, we would not have to convert the twiddle factors from Slade's paper, which reduces complications. For the minor issue of microphone sampling, we suggest using the ATSAM's PDC register to sample the mic in parallel with the rest of the code, allowing for the control of the screen's refresh rate. We would also suggest using an FSM for the FPGA implementation of the SPI module, for simplicity.

6. References

[1] Slade, George William. "The Fast Fourier Transform in Hardware: A Tutorial Based on an FPGA Implementation." ResearchGate, Mar. 2013, web.mit.edu/6.111/www/f2017/handouts/FFTtutorial121102.pdf.

7. Appendix

7.1. SystemVerilog Code (fftTop.sv)

```
/* Define:
 * - N = 32: The number of points in the FFT.
 * - R = 16: The bit resolution of each point in the FFT.
 */

// ATSAM asserts load when it is sending data.
// ATSAM asserts ce when it is listening (regardless of sending or receiving data).
// SDI, SDO are expected to be MSB.
// SPI mode is expected to be 0:
//  - CPOL = 0: Idles at 0, cycles with pulse 1; leading edge is rising
//  - CPHA = 0: Data is sampled (SDI) at the leading edge, and shifted (SDO) on the falling edge.

module fftTop (input  logic clk, load,
               input  logic ce, sck, sdi,
               output logic fft_done, sdo);
  // First, set up the RAM.
  // Each RAM takes in a (clk, we, adr_A, adr_B, wd_A, wd_B), and outputs (rd_A, rd_B).
  //    - The SPI module must be able to write to (adr_A, wd_A), and tap into (rd_A).
  //      It must also know which RAM to read from (i.e: it must tap into rd_select).
  //    - The AGU directly controls (we, adr_A, adr_B), as well as (rd_select).
  //    - The BFU and combine modules directly control (wd_A, wd_B).
  // This means a few things.
  // (1) The SPI module does not have to worry about (we) or (rd_select), which are controlled by
  //     the AGU. Assume that the AGU always tells the SPI module where to read and write, correctly.
  // (2) There are some write conflicts here that need to be settled by muxes, based on whether
  //     or not the SPI module is supposed to be loading in data (based on load).
  //     These conflicts, specifically, are on:
  //        - adr_A: AGU, SPI
  //        - wd_A:  BFU/combine, SPI

  logic        we1, we2, rd_select;
  logic [4:0]  adr_A, adr_B;    logic [4:0]  adrAGU_A, adrAGU_B, adrSPI;
  logic [31:0] wd_A, wd_B;      logic [31:0] wdBFU_A,  wdBFU_B,  wdSPI;
  logic [31:0] rd_A, rd_B;      logic [31:0] rd1_A, rd1_B, rd2_A, rd2_B;

  logic [3:0]  adr_tw;  // size is N/2
  logic [15:0] tw_Re, tw_Im;

  // Set up the RAMs.
  RAM2        ram1    (clk,we1,adr_A,adr_B,wd_A,wd_B , rd1_A,rd1_B);
  RAM2        ram2    (clk,we2,adr_A,adr_B,wd_A,wd_B , rd2_A,rd2_B);

  // Figure out which RAM you're reading from.
  mux2 #(32)  selA    (rd1_A,rd2_A,rd_select , rd_A);  // read from RAM1 if !rd_select
  mux2 #(32)  selB    (rd1_B,rd2_B,rd_select , rd_B);  // read from RAM1 if !rd_select

  // Instantiate modules that write to adr_A (not SPI).
  AGU         agu     (clk,load,
                       adrAGU_A,adrAGU_B,adr_tw,                          // note the adrAGU here
                       rd_select,we1,we2,
                       fft_done);

  // Instantiate modules that write to wd_A.
  BFU         bfu     (tw_Re,tw_Im,rd_A[31:16],rd_B[31:16],rd_A[15:0],rd_B[15:0],
                       wdBFU_A,wdBFU_B);                                  // note the wdBFU here

  // Instantiate SPI, which writes both to adr_A and wd_A.
  RAM_SPI     spi     (ce,clk,sck,sdi,rd_A,load,fft_done,
                               sdo,adrSPI,wdSPI);                         // note the adr/wdSPI here

  // Select the correct address between the AGU and SPI, depending on (load).
  mux2 #(5)   selAdrA (adrAGU_A,adrSPI,(load|fft_done) , adr_A); // use AGU if !load
  mux2 #(5)   selAdrB (adrAGU_B,adrSPI,(load|fft_done) , adr_B);
  // Select the correct write data between the BFU and SPI, depending on (load).
