ROSE-HULMAN INSTITUTE OF TECHNOLOGY

LED Audio Visualizer

ECE331 Final Project

Brian Hulette and Mark Swanson

5/25/2010

TABLE OF CONTENTS

INTRODUCTION

The objective for our ECE331 final project was to create an Audio Visualizer. We decided to use a microprocessor to compute a 16 point Fast Fourier Transform (FFT) and display the result as a bar graph on an 8x8 LED dsiplay[*]. The result is a low resolution, real-time frequency spectrum display of an audio input.

We added a few other features to make the device more convenient to use. A potentiometer allows the user to adjust the gain on the audio signal before it is sampled by the microcontroller. Using this feature, the user can adjust the audio signal to maximize the magnitude of the signal, while preventing clipping. We also added a pass-through audio output, so that the user can listen to the audio signal as its spectrum is displayed.

Below is a picture of the completed device in use.

[*] A 16 point FFT actually only gives you responses at 8 different frequencies because some of the frequencies are repeated as negative frequencies.

USER MANUAL

1. Open up the enclosure, and check if both 9V batteries are connected to their holders. If they are not, connect them both.
2. Close the enclosure and connect the DC power supply to the connector labeled ‘Power’ on the back of the device.
3. Connect your input audio signal to the connector labeled ‘IN’ on the back of the device. If working properly, the device should display a frequency spectrum on the LED display at this point.
4. Adjust the potentiometer on the right side of the device until the highest parts of the spectrum are just reaching the top of the display.
5. If you would like to listen to the audio as it is being visualized connect speakers or headphones to the connection labeled ‘OUT’ on the back of the device.

INTERNAL OPERATION

HARDWARE

We chose to use the Freescale MC9HCS12C128 microprocessor as the core of our project. We chose to use it, rather than a processor specialized for Digital Signal Processing (DSP), because we had already used it throughout the quarter, and had become quite familiar with it.

We used Channel 5 on the processor's onboard 10-bit Analog-to-Digital Converter (ADC) to sample the audio input. Because the ADC can only sample positive voltages, we had to add some more hardware to level shift the input audio signal.

The audio signal first enters a non-inverting amplifier, where the gain is controlled by a 10k potentiometer. That output is then passed to a summing amplifier along with a 2.5V input. The result of this entire circuit should be to amplify the output to a level of 5 Volts peak-to-peak, and shift the signal so it is centered at 2.5V. This creates an ideal signal for sampling with our ADC. One additional op-amp was wired up as a voltage follower with the raw audio signal as its input. Its output drives the pass-through audio output. Every op-amp used in this project was a TL072 low-noise op-amp.

The LED display was an 8x8 LED Display. The display is driven by a MAX7219 Serially Interfaced, 8-Digit, LED Display Driver. The driver is a very convenient way to control any LED display. It will automatically scan the display so that it shows the values stored in its data registers. The Freescale microprocessor can write to those data registers through a SPI bus.

SOFTWARE

There are three main portions of the Audio Visualizer code: Audio Signal Sampling, the FFT Algorithm, and the LED display.

SAMPLING

The audio signal is read from ADC Channel 5 every time an interrupt occurs on Timer Channel 0. This timer is configured to sample the audio input at a rate of 40 kHz, providing a maximum input frequency of 20 kHz.

FFT ALGORITHM

The majority of the FFT algorithm is implemented with a fixed-point FFT Library that we found online. We had to modify the library slightly, however, to make it compatible with our processor. Those changes are highlighted in the fix_fft.c file in Attachment B. There are several other functions, implemented in fft_routines.c, which pre- and post-process the data going into and out of the fixed-point FFT library.

LED DISPLAY

There is one function used for writing to the LED display, display_bar in led_routines,c. That function takes an array of eight integers as inputs, and displays those integers on the LED display as a bar graph.

TESTING

Test Name	Description	Result
LED Bar Graph	Use the LED Display driver and the Freescale microcontroller to display a hard-coded bar graph on the LED display	Passed
Sampling	Use the microcontroller's ATD to sample a 2.5 kHz sinusoidal waveform from a function generator, and graph the results. Verify that the samples look sinusoidal.	Passed
Fast Fourier Transform	Use the fixed-point FFT library to compute the FFT of a set of samples with known frequency. Verify that the FFT shows the correct frequency	Passed
Integration Testing	Take samples of a 2.5 kHz waveform from a function generator, compute the FFT, and display it on the bar graph. Ensure that the graph is largest at the lowest bar, which represents 2.5 kHz.	Passed
Final Testing	Use the device to compute the FFT of an audio signal in real-time and display it on the LED display. Verify that it is a proper visualization of the audio signal (checking for spikes at beats in the audio is a good qualitative test).	Passed

BILL OF MATERIALS

Item	Quantity	Cost
Freescale MC9HCS12C128	1	$25.00
MAX7219 LED Driver	1	Free Sample
8x8 LED Array	1	$7.00
Circuit Board	1	$4.00
TL072 Op-Amp ICs	2	$1.00
9V Batteries	2	$1.00
3.5 mm Audio Connectors	2	In Kind
9V Battery Connectors	2	In Kind
SparkFun Enclosure	1	In Kind
Misc. Resistors and Headers	N/A	$0.50
Total		$38.50

