Lucas Schuermann
Carlos Martin
Jorge Orbay

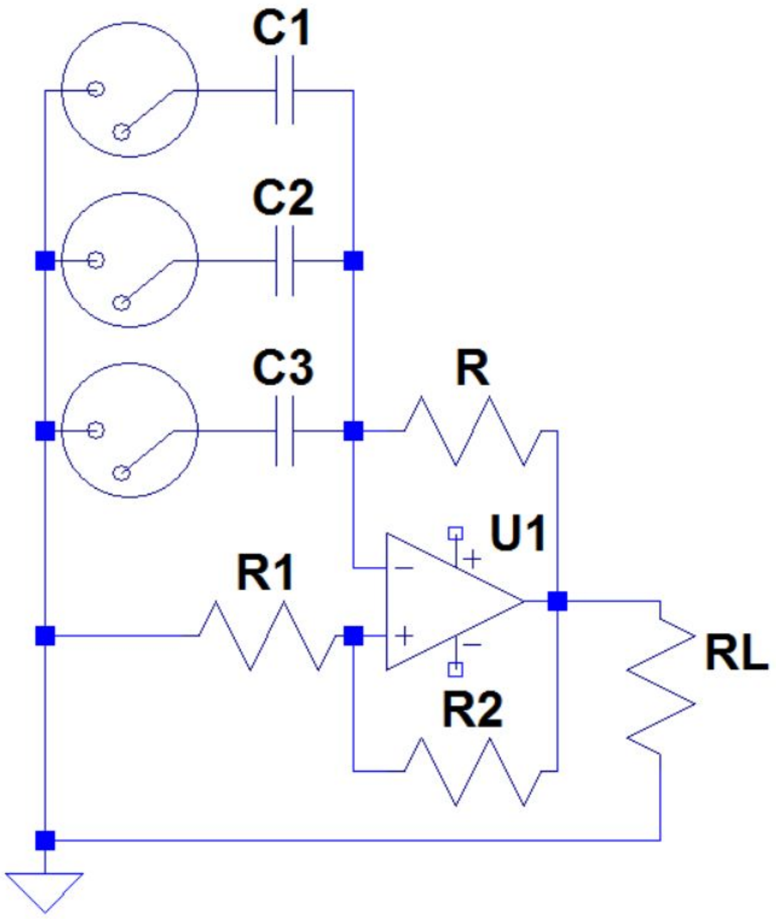
E1201 Project Report:
Periodic Function Generator with FFT-Based LED Waveform Display
7 December 2014