  mux2 #(32)  selWdA  (wdBFU_A, wdSPI, (load|fft_done) , wd_A);  // use BFU if !load
  mux2 #(32)  selWdB  (wdBFU_B, wdSPI, (load|fft_done) , wd_B);

  // Miscellaneous independent modules:
  twROM_Re      w_Re(adr_tw , tw_Re);
  twROM_Im      w_Im(adr_tw , tw_Im);
endmodule



module RAM_SPI (input  logic        ce, clk, sck,
                input  logic        sdi,
                input  logic [31:0] rd,
                input  logic        load,
                input  logic        fft_done,
                output logic        sdo,
                output logic [4:0]  adrSPI,
                output logic [31:0] wdSPI);
```

```
  logic       en_reverse;
  logic [4:0] adr;

  // only reverse bits if you're loading in data
  assign en_reverse = load & !fft_done;

  // if input, bit-reverse
  // if output, don't bit-reverse
  bitReverse    br(en_reverse, adr, adrSPI);

  // Logic signals for relevant edges to watch
  logic wasdone, wasload;
  logic done_posedge, load_posedge;
  assign done_posedge = fft_done & !wasdone;
  assign load_posedge = load & !wasload;

  // Logic signals for watching value changes
  logic [4:0] adr_prev;
  logic       adr_changed;
  assign adr_changed = (adr_prev != adr);

  // Logic signals for counters
  logic [3:0] currBit_in ;
  logic [4:0] currBit_out;

  // Misc. logic signals
  logic [31:0] captured_rd = 0;
  logic        sdodelayed;

  // Last 16 bits of wdSPI are 0's (imaginary)
  assign wdSPI[15:0] = 16'b0;

  // We need to load in the first point of the output RAM before first clock cycle
  logic quick_read;
  assign quick_read = fft_done & adr_changed;

  // (1) Capture on leading rising edge
  always_ff @(posedge sck, posedge quick_read) begin
    if (quick_read) begin
      captured_rd <= rd;
    end
    else if (ce) begin
      if (!wasdone) begin // Shift in  16 bits/point: 0th point, MSB to LSB, then 1st point, MSB to LSB... up to 31st
                        {captured_rd, wdSPI[31:16]} <= {captured_rd,       wdSPI[30:16], sdi};
      end
      else begin          // Shift out 32 bits/point: 0th point, MSB to LSB, then 1st point, MSB to LSB... up to 31st
        if (adr_changed)  captured_rd <= rd; // update all 32 bits with new value
        else              {captured_rd, wdSPI[31:16]} <= {captured_rd[30:0], wdSPI[31:16], sdi};
      end
    end
    // just here to prevent latch
    else begin
      captured_rd <= rd;
    end
  end

  // (2a) Shift on trailing falling edge
  always_ff @(negedge sck) begin
    sdodelayed <= captured_rd[30];
    adr_prev <= adr; // adr updates are here instead of on (3) because adr only matters on SCK
  end

  // (2b) Deal with async counters for bit tracking on input and output
  always_ff @(negedge sck, posedge load_posedge, posedge done_posedge) begin
    if (load_posedge | done_posedge) begin
      currBit_in   <= 0;
      currBit_out  <= 0;
      adr          <= 0;
    end
    else begin
      // Freeze the last address so we don't overflow
      if      (adr == 5'd31) adr <= 5'd31;
      // Otherwise, it's okay to increment the address
      else if (load     & (currBit_in  == 4'd15)) adr <= adr + 5'b1;
      else if (fft_done & (currBit_out == 5'd31)) adr <= adr + 5'b1;
      currBit_in  <= currBit_in  + 4'b1;
      currBit_out <= currBit_out + 5'b1;
    end
  end

  // (3) Ensure that the statuses of done and load are being updated at all times, even when SCK isn't running
  always_ff @(posedge clk) begin
    wasdone <= fft_done;
    wasload <= load;
  end

  // Shift out 31st bit (MSB) if we're just starting to shift data out, OR if we changed addresses
  assign sdo = (done_posedge | adr_changed) ? captured_rd[31]:sdodelayed;
endmodule
```

```
// AGU doesn't strictly do anything with load, it just listens to it to see if it
// needs to reset.
module AGU (input  logic       clk, load,
            output logic [4:0] adr_A, adr_B,
            output logic [3:0] adr_tw,
            output logic       rd_select, we1, we2,
            output logic       fft_done);
  logic [3:0] curr_pair;
  logic [3:0] curr_level;
  logic       levelUp;
  logic [3:0] alternate;  // essentially a 1-bit logic element that flips every time level changes
  logic [4:0] curr_powerOf2;
  logic [4:0] curr_powerOf2_inv;

  assign curr_powerOf2     = (5'd2)**curr_level;
  assign curr_powerOf2_inv = (5'd2)**(4-curr_level);
  assign alternate         = curr_level % 4'd2;  // curr_level % 2 is 4 bits, so we store it in alternate

  counter_pair    pairCount (clk,load,levelUp,         curr_pair);
  counter_level   levelCount(clk,load,levelUp,fft_done,curr_level);

  // In theory, (pair)/(2^level) should truncate. e.g: If pair = 7 and 2^level is 8, then
  // it should be 0.
  assign adr_A  = curr_pair + (curr_powerOf2)*(curr_pair/curr_powerOf2);
  assign adr_B  = adr_A + curr_powerOf2;
  assign adr_tw = curr_powerOf2_inv*(curr_pair/curr_powerOf2_inv);

  // Note: rd_select, on fft completion, will be stuck on the RAM that is last written to due to
  //       how the module counter_level freezes on (final level + 1).
  assign rd_select = alternate[0];  // we only care about the LSB
  // if done is asserted, then stop writing
  // if load is asserted, then force we1 to be 1 (allow SPI to write to it)
  // start by reading from RAM1 and writing to RAM2
  assign we1 = load|(!fft_done &  rd_select);  // rd_select starts at 0 since level starts at 0
  assign we2 = !fft_done & !rd_select;
endmodule



/* <BFU>
 * Description:
 *    The 4-input complex butterly unit used for the radix FFT algorithm.
 * Inputs:
 * - A_r       (size R): The first real input to the BFU, NOT to be multiplied by the twiddle factor.
 * - B_r       (size R): The second real input to the BFU, to be multiplied by the real twiddle factor tw_r.