REFERENCE AND ACKNOWLEDGEMENTS

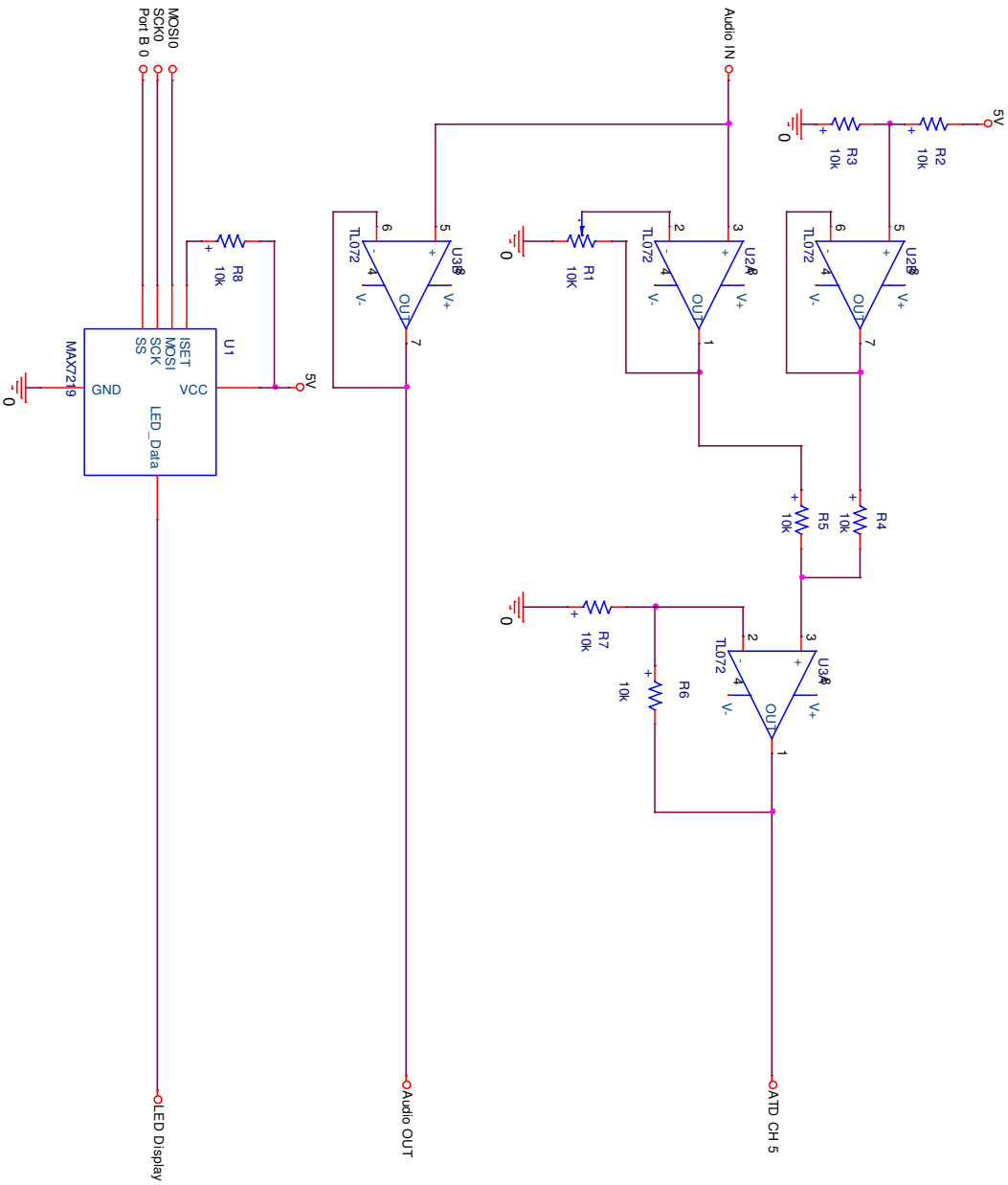
We could not have completed this project without the fixed-point FFT library originally written by Tom Roberts in 1989, with contributions from Malcolm Slaney and Dimitrios P. Bouras in 1994 and 2006, respectively.

We would also like to thank two professors who were very helpful during the design and implementation of our LED Audio Visualizer:

Dr. JianJian Song
For providing us with several parts necessary for completion of the project as well as advice throughout the project's development.

Dr. Wayne Padgett
For helping us understand how an FFT works, and teaching us what we needed to change in our fixed-point FFT library to make it work with our processor.

