Please note that this USB spectrometer project has a new HTML page, which may contain updates and further information:
http://www.fzu.cz/~dominecf/spek2/index.html

The original PDF text follows.

- author

Czech Technical University in Prague
Faculty of Nuclear Sciences and Physical Engineering
Department of Physical Electronics

Design and construction of a digital CCD spectrometer

Research project

Author: **Filip Dominec**
Supervisor: **Ing. Jaroslav Pavel**
Academic year: **2009/2010**

.

I declare hereby that I have elaborated this research project independently and all used literature is listed in the reference section.

Praha, 30. 4. 2010 Filip Dominec

Abstrakt

Název práce: **Návrh a konstrukce digitálního CCD spektrometru**

Autor: Filip Dominec

Obor: Fyzikální inženýrství

Druh práce: Výzkumný úkol

Vedoucí práce: Ing. Jaroslav Pavel

Abstrakt: Předmětem této práce je návrh, vývoj a výroba kompaktního digitálního spektrometru. Spektrum je promítáno přes difrakční mřížku na CCD snímač, vzorkuje se 10-bitovým A/D převodníkem do paměti mikrokontroleru a nakonec je odesláno do osobního počítače přes rozhraní USB. V počítači je měřené spektrum digitálně zpracováno a zobrazuje se pomocí grafického uživatelského rozhraní. Spektrometr pracuje v optické a blízké infračervené oblasti přibližně od 400 do 900 nm, dané použitým křemíkovým CCD snímačem.

V práci je popsána jak montáž optické soustavy a stavba obvodu zpracovávajícího analogový signál, tak i uživatelský software schopný vyhodnocovat a ukládat měřená spektra. V závěru práce jsou uvedeny postupy pro kalibraci přístroje a příklady jeho použití v praxi.

Klíčová slova: spektrometr, CCD snímač, komunikace přes USB

Title: **Design and construction of a digital CCD spectrometer**

Author: Filip Dominec

Abstract: This project aims to design, develop and build a compact digital spectrometer. The spectrum is projected with a diffraction grating on a CCD detector, sampled in a 10-bit A/D converter to the microcontroller memory and finally transmitted to the computer via USB. In the computer it is digitally processed and displayed in a graphical user interface. The spectrometer operates in the optical and near-infrared region, approximately from 400 to 900 nm, which is given by the silicon CCD detector.

Not only the optical setup and electronic circuit used to process the analog signal are described in this work, but also the software enabling the user to evaluate and save the measured spectra. The calibration procedure and practical examples are discussed at the end.

Key words: spectrometer, CCD detector, USB communication

Contents

1 Introduction

Optical spectrometry extensively contributed to the development of physics in recent two centuries. For instance, it was used to determine the ion composition of distant stars, although this task had been enlisted several years before as one of questions which mankind can never answer. Moreover, the solar spectra revealed new, at that time unknown spectral lines, which were later resolved to come from helium. At the turn of the twentieth century, the absorption and emission spectra of various materials supported the need of building a new, quantum theory. Optical spectroscopy developed into an essential measurement method in analytic chemistry, astronomy and many other fields.

Most optical spectrometers that produce a spectral waveform spanning over a selected region of wavelengths may be divided by their principle of operation into 3 most important groups:

1. **Fourier-transform spectrometers** use Michelson interferometer to measure how the interference signal oscillates when one of the mirrors moves. The interference function is later digitally processed to get the intensity spectrum. They can achieve very high spectral resolution and can measure even far-infrared spectra, assuming IR optics and detector are used.

2. **Spectrometers with a monochromator** usually utilize rotating diffraction grating. They are simpler and less sensitive to environment, as no interference fringes are measured. The Czerny-Turner setup is often used, but other different setups may suit a particular purpose better. For example, to replace the rotating grating by a prism with moving mirror may be favourable for UV spectroscopy or when higher diffraction orders are to be eliminated.

3. The latter setup can be modified to obtain a **spectrometer with multi-channel detector** and no moving components. The grating is then fixed and the spectrum is projected on a broadband detector, which is usually a CCD.

   The spectrometers in this category largely differ by sensitivity, resolution, application, size and price. For instance, the *High Accuracy Radial Velocity Planet Searcher* at La Silla Observatory in Chile is able to detect 1 m/s velocity difference in star's movement, enabling to indirectly search for extrasolar planets. On the other hand, spectrometers of this type can be made very compact and they are optimal for applications where resolution and sensitivity is not critical. Some models are available as a PCI card fitting into a computer.