 * - A_i       (size R): The first imaginary input to the BFU, not to be multiplied by the twiddle factor.
 * - B_i       (size R): The second imaginary input to the BFU, to be multiplied by the imaginary twiddle factor tw_i.
 * - tw_r      (size R): The appropriately-selected twiddle factor to multiply B_r with.
 * - tw_i      (size R): The appropriately-selected twiddle factor to multiply B_i with.
 * Outputs:
 * - A_out     (size R): The corresponding output to A, whose value is A_rout concatinated with A_iout.
 * - B_out     (size R): The corresponding output to B, whose value is B_rout concatinated with B_iout.
 * (Note: "Corresponding" means that if A came from the 3rd position in RAM1, then
 *        A_out must go into the 3rd position in RAM2.)
 */
module BFU (input  logic [15:0] tw_r, tw_i,
            input  logic [15:0] A_r, B_r,
            input  logic [15:0] A_i, B_i,
            output logic [31:0] Aout, Bout);
  // Signed logic for signed multiplication
  logic signed [15:0] tw_rS, tw_iS;
  logic signed [15:0] B_rS,  B_iS;
  logic signed [31:0] temp_r, temp_i;
  logic        [15:0] A_rout, B_rout, A_iout, B_iout;

  assign tw_rS = tw_r;
  assign tw_iS = tw_i;
  assign B_rS = B_r;
  assign B_iS = B_i;

  // Compute the BFU output
  assign temp_r = (B_rS*tw_rS) - (B_iS*tw_iS);
  assign temp_i = (B_rS*tw_iS) + (B_iS*tw_rS);

  // We're using Q5.10, and multiplication for Q5.10 requires
  // that (from our findings) the bits from 25:10 are taken, rather
  // than 30:15 (a shift down by 5, which is the number of integer
  // bits we have).
  assign A_rout = A_r + temp_r[25:10];
  assign B_rout = A_r - temp_r[25:10];
  assign A_iout = A_i + temp_i[25:10];
  assign B_iout = A_i - temp_i[25:10];

  // Combine real and imaginary
  assign Aout = {A_rout,A_iout};
  assign Bout = {B_rout,B_iout};
endmodule



/***********************************************
 * Clocks Section
```

```
 ************************************************
 */

/* <counter_pair>
 * Description:
 *    Simple counter that keeps track of how many pairs have gone through the BFU
 *    during one level. When the counter overflows, it will notify the AGU to increment
 *    the level.
 *    Counter must necessarily overflow on N/2.
 *
 */
module counter_pair (input  logic       clk, reset,
                     output logic       levelUp,
                     output logic [3:0] counter);
  always_ff @(posedge clk) begin
    // If load is asserted, then we are calculating a new FFT. Reset.
    if (reset) counter <= 4'b0;
    // If load is not asserted, then calculate as normal.
    else begin
      counter <= counter + 4'b1; //once loaded, begin the counter
      if      (counter == 14) levelUp <= 1'b1;  // prev value is 15 = you overflowed --> level up!
      else                    levelUp <= 1'b0;  // levelUp should not be asserted on non-overflow values
    end
  end
endmodule



/* <counter_level>
 */
module counter_level (input  logic       clk, reset,
                      input  logic       levelUp,
                      output logic       fft_done,
                      output logic [3:0] counter);
  always_ff @(posedge clk) begin
    // If load is asserted, then we are calculating a new FFT. Reset.
    if (reset) begin
      counter <= 4'b0;
      fft_done <= 1'b0;
    end
    // There are 5 layers in a 32-point FFT. Thus, if counter = 5, then
    // we have finished the final layer.
    else if (counter == 4'd5) begin
      counter <= 4'd5;
      fft_done <= 1'b1; // assert done
    end
    else begin
      if (levelUp) counter <= counter + 4'b1;
    end
  end
endmodule



/* <RAM2>
 * Description:
 *    A 2-input RAM.
 * CLOCK BEHAVIOR:
 * - posedge: Writes data to RAM, if applicable.
 * - negedge: No action.
 * Inputs:
 * - we:    Write enable.
 * - adr_A: First address for the RAM to perform a read/write on.
 * - adr_B:
 * (Note: Both adr_A and adr_B must be large enough to store N. For example,
 *        if N = 32, then adr_A/B must be 5 bits long. If N = 64, 6 bits, etc.)
 * - wd_A (size 2R):
 */
module RAM2 (input         clk, we,
             input  [4:0]  adr_A, adr_B,
             input  [31:0] wd_A, wd_B,
             output [31:0] rd_A, rd_B);

  // Instantiate the RAM with N entries of 2R bits.
  // The RAM should store a one-to-one output for an N-point input, and thus
  // it must necessarily have N addresses. For example, if N = 32, then the
  // instantiation must be "... RAM[4:0]", to accomodate 5 bits of addresses.
  logic [31:0] RAM[0:31];

  always_ff @(posedge clk) begin
    if (we) begin       // if write enabled...
      RAM[adr_A] <= wd_A; // write to address A
      RAM[adr_B] <= wd_B; // write to address B
    end
  end

  // constantly read values out of RAM
  assign rd_A = RAM[adr_A];
  assign rd_B = RAM[adr_B];
endmodule
```

```
/* <mux2>
 */
module mux2 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1,
              input  logic             s,
              output logic [WIDTH-1:0] y);
  assign y = s ? d1:d0;
endmodule



/*  <bitReverse>
 * Description:
 *    For 5 bits.
