High quality, low cost real time motion capture unit for a PC
(some assembly required)

Stefan Gustavson, ITN-LiTH (stegu@itn.liu.se) 2002-01-26

I have built a real time motion capture input system which connects to the parallel port of a low-end standard PC and is capable of sampling up to 64 channels with 12 bits of precision at a sampling rate of around 60,000 samples per second, i.e. up to 1000 Hz sample rate in a 64 channel setup, and even higher rates for fewer channels. The system is not based on telemetry, but on physical angle sensors in the form of high quality potentiometers attached to the joints of a skeleton rig. I use the device for a puppet-size digital armature, but the potentiometers could also be mounted on a human-size exoskeleton to be worn by an actor.

Exoskeleton motion capture systems are a bit clumsy to wear and do not allow wild acrobatic movements, but commercial full-body motion capture systems based on this quite straightforward and simple idea are in fact available from the company Meta Motion (http://www.metamotion.com/). However, the price tags for their systems start at $10,000, so it’s still extremely expensive stuff compared to what I spent on my hardware. Of course, their system is comfortable and cordless, and comes with proper support for commercial animation software, but I wasn’t really interested in that. I just wanted fast and accurate real time position information from a digital armature into my own software, on a very low budget.

Compared to telemetric systems (optical or magnetic tracking devices), physical angle sensor systems have high accuracy, high speed, totally unnoticeable time delay for the capture data, and a low hardware cost. An entire prototype system with eight sensors cost me less than $300 in hardware, and I built it in a few days. This is definitely not rocket science. I figure anyone with some hobby electronics experience can assemble the circuit, and the mechanical design of the skeleton rig is also quite simple. I am not a professional hardware designer, nor an avid prototype builder, just a reasonably experienced hobbyist with a degree in electronic engineering to help me know what I am doing when I design stuff.

Computer interface

For the computer interface, I found that a serial port would not provide enough bandwidth for dozens of channels of 12 bit data captured at a high sample rate. For a short while I looked into building a USB interface, but although its transfer speed is more than sufficient, making the interface a USB device would require it to be built using a microcontroller, and I did not really want to design anything that complicated. I wanted to make the hardware cheap and simple to construct without any special equipment. Therefore, I opted for the trusty parallel port.

The parallel port is an old and outdated interface, and under normal circumstances I would really discourage anyone from using it, but it has some pretty nice features for this particular application. First, it is fully programmable on a very low level. If you choose not to use system-level I/O routines to access the port, it can be controlled right down to individual bits with microsecond timing accuracy. The port has quite a few input and output pins, and if you are only going to hook up your own custom device to the port, each pin can be put to any use, regardless of its intended use for a standard parallel port device. There are no hardware or software timing constraints and no protocol, and everything is sent and received with TTL logic levels. Taking all this into consideration, the parallel port was a pretty obvious choice for this project.
Mechanical hardware

Four of the input sensors for my prototype system are mounted on a skeleton arm that was thrown together rather quickly from brass and aluminum. The arm is shown in figure 1. It is about 50 cm long, and equipped with a shoulder joint and an elbow joint. For clarity, I also made some quick 3D renderings of the principle behind the three different types of joints needed: a saddle joint, a hinge joint and a rotational joint.
Hinge and rotation joints have one degree of freedom (one potentiometer), saddle joints have two. These three types of joints are sufficient to build a general articulated structure, including but not limited to a humanoid figure. Ball joints, with three degrees of freedom, can be built by combining a saddle joint and a rotation joint. A human arm, not including the hand, would consist of the following five joints in order from the shoulder to the wrist, corresponding to a total of seven degrees of freedom. My prototype arm implements the first three of these joints and thus has four degrees of freedom.

<table>
<thead>
<tr>
<th>Shoulder</th>
<th>Shoulder twist</th>
<th>Elbow</th>
<th>Lower arm twist</th>
<th>Wrist</th>
</tr>
</thead>
<tbody>
<tr>
<td>saddle</td>
<td>rotation</td>
<td>hinge</td>
<td>rotation</td>
<td>saddle</td>
</tr>
</tbody>
</table>

*Figure 3: Overview of the entire system*
Figure 4: Schematic diagram for the circuitry
Functionally, the circuit is very straightforward, but I’ll explain it in some detail anyway.