Abstract

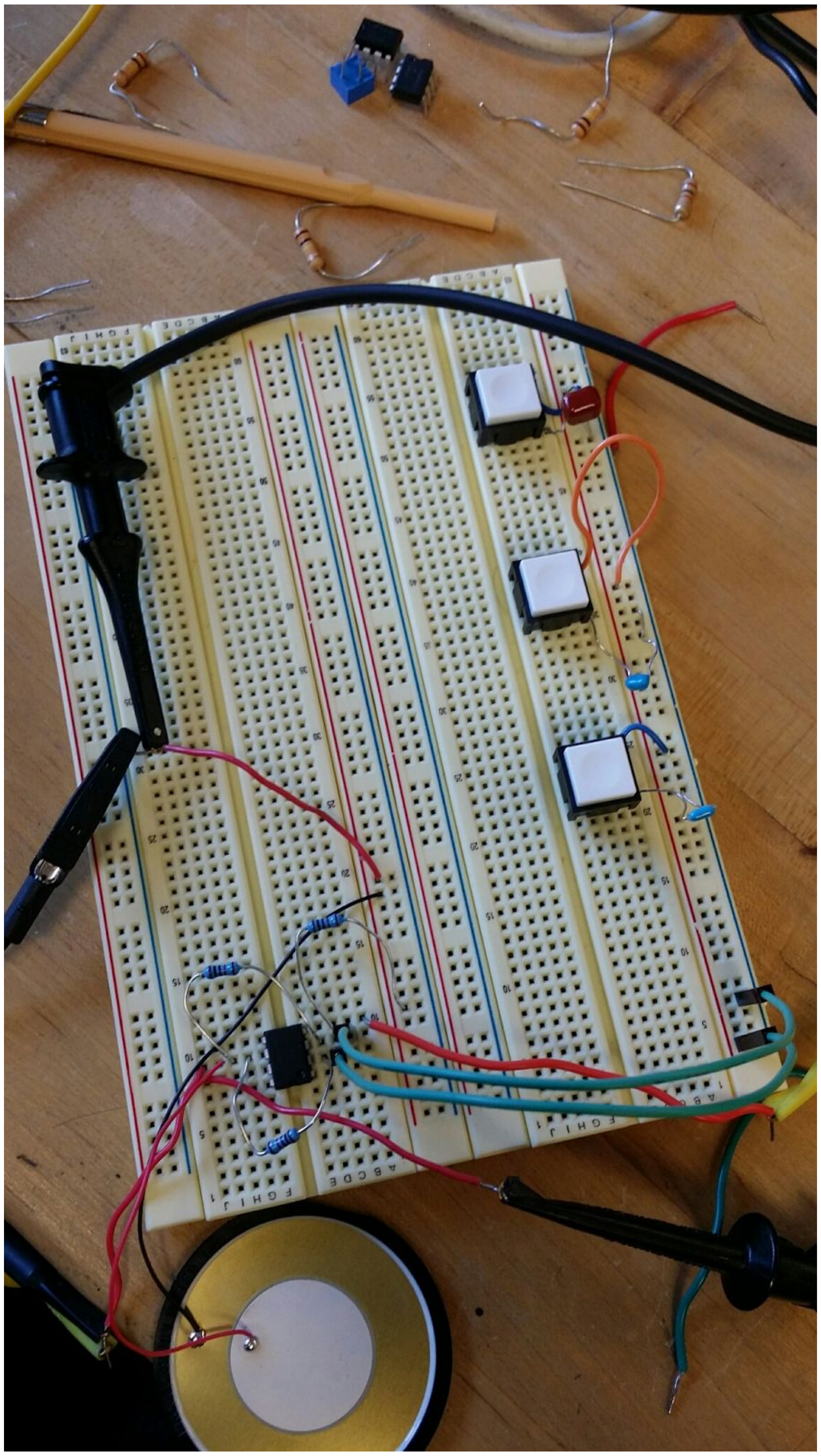
Our project consists of two parts: a controllable (variable) periodic function generator and a FFT-based waveform display system. The periodic function generator uses an op-amp to create a square wave. This square wave is adjusted with a system of switches that wire different capacitor values to the op-amp, essentially increasing or decreasing the frequency of the square wave. Using a 3 × 4 grid of LEDs connected to an Arduino running processing code, we analyze the generated analog waveform and display it in the frequency domain as levels in frequency bins (similar to the equalizer output on old stereos with lines bouncing up and down). The audio processing leverages the Arduino's analog-to-digital converter and a custom finite fast Fourier transform analysis algorithm.