 */
module bitReverse (input  logic       en,
                   input  logic [4:0] in,
                   output logic [4:0] out);
  logic [4:0] n;  // dummy for for-loop
  always_comb begin
    if (en) begin
      for (n = 0; n < 5; n++) begin
        out[n] = in[4-n];
      end
    end
    else begin
      n   = 0;
      out = in;
    end
  end
endmodule

// Note: Twiddles are converted to Q5.10 from Q15, from Slade's paper.

/* <twRom_Re>
 */
module twROM_Re (input  logic [3:0]  adr_tw,
                 output logic [15:0] twiddle);
  always_comb begin
    case(adr_tw)
      4'b0000: twiddle = 16'h0400;
      4'b0001: twiddle = 16'h03EC;
      4'b0010: twiddle = 16'h03B2;
      4'b0011: twiddle = 16'h0353;
      4'b0100: twiddle = 16'h02D4;
      4'b0101: twiddle = 16'h0238;
      4'b0110: twiddle = 16'h0187;
      4'b0111: twiddle = 16'h00C7;
      4'b1000: twiddle = 16'h0000;
      4'b1001: twiddle = 16'hFF39;
      4'b1010: twiddle = 16'hFE79;
      4'b1011: twiddle = 16'hFDC8;
      4'b1100: twiddle = 16'hFD2C;
      4'b1101: twiddle = 16'hFCAD;
      4'b1110: twiddle = 16'hFC4E;
      4'b1111: twiddle = 16'hFC14;
      default: twiddle = 16'h0400;
    endcase
  end
endmodule


/* <twRom_Im>
 */
module twROM_Im (input  logic [3:0]  adr_tw,
                 output logic [15:0] twiddle);
  always_comb begin
    case(adr_tw)
      4'b0000: twiddle = 16'h0000;
      4'b0001: twiddle = 16'h00C7;
      4'b0010: twiddle = 16'h0187;
      4'b0011: twiddle = 16'h0238;
      4'b0100: twiddle = 16'h02D4;
      4'b0101: twiddle = 16'h0353;
      4'b0110: twiddle = 16'h03B2;
      4'b0111: twiddle = 16'h03EC;
      4'b1000: twiddle = 16'h0066;
      4'b1001: twiddle = 16'h03EC;
      4'b1010: twiddle = 16'h03B2;
      4'b1011: twiddle = 16'h0353;
      4'b1100: twiddle = 16'h02D4;
      4'b1101: twiddle = 16'h0238;
      4'b1110: twiddle = 16'h0187;
      4'b1111: twiddle = 16'h00C7;
      default: twiddle = 16'h0000;
    endcase
  end
endmodule
```

7.2. Main ATSAM Behavioral Loop (main.c)

```
#include "SAM4S4B.h"
#include "ST7735.h"
#include <stdlib.h>
#include <math.h>

// The channel we will use for ADC sampling rate
#define ADC_SAMPLE_TC TC_CH1_ID

// FPGA-related pins
#define FPGA_CE   PIO_PA17  // Chip enable pin (1 enable) [OUT to   P58]
#define FPGA_LOAD PIO_PA18  // Load pin (1 enable)        [OUT to   P59]
#define FPGA_DONE PIO_PA19  // Done pin                   [IN  from P60]
// SCK goes to P42
// MOSI/sdi goes to P39
// MISO/sdo goes to P38

// Defining enable level for various FPGA pins
#define FPGA_CE_EN   1
#define FPGA_LOAD_EN 1

// Define ADC pin for microphone usage
#define MIC  ADC_CH9

// Structure to contain FFT values
typedef struct {
  int16_t REAL;
  int16_t IMAG;
} complex16_t;

// Structure to contain normalized FFT values (need decimals)
typedef struct {
  float REAL;
  float IMAG;
} complexf_t;


void init(void) {
  samInit();
  pioInit();
  tcInit();
  tcDelayInit();
  // The ST7735R SPI communication method requires MSB first and SPI mode 0.
  // The ATSAM is already locked to MSB first (see Figures 33-3 and 33-4).
  // SPI mode 0 is analogous to CPOL = 0 and NPCHA = 1 for the ATSAM.
  // Note to self: Figure out a proper frequency for SCLK later.
  spiInit();

  // Setting pinmodes
  pioPinMode(CS,  PIO_OUTPUT);
  pioPinMode(DC,  PIO_OUTPUT);
  pioPinMode(RST, PIO_OUTPUT);
  pioPinMode(FPGA_LOAD, PIO_OUTPUT);
  pioPinMode(FPGA_CE,   PIO_OUTPUT);
  pioPinMode(FPGA_DONE, PIO_INPUT);

  // Setting initial pin values
  pioDigitalWrite(CS,  !CS_EN);   // ensure that the display is not SPI-enabled
  pioDigitalWrite(DC,  PIO_HIGH); // always assume DATA (high) is written unless CMD (low) is specified for write
  pioDigitalWrite(RST, PIO_HIGH); // ensure that RST isn't immediately on
  pioDigitalWrite(FPGA_CE,   !FPGA_CE_EN);
  pioDigitalWrite(FPGA_LOAD, !FPGA_LOAD_EN);

  initR();

  adcInit(ADC_MR_LOWRES_BITS_10);
  adcChannelInit(MIC, ADC_CGR_GAIN_X1, ADC_COR_OFFSET_OFF);

  // Use TIMER_CLOCK1, which is MCK/2 (where MCK = 3578000) --> 1789000 Hz;
  // The counter is 16 bits, which counts up to 65535, which means it caps in 65535/1789000 = 36.6 ms;
  // For a 2000 Hz sampling frequency, the TC would be at:
  // (1/2000 sec)*(1789000 Hz) = 894.5 = ~894
  // Thus, reset if the counter is greater than 894.