Six address bits are sent from the data lines D0 through D5 of the parallel port. Bits D5, D4, D3 enable one of eight analog multiplexers (74HC4351) by means of a 3-to-8 decoder (74HC138). Bits D2, D1, D0 select one of eight inputs on the analog multiplexer and connect it to the A/D converter (ADS774). An operational amplifier (OP184) in a unity gain connection is used as an impedance buffer between the multiplexer outputs and the A/D converter. The A/D conversion is initiated by a control bit from the parallel port (C2). The 12 bits of digital data are read out in four-bit nybbles, because the parallel port does not have enough input lines. The selection between the three nybbles is done by two 8-to-4 multiplexers (74HCT138) addressed by the control lines C0 and C1, and the data is sent to status lines S3-S6. Status line S7 is used to signal when the conversion is finished. The data lines D0 to D5 are buffered and level converted by open-collector inverters (74LS05) and glitch filtered by capacitors (1nF). The control and status lines are also sent though buffers (74LS244), with glitch filtering capacitors for the data sent from the PC (3.3 nF).

As you can see from the diagram, there are two supply voltages involved: 5V and 9V. I designed it like that to make good use of the fixed unipolar range 0-10 V of the A/D converter. This circuit needs a very stable analog supply voltage, or you will get problems with the accuracy of the digital output. 12 bits is 4096 steps, so the LSB of the data word corresponds to a mere 2.2 mV change in input voltage. First I tapped the internal power supply of the PC for +5V and +12V, and regulated the 12V down to 9V with a 7809 voltage regulator and a couple of capacitors. That provided a stable enough voltage for the 12-bit A/D conversion to stay within ±0.5 LSB fluctuations. A 10V supply would have improved the signal-to-noise ratio some more, but the relatively uncommon 7810 10V regulators were not in stock when I shopped for components.

Later, I switched to a separate external power supply for the circuit, because I wanted to be able to use the device with a laptop computer. I happened to have an old 12V/5V dual supply around, but a single 12V supply could of course have been used with two voltage regulators to get both 9V and 5V supply.

For proper 9V operation, make sure you use 74HC138 and 74HC4351, not the HCT versions. The 74LS06 open collector gates are required to convert TTL levels to 9V CMOS levels. You could use HCT versions powered by 5V to get rid of the dual supply voltages and the open collector gates, but that would cost you one bit of A/D resolution, because half of the analog 10V range would be inaccessible. If a 5V supply were used for the analog part, it should in any case be made separate from the digital supply to reduce noise, so you would have a dual supply design anyway.

The glitch filter capacitors were very much required on the computer I used for testing. Brief but strong noise spikes appeared on all outputs whenever anything was changed on either the data or control lines, and the circuit went crazy without the capacitors. I needed an oscilloscope to find that problem. Other parallel ports might be of better quality, but the capacitors won’t hurt. The capacitor values might need to be changed for your particular parallel port, but I tested my circuit with a number of different PCs, and the values shown in the diagram always worked for me.

The ADS774 A/D converter, manufactured by Burr & Brown, was a perfect fit: 12 bits, parallel output, high speed, stand-alone and self-clocked operation, internal sample-and-hold, built-in voltage reference and single +5V supply voltage. It’s a bit expensive (around $25), but it requires almost no external components, only a single 50Ω resistor. There are other A/D converters on the market that would do the job, but this one was the best I found.

One of the two 74HCT157 multiplexers could actually be replaced with an internal multiplexing function of the A/D converter, but that would have required one more control line than this solution, so I chose to use external multiplexers only. 74HCT157 may safely be replaced by 74LS157 if you wish.
The OP184 amplifier between the 74HC4351 analog multiplexers and the A/D converter is a key element of the analog circuit. The input impedance of the ADS774 is quite low, but the output impedance from a potentiometer piped through an analog multiplexer is high and variable. The OP184 is a very suitable component for buffering between them. It combines low noise with a fairly high speed, and it can cope with input and output swings all the way from 0V to the positive supply voltage. For a first prototype, I actually used a cheap LM324 with reasonably good results, but it was too slow, and its swing for unity gain was limited to some 1.5V below the positive supply voltage, thus reducing the working range for the conversion. The OP184 is manufactured by Analog Devices and costs around $7. The RC link (100\,\Omega, 4.7 \,\text{nF}) at its output is not really required, but it reduces the high-frequency ringing oscillations at the sharp voltage transitions between channels, and it might also reduce high-frequency analog noise somewhat. Do not use a too large capacitor here, as that would make the voltage transitions too slow for accurate full speed operation. 4.7 \,\text{nF} is a safe value.

