Report

Digital Project

USB Data Acquisition Card

Lund, October 2006

Joe Evans
Bernhard Mayr

Table of Contents

1 Abstract

The aim of the project is to build a data acquisition module that is compatible with the universal serial bus standard. When the module is connected to a PC through a normal USB-cable, it should be able to deliver any data it has acquired to the PC. The micro controller that we used in the project is AVR Atmega32 and the USB-chip is from National Semi-Conductors USBN9604-28M. These products are readily available and easy to build on.

2 Introduction

We decided to do this particular project because USB technology is the future when it comes to data transfer between a PC and a ducking peripheral. The project gave us a good insight about how to build a bridge between hardware and software, and also a deeper understanding of the universal serial bus technology. The project was supposed to take us six weeks; however, we spent two study periods on it. This is owing to the complexities of the USB technology itself.

2.1 Project Specification

With the USB data acquisition card it should be possible to capture a fixed number of samples. The card is remote controlled with command messages which are sent through USB. The Micro-controller should have enough FIFO memory to store all the samples before sending. For the analog to digital conversion the build in ADC in the Atmega device should be used. Only in this way correct timing can be ensured. After capturing, the data should be sent in bulk mode to the client. The client software writes the measurement data to a file. The presentation of the data can be done with Matlab® or a Java- GUI. (For e.g. with jfree-chart) The client software can be done as command line application with parameters.
For debugging the RS232 could be useful, to send debug messages to Hyperterminal.

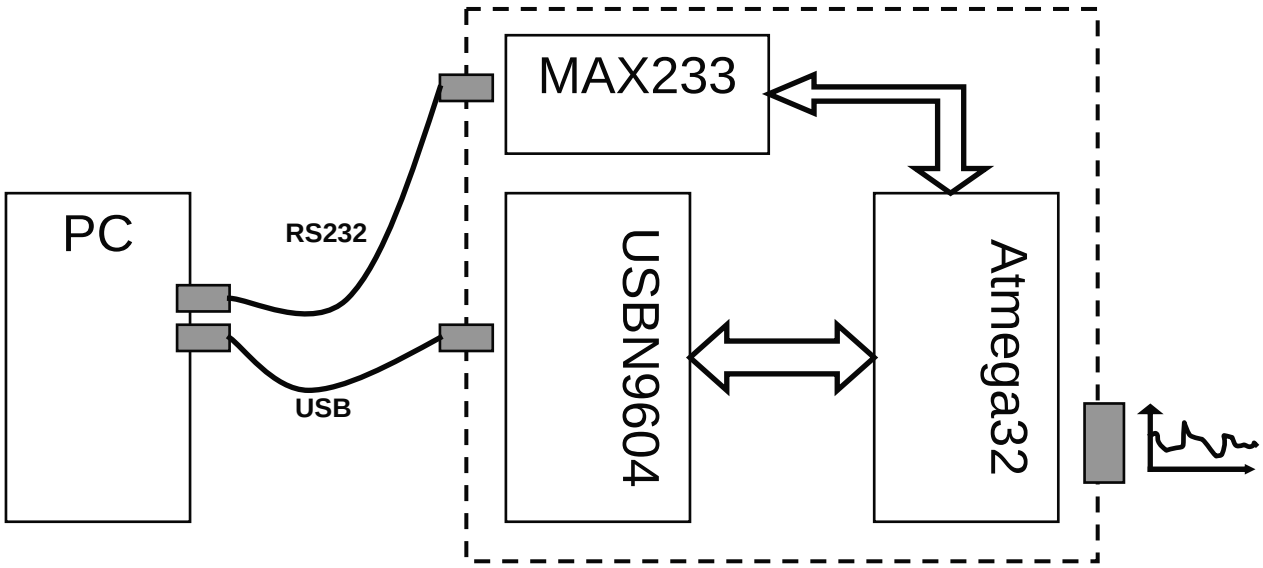

Figure 1: Project Overview

Main devices used in this project:

- Atmega32 clocked with 16MHz
- USBN9604-28M Node Controller clocked with 24MHz
- Max233 for serial debugging interface
- LM35 temperature sensor (only to have analog input data)

Developing environmental tools:

- WinAVR + Programmers Notepad
- Avr-gcc compiler (included in WinAVR)
- AVR – Studio 4.0
- Libusb.lib (USB library with API)
- Visual C and Borland C++ Compiler

- JDK

3 USB Protocol

An USB interface can be very useful for microcontroller applications. Nevertheless the USB interface is not so much used in electronic projects. Although USB is available since 1994 and a very popular interface at PCs, however, compared to the simple RS232 – interface its implementation is much more difficult. A further problem is that there exist not so good documentations and tutorials about USB. The completely detailed USB – specification[1] is also not the best to start with USB. At least we could find a good German master thesis [5] about the usage of USB for the data communication in real time systems.

3.1 USB speed standards

The following table should give an overview about the USB standards. In our case we use USB1.1 (Full-speed) which means a theoretic maximum data transfer of 10 Mbit/s.

speed class	application	physical interface
Low-Speed (USB 1.0) 10 – 100 KBit/s	interactive user interface (keyboard, mouse, joystick)	3,3V differential level 1,5MHz clock
Full- Speed (USB1.1) 500 – 10000 KBit/s	mid speed data transfer modems, digital audio, scanner, printer	3,3V differential level 12MHz clock
High – Speed (USB2.0) 25 – 450 MBit/s	video, external storages (TV and video devices, memory sticks, external hard disks)	instead of voltage interface a high speed current interface is used, no pull up resistor used, 45Ohm termination

Table 1: USB standards

3.2 Architecture

Compared to other serial interfaces, USB is organized in hierarchical manner. The USB architecture is organized in a tree topology. The root of the tree is the host controller, which administer all node devices. A host controller is for example used in a PC. When we buy a microcontroller with USB interface we normally have only an USB node controller implemented. The functionality of a host controller is much more than for a node. Between a host and a node controller, there can be an USB hub to branch the signal to more than one USB device.
This topology explains why we have two different USB connectors. USB-A connector is used for host side and USB-B connector is used for node side. In this way a connection intended for a node cannot be used to connect a hosts or vice versa.