  tcChannelInit(ADC_SAMPLE_TC, TC_CLK1_ID, TC_MODE_UP_RC);
  tcSetRC_compare(ADC_SAMPLE_TC, 894);
  // took a million years to figure this out but as it turns out you need to reset the counter first to get it to run
  tcResetChannel(ADC_SAMPLE_TC);

}

// (1) Update mic data from the connected mic using ADC channels.
void update_mic_data(uint16_t mic_data[], int fft_size) {
  if (tcCheckRC_compare(ADC_SAMPLE_TC)) {
    // Shift elements of mic_data to the right by 1 index (kick out the oldest value)
    for (int i = fft_size-1; i > 0; i--) { mic_data[i] = mic_data[i-1]; }
    // Put the newest value of mic_data (direct ADC steps) into the 0th index
    // Basically, the 10-bit ADC goes into a 16-bit point, which is totally fine
    // because we'll send it directly to the FPGA, which will interpret it as a Q5.10 number
    // (which is fine since these numbers can't be negative, so we're technically just
```

```
    // dividing by 2^5).
    mic_data[0] = (uint16_t) adcRead_step(MIC);
    // Reset counter
    tcResetChannel(ADC_SAMPLE_TC);
  }
}

// (2) Convert mic data into fft
void get_fft(complex16_t fft[], uint16_t mic_data[], int fft_size) {
  // Assert LOAD and CE to prime the FPGA
  pioDigitalWrite(FPGA_LOAD, FPGA_LOAD_EN);
  tcDelayMicroseconds(1);
  pioDigitalWrite(FPGA_CE,   FPGA_CE_EN);

  // Send out the data to the FPGA
  int i; uint16_t sdi;
  for (i = 0; i < fft_size; i++) {
    sdi = mic_data[i];
    spiSendReceive16(sdi);
  }
  // Deassert load and wait for FPGA to finish
  pioDigitalWrite(FPGA_LOAD, !FPGA_LOAD_EN);
  while(!pioDigitalRead(FPGA_DONE));
  // Shift data in from FPGA

  for (i = 0; i < fft_size; i++) {
    // Send garbage data and get MSB (real) and LSB (imag) in return
    fft[i].REAL = spiSendReceive16(0xFFFF);
    fft[i].IMAG = spiSendReceive16(0xFFFF);
  }
  // Deassert CE; transaction complete
  tcDelayMicroseconds(1);
  pioDigitalWrite(FPGA_CE,!FPGA_CE_EN);
}

// (3) Normalize the fft
void norm_fft(complexf_t fft_norm[], complex16_t fft[], int fft_size) {
  int i;
  for (i = 0; i < fft_size/2; i++) {
    // Divide by 32768 because the value we have is, despite being a Q5.10
    // number, technically a signed short. Hence, even though the max of
    // a Q5.10 number is 32, we instead divide by the max of a signed short.
    fft_norm[i].REAL = ((float) fft[i].REAL)/32768.0;
    fft_norm[i].IMAG = ((float) fft[i].IMAG)/32768.0;
  }
}

// (4) Calculate the normalized magnitude for each point in the fft
void get_mag(float fft_mag[], complexf_t fft_norm[], int size) {
  // Don't need to worry about overflow or whatnot since we normalized
  // the magnitudes here. However, we do need to divide by root 2
  // since that is the maximum a magnitude can be given both inputs
  // are 1 (sqrt(1 + 1) = sqrt(2)).
  int i; float sqRe; float sqIm;
  for (i = 0; i < size; i++) {
    sqRe = powf(fft_norm[i].REAL,2);
    sqIm = powf(fft_norm[i].IMAG,2);
    // Taking magnitude and dividing by root 2
    fft_mag[i] = sqrt((sqRe + sqIm)/2.0);
  }
}

// (5) Convert the magnitude to a display height on the ST7735
void get_disp(uint16_t fft_disp[], float fft_mag[], int size) {
  int i;
  for (i = 0; i < size; i++) {
    // Magnitude is normalized to 1, just multiply by max height
    fft_disp[i] = (uint16_t) (fft_mag[i]*MAXHEIGHT);
  }
}

int main(void) {
  init();
  draw_cal(); // integrity check
  fillRect(0,0,MAXWIDTH,MAXHEIGHT,RGB565_BLACK); // black background

  int fft_size  = 32;
  int bin_num   = fft_size/2; // number of bins is half the size of fft
  int bin_spc   = 4;          // pixels between bins
  int bin_width = 4;          // pixel width of bins
  // Calculate blank space on the sides
  int edge_spc  = (MAXWIDTH-(bin_num*bin_width + (bin_num-1)*bin_spc))/2;

  // Data structures necessary for processing sound data
  uint16_t*    mic_data = malloc(fft_size   * sizeof(uint16_t));
  complex16_t* fft      = malloc(fft_size   * sizeof(complex16_t));
  complexf_t*  fft_norm = malloc(fft_size/2 * sizeof(complexf_t));
  float*       fft_mag  = malloc(fft_size/2 * sizeof(float));
  uint16_t*    fft_disp = malloc(fft_size/2 * sizeof(uint16_t));

  // Assorted dummy variables
  int i, x, y, w, h;
```

```
  while (1) {
    // erase the old data by painting it with black
    for (i = 0; i < bin_num; i++) {
      x = edge_spc + i*(bin_width + bin_spc);
      y = 0;
      w = bin_width;
      h = fft_disp[i];
      //h = (int) (((double) mic_data[i])/1024.0*128.0);
      fillRect(x,y,w,h,RGB565_BLACK);
    }
    // update mic data
    update_mic_data(mic_data,fft_size);
    // update fft
    get_fft(fft,mic_data,fft_size);
    // normalize fft
    norm_fft(fft_norm,fft,fft_size);
    // update fft_mag (normalized to 1)
    get_mag(fft_mag,fft_norm,fft_size/2);
    // update fft_disp (normalized to 128)
    get_disp(fft_disp,fft_mag,fft_size/2);
    // show the current data by painting over with white
    for (i = 0; i < bin_num; i++) {
      x = edge_spc + i*(bin_width + bin_spc);
      y = 0;
      w = bin_width;
      h = fft_disp[i];
      //h = (int) (((double) mic_data[i])/1024.0*128.0);
      fillRect(x,y,w,h,RGB565_WHITE);
    }
    // Pause for a bit
    //tcDelayMillis(33);
    // We can't pause because we need to sample the mic!
