**Signal analysis**

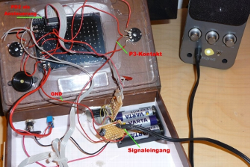
by Piotr Platek

[](http://www.elo-web.de/ximage/1001FFT1.jpg)

The Retro Arcade circuit board from Haynes provides not only a retro gaming platform, but also a development environment, with which very varied fields of application can be tested out. Here I would like to introduce another idea. The Atmega8 microcontroller has an analogue to digital converter (ADC = Analogue-to-Digital-Converter), which is used by the firmware in the Retro Arcade game to read the potentiometer settings and the corresponding driving of the table tennis bats. In addition, the circuit board, with its 12x10 LED matrix is well suited to the simple, graphic presentation of measured values. The two characteristics of the circuit board – the analogue-to-digital converter and 120 freely controllable LEDs – have inspired me with the idea of the ‘visualisation’ of music and audio signals. Everyone knows about this from music systems and computer programs, which along with playing back the music, display coloured bars in time with the rhythm of the music. The bars correspond to the strength and frequency of the signal. The signal analysis can be carried out via various means. One way is the ‘Discrete Fourier Transformation’. The complex mathematical transformation is carried out using a fast algorithm – ‘FFT’ (Fast Fourier Transform). As its result, the FFT delivers the principally occurring frequencies of the sampled signal, with their respective amplitudes, which are shown as ‘coloured bars’ on the Retro Arcade display.

In itself, the FFT algorithm uses floating-point arithmetic (for the calculation of trigonometric functions), which nonetheless presents something of a challenge for the small Atmega8 micro controller. The calculation also has to be carried out in real time, so that the rhythm of the music can be followed. For the poor ATmega8 that is definitely too much. Due to the relatively simple graphic display, the highest level of accuracy can be dispensed with and the simpler floating-point arithmetic can be used.

In addition, the trigonometric values can be determined in advance and saved in a table as fixed values. With this procedure, the FFT can be coded in Assembler very effectively. So as not to reinvent the wheel, I’ve done some research on the Internet and I came across the site [**http://elm-chan.org/works/akilcd/report\_e.html**](http://elm-chan.org/works/akilcd/report_e.html), where a finished FFT solution is helpfully presented with a GPL model. This solution has 32 values per analysis run as its lowest resolution. However, the Retro Arcade circuit board only offers 12 LED columns. Hence the output result has to be adjusted in the program accordingly. This can be implemented as part of the FFT or afterwards. I have implemented and tested both cases. I have limited the resolution of the FFT to 16 values (and have also made a small adjustment in the file ffft.S). From the original 32 values, in each case, two neighbouring frequency values are combined and only the larger of the two is displayed. In addition, the two lowest as well as the two highest frequencies (plus direct-current components) are omitted.

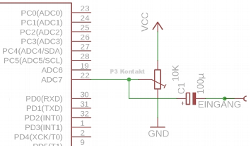
[](http://www.elo-web.de/ximage/1001FFT3.jpg)

Besides this, my program also provides various options for the display. The bars can fall slowly, follow the signal rapidly and only leave behind a maximum value or only display the maximum values. To change the display, I use the available port D, pin PD2 as a driver. Pin PD2 is connected to the mass of the device via a micro button. Consequently, please note that I have slightly changed the initialisation of the port and the Interrupt routine.

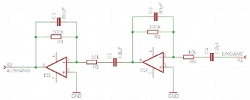
[](http://www.elo-web.de/ximage/1001FFT2.jpg)

As a further option (of the display), I have implemented a kind of oscilloscope. In this mode the time basis can be set with the left potentiometer. However, the resolution of the display, (low) sampling rate and signal input are not suitable for a ‘professional’ oscilloscope.

Even if the display and accuracy of the frequency analysis outlined above are surprisingly good, we should still briefly discuss the signal input here, once again. In the basic version, the audio signal is connected directly from the headphone input via a capacitor (100 µF) with the P3 contact.