However, even the price of these handheld spectrometers usually exceeds the $1000 barrier, making them hardly affordable as a educational gadget for a high-school science practicals, for example. Moreover, the supplied proprietary software usually does not permit any modifications or scripting, which may be needed when automating routine work in research laboratory.

This research project focuses on building a simple, cheap and compact spectrometer meeting following objectives:

1. The spectral resolution ($\Delta\lambda$) should be better than 5 nm.

2. No moving components should be used.

3. The device and supplied software should be easy to use so that anybody with basic knowledge of spectroscopy can use it immediately.

4. The supplied software must run on most major operating systems with dependence on freely available programs only.

5. Except for initial calibration, no further maintenance would be required. Each device built will store its own calibration data in its permanent memory to prevent the need for recalibration when another model is connected.

6. Communication with computer and electrical supply should be performed via single USB cable.

The purpose of this research project is to decide whether such device can be built at a substantially lower price and yet remain suitable for practical tasks, as well as to present a possible way of doing that. Additionally, the computer software is released under free license, allowing the aforementioned modifications and scripting. Last but not least, the data acquisition, processing and displaying techniques can be reused in many other applications involving measurement automation.

2 Theory

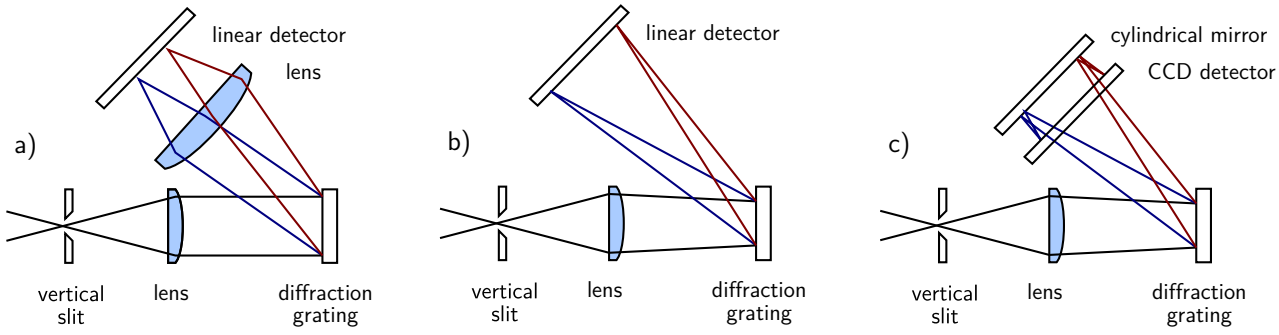
2.1 General dispersive spectrometer design

In this research project, a diffraction grating design with a CCD detector was used. This section describes how the light components are separated and delivered to the CCD.

In the horizontal plane, the incoming light must be spatially limited by a vertical slit. Then the spatial limitation is converted to angular limitation using a concave mirror or a convex lens. Ideally, we obtain a parallel beam of white light. Its width is limited either by the size of the mirror or lens used, of by the angle of the beam passing through the slit.

The light components of different wavelengths separate after being reflected into different angles by a diffraction grating. Finally, the angular separation might be converted back to spatial separation using another lens or mirror as drawn in Fig. 1a. However, this would require a big lens with diameter similar to the length of CCD. The spatial difference would be proportional to its focal length, probably resulting in a small spatial separation. Fortunately, the second lens is not necessary, as the first lens can be shifted several millimeters away from the slit to create a real image in arbitrary distance from the grating (Fig. 1b). The beam incident on the diffraction grating becomes slightly convergent when a single lens is used. This results in nonuniform angular difference even for single wavelength and the projected image of the slit becomes slightly blurred. However, for the sharp angle of convergence this effect can be neglected. There are several other phenomena limiting the spectral resolution of this

Fig. 1: Horizontal schematics and light paths



setup, but assuming an ideal diffraction grating is used, the nonzero width of the input slit dominates.