[1] The complete USB specification can be downloaded on www.usb.org

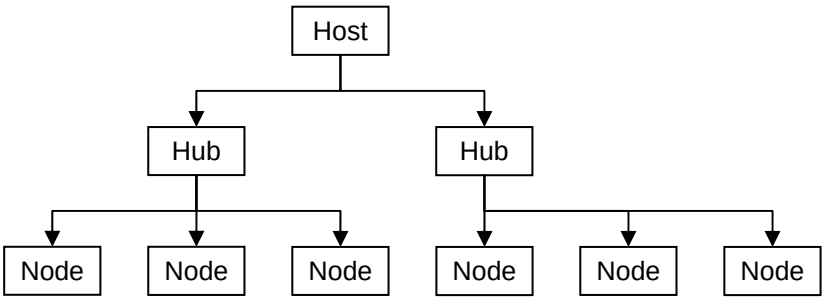

Figure 2: topology of USB network

Compared to other serial Interfaces the power of USB its protocol. Instead of seeing the interface in RX and TX channel, USB is organized in Endpoints. The Endpoints can be seen in FIFO – data pipes (see figure 3). We can define an Endpoint as receiving or transmitting Endpoint. The view of the data direction is always seen from the host side. It is important to know, that all this settings are dynamically.
However, the logical representation of a node is much more than some endpoints.
Every node device has a product ID and a Vendor ID. To produce USB hardware you have to request for a Vendor ID at the USB developer group [4]. In our case we use the Vendor ID from National Semiconductors which is 0x0400.

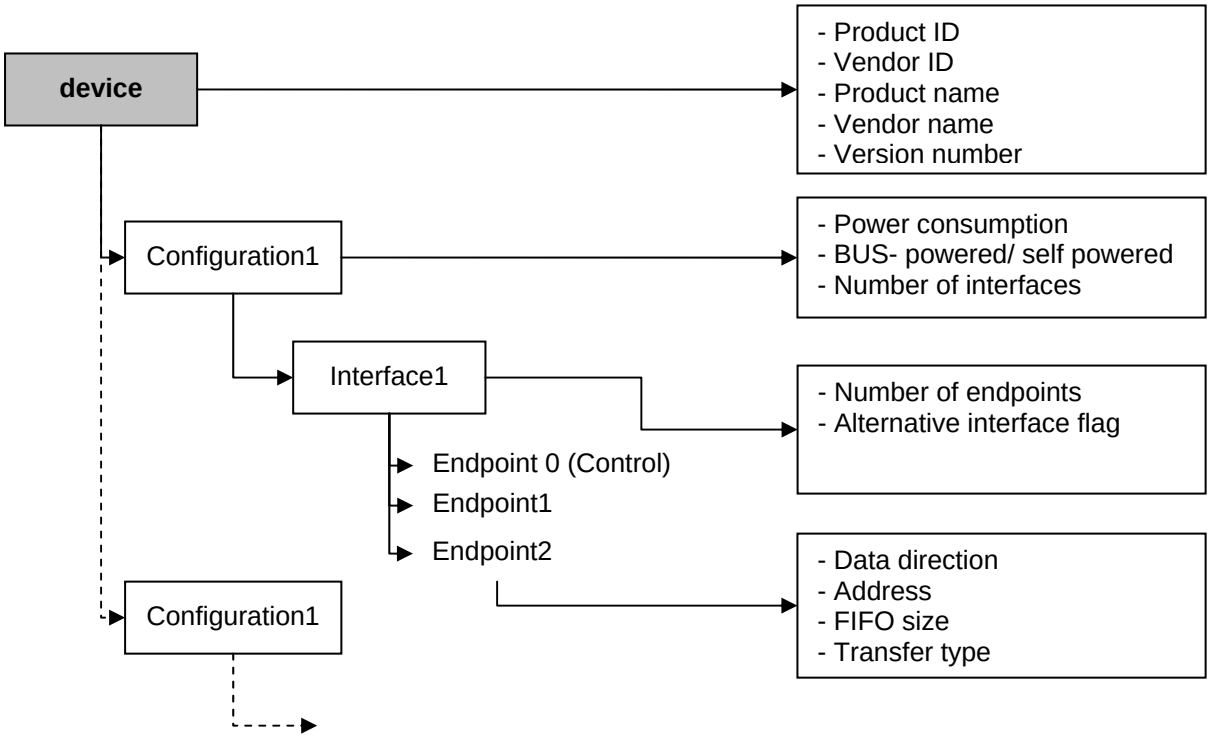

Figure 4: logical organization of an USB node device

3.3 Transfer Types

The USB standard has defined four different transfer types. The used transfer type is depended of the application.

- **Control transfer** is used for special requests from the host side. This transfer type is only used for control messages and not for normal data transfer. Every USB connection uses this mode at least in the initialization phase.

- **Interrupt transfer** is used for devices which send the data very discontinuous. USB itself doesn't support interrupts. The host polls the node devices periodically. This mode is useful for Keyboard or mouse.

- **Bulk transfer** is very useful for huge data packets. It is a fast and secure data transfer with CRC check. (interrupt- and control- transfer is also done with CRC check)
  We are using this mode for sending and receiving data. This mode is used for many applications

- **Isochronous transfer** is for data transfer where a small latency time is very important. A typical application is an audio- communication. The user doesn't want to hear a delay. The disadvantage of this mode is that it doesn't use error detection. But for real time audio- or video data is a bit error not from importance.