[](http://www.elo-web.de/ximage/1001FFT4.jpg)

If anyone would like to develop the ‘correct’ audio frequency analyser, the signal input should have a low-pass filter to achieve a smoothing of the signal. A frequency range filter would be even better, which is designed (calculated) for the audio frequency range present. As an example, here I’d like to suggest a frequency range filter that works with active elements (operational amplifier with passive elements), which is designed (calculated) for a frequency range of 20 Hz-20 kHz. The two inverting amplifiers in series ensure a clean output signal and a good frequency response.

[](http://www.elo-web.de/ximage/1001FFT5.jpg)

Download: [**Pong FFT**](http://www.elo-web.de/xattachment/pong-fft-v2.02.zip)

See also: [**http://blogs.zobniow.net/micro**](http://blogs.zobniow.net/micro)

/\*  
\* Audio spectrum analyser for Retro Arcade board  
\*  
\* Fixed-point FFT routines for megaAVRs (C)ChaN, 2005  
\* http://elm-chan.org/works/akilcd/report\_e.html  
\*  
\* Ready for AVR Studio 4 or WinAVR  
\* Modified multiplexing display procedure based on the code from  
\* Laufschrift program written by Sascha Bader  
\*  
\* Ver. Date Author Comments  
\* ------- ---------- -------------- ------------------------------  
\* 1.00 25.12.2009 Piotr Platek initial  
\* 2.00 16.01.2009 Piotr Platek correct sampling method, adding display modes   
\* 2.01 17.01.2009 Piotr Platek changed display frequency to get better sampling accuracy  
\* 2.02 17.01.2009 Piotr Platek Mode switch moved to PD3 (PD2 is connected to potentiometer  
\* added oscoloscope mode with different time x-scale  
\*  
\*  
\* Feel free to provide me with any feedback   
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
\*\*  
\*\* This program is free software; you can redistribute it and/or  
\*\* modify it under the terms of the GNU General Public License  
\*\* as published by the Free Software Foundation; either version 2  
\*\* of the License, or (at your option) any later version.  
\*\*  
\*\* This program is distributed in the hope that it will be useful,  
\*\* but WITHOUT ANY WARRANTY; without even the implied warranty of  
\*\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the  
\*\* GNU General Public License for more details.  
\*\*  
\*\* You should have received a copy of the GNU General Public License  
\*\* along with this program; if not, write to the Free Software Foundation,   
\*\* Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
\*/  
  
#define F\_CPU 8000000UL  
  
#include <avr/io.h>  
#include <stdlib.h>  
#include <avr/pgmspace.h>  
#include <avr/interrupt.h>  
#include <avr/eeprom.h>  
#include <avr/sleep.h>  
#include <stdio.h>  
#include <inttypes.h>  
#include <ctype.h>  
#include <util/delay.h>  
#include <math.h>  
#include "display.h"  
#include "ffft.h"  
  
  
/\* defines \*/  
  
#define FFT\_SIZE (FFT\_N/2) /\* number of result to be plotted \*/

uint16\_t leds[SWIDTH]; /\* Screen memory definition \*/  
volatile uint16\_t offset = 0; /\* index for data buffer collection \*/

#define FALLDELAY (80) /\* falling down delay for the frequency bars \*/  
volatile uint16\_t delta1 = 0; /\* delay counter \*/

#define PD3DELAY (25) /\* delay for button detections \*/  
#define DEBOUNCING (1) /\* debouncing button \*/

volatile uint16\_t pd3\_delay = 0; // counter for button test procedure   
volatile int8\_t pd3\_processed= 1; // all events processed

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
\* SIG\_OVERFLOW0  
\*  
\* This routine is run every time the timer counter TCCR0 overflows.   
\*   
\*   
\* Task:  
\* 1) decrease counter of falling down freq. bars based on the peak value  
\* 2) gather data from ADAC  
\* 3) drive the screen  
\*   
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
SIGNAL(SIG\_OVERFLOW0) {  
uint16\_t ledval;  
uint8\_t portdout;  
uint8\_t portcout;

static uint8\_t col = 0;

cli();

if (pd3\_processed) // check only if the previous key event was processed  
{  
if (bit\_is\_clear(PIND, PD3)) // check the button PD3  
{  
if (pd3\_delay < PD3DELAY) pd3\_delay++;  
}  
else  
{  
if (pd3\_delay < DEBOUNCING) // filter out button bouncing  
{  
pd3\_delay = 0; // start new process it was just a short bump  
}  
else  
{  
pd3\_processed = 0; // stop checking PD3 as long as not proceed in main loop  
pd3\_delay = 0; // start new process it was just a short bump  
}  
}  
}

if (delta1) delta1--; // as long as not 0 decrease the delay for falling down bars

if (col == SWIDTH) col = 0;