Even though the specifications for OP184 boldly state that it should be stable at unity gain and be able to cope with input and output swings all the way to the negative and positive supply voltages, I ran into intermittent oscillation problems when the input was rapidly brought to precisely 0V or 9V. Hence the small fixed resistors at the potentiometers, which limit the extreme sensor voltages to about 0.1 V short of either supply rail.

The input sensors should be high quality potentiometers. It is crucial that you use premium quality hardware here, or else your data will be noisy and inaccurate, and your system will wear out quickly. A lower grade potentiometer will cost you less than $1, but it is designed to last only for about 10,000 rotations under a minimal amount of mechanical strain. The ones I used are BI Technologies model number 6187 (see http://www.bitechnologies.com), 10k\,\Omega. They have a steel axle in a sturdy bronze bearing, a sealed housing and a resistor made of a conductive polymer instead of carbon. Their specified lifetime expectancy is 5,000,000 turns. The plastic resistor material is very smooth and homogeneous, which makes the slider almost noise free, provides a very high angle resolution, and keeps the linearity error to within 1%. They cost around $10 each, but they are well worth the money, believe me. They are available with a free 360° rotating axle, which is a very handy feature for this application.

The analog inputs are connected directly to the slider terminal of each potentiometer. The end terminals of the potentiometer are connected to +9V and ground, respectively, though a small resistor of 120\,\Omega (the exact value is not critical, I just happen to have a lot of 120\,\Omega resistors), primarily to limit the input voltage to a slightly smaller swing than the full 9V, and also in order not to risk a high-current short circuit in case of some fault in the connections or in the sensor cable.

The electromagnetic environment near a PC is quite busy with lots of different frequencies humming around in the air, so to reduce noise pickup, I used shielded 3-lead cable to connect the potentiometers to the inputs. The shield was connected to ground at the analog input end. If you have trouble finding 3-lead shielded cable, a 2-lead shielded cable with the shield used for the most negative sensor terminal might work well enough. Using an unshielded cable for anything else than very short connections would most certainly be a bad idea.
The prototype board

Top row of IC:s from left: 7809, 74HC4351, OP184, ADS774.
Bottom row of IC:s from left: 74LS06, 74HC138, 74LS244, 74HCT157, 74HCT157.
Left edge: +12V, +5V and GND from power supply. Top edge: four analog sensor inputs. Bottom edge: parallel port connections.

The prototype above has only one 74HC4351, but includes the 74HC138 anyway for expandability. If the analog inputs were to be connected in a more compact way, there is room for several additional 4351’s on the board.

If you look carefully, you notice that the OP184 is a bug-sized SO8 surface mount package, hand soldered to a small home-made PCB to plug into the prototype board. I did that because the DIL8 package version was out of stock, but I would not recommend you to try it. SO8 packages are really too small to handle.
Control software

The hardware described above is extremely dumb, and relies on the host computer software to provide all the control and timing information. My control software uses the CPU for such mundane tasks as waiting in a 100% busy loop for a bit to flip, or creating a delay of a few microseconds. This is not exactly good CPU utilisation, but it was of little concern to me. I designed this system to be connected to a PC that would be dedicated to reading motion capture data and serving it over the network. The PC in question does not have to be fast or good. In fact, any old piece of junk in working condition will do, so finding a suitable dedicated interface computer was not a problem. I used a 200 MHz K6 system with 64 MB of memory, which actually turned out to be a lot more than I needed. The software I wrote would probably run just fine on an old 486, or even a 386, even though I haven’t tested that.