ATTACHMENT A: SCHEMATIC



ATTACHMENT B: CODE

MAIN.C

```
/*****************************************************************
 *    DESCRIPTION:    An LED array displaying FFT of an audio signal  *
 *    Date:           5-12-2010                                       *
 *    AUTHOR:         Brian Hulette & Mark Swanson                    *
 *****************************************************************/

#include "per_C32_L45J.h"

#define CLI() {asm cli;}  // Enable global interrupts
#define SEI() {asm sei;}      // Disable global interrupts
#define WAIT  150         // Loop WAIT^2 between display refresh

extern void initialize_indicator(void);
extern void initialize_spi(void);
extern void write_to_spi(unsigned int data);
extern void initialize_ATD(void);
extern void initialize_sampler(void);
extern void sample(signed int *samps);
extern void normalize(signed int *atd);
extern int fft(signed int *order, signed int *imaginary);
extern void denormalize(signed int *disps);
extern void initialize_LED(void);
extern void display_bar(char* data);
void SetOscillatorSpeed(void);

/* Main Routine */
void main ()
{
  int ffted, i, j;
  static signed int samp_data[16], imaginary[16];

  SetOscillatorSpeed();             // Set bus clock
      SEI();                            // Disable I bit interrupts
      initialize_indicator();           // Initialize LED clock indicator
      initialize_spi();                 // Initialize SPI communication
      initialize_ATD();                 // Initialize ATD
      initialize_sampler();             // Initialize sample timer
      initialize_LED();                 // Initialize LED display driver
  CLI();                            // Enable I bit interrupts

  while(1) {
    sample(samp_data);              // Sample data from ATD
    // Insert zeros in imaginary array for real-valued FFT calculation
    for(i=0; i<16; i++){imaginary[i]=0;}
    normalize(samp_data);           // Normalize sampled data for FFT
calculation
    // Calculate FFT using a real array and zeroed imaginary array
    ffted = fft(samp_data, imaginary);
    denormalize(samp_data);         // Denormalize FFT results for display
    display_bar((char *)samp_data); // Refresh dislay with FFT result
    // Pause in order to aesthetically display FFT of signal
    for(i=0; i < WAIT; i++){for(j=0; j < WAIT; j++){}}
```

```
  }
}

/************************************************************************
 * Initialize the PLL clock frequency to PLLClk                         *
 ************************************************************************/

#define PLLClk 20000000 //PLL clock rate in Hz

void SetOscillatorSpeed(void) {
  // PLLCLK=2xOSCCLKx(SYNR + 1)/(REFDV + 1)
  // Set clock speed to be 20MHz for 4MHz crystal
  // SYNR=9 and REFDV=3
  Crg.clksel.byte &=~PLLSEL;            // Disconnect PLL
  Crg.pllctl.byte |=PLLON;              // Turn on PLL
  Crg.synr.byte |=SYN3|SYN0;            // SYNR=9
  Crg.refdv.byte |=REFDV1|REFDV0;       // REFDV=3
      while(!(Crg.crgflg.byte & LOCK)){ };  // Wait for PLL to connect
      Crg.clksel.byte |=PLLSEL;             // Connect PLL
}
```