PORTB = 0;  
portdout = PORTD & 0b00001111; // save lower first 4 bits and zero higher 4  
PORTD = portdout;  
portcout = PORTC & 0b11110000; // save higher first 4 bits and zero lower 4  
PORTC = portcout;

if ( col == 0 ) PORTB &= ~(1 << 4);   
else PORTB |= (1 << 4);

PORTB |= (1 << 3);  
PORTB &= ~(1 << 3);

PORTB |= (1 << 2);  
PORTB &= ~(1 << 2);

ledval = leds[col++];  
  
PORTC = (ledval & 0x000f) | portcout;  
PORTD = (ledval & 0x00f0) | portdout;  
PORTB = (ledval >> 8) & 0x0003;

sei();  
}  
  
  
int main(void)  
{

int16\_t capture[FFT\_N]; /\* Wave capturing buffer \*/  
complex\_t bfly\_buff[FFT\_N]; /\* FFT buffer \*/  
uint16\_t spektrum[FFT\_N/2]; /\* Spectrum output buffer \*/

uint8\_t k, m;  
int16\_t SpecVal;  
uint16\_t peaks[SWIDTH];

int8\_t mode = 0;

uint8\_t PotiSwitch = 0b0111;  
int16\_t TimeScale = 100;  
int16\_t TimeScaleCounter = 0;

/\* initialize port data directions \*/  
DDRC = 0x0f; // 0x0f PORTC7.6 as AD input

// this is important to setup PD3 as input  
DDRD = 0b11110111; // Portd = Output but PD3  
PORTD |= 0b00001000; // activate pullup at PD3

/\* initialize timers \*/  
TCCR0 |= \_BV(CS01) | \_BV(CS00); // clk prescale (clk/64)  
TIMSK |= \_BV(TOIE0); // enable timer/counter0 overflow interrupt   
TCNT0 = 0; // reset the timer  
  
/\* initialize the ADC \*/  
// ADMUX |= \_BV(ADLAR); // we only want 8-bit  
ADMUX |= \_BV(MUX2) | \_BV(MUX1) | \_BV(MUX0);  
// ADMUX |= \_BV(REFS1) | \_BV(REFS0);  
ADMUX |= \_BV(REFS0);

// Setup ADC sampling clock   
// ADCSRA &= ~\_BV(ADPS2) & ~\_BV(ADPS1) & ~\_BV(ADPS0); // ADC=CLK/2  
// ADCSRA |= \_BV(ADPS1); // ADC=CLK/4  
// ADCSRA |= \_BV(ADPS1) | \_BV(ADPS0); // ADC=CLK/8  
// ADCSRA |= \_BV(ADPS2); // ADC=CLK/16  
ADCSRA |= \_BV(ADPS2) | \_BV(ADPS0); // ADC=CLK/32  
// ADCSRA |= \_BV(ADPS2) | \_BV(ADPS1); // ADC=CLK/64  
// ADCSRA |= \_BV(ADPS2) | \_BV(ADPS1) | \_BV(ADPS1); // ADC=CLK/128

ADCSRA |= \_BV(ADEN);   
ADCSRA |= \_BV(ADSC); // Start the first conversion  
  
sei(); //enable global interrupts  
  
for (m=0; m < SWIDTH; peaks[m++]=0);

while(1)  
{  
if (offset >= FFT\_N) // buffer of samples is full - ready to process FFT  
{   
// process FFT  
fft\_input(capture, bfly\_buff);   
fft\_execute(bfly\_buff);  
fft\_output(bfly\_buff, spektrum);  
offset = 0; // release the next buffer data collection