The software to control the device is written for Linux. It would have been perfectly possible to write it for Windows, or even for DOS, but Linux was a more convenient and more familiar programming environment to me. When I get around to building the full skeleton rig and want to actually start using it, I will probably write a Linux server that serves motion capture data over UDP/IP (or maybe even ask a colleague to write it), but for now, this is what I have. The code below compiles with GCC to a small program that tests the basic functionality of the hardware by alternately converting analog channels number 0 and 1. With the high speed analog amplifier OP184, the code below can read up to 60,000 accurate samples per second. With an LM324, the speed needs to be reduced by nanosleep() calls, or simply a few extra dummy port reads, to a few thousand samples per second, or else the channels will interfere with each other because of a too long settling time for the amplifier output. If you use a buffer amplifier other than the OP184, use an oscilloscope to find out how your sample timing matches the settling time of the analog signal.

```c
/* Test motion capture unit on parallel port. */
/* Stefan Gustavson 2001-09-15 */
/* NOTE: compile with -O2, or things will break! */
/* Also, this program needs root permissions. */

#include <stdio.h>
#include <unistd.h>
#include <asm/io.h>

#define BASEPORT 0x378
#define DATAPORT BASEPORT
#define STATUSPORT BASEPORT+1
#define CONTROLPORT BASEPORT+2

int main()
{
    int i=0, nwaits=0, dataword[2]={0,0}, nybble=0;
    if(ioperm(BASEPORT,3,1)) { perror("ioperm"); exit(1); }

    // Select channel 0 for the first conversion
    outb(0x07, DATAPORT); // Channel 0 - address bits are inverted
    // Init control port
    outb(0x07, CONTROLPORT);
```
while(1) {
    // Convert channel 0
    // Start conversion by flipping C2 to zero briefly
    outb(0x03, CONTROLPORT);
    outb(0x07, CONTROLPORT); // Prepare for reading, select low nybble
    // We have a sample-and-hold ADC, so we can switch channels now
    outb(0x06, DATAPORT); // Select channel 1 for next conversion
    // Wait for conversion to finish - about 5 waits are needed
    for(nwaits=0; (inb(STATUSPORT) & 0x80) == 0; nwaits++);
    //printf("Conversion took %d waits.\n",nwaits);
    nybble = inb(STATUSPORT) & 0x78; // Mask out bits S6,S5,S4,S3
    dataword[0] = nybble >> 3;
    outb(0x06, CONTROLPORT); // Select middle nybble
    nybble = inb(STATUSPORT) & 0x78;
    dataword[0] |= (nybble << 1);
    outb(0x05, CONTROLPORT); // Select high nybble
    nybble = inb(STATUSPORT) & 0x78;
    dataword[0] |= (nybble << 5);
    // Delay here if a slower analog buffer is used

    // Convert channel 1 - identical code except for the address
    // Start conversion by flipping C2 to zero briefly
    outb(0x03, CONTROLPORT);
    outb(0x07, CONTROLPORT); // Prepare for reading, select low nybble
    outb(0x07, DATAPORT); // Select channel 0 for next conversion
    for(nwaits=0; (inb(STATUSPORT) & 0x80) == 0; nwaits++);
    //printf("Conversion took %d waits.\n",nwaits);
    nybble = inb(STATUSPORT) & 0x78; // Mask out bits S6,S5,S4,S3
    dataword[1] = nybble >> 3;
    outb(0x06, CONTROLPORT); // Select middle nybble
    nybble = inb(STATUSPORT) & 0x78;
    dataword[1] |= (nybble << 1);
    outb(0x05, CONTROLPORT); // Select high nybble
    nybble = inb(STATUSPORT) & 0x78;
    dataword[1] |= (nybble << 5);
    // Delay here if a slower analog buffer is used

    if(i++ >= 1000) { // Print one sample every 1000 conversions
        printf("0: %4d 1: %4d\n", dataword[0], dataword[1]);
        i=0;
    }
}
}
OpenGL test program