Circuit Schematics & Supporting Materials

1. Function Generator Circuit Diagram

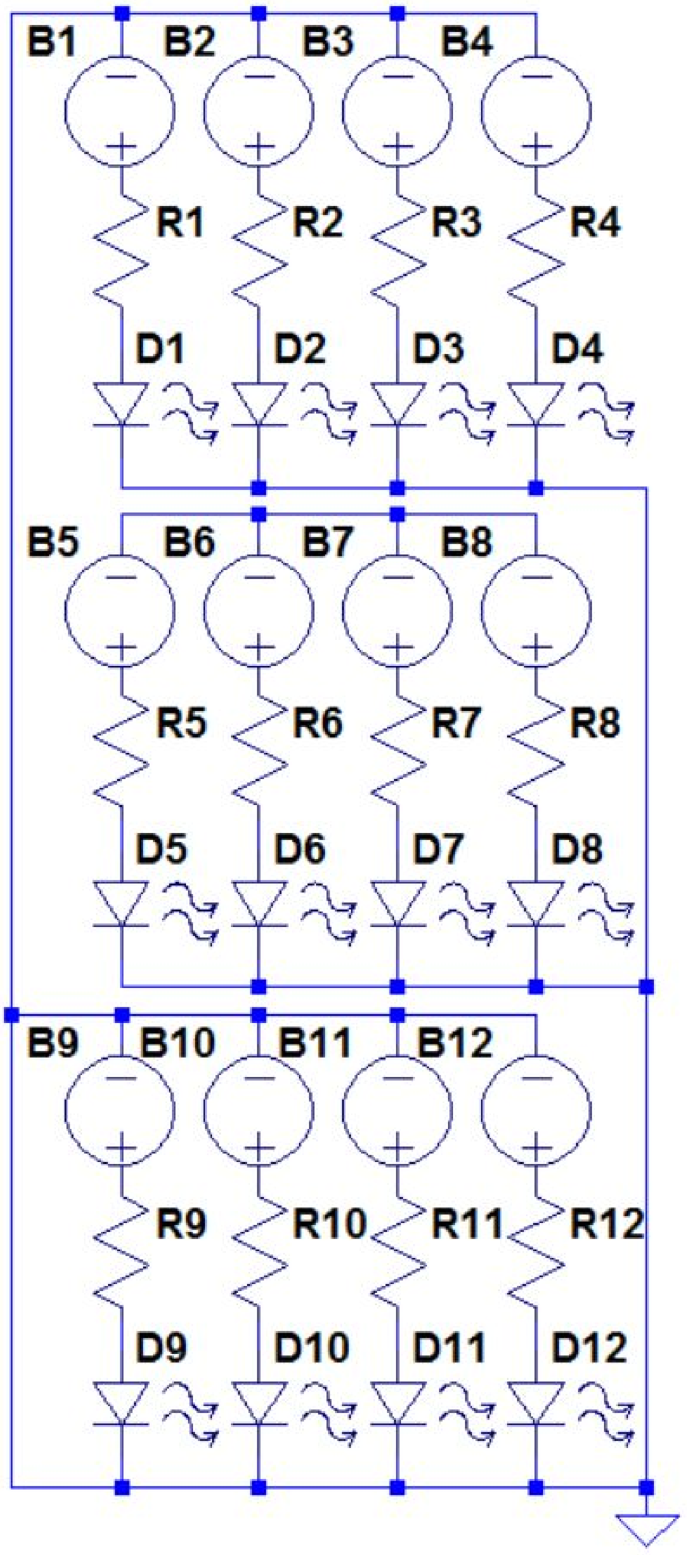


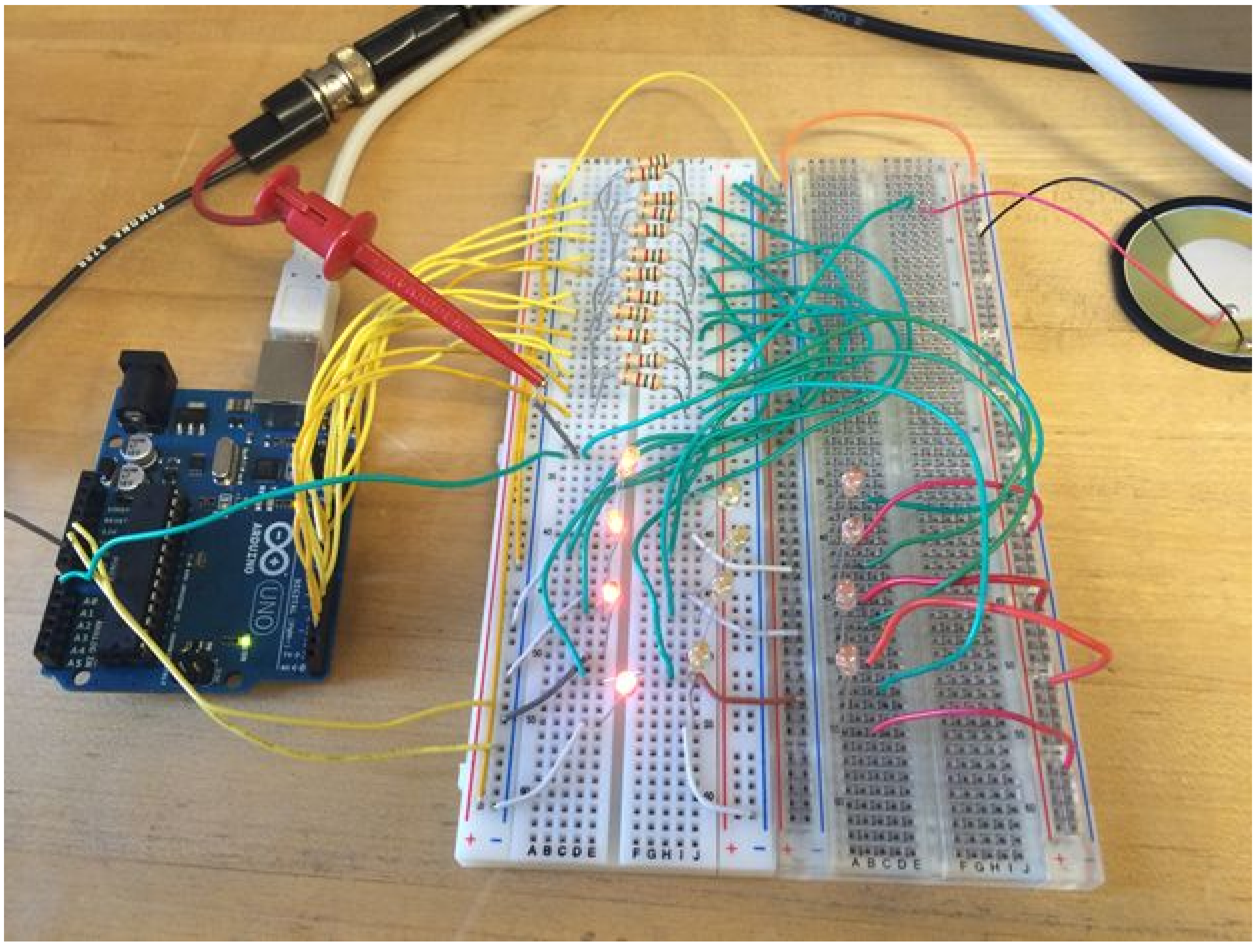
The period of the voltage signal across R_L depends on R, R_1, R_2 and the capacitance value used, as explained in the section below. The capacitance value is controlled by the switches shown on the left side of the schematic. If a switch is closed, the capacitor adjacent to that switch is used. If multiple switches are closed, the total capacitance becomes the sum of the individual capacitances corresponding to each switch (since the capacitors are in parallel).