FIX_FFT.C

(HIGHLIGHTED PORTIONS ADAPTED FOR OUR MICROPROCESSOR)

```
/* fix_fft.c - Fixed-point in-place Fast Fourier Transform  */
/*
  All data are fixed-point short integers, in which -32768
  to +32768 represent -1.0 to +1.0 respectively. Integer
  arithmetic is used for speed, instead of the more natural
  floating-point.

  For the forward FFT (time -> freq), fixed scaling is
  performed to prevent arithmetic overflow, and to map a 0dB
  sine/cosine wave (i.e. amplitude = 32767) to two -6dB freq
  coefficients. The return value is always 0.

  For the inverse FFT (freq -> time), fixed scaling cannot be
  done, as two 0dB coefficients would sum to a peak amplitude
  of 64K, overflowing the 32k range of the fixed-point integers.
  Thus, the fix_fft() routine performs variable scaling, and
  returns a value which is the number of bits LEFT by which
  the output must be shifted to get the actual amplitude
  (i.e. if fix_fft() returns 3, each value of fr[] and fi[]
  must be multiplied by 8 (2**3) for proper scaling.
  Clearly, this cannot be done within fixed-point short
  integers. In practice, if the result is to be used as a
  filter, the scale_shift can usually be ignored, as the
  result will be approximately correctly normalized as is.

  Written by:  Tom Roberts  11/8/89
  Made portable:  Malcolm Slaney 12/15/94 malcolm@interval.com
  Enhanced:  Dimitrios P. Bouras  14 Jun 2006 dbouras@ieee.org
*/

#define N_WAVE      1024    /* full length of Sinewave[] */
#define LOG2_N_WAVE 10      /* log2(N_WAVE) */

/*
  Henceforth "short" implies 16-bit word. If this is not
  the case in your architecture, please replace "short"
  with a type definition which *is* a 16-bit word.
*/

/*
  Since we only use 3/4 of N_WAVE, we define only
  this many samples, in order to conserve data space.
*/
const signed int Sinewave[N_WAVE-N_WAVE/4] = {
      0,    201,    402,    603,    804,   1005,   1206,   1406,
   1607,   1808,   2009,   2209,   2410,   2610,   2811,   3011,
   3211,   3411,   3611,   3811,   4011,   4210,   4409,   4608,
   4807,   5006,   5205,   5403,   5601,   5799,   5997,   6195,
   6392,   6589,   6786,   6982,   7179,   7375,   7571,   7766,
   7961,   8156,   8351,   8545,   8739,   8932,   9126,   9319,
   9511,   9703,   9895,  10087,  10278,  10469,  10659,  10849,
  11038,  11227,  11416,  11604,  11792,  11980,  12166,  12353,
  12539,  12724,  12909,  13094,  13278,  13462,  13645,  13827,
```

```
14009,  14191,  14372,  14552,  14732,  14911,  15090,  15268,
15446,  15623,  15799,  15975,  16150,  16325,  16499,  16672,
16845,  17017,  17189,  17360,  17530,  17699,  17868,  18036,
18204,  18371,  18537,  18702,  18867,  19031,  19194,  19357,
19519,  19680,  19840,  20000,  20159,  20317,  20474,  20631,
20787,  20942,  21096,  21249,  21402,  21554,  21705,  21855,
22004,  22153,  22301,  22448,  22594,  22739,  22883,  23027,
23169,  23311,  23452,  23592,  23731,  23869,  24006,  24143,
24278,  24413,  24546,  24679,  24811,  24942,  25072,  25201,
25329,  25456,  25582,  25707,  25831,  25954,  26077,  26198,
26318,  26437,  26556,  26673,  26789,  26905,  27019,  27132,
27244,  27355,  27466,  27575,  27683,  27790,  27896,  28001,
28105,  28208,  28309,  28410,  28510,  28608,  28706,  28802,
28897,  28992,  29085,  29177,  29268,  29358,  29446,  29534,
29621,  29706,  29790,  29873,  29955,  30036,  30116,  30195,
30272,  30349,  30424,  30498,  30571,  30643,  30713,  30783,
30851,  30918,  30984,  31049,  31113,  31175,  31236,  31297,
31356,  31413,  31470,  31525,  31580,  31633,  31684,  31735,
31785,  31833,  31880,  31926,  31970,  32014,  32056,  32097,
32137,  32176,  32213,  32249,  32284,  32318,  32350,  32382,
32412,  32441,  32468,  32495,  32520,  32544,  32567,  32588,
32609,  32628,  32646,  32662,  32678,  32692,  32705,  32717,
32727,  32736,  32744,  32751,  32757,  32761,  32764,  32766,
32767,  32766,  32764,  32761,  32757,  32751,  32744,  32736,
32727,  32717,  32705,  32692,  32678,  32662,  32646,  32628,
32609,  32588,  32567,  32544,  32520,  32495,  32468,  32441,
32412,  32382,  32350,  32318,  32284,  32249,  32213,  32176,
32137,  32097,  32056,  32014,  31970,  31926,  31880,  31833,
31785,  31735,  31684,  31633,  31580,  31525,  31470,  31413,
31356,  31297,  31236,  31175,  31113,  31049,  30984,  30918,
30851,  30783,  30713,  30643,  30571,  30498,  30424,  30349,
30272,  30195,  30116,  30036,  29955,  29873,  29790,  29706,
29621,  29534,  29446,  29358,  29268,  29177,  29085,  28992,
28897,  28802,  28706,  28608,  28510,  28410,  28309,  28208,
28105,  28001,  27896,  27790,  27683,  27575,  27466,  27355,
27244,  27132,  27019,  26905,  26789,  26673,  26556,  26437,
26318,  26198,  26077,  25954,  25831,  25707,  25582,  25456,
25329,  25201,  25072,  24942,  24811,  24679,  24546,  24413,
24278,  24143,  24006,  23869,  23731,  23592,  23452,  23311,
23169,  23027,  22883,  22739,  22594,  22448,  22301,  22153,
22004,  21855,  21705,  21554,  21402,  21249,  21096,  20942,
20787,  20631,  20474,  20317,  20159,  20000,  19840,  19680,
19519,  19357,  19194,  19031,  18867,  18702,  18537,  18371,
18204,  18036,  17868,  17699,  17530,  17360,  17189,  17017,
16845,  16672,  16499,  16325,  16150,  15975,  15799,  15623,
15446,  15268,  15090,  14911,  14732,  14552,  14372,  14191,
14009,  13827,  13645,  13462,  13278,  13094,  12909,  12724,
12539,  12353,  12166,  11980,  11792,  11604,  11416,  11227,
11038,  10849,  10659,  10469,  10278,  10087,   9895,   9703,
 9511,   9319,   9126,   8932,   8739,   8545,   8351,   8156,
 7961,   7766,   7571,   7375,   7179,   6982,   6786,   6589,
 6392,   6195,   5997,   5799,   5601,   5403,   5205,   5006,
 4807,   4608,   4409,   4210,   4011,   3811,   3611,   3411,
 3211,   3011,   2811,   2610,   2410,   2209,   2009,   1808,
 1607,   1406,   1206,   1005,    804,    603,    402,    201,
    0,   -201,   -402,   -603,   -804,  -1005,  -1206,  -1406,
-1607,  -1808,  -2009,  -2209,  -2410,  -2610,  -2811,  -3011,
```

```
  -3211,  -3411,  -3611,  -3811,  -4011,  -4210,  -4409,  -4608,
  -4807,  -5006,  -5205,  -5403,  -5601,  -5799,  -5997,  -6195,
  -6392,  -6589,  -6786,  -6982,  -7179,  -7375,  -7571,  -7766,
  -7961,  -8156,  -8351,  -8545,  -8739,  -8932,  -9126,  -9319,
  -9511,  -9703,  -9895, -10087, -10278, -10469, -10659, -10849,
 -11038, -11227, -11416, -11604, -11792, -11980, -12166, -12353,
 -12539, -12724, -12909, -13094, -13278, -13462, -13645, -13827,
 -14009, -14191, -14372, -14552, -14732, -14911, -15090, -15268,
 -15446, -15623, -15799, -15975, -16150, -16325, -16499, -16672,
 -16845, -17017, -17189, -17360, -17530, -17699, -17868, -18036,
 -18204, -18371, -18537, -18702, -18867, -19031, -19194, -19357,
 -19519, -19680, -19840, -20000, -20159, -20317, -20474, -20631,
 -20787, -20942, -21096, -21249, -21402, -21554, -21705, -21855,
 -22004, -22153, -22301, -22448, -22594, -22739, -22883, -23027,
 -23169, -23311, -23452, -23592, -23731, -23869, -24006, -24143,
 -24278, -24413, -24546, -24679, -24811, -24942, -25072, -25201,
 -25329, -25456, -25582, -25707, -25831, -25954, -26077, -26198,
 -26318, -26437, -26556, -26673, -26789, -26905, -27019, -27132,
 -27244, -27355, -27466, -27575, -27683, -27790, -27896, -28001,
 -28105, -28208, -28309, -28410, -28510, -28608, -28706, -28802,
 -28897, -28992, -29085, -29177, -29268, -29358, -29446, -29534,
 -29621, -29706, -29790, -29873, -29955, -30036, -30116, -30195,
 -30272, -30349, -30424, -30498, -30571, -30643, -30713, -30783,
 -30851, -30918, -30984, -31049, -31113, -31175, -31236, -31297,
 -31356, -31413, -31470, -31525, -31580, -31633, -31684, -31735,
 -31785, -31833, -31880, -31926, -31970, -32014, -32056, -32097,
 -32137, -32176, -32213, -32249, -32284, -32318, -32350, -32382,
 -32412, -32441, -32468, -32495, -32520, -32544, -32567, -32588,
 -32609, -32628, -32646, -32662, -32678, -32692, -32705, -32717,
 -32727, -32736, -32744, -32751, -32757, -32761, -32764, -32766,
};

/*
  FIX_MPY() - fixed-point multiplication & scaling.
  Substitute inline assembly for hardware-specific
  optimization suited to a particluar DSP processor.
  Scaling ensures that result remains 16-bit.

  MODIFIED: Brian Hulette & Mark Swanson
  ADAPTED:  MC9S12C128
*/
signed int FIX_MPY(signed int a, signed int b)
{
      /* shift right one less bit (i.e. 15-1) */
      //c = (a * b) >> 14;
      /* last bit shifted out = rounding-bit */
      //carry = c & 0x01;
      /* last shift + rounding bit */
      //result = (c >> 1) + carry;
      static signed int a1, b1;
      static unsigned int highb, lowb, lowmsb, lowsmsb;
      a1 = a;
      b1 = b;
      _asm{
        pshd;
        pshy;
        ldd a1;
```

```
          ldy b1;
          emuls;
          sty highb;
          std lowb;
          puld;
          puly;
        }
        lowmsb = lowb >> 15;
        lowsmsb = (lowb << 1) >> 15;
        a1 = (highb<<1)+lowmsb;
        b1 = a1 + lowsmsb;
        return b1;
}

/*
  fix_fft() - perform forward/inverse fast Fourier transform.
  fr[n],fi[n] are real and imaginary arrays, both INPUT AND
  RESULT (in-place FFT), with 0 <= n < 2**m; set inverse to
  0 for forward transform (FFT), or 1 for iFFT.
*/
long fix_fft(signed int fr[], signed int fi[], signed int m, signed int
inverse)
{
        static long mr, nn, i, j, l, k, istep, n, scale, shift;
        static signed int qr, qi, tr, ti, wr, wi;

        n = 1 << /*m*/4;

        /* max FFT size = N_WAVE */
        if (n > N_WAVE)
                return -1;

        mr = 0;
        nn = n - 1;
        scale = 0;

        /* decimation in time - re-order data */
        for (m=1; m<=nn; ++m) {
                l = n;
                do {
                        l >>= 1;
                } while (mr+l > nn);
                mr = (mr & (l-1)) + l;

                if (mr <= m){
                        continue;
                }
                tr = fr[m];
                fr[m] = fr[mr];
                fr[mr] = tr;
                ti = fi[m];
                fi[m] = fi[mr];
                fi[mr] = ti;
        }

        l = 1;
        k = LOG2_N_WAVE-1;
```

```
while (l < n) {
      if (inverse) {
            /* variable scaling, depending upon data */
            shift = 0;
            for (i=0; i<n; ++i) {
                  j = fr[i];
                  if (j < 0)
                        j = -j;
                  m = fi[i];
                  if (m < 0)
                        m = -m;
                  if (j > 16383 || m > 16383) {
                        shift = 1;
                        break;
                  }
            }
            if (shift)
                  ++scale;
      } else {
            /*
              fixed scaling, for proper normalization --
              there will be log2(n) passes, so this results
              in an overall factor of 1/n, distributed to
              maximize arithmetic accuracy.
            */
            shift = 1;
      }
      /*
        it may not be obvious, but the shift will be
        performed on each data point exactly once,
        during this pass.
      */
      istep = l << 1;
      for (m=0; m<l; ++m) {
            j = m << /*k*/9;
            /* 0 <= j < N_WAVE/2 */
            wr =  Sinewave[j+N_WAVE/4];
            wi = -Sinewave[j];
            if (inverse)
                  wi = -wi;
            if (shift) {
                  wr >>= 1;
                  wi >>= 1;
            }
            for (i=m; i<n; i+=istep) {
                  j = i + l;
                  tr = FIX_MPY(wr,fr[j]) - FIX_MPY(wi,fi[j]);
                  ti = FIX_MPY(wr,fi[j]) + FIX_MPY(wi,fr[j]);
                  qr = fr[i];
                  qi = fi[i];
                  if (shift) {
                        qr >>= 1;
                        qi >>= 1;
                  }
                  fr[j] = qr - tr;
                  fi[j] = qi - ti;
                  fr[i] = qr + tr;
```

```
                        fi[i] = qi + ti;
                }
            }
            --k;
            l = istep;
      }
      return scale;
}

/*
  fix_fftr() - forward/inverse FFT on array of real numbers.
  Real FFT/iFFT using half-size complex FFT by distributing
  even/odd samples into real/imaginary arrays respectively.
  In order to save data space (i.e. to avoid two arrays, one
  for real, one for imaginary samples), we proceed in the
  following two steps: a) samples are rearranged in the real
  array so that all even samples are in places 0-(N/2-1) and
  all imaginary samples in places (N/2)-(N-1), and b) fix_fft
  is called with fr and fi pointing to index 0 and index N/2
  respectively in the original array. The above guarantees
  that fix_fft "sees" consecutive real samples as alternating
  real and imaginary samples in the complex array.
*/
int fix_fftr(signed int f[], int m, int inverse)
{
      int i, N = 1<</*(m-1)*/4, scale = 0;
      signed int tt, *fr=f, *fi=&f[N];

      if (inverse)
            scale = fix_fft(fi, fr, m-1, inverse);
      for (i=1; i<N; i+=2) {
            tt = f[N+i-1];
            f[N+i-1] = f[i];
            f[i] = tt;
      }
      if (! inverse)
            scale = fix_fft(fi, fr, m-1, inverse);
      return scale;
}
```