  }
}
```

7.3. Translated ST7735 Library (ST7735.h)

```
#ifndef ST7735_H
#define ST7735_H

// ----------<CODE BEGIN>--------------------------------------------
#include "SAM4S4B.h"

// Datasheet: https://cdn-shop.adafruit.com/datasheets/ST7735R_V0.2.pdf

// See Table 6.2 for a description of pin functions on the ST7735.
#define CS   PIO_PA16   // Chip select pin (0 enable)
#define DC   PIO_PA9    // Display data/command select pin (0 cmd, 1 data/param)
// Note: In the Arduino library, DC is instead RS.
#define RST  PIO_PA10   // Reset pin (0 reset) - req. to properly init chip
// MISO is not required for this write-only display

// initR() initializes the ST7735R series (instead of the B or G series).
// initR() behavior changes based on the tab color of your ST7735's plastic
// wrap; for example, our ST7735 has a green tab, and thus uses INITR_GREENTAB.
// Functionally speaking these macros will not be used; it's just here to
// explain the original C++ library code. We know that we are using a 128 by
// 128 ST7735R with a green tab.
#define INITR_GREENTAB    0x0
#define INITR_REDTAB      0x1
#define INITR_BLACKTAB    0x2

// Our ST7735 has dimensions of 128 by 128.
#define MAXWIDTH   128
#define MAXHEIGHT  128

// System command list as per Table 10.1.1
#define ST7735_NOP        0x00  // No operation
#define ST7735_SWRESET    0x01  // Software reset
#define ST7735_RDDID      0x04  // Read display ID
#define ST7735_RDDST      0x09  // Read display status

#define ST7735_SLPIN      0x10  // Sleep in and booster off
#define ST7735_SLPOUT     0x11  // Sleep out and booster on
#define ST7735_PTLON      0x12  // Partial mode on
#define ST7735_NORON      0x13  // Parital off (normal)

#define ST7735_INVOFF     0x20  // Display inversion off
#define ST7735_INVON      0x21  // Display inversion on
#define ST7735_DISPOFF    0x28  // Display off
#define ST7735_DISPON     0x29  // Display on
#define ST7735_CASET      0x2A  // Column address set
#define ST7735_RASET      0x2B  // Row address set
#define ST7735_RAMWR      0x2C  // Memory write
#define ST7735_RAMRD      0x2E  // Memory read
```

```
#define ST7735_PTLAR      0x30  // Partial start/end address set
#define ST7735_COLMOD     0x3A  // Interface pixel format
#define ST7735_MADCTL     0x36  // Memory data access control

// Panel function command list as per Table 10.2.1
#define ST7735_FRMCTR1    0xB1  // In normal mode (full colors)
#define ST7735_FRMCTR2    0xB2  // In idle mode (8-colors)
#define ST7735_FRMCTR3    0xB3  // In partial mode + full colors
#define ST7735_INVCTR     0xB4  // Display inversion control
#define ST7735_DISSET5    0xB6  //

#define ST7735_PWCTR1     0xC0  // Power control setting
#define ST7735_PWCTR2     0xC1  // Power control setting
#define ST7735_PWCTR3     0xC2  // In normal mode (full colors)
#define ST7735_PWCTR4     0xC3  // In idle mode (8-colors)
#define ST7735_PWCTR5     0xC4  // In partial mode + full colors
#define ST7735_VMCTR1     0xC5  // VCOM control 1

#define ST7735_RDID1      0xDA  // Read ID1
#define ST7735_RDID2      0xDB  // Read ID2
#define ST7735_RDID3      0xDC  // Read ID3
#define ST7735_RDID4      0xDD  // Read ID4

#define ST7735_GMCTRP1    0xE0  // Gamma ('+' polarity) correction chars
#define ST7735_GMCTRN1    0xE1  // Gamma ('-' polarity) correction chars

// Color definitions, 16 bits each --> RGB 5-6-5 allocation (Section 9.8.5)
#define  RGB565_BLACK      0x0000  // (00000,000000,00000)
#define  RGB565_BLUE       0x001F  // (00000,000000,11111)
#define  RGB565_RED        0xF800  // (11111,000000,00000)
#define  RGB565_GREEN      0x07E0  // (00000,111111,00000)
#define  RGB565_CYAN       0x07FF  // (00000,111111,11111)
#define  RGB565_MAGENTA    0xF81F  // (11111,000000,11111)
#define  RGB565_YELLOW     0xFFE0  // (11111,111111,00000)
#define  RGB565_WHITE      0xFFFF  // (11111,111111,11111)