m = 0; // start the index for peaks and screen memory

for (k=2; k <= FFT\_SIZE-1; k+=1) // the other spectrum modes  
{  
SpecVal = spektrum[k];  
// SpecVal = spektrum[k] > spektrum[k+1] ? spektrum[k] : spektrum[k+1];

if (m <= SWIDTH-1) // Process first SWIDTH-1 results  
{  
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
\* There is no true filter on input.  
\* I tried to eliminate the constant part of the signal and some disruption   
\* and noise by introducing a software filter  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

if (SpecVal <= 4 ) SpecVal = 0;  
else SpecVal /= 3;

if (SpecVal > 10 ) SpecVal = 10; // if there is a peak bigger then 10, make it 10

if (SpecVal > peaks[m]) peaks[m] = SpecVal; // save a maximum value in the peak table   
  
  
// Display results in one of 3 modes  
switch (mode)  
{  
case 0: // falling down bars  
leds[m] =  
((FULLBAR << (10-SpecVal)) & FULLBAR) |  
((FULLBAR << (10-peaks[m])) & FULLBAR);  
break;  
case 1: // real time bars + falling down peaks  
leds[m] =  
((FULLBAR << (10-SpecVal)) & FULLBAR) |  
((0b0000000000000001 << (10-peaks[m])) & FULLBAR);  
break;  
case 2: // falling down peaks  
leds[m] = ((0b0000000000000001 << (10-peaks[m])) & FULLBAR);  
break;  
case 3: // real time bars  
leds[m] = ((FULLBAR << (10-SpecVal)) & FULLBAR);  
break;  
case 4: // Oscilloscope  
break;  
default: // clear display  
leds[m] = 0;  
}  
m++; // proceed to the next point  
}  
}  
}  
else  
if (mode==4) // Oscilloscope mode  
{

if (!bit\_is\_set(ADCSRA, ADSC)) // check if conversion is ready  
{  
if ( PotiSwitch == 0b0111) // check the side ADC7 = Osci. Input  
{  
for (m=0; m < SWIDTH-1 ; m++) // scroll screen from right to left  
{  
leds[m] = leds[m+1];   
}  
SpecVal = ADC;  
SpecVal /= 51; // depends on the input voltage  
leds[SWIDTH-1] = 1 << (9-SpecVal);  
// if (TimeScaleCounter > 50)  
{  
ADMUX &= ~\_BV(MUX0);   
PotiSwitch = 0b0110; // Next measurement for ADC6   
TimeScaleCounter = 0; // Next ADC for Time Scale  
}  
// else TimeScaleCounter++;  
ADCSRA |= \_BV(ADSC); // Already start conversion  
\_delay\_ms(TimeScale);  
}  
else // the other side -< ADC6 = Left potentiometer  
{  
TimeScale = ADC;  
TimeScale /= 33;  
ADMUX |= \_BV(MUX0);  
PotiSwitch =0b0111;  
ADCSRA |= \_BV(ADSC); // start the next ADC conversion  
}  
}  
}  
else  
{ // Gather next set of samples  
// cli(); // for high screen refresh frequency  
// it is recommended to disable interrupts  
// for sampling time

ADMUX |= \_BV(MUX2) | \_BV(MUX1) | \_BV(MUX0); // Ensurethe correct input line  
do   
{  
if (!bit\_is\_set(ADCSRA, ADSC)) // check if conversion is ready  
{   
capture[offset++] = ADC-32768; // convert to signed linear  
ADCSRA |= \_BV(ADSC); // start the next ADC conversion  
}   
} while (offset < FFT\_N);  
// sei(); // see cli() above remarks

if (!delta1) // the time for bar fall down has elapsed  
{   
cli(); // see comment above  
delta1 = FALLDELAY; // setup a new delay for next fall down  
sei();  
for (m=0; m <= SWIDTH-1; m++)  
{  
if (peaks[m]) peaks[m]--; // decrease all peaks by 1  
};   
}   
}  
// check keyboard  
if (!pd3\_processed) // there is not yet processed key event  
{  
  
mode ++;  
if (mode > 4) mode = 0;

pd3\_delay = 0; // start the new check for key event  
pd3\_processed = 1; // --//--  
}  
}  
}