4 USB - Node Controller

4.1 Overview

To develop an USB device you need an USB node controller. On the market you can find many different chips for different purpose. Some node controllers have an additional 8051 core. This is useful for small applications so that you need not an additional microcontroller. There are also many microcontrollers with an USB interface available like the C541U from Infineon or AT8xC5131 from Atmel. It would be nice to know, why Atmel didn't use an Atmel – device for handling the USB connection to the Atmel JTAG ICE MK2 device. (Atmel used the Phillips USB controller PDIUSBD12)
Mainly, the node controllers differ in their complexity. There are some controllers available which have implemented only a SIE (serial interface engine) with a hardware interface but also some devices that have implemented a big part of the USB protocol. The following table should give you an overview of different USB node controllers.

Manufacturer	Device name	Description
National	USBN9604	Full speed node controller with parallel output and DMA support
FTDI	FT232	USB to UART interface, simple to use, Win- driver available, good support
FTDI	FT245	USB controller with parallel interface, Win driver available, good support
Cypress	CY7C63000	One of the first USB chips, only for low speed devices like mouse or joystick, memory is only OTP
Cypress	AN2131	full speed controller with 8051 core (8KByte Data- Program memory), max. 32 Endpoints with 64Byte pipe, good for applications with many endpoints
Phillips	PDIUSBD11/12	full speed node controllers with I^2C and parallel interface respectively

Table 2: USB node controllers from different manufactures

4.2 USBN9604

The USBN9604 is an integrated USB Node controller. The device provides enhanced DMA support with many automatic data handling features. It is compatible with USB specification versions 1.0 and 1.1.

The device integrates the required USB transceiver with a 3.3V regulator, a Serial Interface Engine (SIE), USB endpoint (EP) FIFOs, a versatile 8-bit parallel interface, a clock generator and a MICROWIRE/PLUS™ interface. Seven endpoint pipes are supported: one for the mandatory control endpoint and six to support interrupt, bulk and isochronous endpoints. Each endpoint pipe has a dedicated FIFO, 8 bytes for the control endpoint and 64 bytes for the other endpoints. The 8-bit parallel interface supports multiplexed and non-multiplexed style CPU address/data buses. A programmable interrupt output scheme allows device configuration for different interrupt signaling requirements.

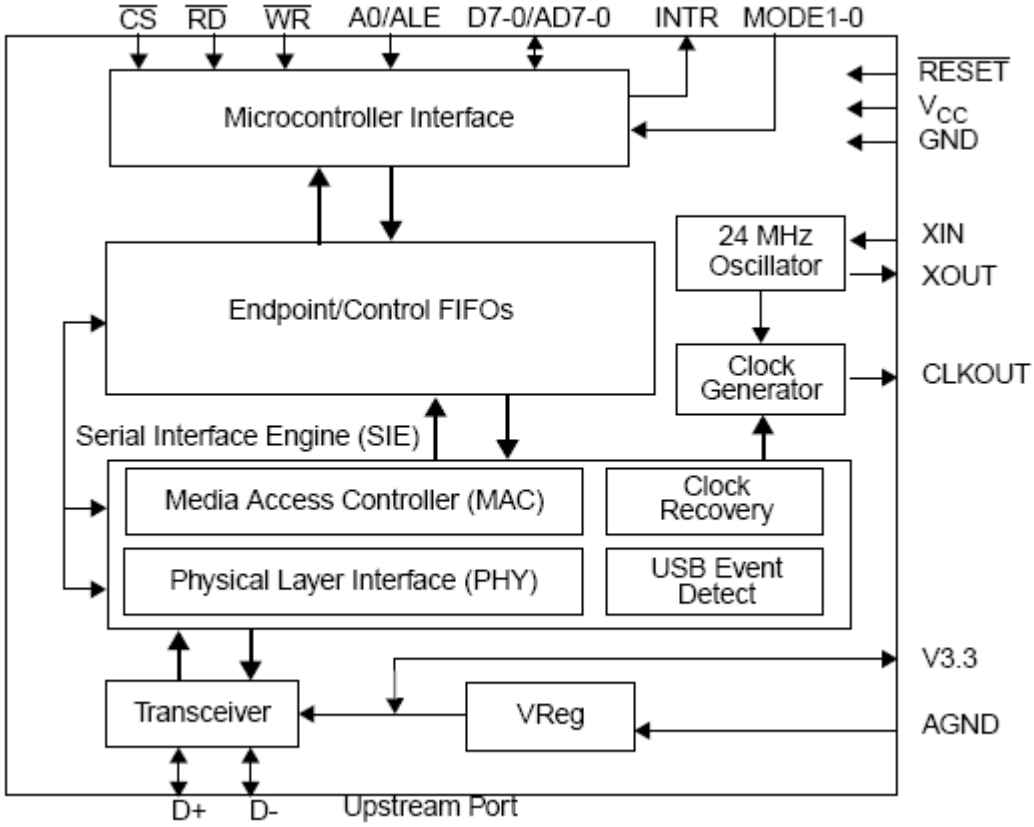

Figure 5: Block diagram of USBN9604

Features:

- Full-speed USB node device
- Integrated USB transceiver
- Supports 24 MHz oscillator circuit with internal 48 MHz clock generation circuit
- Programmable clock generator
- Serial Interface Engine (SIE) consisting of Physical Layer Interface (PHY) and Media Access Controller (MAC), USB Specification 1.0 and 1.1 compliant
- Control/Status register file
- USB Function Controller with seven FIFO-based Endpoints:
    - One bidirectional Control Endpoint 0 (8 bytes)
    - Three Transmit Endpoints (64 bytes each)

- Three Receive Endpoints (64 bytes each)

- 8-bit parallel interface with two selectable modes:
  Non-multiplexed
  Multiplexed (Intel compatible)
- Enhanced DMA support
  Automatic DMA (ADMA) mode for fully CPU-independent transfer of large bulk or ISO packets
  DMA controller, together with the ADMA logic, can transfer a large block of data in 64-byte packets via the USB