Picture of the Function Generator

2. LED Display Circuit Diagram





Picture of LED Display Circuit with Arduino

This circuit consists of a grid of LEDs to display the waveform. Resistors are placed to step down the 5 V logic high from the Arduino to a usable value that won't fry the LED (1.5 kilo-ohm resistors are used). The voltage sources represent the logic values coming from the Arduino IO-pins (0 V or 5V DC). All LEDs share the common ground of the Arduino.

See Appendix A for complete Arduino Fourier transform code.

Methods

1. Function Generation

The function generator uses an op-amp, a capacitor, and resistors to transform a DC power source into an analog wave. This is done with a capacitor at the negative input voltage, which is initially charged up to $V_{cc}C\ R_1/(R_1+R_2)$. For simplicity's sake, we will represent $R_1/(R_1+R_2)$ with λ. Before the capacitor is charged up, the output voltage of the op-amp is equal to V_{cc}. Once the capacitor is charged up, it begins to discharge, at which point the voltage of the op-amp is equal to $-V_{cc.}$ This oscillating charging and discharging creates a square analog wave. By using the charging equation of the capacitor, we can compute a time for which it takes a capacitor to charge up, discharge, charge up again, and discharge again, which would be observed as one unit of the square wave. This amount of time, or the period of the square wave, would be:

$$T = 2RC \ln((1+\lambda)/(1-\lambda))$$

We used this equation to choose our values for resistors and the capacitors. First, for the values of the resistors R_1 ,R_2, and R, we chose 10k ohms for all three because we saw these resistor values as arbitrary. We did not vary the resistor values to change the frequency, so we simply picked resistor values that we had a great amount of in the lab. The resistors themselves are used for the voltage divider in the circuit and in the negative feedback of the op-amp, so although their values are arbitrary, there importance can not be understated. For the values of capacitance, we used 1 μF (1000 nF), 33 nF, and 460 nF. The reason for these values is due to the equation for frequency:

$$f = 1/T$$

This equation means that increasing values for C will decrease frequency, and decreasing values for C will increase frequency. Since we wanted to know the exact frequencies delivered to the waveform display, we chose capacitance values both readily available and fairly distinct from each other. The frequency obtained, since R is constant for all values, is then about $f = 1/(C \times 21972)$. So our three frequencies produced by the square wave generator were 45 Hz (1 μF), 99 Hz (460 nF), and 1379 Hz (33nF).