// Used for selecting whether you're sending data or command
#define DATA 0
#define CMD  1

// Define what values enable which slaves
#define CS_EN    0

// For an ST7735R with a green tab, colstart and rowstart are offset
#define COLSTART 2
#define ROWSTART 3

// Imported from Arduino library
#define DELAY 0x80

// Imported from Arduino library (comments included)
const static unsigned char Rcmd[] = {
  21,                       // -----<21 commands total for green tab>-----
                            // -----<15 commands for red/green tab only>-----
  ST7735_SWRESET, 0+DELAY,  //  1: Software reset, 0 args, w/delay
    150,                    //     * 150 ms delay
  ST7735_SLPOUT , 0+DELAY,  //  2: Out of sleep mode, 0 args, w/delay
    255,                    //     * 500 ms delay
  ST7735_FRMCTR1, 3      ,  //  3: Frame rate ctrl - normal mode, 3 args:
    0x01, 0x2C, 0x2D,       //     - Rate = fosc/(1x2+40) * (LINE+2C+2D)
                            //       (Note: fosc = 625 kHz (Section 10.2.1))
  ST7735_FRMCTR2, 3      ,  //  4: Frame rate control - idle mode, 3 args:
    0x01, 0x2C, 0x2D,       //     - Rate = fosc/(1x2+40) * (LINE+2C+2D)
  ST7735_FRMCTR3, 6      ,  //  5: Frame rate ctrl - partial mode, 6 args:
    0x01, 0x2C, 0x2D,       //     - Dot inversion mode
    0x01, 0x2C, 0x2D,       //     - Line inversion mode
  ST7735_INVCTR , 1      ,  //  6: Display inversion ctrl, 1 arg, no delay:
    0x07,                   //     - No inversion
  ST7735_PWCTR1 , 3      ,  //  7: Power control, 3 args, no delay:
    0xA2,                   //
    0x02,                   //     - GVCL = -4.6V
    0x84,                   //     - AUTO mode
  ST7735_PWCTR2 , 1      ,  //  8: Power control, 1 arg, no delay:
    0xC5,                   //     - VGH25 = 2.4C VGSEL = -10 VGH = 3 * AVDD
  ST7735_PWCTR3 , 2      ,  //  9: Power control, 2 args, no delay:
    0x0A,                   //     - Opamp current small
    0x00,                   //     - Boost frequency
  ST7735_PWCTR4 , 2      ,  // 10: Power control, 2 args, no delay:
    0x8A,                   //     - BCLK/2, Opamp current small & Medium low
    0x2A,                   //
  ST7735_PWCTR5 , 2      ,  // 11: Power control, 2 args, no delay:
    0x8A, 0xEE,             //
  ST7735_VMCTR1 , 1      ,  // 12: Power control, 1 arg, no delay:
    0x0E,                   //
  ST7735_INVOFF , 0      ,  // 13: Don't invert display, no args, no delay
  ST7735_MADCTL , 1      ,  // 14: Memory access control (directions), 1 arg:
    0xC8,                   //     - row addr/col addr, bottom to top refresh
  ST7735_COLMOD , 1      ,  // 15: set color mode, 1 arg, no delay:
    0x05                    //     - 16-bit color
                            // -----<2 commands for green tab only>-----
  ST7735_CASET  , 4      ,  //  1: Column addr set, 4 args, no delay:
```

```
    0x00, 0x02           ,  //     - XSTART = 0
    0x00, 0x7F+0x02      ,  //     - XEND = 127
  ST7735_RASET  , 4      ,  //  2: Row addr set, 4 args, no delay:
    0x00, 0x01           ,  //     - XSTART = 0
    0x00, 0x9F+0x01      ,  //     - XEND = 159
                            // -----<4 commands for red/green tab only>-----
  ST7735_GMCTRP1,16      ,  //  1: Magical unicorn dust, 16 args, no delay:
    0x02, 0x1c, 0x07, 0x12, //
    0x37, 0x32, 0x29, 0x2d, //
    0x29, 0x25, 0x2B, 0x39, //
    0x00, 0x01, 0x03, 0x10, //
  ST7735_GMCTRN1,16      ,  //  2: Sparkles and rainbows, 16 args, no delay:
    0x03, 0x1d, 0x07, 0x06, //
    0x2E, 0x2C, 0x29, 0x2D, //
    0x2E, 0x2E, 0x37, 0x3F, //
    0x00, 0x00, 0x02, 0x10, //
  ST7735_NORON  , 0+DELAY,  //  3: Normal display on, no args, w/delay
    10,                     //     * 10 ms delay
  ST7735_DISPON , 0+DELAY,  //  4: Main screen turn on, no args w/delay
    100                     //     * 100 ms delay
};

void resetR(void) {
  // The ST7735R resets on a low signal.