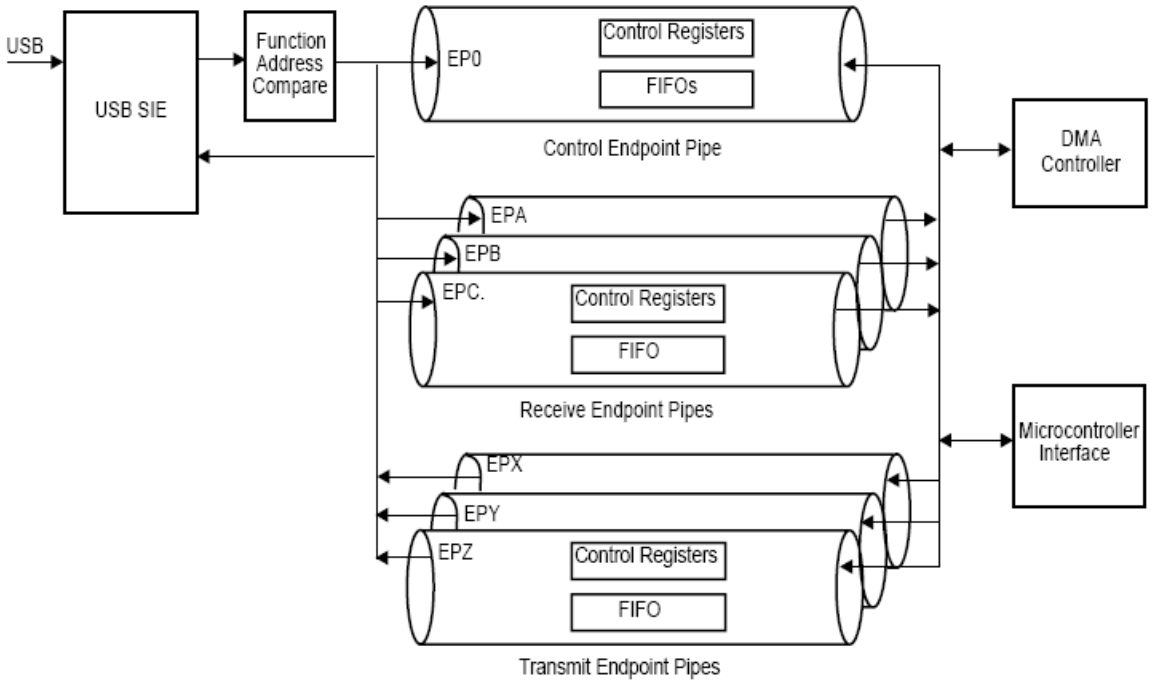

Figure 6: Pipe structure of USBN9604

5 Description of the circuit

The aim of this circuit is to acquire data with the ADC converter of the microcontroller and send this data via USB to the PC. As mentioned before, we are using the 8 Bit RISC microcontrollers Atmega32 and the USB Bridge USBN9604 from National. The Atmega32 is clocked with 16MHz. First the LM2937 was used to supply the microcontroller with 3,3V. But as mentioned in the datasheet, the Atmega device has to be supplied with 5V, if it is connected to a 16MHz crystal. The crystal for the USB Bridge is a 24MHz crystal. Also the USBN9604 is supplied with 5V. The 3,3V for the USB interface are generated with the internal voltage regulator of the USB-Bridge. This choice is reasonable compared to supply both devices with 3,3V because in this configuration the microcontroller can be used with maximum computational power and the startup of the device is more robust. In this case after connecting the USB device to the PC, the microcontroller and the USB bridge resets. Before the USB– data transfer can start, the microcontroller has to set all configuration registers in USBN9604. After the complete initialization phase, the microcontroller can open the USB interface in the way that it activates the V3.3 output at the USBN9604. This causes that the D+ data line will be pulled up by the 1,5k resistor. With pulling up the D+ line the host controller detects that a new USB device is connected to the USB bus. Then the host can start to read all device information, which is necessary to access the device. The two data lines D+ and D- are connected directly from the USBN9604 to the USB connector. There are no additional matching resistors necessary.
To the ADC- Converter there is only a simple potentiometer and the temperature sensor LM35 connected. To demonstrate fast data acquisition it would make more sense to connect an audio signal or a measurement signal with at least higher frequencies than a temperature signal to the ADC.

5.1 Interface Atmega32 ⇔ USBN9604

The Atmel device is connected with the USBN9604 through an 8Bit parallel interface. Before we can write or read data, we have to write an address to the address register in such a way that A0 is high. We are able to access mostly all configuration registers in the USBN9604. After writing the address of the specific register, we can write or read data to this register. An address table of all configuration and status registers can be found in the datasheet of USBN9604 [2].

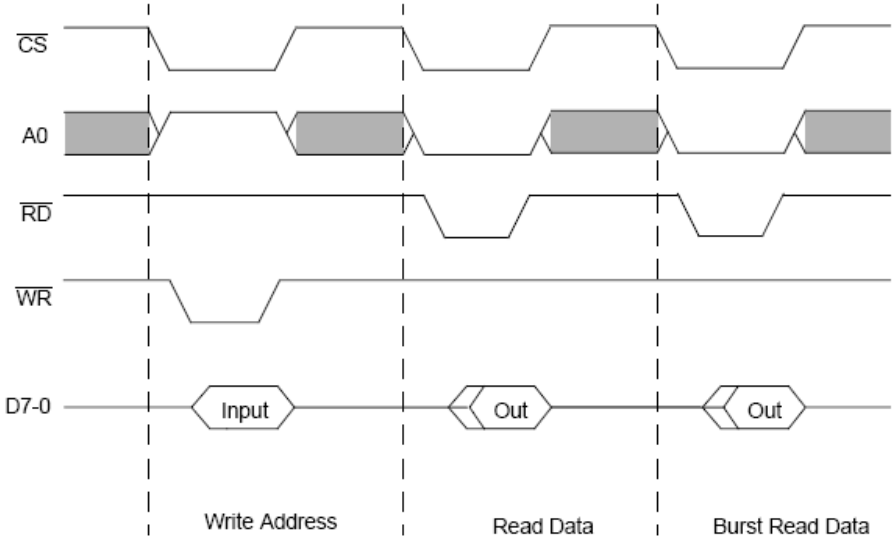

Figure 7: Timing diagram of parallel interface to USBN9604

5.2 Port usage

Port A: Input ADC – Converter
Port B: 8Bit parallel data/address bus to USBN9604
Port C: JTAG interface, general purpose indicators
Port D: Control Signal for USBN9604 (CS, RD, WR, INTR, ALE)