FFT_ROUTINES.C

```
/*********************************************************************
 *  DESCRIPTION:    FFT Subroutines                                  *
 *  DATE:           5-12-2010                                        *
 *  AUTHOR:         Mark Swanson                                     *
 *********************************************************************/

#include "per_C32_L45J.h"

#define FFTSIZE 16    // Number of samples in FFT set
#define HALFFFTSIZE 8
#define HALFATD 512  // Half the maximum ATD value
#define FFTAMP 64    // Max FFT value/HALFATD - 1 = 32768/512 = 64
#define log2 4       // log2(FFTSIZE) + 1
#define INVERSE 0    // Compute inverse FFT or forward FFT
#define DOTAMP 500//3641  // Max FFT value/(Number LEDs + 1) = 32768/9 = 3641
#define HALFDAMP 1820

#define MAGNITUDE(X)
FIX_MPY(order[X],order[X])/2+FIX_MPY(imaginary[X],imaginary[X])/2;

unsigned int findex;
static signed int normarr[2*FFTSIZE];
static signed int result;
static signed int fftarr[FFTSIZE];
static signed int disparr[FFTSIZE/2];
static signed int splitarr[2*FFTSIZE];
static signed int fftamp;
unsigned int dindex;
static unsigned int disp[8] =
{HALFDAMP,2*HALFDAMP,3*HALFDAMP,4*HALFDAMP,5*HALFDAMP,6*HALFDAMP,7*HALFDAMP,8
*DOTAMP};

extern int fix_fft(signed int fr[], signed int fi[], signed int m, signed int
inverse);
extern int fix_fftr(signed int f[], int m, int inverse);
extern signed int FIX_MPY(signed int a, signed int b);


/*********************************************************************
 * Normalize ATD results to range expected by fixed point FFT        *
 *********************************************************************/

void normalize(signed int *atd) {
  for(findex=0; findex < FFTSIZE; findex++) {
    normarr[findex] = (atd[findex]-HALFATD)*FFTAMP; // Normalize ATD value [-
2^15 2^15]
    atd[findex] = normarr[findex];
  }
  return;                                  // Return normalized values
}

/*********************************************************************
 * Split array in halves with even indices in front half odd in back  *
 *********************************************************************/
```

```
void split(signed int *norm) {
  for (findex=0; findex < 2*FFTSIZE; findex++) {
            if (findex & 0x01)
                  splitarr[((2*FFTSIZE)+findex)>>1] = norm[findex];
            else
                  splitarr[findex>>1] = norm[findex];
  }
  for (findex=0; findex < 2*FFTSIZE; findex++) {
    norm[findex] = splitarr[findex];
  }
  return;
}

/*****************************************************************
 * Computes FFT of given normalized samples FOLLOWING normalize call  *
 *****************************************************************/

int fft(signed int *order, signed int* imaginary) {
  result = fix_fft(order, imaginary, log2, INVERSE);
  //shift = fix_fftr(order, log2, INVERSE);  // Call real number FFT for 8
samples
  for(findex=0; findex  < HALFFFTSIZE-1; findex++) {
    order[findex+1]=2*order[findex+1];
    order[findex]=MAGNITUDE(findex+1);
    /*if(order[findex+1]<0) {
      order[findex+1] = order[findex+1]*(-1);
    }
    fftarr[findex] = order[findex+1];           // Store result of fft
skipping the DC component
    order[findex] = fftarr[findex];
    */
  }
  order[HALFFFTSIZE-1]=MAGNITUDE(HALFFFTSIZE);
  return result;                  // Return amount results must be shifted
}

/*****************************************************************
 * Denormalize FFT results to range expected by LED display array     *
 *****************************************************************/

void denormalize(signed int *disps) {
  for(findex=0; findex < FFTSIZE/2; findex++) {
    fftamp = disps[findex];
    if(findex != (FFTSIZE/2 - 1)) {
      fftamp = 2*fftamp;
    }
    for(dindex=0; dindex < (FFTSIZE/2)+1; dindex++) {
      if(fftamp<((dindex+1)*DOTAMP) || fftamp==(dindex*DOTAMP)) {   // TODO:
Create for-loop for DOTAMP array
        disparr[findex] = dindex;
        disps[findex] = disparr[findex];
        break;
      }
    }
  }
  return;
}
```