  pioDigitalWrite(RST, PIO_HIGH);
  tcDelayMillis(10);
  pioDigitalWrite(RST, PIO_LOW);
  tcDelayMillis(10);
  pioDigitalWrite(RST, PIO_HIGH);
  tcDelayMillis(10);
}

void writeR(char DC_sel, char send) {
  // NOTE: If you are using a non-peripheral pin as the chip select for SPI
  //       communication, then you must configure that pin as high/low OUTSIDE
  //       of this function. (e.g: setHigh; setLow; writeR; writeR; setHigh;)
  // See Section 9.4.1 for expected behavior for multi-byte writes.
  if (DC_sel == CMD)  { pioDigitalWrite(DC, PIO_LOW); }  // sending CMD - pull low
  spiSendReceive(send);
  if (DC_sel == CMD)  { pioDigitalWrite(DC, PIO_HIGH); } // only pull low while sending CMD
}

void initR(void) {
  resetR(); // before anything else, perform a reset

  char     numComms;  // stores number of commands in Rcmd[]
  // Note: numArgs is in a format such that numArgs[7:4] = DELAY? (where
  //       an additional argument is necessary if DELAY? = 1111) and
  //       numArgs[3:0] is the actual number of parameters.
  //       e.g: numArgs = 1111 0001 will have 1 actual argument and 1 DELAY
  //            argument.
  char     numArgs;   // stores number of args for an individual command
  // Note: numParams will just be numArgs[3:0], effectively.
  char     numParams; // stores number of params for an individual command
  uint16_t addr = 0;  // stores current index in Rcmd[]
  uint16_t ms;        // stores delay necessary after executing command

  numComms = Rcmd[addr++];      // Number of commands to follow
  while(numComms--) {           // For each command...
    // -----<START: SPI Transaction>-----
    pioDigitalWrite(CS,CS_EN); // chip enable
    writeR(CMD,Rcmd[addr++]);     // Issue command
    numArgs   = Rcmd[addr++];     // Number of args to follow
    ms        = numArgs &  DELAY; // Extract delay bits (if no DELAY, ms = 0)
    numParams = numArgs & ~DELAY; // Mask out delay bit to get numParams
    while(numParams--) {          // For each param...
      writeR(DATA,Rcmd[addr++]);    // Issue param
    }
    pioDigitalWrite(CS,!CS_EN); // chip disable
    // -----<END: SPI Transaction>-----
    if(ms) {                    // If we have a delay (i.e: nonzero ms)...
      ms = Rcmd[addr++];          // Read post-command delay time (ms)
      if(ms == 255) ms = 500;     // If 255, delay for 500 ms instead
      tcDelayMillis(ms);
    }
  }
}

void setAddrWindow(uint8_t x0, uint8_t y0, uint8_t x1, uint8_t y1) {
  writeR(CMD,  ST7735_CASET); // Column addr set
  writeR(DATA, 0x00);
  writeR(DATA, x0+COLSTART);  // XSTART
  writeR(DATA, 0x00);
  writeR(DATA, x1+COLSTART);  // XEND

  writeR(CMD,  ST7735_RASET); // Row addr set
  writeR(DATA, 0x00);
  writeR(DATA, y0+ROWSTART);  // YSTART
  writeR(DATA, 0x00);
  writeR(DATA, y1+ROWSTART);  // YEND

  writeR(CMD,  ST7735_RAMWR); // write to RAM
```

```
}

/* Note: Coordinates are such that (x,y) = (0,0) is on the bottom right corner of
         the TFT (when all the pins are at the bottom), and x increases leftwards
         while y increases upwards.
 */
void fillRect(int16_t x, int16_t y, int16_t w, int16_t h, int16_t color) {
  /*
  // Must offset x and y to the starting row and column
  x += COLSTART;
  y += ROWSTART;

  // Set x and y bounds
  int x_max = COLSTART + MAXWIDTH  - 1;
  int y_max = ROWSTART + MAXHEIGHT - 1;
  */

  int x_max = MAXWIDTH-1;
  int y_max = MAXHEIGHT-1;

  // clipping code
  if((x > x_max) || (y > y_max)) { return; }
  if((x + w-1) > x_max)          { w = x_max+1-x;  }
  if((y + h-1) > y_max)          { h = y_max+1-y; }

  // -----<START: SPI Transaction>-----
  pioDigitalWrite(CS,CS_EN);       // chip enable
  setAddrWindow(x, y, x+w-1, y+h-1);

  unsigned char hi = color >> 8;     // MSB to send
  unsigned char lo = color;          // LSB to send

  // Write data for every pixel in the address window
  for(y=h; y>0; y--) {
    for(x=w; x>0; x--) {
      writeR(DATA, hi);
      writeR(DATA, lo);
    }
  }
  pioDigitalWrite(CS,!CS_EN);     // chip disable
  // -----<END: SPI Transaction>-----
}

/* <draw_cal>
   Draw the calibration screen.
   (Just fills the screen black and draws a set of axes along the edges.)
 */
void draw_cal(void) {
  fillRect(0,0,MAXWIDTH,MAXHEIGHT,RGB565_BLACK);

  int color = RGB565_RED;
  for(int16_t i = 0; i < 128; i++) {
    fillRect(i,0,1,1,color);
    fillRect(0,i,1,1,color);
    if (i%4 == 3) {
      switch(color) {
        case RGB565_RED:    color = RGB565_YELLOW; break;
        case RGB565_YELLOW: color = RGB565_BLUE;   break;
        case RGB565_BLUE:   color = RGB565_GREEN;  break;
        case RGB565_GREEN:  color = RGB565_RED;    break;
        default:            color = RGB565_BLACK;  break;
      }
    }
  }
}

// ----------<CODE END>----------

#endif
```