5.3 USB interface

name	Pin Nr.	description
VCC	1	+5V
D-	2	Data -
D+	3	Data +
GND	4	Ground

5.4 Debugging Interface

For debugging we used mainly the serial interface (RS232). The JTAG ICE MK2 was in our case not so useful for debugging. The reason is that we had to run the system in real-time to prevent timing problems. So we used the RS232 Interface to send Debug messages to MS- Hyper-Terminal. We also implemented the possibility to run functions with RS232 commands. This was useful to read status registers from USBN9604.
The JTAG ICE MK2 was useful to verify the time uncritical ADC – function.

5.5 Programming Interface

As programming Interface we used the JTAG Interface. For programming and setting use flags, the JTAG MK2 module was very useful. In the following figure, the pin assignment is mentioned. The signal nTRST is not used.

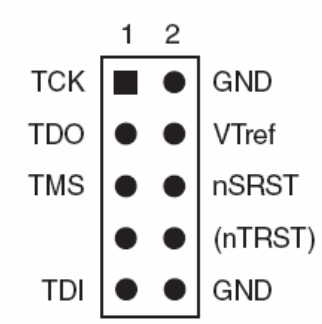

Figure 8: Pin assignment of 10 Pin JTAG connector

6 Schematic

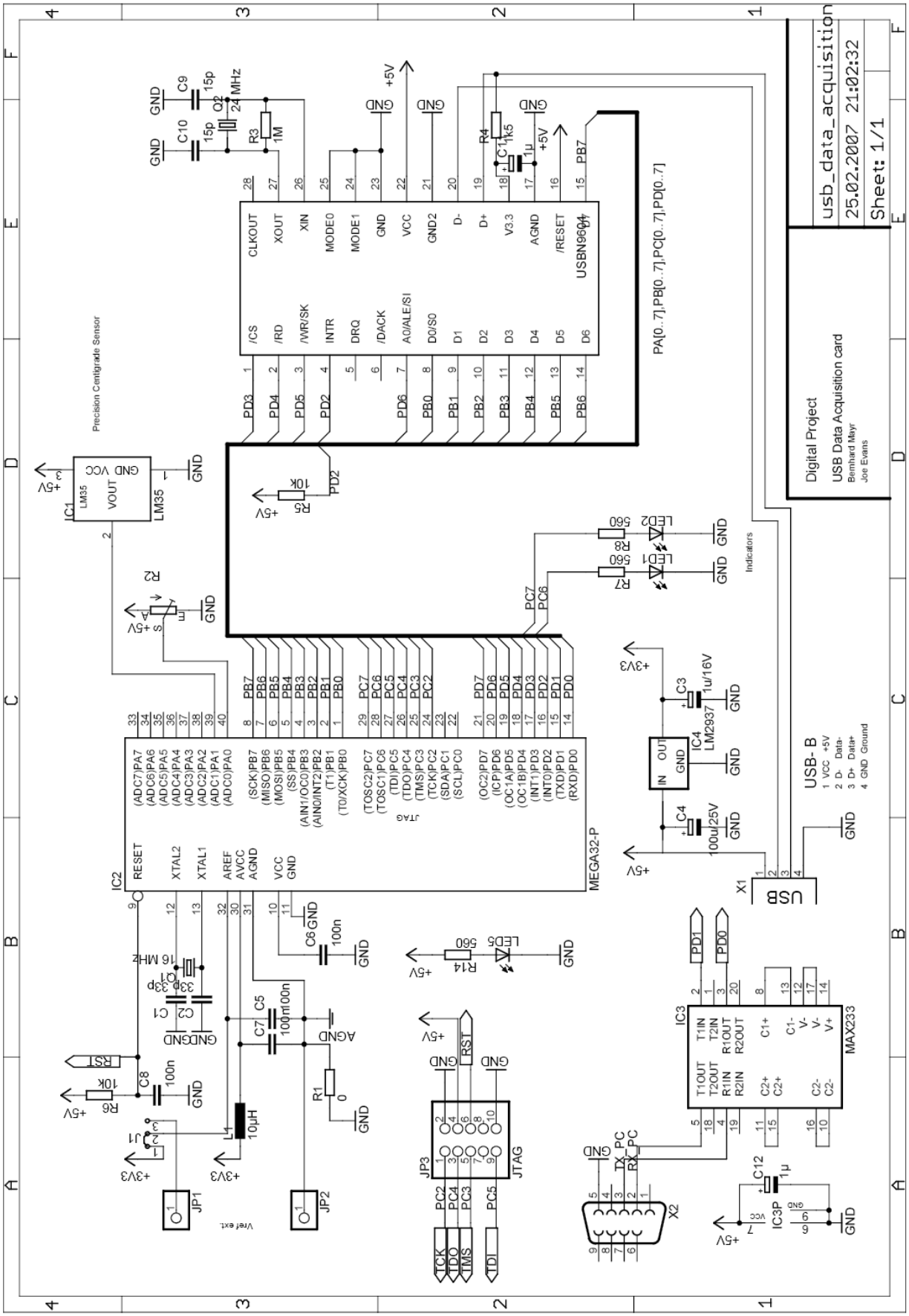

Figure 9: Schematic of USB data acquicition card

7 Board

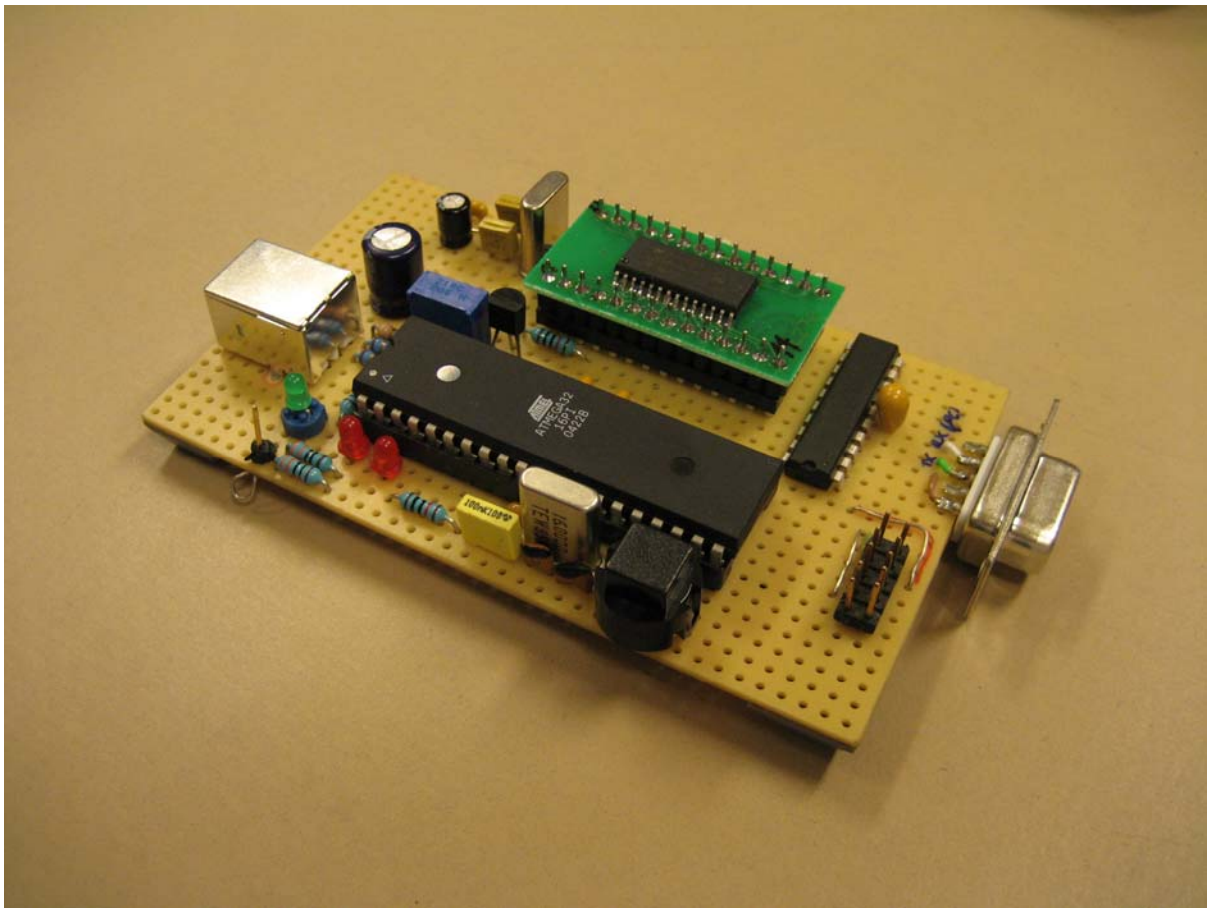

Figure 10: Picture of USB data acquisition board

8 Software

The hardware peripheral i.e. the micro-controller and the USB node controller are coded entirely in c-language. However, on the PC side the client is coded in c-language while the small user interface is written in java.

8.1 Firmware and Application (Atmega)

The firmware is the software on the microcontroller. It should handle the communication to the USB node controller and capture the measurement data with the ADC converter. We tried to build our firmware in a way that we can remote our device as given in the following line:

- `Usb_card.exe –number of samples –sampling rate`

In this case the accurate timing is implemented with the timers of Atmega32. A typical sample rate is 10 or 100µs, the maximum number of samples is 1000.
Unfortunately, we didn't have enough time to implement the timer solution. The current solution captures only 32 values (64 Bytes) and sends it to the host. This program can be found in *_usb_data_acquistion_card/source_code/applications/usb_card/client/* and can be executed as follows:

- `Usb_card.exe 1 (usb_card –ADC-port)`

The argument selects the channel of the ADC. The complete Atmega- source code and the .hex-file can be found in *_usb_data_acquistion_card/source_code/applications/usb_card/*.

To access the USBN9604 we used the code template of USBN2MC [7]. This template from Benedikt Sauter was very useful and helpful for our design. The template offers a complete API to define a device, interface, configuration and endpoints. The code is written very dynamically (also using malloc). Instead of hard coding parameters, everything is configurable with methods. On reason, why we use more than 16KByte program memory is that USB has lot descriptors which are saved on the Atmega device. But we also have to take into account several numbers of debug messages, which are saved as strings.

In Figure 11 you can see some API functions to access the USBN9604 in a very elegant way. More details how to use this API functions can be seen in the source code or on the website of Benedikt Sauter [7].

```
USBNInit()                              //activate usb kernel
USBNDeviceVendorID(0x0400)              //0x0400 is the number from national
USBNDeviceProductID(0x9876)             //add your product id
USBNDeviceManufacture ("MyFirm")
USBNDeviceProduct     ("mydevice")

USBNAddConfiguration()                  //configurations a numbered automati-
cally
USBNConfigurationPower(conf,50)         //set the current of the device e.g.
50mA
USBNAddInterface(conf,0)
USBNAddInEndpoint(conf,interf,1,0x03,BULK,64,0)

USBNSendData(fifonumber, data)
```

Figure 12: Some API - functions from USBN2MC

8.2 Application (Win32)

In the client side, the native c classes pass data to the java user interface through a file during a read operation. The data passing is the other way round during a write operation. The user interface runs a thread that reads the file every 10-micro seconds. Depositing acquired data to a file, and allowing the user interface to subsequently fetch it, might seem slow. However, reading and updating a temperature in this project is not a time critical event. Consequently, using a file as temporary storage here is appropriate.