SPI_ROUTINES_74HC595.C

```
/*****************************************************************
 *    DESCRIPTION:  SPI routines for PBMCUSLK board from Freescale
 *                  send data to a 74HC595 shift register on the PBMCUSLK
 *    DATE:         4-21-2009
 *    AUTHOR:       Jianjian Song
 *     MODIFIED:       May 2010, for use with the MAX7219CNG SPI LED Display
driver
 *                  Brian Hulette & Mark Swanson
 *****************************************************************/

#include "per_C32_L45J.h"


#define SPI_PORT_DIRECTION Pim.ptm    //in S12C32PIMV1.h
#define SCK   Pim.ptm.bit.ptm5
#define MOSI  Pim.ptm.bit.ptm4
#define SLAVE_SELECT  Pim.ptm.bit.ptm3
#define            SPI_DDRM_init                          Pim.ddrm.bit.ddrm5=1;
Pim.ddrm.bit.ddrm4=1;Pim.ddrm.bit.ddrm3=1;

#define LED_SS        Regs.portb.bit.ptb0
#define LED_SS_DDR  Regs.ddrb.bit.ddrb0
#define SS_WAIT     50

#define SPICR1_INIT 0b01010000 //most significant bit first, sampling occurs
on rising edges
#define SPICR2_INIT 0x00
#define SPIBAUD_INIT 0b010010010 //select serial clock baud rate = 130 kHz at
25MHz bus clock
                    // 0b01000011

void initialize_spi(void){
  unsigned char read_spisr;

  LED_SS_DDR=1;

  SPI_DDRM_init
  Spi0.spicr1.byte=SPICR1_INIT;  //spi registers are defined in S12SPIV3.h
  Spi0.spicr2.byte=SPICR2_INIT;
  Spi0.spibr.byte=SPIBAUD_INIT;
//to clear SPIF in 2 steps
  read_spisr=Spi0.spisr.byte;
  read_spisr=Spi0.spidr.byte;
} // end initialize_spi

//write a byte through spi
void write_to_spi(unsigned int data){
     unsigned char read_spisr;
     int i;
// wait until transmit empty interrupt flag is "1"
  LED_SS=0;
  SLAVE_SELECT=0;    //this pin is used to enable the 74HC595 shift register
  while(Spi0.spisr.bit.sptef==0){};
  read_spisr=Spi0.spisr.byte;
  Spi0.spidr.byte=data>>8;
```

```
  while(Spi0.spisr.bit.sptef==0) // wait for transfer to complete
  read_spisr=Spi0.spisr.byte;
  Spi0.spidr.byte=data;
  while(Spi0.spisr.bit.sptef==0) // wait for transfer to complete
  for(i=0; i<SS_WAIT; i++){}
  SLAVE_SELECT=1;        //transfer data to the storage register on positive
edge
  LED_SS=1;
}
```