Just to see the prototype in action, I also threw together a very simple OpenGL/GLUT test program to animate a simple 3D model using real time motion capture data. This piece of code is an extremely transient hack, but I include it here anyway to show the principle behind device calibration and using angle data for animation in OpenGL.

```c
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <asm/io.h>
#define BASEPORT 0x378
#define DATAPORT BASEPORT
#define STATUSPORT BASEPORT+1
#define CONTROLPORT BASEPORT+2

static float shoulderX, shoulderY, shoulderZ, elbow;
static int capture[8]={0,0,0,0,0,0,0,0};
static int lastcapture[8]={0,0,0,0,0,0,0,0};
static int diffs[8]={0,0,0,0,0,0,0,0};
static int offset[8]={0,0,0,0,0,0,0,0};

void init(void) {
    // Init parallel port device
    if(ioperm(BASEPORT,3,1)) { perror("ioperm"); exit(1); }
    // Select channel 0 for the first conversion
    outb(0x3F, DATAPORT); // Channel address bits are inverted
    // Init control port
    outb(0x07, CONTROLPORT);
    offset[0]=2550;
    offset[1]=850; // Noisy channel in my AMUX, for some reason
    offset[2]=1700;
    offset[3]=1700; // Also slightly bad channel in the AMUX
    offset[4]=850; // Replacement for bad channel 1
    offset[5]=1700; // Replacement for bad channel 3
    // Init graphics state
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glShadeModel (GL_FLAT);
}

void sample(void) {
    #define NCHANNELS 8
    int nwaits=0, nybble=0;
    char channel;
    for(channel=0; channel<NCHANNELS; channel++) {
        // Perform dummy reads here to increase the channel settling time
        // nybble = inb(STATUSPORT);
        // Start conversion by flipping C2 to zero briefly
        outb(0x03, CONTROLPORT);
        outb(0x07, CONTROLPORT); // Prepare for reading, select low nybble
```

```c
    // Actual conversion
    outb(0x03, CONTROLPORT);
    outb(0x07, CONTROLPORT); // Prepare for reading, select low nybble
```
// We have a sample-and-hold ADC, so we can switch channels now
outb(0x3F-((channel+1) & 0x07), DATAPORT); // Next channel for conversion
// Wait for conversion to finish - about 5 waits are needed
for(nwaits=0; (inb(STATUSPORT) & 0x80) == 0; nwaits++);
//printf("Conversion took %d waits.\n",nwaits);
nybble = inb(STATUSPORT) & 0x78; // Mask out bits S6,S5,S4,S3
capture[channel] = nybble >> 3;
outb(0x06, CONTROLPORT); // Select middle nybble
nybble = inb(STATUSPORT) & 0x78;
capture[channel] |= (nybble << 1);
outb(0x05, CONTROLPORT); // Select high nybble
nybble = inb(STATUSPORT) & 0x78;
capture[channel] |= (nybble << 5);
//printf("Channel %d is 0x%04x (decimal %5d).\n",channel,capture[channel],
capture[channel]);
}
}

doxygen

void display(void)
{
    int i,n;
    int averaged[8]={0,0,0,0,0,0,0,0};
    glClear (GL_COLOR_BUFFER_BIT);
    for (i=0; i<NCHANNELS; i++) lastcapture[i]=capture[i];
#define NSAMPLES 10
    for(n=0; n<NSAMPLES; n++) {
        sample();
        for(i=0; i<NCHANNELS; i++)
            averaged[i]+=capture[i];
    }
    for(i=0; i<NCHANNELS; i++)
        if(averaged[i]!=NSAMPLES)
            capture[i]=averaged[i];

    for (i=0; i<NCHANNELS; i++)
        diffs[i] = capture[i]-lastcapture[i];
    printf("Diffs are: %5d %5d %5d %5d ** %5d %5d %5d %5d\n", diffs[2], diffs[4], diffs[0], diffs[5], diffs[1], diffs[3]);
    shoulderZ = -(float)((capture[2]-offset[2]*NSAMPLES)/3500.0*340/NSAMPLES);
    shoulderY = (float)((capture[4]-offset[4]*NSAMPLES)/3500.0*340/NSAMPLES);
    shoulderX = (float)((capture[0]-offset[0]*NSAMPLES)/3500.0*340/NSAMPLES);
    elbow = -(float)((capture[5]-offset[5]*NSAMPLES)/3500.0*340/NSAMPLES);

    glPushMatrix();
    glTranslatef(-2.0, -2.5, 0.0);
    glScalef(2.0, 5.0, 1.0);
    glutWireCube(1.0);
    glPopMatrix();

    glPushMatrix();
    glTranslatef(-2.0, 1.0, 0.0);
    glRotatef(90, 1, 0, 0);
    glutWireSphere(1.0, 10.0, 10.0);
    glPopMatrix();

    glPushMatrix();