8.3 Driver

Initially we wanted to make the data transfers without a USB-node-controller chip, however, after many hours of writing codes and manipulation of our components, it became obvious that we needed a USB-node-controller in other to implement a USB-data-acquisition device. After introducing the node controller, we had to use the open source Libusb-win32-device driver from SOURCFORGE [10]. The diagram below depicts the layers involved in the project.

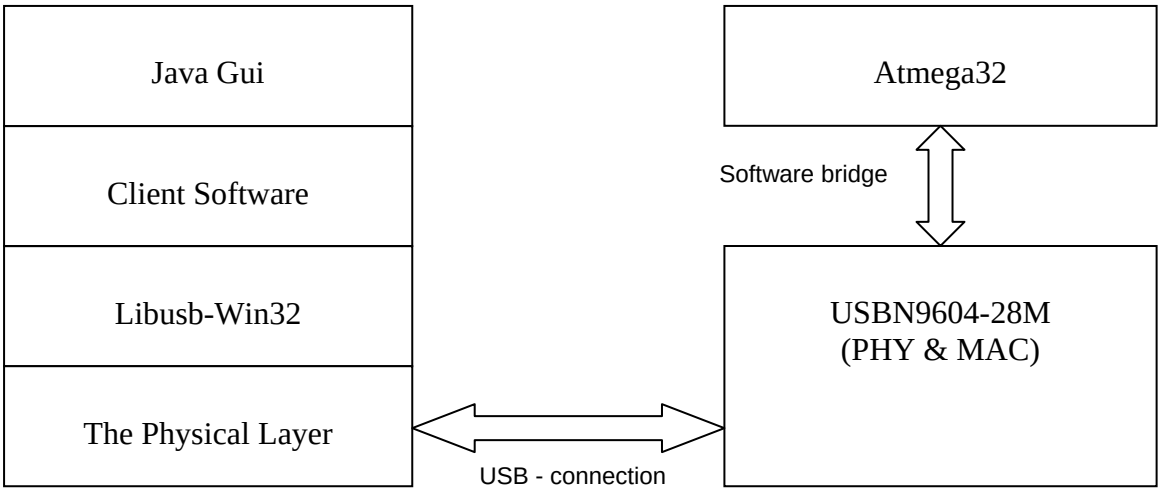

Figure 13: Layer model for USB connection

8.4 Libusb-win (API)

At the PC side the client runs on Libusb-win32 API. We used the specifications of the API with changed parameters to suit our need. We are reading and writing during a bulk-read or bulk-write to a specified endpoint. The maximum bulk-read and bulk-write is 64 bytes in either direction. Our client class interacts with the Libusb API through the <usb.h> interface. The provision of the usb.h interface by the Libusb API greatly simplifies things at the client side if one knows how to build and compile it. As a matter of fact, compiling and building the Libusb-win32 is not as trivial as it seems.

8.5 Software development environment

The software development environment used for writing the c codes is Programmers Notepad. For downloading we used AVR Studio 4, this is connected to our device through JTAG interface. The client side c programs where compiled with Microsoft visual studio command prompt and Borland compilers command prompt at different times. The choice of compilers has to do with the Libusb-win32 device module, which would not compile in Gcc compiler. To build the libusb-win32-device we used the Inno Setup environment. The small java user interface was developed in eclipse.

9 Problems during Project

9.1 Architecture problems

As aforementioned in the introduction we spent more time than expected because of the problems we encountered. Implementation of USB was a new area for us.
First we tried to implement the USB data card without external node controller. A sample project can be found at the homepage of objective development [8]. If you connect everything exactly in the same way as they did, it should work also without external node controller. The Atmel device acts in this way as node controller. The big disadvantage in this architecture is that it uses all the computational power of the microcontroller for the implementation of the node controller. To prevent timing errors the microcontroller cannot be used for complex calculations or time consuming operations.

A further point why it didn't work without external node controller could be the clock frequency. For the USB 1.1 Standard we have to supply the device with a 12MHz CLK- signal. On the other hand the USB standard requires 3,3V- levels on the data lines. But Atmel devices are not specified for more than 8MHz at 3,3V operation.

9.2 Libusb-win32 problems

The next problem we encountered was that of building and compiling the client program. The Libusb-win32 provided an interface API (usb.h), but compiling and liking that to our client programs was a none-trivial issue. There is no proper documentation about how to build and compile the driver. After, several hours of work we come up with how it is done. Below are the steps:

- Download libusb-win32-device from sourceforge.net, note; not libusb-win32-filter
- Download the Inno-Setup compiler [11] from jrsoftware.org
- Connect the USB device you want to use to the pc. In this project, that will be the Acquisition card
- In the Bin folder of the libusb-win32-device there is a file called inf-wizard.exe double-click on it
- The above action will create an .inf file; you can name it "libusb.inf".
- Copy the .inf file and the files libusb0.dll and libusb0.sys from the Bin folder to the examples folder.
- Double click on the Inno-Setup to install it.
- In the examples folder of the libusb-win32-device there is a file called driver_installer-template.iss, open this file in Inno-Setup.
- In the last line of the driver_installer-template.iss opened in the Inno-Setup, replace this phrase <your_inf_file.inf> with the name of your .inf file. Then build the driver in the Inno-Setup.
- At this point you have the libusb-win32-device driver install application created.

- The next step will be to write your client program and compile it. Your client program should be placed in the examples folder and it should include <usb.h>. For a start, you could try with the test.c or bulk.c provided in the example folder. However, you would need to compile it and here starts the compiler problem. You could either use the Microsoft visual studio command line prompt if you have that or try to download the free Borland Compiler. Steps to compile:

  Once you have the compiler installed:

  - From the libusb-win32-device; if you are using ms-visual studio copy the libusb.lib file from the /lib/msvc folder to the lib folder of the ms-visual studio compiler. If you are using Borland compiler, copy the libusb.lib file from the /lib/bcc folder to the lib folder of your Borland compiler. Finally, copy usb.h file from the include folder of libusb-win32-device to the include folder of the compiler you are using.