LED_ROUTINES.C

```
extern void write_to_spi(unsigned int data);

void initialize_LED(){
  write_to_spi(0x0C01); // Set to normal operation (as opposed to shutdown
mode)
  write_to_spi(0x0900); // Disable decode register so we can address LEDs
directly
  write_to_spi(0x0AFF); // Set Intensity to max on
  write_to_spi(0x0BFF); // Set scan-limit to all digits
  write_to_spi(0x0F00); // Turn off Display Test mode
}

/*****************************************************
 * Displays a bar graph on an 8x8 LED display using a *
 * MAX7219CNG SPI Display Driver                      *
 * The bar graph data is passed in through the        *
 * int data[] argument where each index represents    *
 * one bar in the graph, and the value at the index   *
 * is an integer representing the height of that bar  *
 * in LEDs                                            *
 *****************************************************/
void display_bar(int data[]){
  write_to_spi(0x01FF & (~(0xFF>>data[0])));
  write_to_spi(0x02FF & (~(0xFF>>data[1])));
  write_to_spi(0x03FF & (~(0xFF>>data[2])));
  write_to_spi(0x04FF & (~(0xFF>>data[3])));
  write_to_spi(0x05FF & (~(0xFF>>data[4])));
  write_to_spi(0x06FF & (~(0xFF>>data[5])));
  write_to_spi(0x07FF & (~(0xFF>>data[6])));
  write_to_spi(0x08FF & (~(0xFF>>data[7])));
}
```