2. Waveform Analysis

The output of our function generator circuit is an analog waveform. This is routed into the A0 analog input pin of the Arduino. In order to analyze, in code, finite frequency domains and intensities, we must convert this signal to a digital representation.

Utilizing the Arduino's analog-to-digital converter (allowing us to take discrete measurements of the waveform, or sample it in the code), coupled with Fourier analysis, we are able to represent the signal as a set of discrete frequency magnitudes. The Fourier transform, by definition, is given by

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x)\, e^{-2\pi i x \xi}\, dx,$$

for any real number epsilon. Mathematically, the Fourier transform expresses a function of time in terms of amplitude of each of the frequencies that comprise it. While formally defined as an improper Riemannian integral (and with an inverse of the Gaussian function in the standard case), the real advantage of the Fourier transform lies in its application to signal processing and its implementation by means of computer algorithms.

The discrete Fourier transform (related to the continuous Fourier transform) converts a finite set of samples of a function into a set of Fourier coefficients that define a finite combination of sinusoids in order of frequency that has the same sample values (known as the frequency domain). The DFT is given by:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i 2\pi k \frac{n}{N}} \qquad k = 0, \dots, N-1.$$

Where N is the number of samples and x, X are the samples and output coefficients, respectively. The standard algorithm for computing a discrete Fourier transform is the Fast Fourier Transform, or FFT. Fourier algorithms are typically of the efficiency $O(n^2)$, whereas the bound on the efficiency of FFT is $O(n \times \log(n))$.

Due to computational limitations for the Arduino, we implement a fixed-value (fixed-place) finite Fourier transform, which is a modification to the discrete Fourier transform. Specifically, as exemplified by the code, our algorithm uses 128 samples per iteration for computing the Fourier transform and then bins frequencies into a set of 64. The standard complex magnitude,

$$|z| = \sqrt{zz^*} = \sqrt{(x+iy)(x-iy)}$$
$$= \sqrt{x^2 + y^2}$$

for any complex number z is utilized to calculate the intensity at a given frequency bin. From the frequency bins, we are able to analyze a representative subset which are then transformed to light the LEDs based upon the intensities of frequency bands. For example, if the intensity of frequencies on the lower end of the human hearing spectrum are above a threshold value, the bottom left LED on the grid would light. If these frequencies were increased in intensity, the column would continue to light, one LED higher at a time, or, if the signal was moved to a higher frequency, the LEDs would light on the row to the right, etc.

We observed this behavior by both testing our own waveform generator and watching the LEDs light in accordance with the frequency of the signal generated (audibly correlating higher pitches with higher LEDs lit). To more robustly test the waveform display, we used the lab Agilent waveform generator with sweep settings from 10 Hz to 1 kHz over a sweep time of 2.5s (logarithmic). We observed that the leds lit incrementally in accordance with the increase in pitch of the signal.

Notes

On the function generator:

For the function generator, we chose to use a series of capacitors and switches. This design is different than our original idea, which was to use a variable capacitor, because the variable capacitor obtained in the lab did not have a wide enough range to vary output frequencies of the waveform generator. We also thought of using a variable resistor to vary the frequency values, but the variable resistors available in the lab did not have the required resistance range either. Also, the interface of switches on the circuit is much more presentable and user friendly when wanting to observe the changes in the waveform display.

On the waveform display:

Using a true LED display such as that of a calculator or a true LED matrix (as developed by a lot of 3rd party companies) would allow for a much better display of the waveform in terms of frequency intensity. Further, the Arduino is a limitation in the signal processing we can do. Since the function generator already has some noise, it throws off the already janky A to D converter, which seems to be inherently bad. In this vein, our Arduino's poor processing capability necessitated an integer-transformed fixed-value finite Fourier transform whereas a more powerful arm processor, for example, or any processor for which we could actually compile C code, would allow for a much better processing algorithm. This is said following the identification of some inaccuracies and/or errors in the waveform to LED converter behavior. While on the whole it works, a better display and processor could yield much better results.