  - To compile and build the object file, from the Microsoft visual studio command prompt, type:

    ```
    cl/EHsc libusb.lib test.c
    ```

  - To compile and build the object file, from the Borland command prompt type:

    ```
    bcc32 -WC libusb.lib test.c
    ```

Note: In the command prompt, you have to be in the examples directory to apply these commands

10 References

[1] Atmel Cooperation; www.atmel.com (Feb.2007)
[2] National Semiconductor; www.national.com
[3] Maxim IC; http://www.maxim-ic.com/
[4] USB specification.; http://www.usb.org/home
[5] Eik Arnold, USB master thesis;
http://www.tu-chemnitz.de/etit/messtech/studienarbeiten/abgeschl/pdf/arnold_da.pdf
[6] Microcontroller forum, Atmel tutorial; http://www.mikrocontroller.net/
[7] USBN2MC, API for USBN9604; http://usbn2mc.berlios.de/index.php?page_id=57
[8] USB-AVR Objective development; http://www.obdev.at/products/avrusb/index.html
[9] USB projects; http://www.usb-projects.net/cwiki.php?page=HomePage
[10] libusb; http://libusb-win32.sourceforge.net/, Feb. 2007
[11] Inno Setup compiler; http://www.jrsoftware.org/isdl.php
[12] Entech driver development; http://www.entechtaiwan.com/dev/index.shtm
[13] Windriver; http://www.entechtaiwan.com/dev/index.shtm

11 Appendix

11.1 Bill of material

quantity	value	name	instances
1	F09H	X2	
1	J2X2MM	J1	
3	LED3MM	LED1,	LED2,LED5
2	PINHD-1X1	JP1,	JP2
1	PN61729	X1	
1	TRIM_EU-ST15	R2	
1	0	R-EU_0207/10	R1
2	1µ	CPOL-EU085CS-1AR	C11,C12
1	1M	R-EU_0207/10	R3
1	1k5	R-EU_0207/10	R4
1	1u/16V	CPOL-EUE2.5-6	C3
1	10µH	L-EU0204/7	L1
2	10k	R-EU_0207/10	R5,R6
2	15p	C-EU025-024X044	C9,C10
1	**16**	**MHz**	**CRYTALHC49S Q1**
1	**24**	**MHz**	**CRYTALHC49S Q2**
2	33p	C-EU025-024X044	C1,C2
4	100n	C-EU050-025X075	C5,C6,C7, C8
1	100u/25V	CPOL-EUE2.5-6	C4
3	560	R-EU_0207/10	R7,R8, R14
1	JTAG	PINHD-2X5	JP3
1	LM35	LM35	IC1
1	LM2937	78XXS	IC4
1	**MAX233**	**MAX233**	**IC3**
1	**MEGA32-P**	**MEGA16-P**	**IC2**
1	**USBN9604**	**USBN9604**	**M1**

11.2 Driver (.inf-file)

```
[Version]
Signature = "$Chicago$"
provider  = %manufacturer%
DriverVer = 03/09/2005,0.1.10.1
CatalogFile = libusb.cat

Class = LibUsbDevices
ClassGUID = {EB781AAF-9C70-4523-A5DF-642A87ECA567}

[ClassInstall]
AddReg=ClassInstall.AddReg

[ClassInstall32]
AddReg=ClassInstall.AddReg

[ClassInstall.AddReg]
HKR,,,,"LibUSB-Win32 Devices"
HKR,,Icon,,"-20"

[Manufacturer]
%manufacturer%=Devices

;-----------------------------------------------------------------------
; Files
;-----------------------------------------------------------------------
```

```
[SourceDisksNames]
1 = "Libusb-Win32 Driver Installation Disk",,

[SourceDisksFiles]
libusb0.sys = 1,,
libusb0.dll = 1,,

[DestinationDirs]
LIBUSB.Files.Sys = 10,System32\Drivers
LIBUSB.Files.Dll = 10,System32

[LIBUSB.Files.Sys]
libusb0.sys

[LIBUSB.Files.Dll]
libusb0.dll

;------------------------------------------------------------------------
; Device driver
;------------------------------------------------------------------------

[LIBUSB_DEV]
CopyFiles = LIBUSB.Files.Sys, LIBUSB.Files.Dll
AddReg    = LIBUSB_DEV.AddReg

[LIBUSB_DEV.NT]
CopyFiles = LIBUSB.Files.Sys, LIBUSB.Files.Dll

[LIBUSB_DEV.HW]
DelReg = LIBUSB_DEV.DelReg.HW

[LIBUSB_DEV.NT.HW]
DelReg = LIBUSB_DEV.DelReg.HW

[LIBUSB_DEV.NT.Services]
AddService = libusb0, 0x00000002, LIBUSB.AddService

[LIBUSB_DEV.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,libusb0.sys

[LIBUSB_DEV.DelReg.HW]
HKR,,"LowerFilters"

;------------------------------------------------------------------------
; Services
;------------------------------------------------------------------------

[LIBUSB.AddService]
DisplayName    = "LibUsb-Win32 - Kernel Driver 03/09/2005, 0.1.10.1"
ServiceType    = 1
StartType      = 3
ErrorControl   = 0
ServiceBinary  = %12%\libusb0.sys

;------------------------------------------------------------------------
; Devices
;------------------------------------------------------------------------

[Devices]
"USB Data Aqucisition"=LIBUSB_DEV, USB\VID_0400&PID_9875

[Strings]
manufacturer = "National Semiconductors"
```

11.3 Source code Firmware

See Zip- File

11.4 Source code Application (Win32)

See Zip- File