ATD_ROUTINES.C

```
/*********************************************************************
 *  DESCRIPTION:    ATD Subroutines                                  *
 *  DATE:           5-12-2010                                        *
 *  AUTHOR:         Mark Swanson                                     *
 *********************************************************************/

#include "per_C32_L45J.h"

#define ON  1
#define OFF 0
#define ATDLOOP 200
//Right justified, unsigned data, continuous conversion, single channel, AN5
#define ATDMODE 0b10100101
#define SINGLE 0b0001   // 1 conversion
#define ATDCYC 0b00     // 2 A/D conversion cycles
#define ATDPRE 0b00000  // Prescaler = 0

unsigned int aindex;
unsigned char ch;

/*********************************************************************
 * Initializes the ATD for continuous 10-bit FIFO conversion cycles   *
 *********************************************************************/

void initialize_ATD() {
  Atd0.atdctl2.bit.adpu = ON;      // Turn on ATD
  for(aindex=0; aindex < ATDLOOP; aindex++){}     // Loop ATDLOOP times
  Tim0.tscr1.bit.ten = ON;         // Enable timer (required for ATD)
  Atd0.atdctl2.bit.affc = OFF;     // Disable Fast Flag Clear All
  Atd0.atdctl2.bit.awai = OFF;     // ATD runs in Wait mode
  Atd0.atdctl2.bit.ascie = OFF;    // Disable ATD Sequence Complete
interrupts
  Atd0.atdctl3.bit.slc = SINGLE;   // One conversion per sequence
  Atd0.atdctl3.bit.fifo = ON;      // Enable result register FIFO
  Atd0.atdctl4.bit.smp = ATDCYC;   // Set 2 A/D conversion clock periods
  Atd0.atdctl4.bit.prs = ATDPRE;   // Set ATD clock = PLLCLK/((PRS+1)*2) =
20Mhz/2 = 10Mhz
  Atd0.atdctl4.bit.sres8 = OFF;    // Enable 10-bit resolution
  Atd0.atdctl5.byte = ATDMODE;     // Start conversion cycles
}

/*********************************************************************
 * Returns 10-bit ATD result from current result register            *
 *********************************************************************/

signed int read_ATD() {
  ch = Atd0.atdstat0.bit.ccx;          // Store next result register index
  switch (ch) {
    case 0: while(Atd0.atdstat1.bit.ccf0==OFF){}  // Wait for conversion
            return Atd0.atddr[0].d10;             // Return ATD result
    case 1: while(Atd0.atdstat1.bit.ccf1==OFF){}  // Wait for conversion
            return Atd0.atddr[1].d10;             // Return ATD result
    case 2: while(Atd0.atdstat1.bit.ccf2==OFF){}  // Wait for conversion
            return Atd0.atddr[2].d10;             // Return ATD result
    case 3: while(Atd0.atdstat1.bit.ccf3==OFF){}  // Wait for conversion
```

```
            return Atd0.atddr[3].d10;             // Return ATD result
        case 4: while(Atd0.atdstat1.bit.ccf4==OFF){}  // Wait for conversion
            return Atd0.atddr[4].d10;             // Return ATD result
        case 5: while(Atd0.atdstat1.bit.ccf5==OFF){}  // Wait for conversion
            return Atd0.atddr[5].d10;             // Return ATD result
        case 6: while(Atd0.atdstat1.bit.ccf6==OFF){}  // Wait for conversion
            return Atd0.atddr[6].d10;             // Return ATD result
        case 7: while(Atd0.atdstat1.bit.ccf7==OFF){}  // Wait for conversion
            return Atd0.atddr[7].d10;             // Return ATD result
        default:while(Atd0.atdstat1.bit.ccf0==OFF){}  // Wait for conversion
            return Atd0.atddr[0].d10;             // Return ATD result

    }

}
```

SAMP_ROUTINES.C

```
/*********************************************************************
 *  DESCRIPTION:    Sampling Subroutines                              *
 *  DATE:           5-12-2010                                         *
 *  AUTHOR:         Mark Swanson                                      *
 *********************************************************************/

#include "per_C32_L45J.h"

#define SAMPLES 32
#define ON 1
#define OFF 0
#define OUTPUT 1
#define INPUT 0
#define SAMP_PIN Pim.ptt.bit.ptt0    // Toggle bit 0 of port T
#define SAMP_DIR Pim.ddrt.bit.ddrt0  // Data direction for bit 0 of port T
#define SAMP_PRESCALER 0b010         // Divide bus clock by 4, bus clock =
20MHz
// Timer interrupt interval = 125x4x(1/20) microsec = 25us
// Sampling frequency of 1/25us = 40kHz
#define SampleInterruptInterval  123
#define SampleCompareReg Tim0.tc[0]  // TC0 Register

unsigned char sampling;
unsigned int sindex;
static signed int samparr[32];

extern signed int read_ATD(void);

/*********************************************************************
 * Initializes timer channel 0 with toggle pin bit 0 of port T       *
 *********************************************************************/

void SetSampleTimer() {
  sampling = OFF;                   // Initially disable sampling
  SAMP_DIR = OUTPUT;            // Set bit 0 of port T as output port
      Tim0.tios.bit.ios0  =  OUTPUT;   //  Select  timer  channel  0  for  output
compare
  Tim0.tie.bit.c0i= 1;          // Enable timer channel 0 interrupt
  // Clear the timer FLAG in TFLG1 by writing a ``1'' to it.
  Tim0.tflg1.bit.c0f = 1;
  // Load offset to begin new outpout compare
  SampleCompareReg = Tim0.tcnt.word+SampleInterruptInterval;
}

/*********************************************************************
 * If sampling enabled, samples ATD and resets timer for next sample  *
 *********************************************************************/

interrupt void ServiceSampleTimer() {
  SAMP_PIN = SAMP_PIN+1;      // Toggle sample output pin
  if(sampling == ON){             // Ensure sampling enabled
    samparr[sindex] = read_ATD();  // Store next sample in FFT set
    sindex++;                       // Increment FFT set index
    if(sindex == SAMPLES){              // Check for final sample in FFT set
      sindex = 0;                   // Reinitialize FFT set index
```

```
      sampling = OFF;             // Indicate sampling complete
    }
  }
  // Clear the timer FLAG in TFLG1 by writing a ``1'' to it.
  Tim0.tflg1.bit.c0f = 1;
  // Load offset to begin new outpout compare
  SampleCompareReg = Tim0.tcnt.word+SampleInterruptInterval;
}

/*********************************************************************
 * Returns 8 ATD samples collected at specified sampling frequency    *
 *********************************************************************/

void sample(signed int *samps) {
  sindex = 0;               // Initialize FFT set index
  sampling = ON;           // Enable sampling
  while(sampling == ON){}  // Wait for 8 samples to complete
  for(sindex=0; sindex < SAMPLES; sindex++) {
      samps[sindex] = samparr[sindex];
  }
  return;      // Return sampling results
}
```