Abandon all hope, ye who enter here…
View code on github

Appendix A:
Arduino Code for LED Waveform Display

```
// FFT Waveform Display by Lucas Schuermann, 2014

// Uses FFFT to process signal on A0 using Arduino A-D converter
// Displays waveform using binned frequency intensities
// Led grid corresponds to
//    [1,4][2,4][3,4]
//    [1,3][2,3][3,3]
//    [1,2][2,2][3,2]
//    [1,1][2,1][3,1]
// using Arduino digital output pins 13-2
// speaker on digital pin 1 for waveform pass through (in circuit as
// well)
// FFT samples = 128 with bins = 64
//
// Based upon work by Olli-Pekka Heinisuo

#include <avr/pgmspace.h>

#include "pitches.h"

// led configuration
int led14 = 13;
int led13 = 12;
int led12 = 11;
int led11 = 10;
int led24 = 9;
int led23 = 8;
int led22 = 7;
int led21 = 6;
int led34 = 5;
int led33 = 4;
int led32 = 3;
int led31 = 2;

// speaker output
int speaker = 1;

 /* 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.

   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
   Adapted for Arduino: Anatoly Kuzmenko 6 Feb 2011
k_anatoly@hotmail.com
 */


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

 #define FFT_SIZE      128
 #define log2FFT       7
 #define N             (2 * FFT_SIZE)
 #define log2N         (log2FFT + 1)

 const prog_int16_t Sinewave[N_WAVE-N_WAVE/4] PROGMEM = {
   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
};

int fix_fft(int fr[], int fi[], int m )
{
  int mr, nn, i, j, l, k, istep, n, scale, shift;
  int qr, qi, tr, ti, wr, wi;

  n = 1 << m;

  /* 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) {
  shift = 1;
  istep = l << 1;
  for (m=0; m<l; ++m) {
    j = m << k;
    /* 0 <= j < N_WAVE/2 */
    wr =  pgm_read_word(&Sinewave[j+N_WAVE/4]);
    wi = -pgm_read_word(&Sinewave[j]);

    wr >>= 1;
    wi >>= 1;

    for (i=m; i<n; i+=istep) {
      j = i + l;
      tr = ((long)wr*(long)fr[j] - (long)wi*(long)fi[j])>>15;
```

```
          ti = ((long)wr*(long)fi[j] + (long)wi*(long)fr[j])>>15;
          qr = fr[i];
          qi = fi[i];

          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;
  }

  int fix_fftr(int f[], int m )
  {
    int i, Nt = 1<<(m-1),  scale = 0;
    int tt, *fr=f, *fi=&f[Nt];

    scale = fix_fft(fi, fr, m-1 );
    return scale;
  }

//*********************************************************************
*******************************

  int x[N], fx[N];
  int incomingByte;
  int i, count, scale;
  int b1 = 0, b2 = 0, b3 = 0;
  int b4 = 0, b5 = 0, b6 = 0;

  int sdvig = 32768; //DC bias of the ADC, approxim +2.5V.
(kompensaciya post. sostavlyauschei).
  int minim = 0;
  int maxim = 32000;
  int vrem;
  float kdmp = 0.95;  //Smoothing constant.
```

```
 float kary = 0.999;  //AGC time constant.

 void setup() {
    // set led pins
  pinMode(led14, OUTPUT);
  pinMode(led13, OUTPUT);
  pinMode(led12, OUTPUT);
  pinMode(led11, OUTPUT);
  pinMode(led24, OUTPUT);
  pinMode(led23, OUTPUT);
  pinMode(led22, OUTPUT);
  pinMode(led21, OUTPUT);
  pinMode(led34, OUTPUT);
  pinMode(led33, OUTPUT);
  pinMode(led32, OUTPUT);
  pinMode(led31, OUTPUT);

  // test speaker
  int melody[] = {
    NOTE_C4, NOTE_G3,NOTE_G3, NOTE_A3, NOTE_G3,0, NOTE_B3, NOTE_C4
};
  int noteDurations[] = {
    4, 8, 8, 4,4,4,4,4   };
  for (int thisNote = 0; thisNote < 8; thisNote++) {
    int noteDuration = 1000/noteDurations[thisNote];
    tone(speaker, melody[thisNote],noteDuration);