glTranslatef(-1.5, 0.0, 0.0);
glRotatef(shoulderZ, 0.0, 0.0, 1.0);
glRotatef(shoulderY, 0.0, 1.0, 0.0);
glRotatef(shoulderX, 1.0, 0.0, 0.0);
glTranslatef(1.5, 0.0, 0.0);
glPushMatrix();
glScalef(3.0, 0.4, 1.0);
glutWireCube(1.0);
glPopMatrix();

glTranslatef(1.5, 0.0, 0.0);
glRotatef(elbow, 0.0, 0.0, 1.0);
glTranslatef(1.0, 0.0, 0.0);
glPushMatrix();
glScalef(2.0, 0.4, 1.0);
glutWireCube(1.0);
glPopMatrix();

glTranslatef(1.0, 0.0, 0.0);
//glTranslatef (wristZ, 0.0, 0.0, 1.0);
glTranslatef(0.5, 0.0, 0.0);
glPushMatrix();
glScalef(1.0, 0.4, 1.0);
glutWireCube (1.0);
glPopMatrix();

glPushMatrix();
glTranslatef (0.75, 0.0, 0.5);
glRotatef (-10, 0, 1, 0);
glScalef (0.5, 0.25, 0.25);
glutWireCube (1.0);
glPopMatrix();

glPushMatrix();
glTranslatef(0.75, 0.0, 0.0);
glScalef (0.5, 0.25, 0.25);
glutWireCube (1.0);
glPopMatrix();

glPushMatrix();
glTranslatef(0.75, 0.0, -0.5);
glRotatef (10, 0, 1, 0);
glScalef (0.5, 0.25, 0.25);
glutWireCube (1.0);
glPopMatrix();

glPushMatrix();
glutSwapBuffers();
glutPostRedisplay();
}

void reshape (int w, int h)
{
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
glMatrixMode (GL_PROJECTION);
    glLoadIdentity();
gluPerspective(50.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
glTranslatef (0.0, 0.0, -5.0);
glScalef(0.4,0.4,0.4);
glRotatef(-30,0,1,0);  // View from the back, slightly to the right

void keyboard(unsigned char key, int x, int y) {
    switch (key) {
    case 'a':
        offset[0] -= 10;
        printf("offset[0] is %d\n", offset[0]);
        break;
    case 'A':
        offset[0] += 10;
        printf("offset[0] is %d\n", offset[0]);
        break;
    case 's':
        offset[1] -= 10;
        printf("offset[1] is %d\n", offset[1]);
        break;
    case 'S':
        offset[1] += 10;
        printf("offset[1] is %d\n", offset[1]);
        break;
    case 'd':
        offset[2] -= 10;
        printf("offset[2] is %d\n", offset[2]);
        break;
    case 'D':
        offset[2] += 10;
        printf("offset[2] is %d\n", offset[2]);
        break;
    case 'f':
        offset[3] -= 10;
        printf("offset[3] is %d\n", offset[3]);
        break;
    case 'F':
        offset[3] += 10;
        printf("offset[3] is %d\n", offset[3]);
        break;
    case 27:
        exit(0);
        break;
    default:
        break;
    }
}
```c
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize (500, 500);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    init ();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}
```

**Action shot**

Here’s a photo of the device in action, with the OpenGL demo program running on the screen.

![Figure 6: The device in action](image)
Finally, just because I have it, here’s an oscilloscope trace of a worst case scenario, with the analog input (top) swinging from maximum to minimum voltage between channels, and the control signal C2 (bottom) that starts the A/D conversion. Each horizontal grid unit is 5 microseconds. As you can see, the sampling frequency is around 60kHz, and there is plenty of time for the analog voltage to settle before the sampling is initiated.

The noise in the analog voltage below is a problem with the oscilloscope, not with the signal as such. The digital converted values only had occasional variations in the least significant bit.

The contents of this document and the original designs described in it are copyright 2001-2002 Stefan Gustavson, Media Technology, ITN, Linköping University, Campus Norrköping (stegu@itn.liu.se). You are most welcome to use this for any non-commercial, not-for-profit purposes with proper credit. If you build your own version of this device, please let me know, and feel free to contact me if you need more information or help. For terms regarding commercial and/or for-profit use of this design, please contact me.