    int pauseBetweenNotes = noteDuration * 1.30;
    delay(pauseBetweenNotes);
    noTone(speaker);
  }
 }

 void loop()
 {

 //Filling up input raw data array x[];
 // 14.6 msec for ADC and 12.4 msec everything else, TOTAL = 27 msec
if FFT size = 64 (N=128).
 // 28.8 msec for ADC and 27.1 msec everything else, TOTAL = 55.9
msec if FFT size = 128 (N).

   ADCSRA = 0x87;
   // turn on adc, freq = 1/128 , 125 kHz.
   ADMUX = 0x60;
   //Bit 5 � ADLAR: ADC Left Adjust Result
   ADCSRA |= (1<<ADSC);
   //    while((ADCSRA&(1<<ADIF)) == 0); //Discard first conversion
result.
   while(!(ADCSRA & 0x10));

   for(i=0; i<N; i++ ) {
     ADCSRA |= (1<<ADSC);
     //    while((ADCSRA&(1<<ADIF)) == 0);
     while(!(ADCSRA & 0x10));

     x[i] = ADCL;
     x[i] += (ADCH << 8);
   }

   ADCSRA = 0x00;

   for (i=0; i<N; i++){
     x[i] -=  sdvig;
     if (i & 0x01)
       fx[(N+i)>>1] = x[i] ;
     else
       fx[i>>1] = x[i] ;
   }

 //Performing FFT, getting fx[] array, where each element represents
 //frequency bin with width 65 Hz.

   fix_fftr( fx, log2N );


 // Calculation of the magnitude:
   for (i=0; i<N/2; i++)
   {
     fx[i] = sqrt((long)fx[i] * (long)fx[i] + (long)fx[i+N/2] *
(long)fx[i+N/2]);
   }

   b1 = 0;
   b2 = 0;
   b3 = 0;
   b4 = 0;
   b5 = 0;
   b6 = 0;

   // examine frequency bins and modulate to bar values (similar to
eq bars in old stereo)
   for ( count = 40; count < 50; count++ )
   {
     b1 = b1 + fx[count];
   }
   if (b1 < 0) {
    b1 /= -1;
   }
   if (b1 > 25000) {
    b1 = 7;
    } else {
      b1 /= 3500;
    }
   for ( count = 51; count < 61; count++ )
   {
     b2 = b2 + fx[count];
   }
   if (b2 < 0) {
    b2 /= -1;
   }
    if (b2 > 25000) {
    b1 = 7;
    } else {
      b2 /= 3500;
    }
   for ( count = 62; count < 72; count++ )
   {
     b3 = b3 + fx[count];
   }
     if (b3 > 25000) {
    b1 = 7;
    } else {
      b3 /= 3500;
    }
   for ( count = 73; count < 83; count++ )
   {
     b4 = b4 + fx[count];
   }
    if (b4 > 25000) {
```

```
   b1 = 7;
   } else {
     b4 /= 3500;
   }
  for ( count = 84; count < 94; count++ )
  {
    b5 = b5 + fx[count];
  }
   if (b5 > 25000) {
   b1 = 7;
   } else {
     b5 /= 3500;
   }
  for ( count = 95; count < 110; count++ )
  {
    b6 = b6 + fx[count];
  }
    if (b6 > 25000) {
   b1 = 7;
   } else {
     b6 /= 3500;
   }

// write to LEDs based upon analyzed bins
if(b1 > 0)
{
  digitalWrite(led11, HIGH);
  if(b1 >= 3)
    digitalWrite(led12, HIGH);
  if(b1 >= 5)
    digitalWrite(led13, HIGH);
  if(b1 >= 7)
    digitalWrite(led14, HIGH);
}
else
{
  digitalWrite(led11, LOW);
  digitalWrite(led12, LOW);
  digitalWrite(led13, LOW);
  digitalWrite(led14, LOW);
}

if(b3 > 48)
```

```
  {
    digitalWrite(led21, HIGH);
    if(b3 >= 50)
      digitalWrite(led22, HIGH);
    if(b3 >= 53)
      digitalWrite(led23, HIGH);
    if(b3 >= 56)
      digitalWrite(led24, HIGH);
  }
  else
  {
    digitalWrite(led21, LOW);
    digitalWrite(led22, LOW);
    digitalWrite(led23, LOW);
    digitalWrite(led24, LOW);
  }

  if(b6 > 104)
  {
    digitalWrite(led31, HIGH);
    if(b6 >= 106)
      digitalWrite(led32, HIGH);
    if(b6 >= 109)
      digitalWrite(led33, HIGH);
    if(b6 >= 112)
      digitalWrite(led34, HIGH);
  }
  else
  {
    digitalWrite(led31, LOW);
    digitalWrite(led32, LOW);
    digitalWrite(led33, LOW);
    digitalWrite(led34, LOW);
  }
}
```