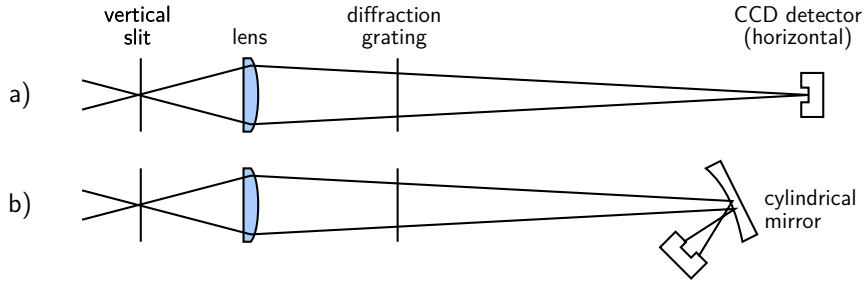
In the vertical projection, a single point in the slit plane is projected onto another point on the sensor as shown at Fig. 2a. As the CCD sensor is very narrow, the spectrometer would be sensitive in only one point at the slit, not depending on the incident angle. This would be optimal if either a thin optical fiber or a point source are measured.

However, it was expected that the incident light would mostly be cover some area on the slit and that the spectrometer should be sensitive at nonzero length along the slit. Therefore a cylindrical mirror was inserted near to the focal plane to concentrate each spectral line (image of the input slit) into a much shorter line at the detector (Fig. 1c, 2b).

If the light entered the slit at a single point and this point slightly moved in vertical axis, its image would move in vertical axis, too. When a cylindrical mirror is used, the point image would still be projected onto the CCD, albeit under different angle. The detector could not be put directly in the focal

plane of this mirror, because the horizontal focus given by the lens must be maintained. Nonetheless, the sensitivity to the light entering the slit was substantially improved.

Fig. 2: Vertical schematics and light paths



2.2 CCD detector

2.2.1 Overview

The charge coupled device, CCD, was originally designed not to be an optical detector, but a memory instead. It should work as an analog shift register, storing arbitrary charge in each of its cells and reading them sequentially. Soon it was discovered that illumination causes internal photoelectric effect in the silicon substrate and fills up each memory cell with additional charge proportional to the incident light intensity. Although CCD was rarely used as a memory, it found widespread application as a sensitive optical detector both in consumer electronics and in high-end scientific instruments such as astronomical telescopes.

CCD detectors may be divided into two groups: linear CCDs, where one row of pixels is shifted to the output, and matrix CCDs, where the bottom row is fully shifted out always when all columns are shifted down by one pixel. For this project, a linear CCD was chosen over a matrix CCD because only a linear detector several centimeters long was needed. Still, it would be possible to build a custom matrix CCD driver, too [1]. Another option, a CMOS sensor, was not used because of worse availability.

2.2.2 Driving a linear CCD

Although the internal design of the particular components may differ, the basic principle of CCD operation should remain similar. In the CCD register itself, a silicon substrate is covered by a thin layer of insulator and the electrodes. When a voltage is applied to an electrode, the electrons get trapped under the layer of insulator. By changing the voltage distribution on the surface, the charge packet can be moved with negligible loss.

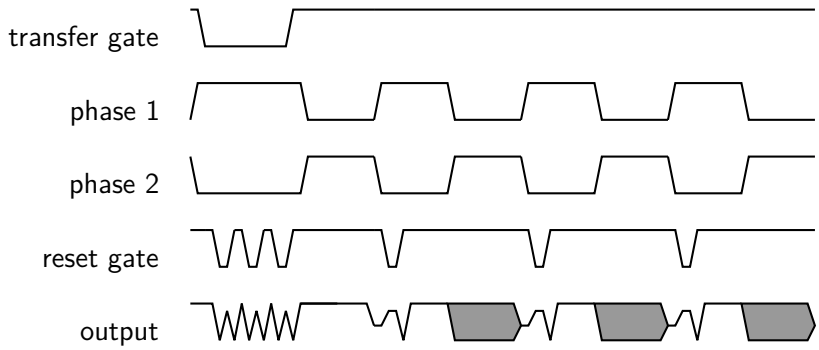
In a common two-phase driven CCD, there are two electrodes on the insulating layer over each pixel. The geometry of electrodes is asymmetrical, dragging the electron packet nearer to one end of electrode [11]. Therefore, a two-phase only signal is needed to transport the charge to the next pixel. The two phase crossing must be nearly symmetrical, which requires them to be switched at once[1].

The silicon surface can integrate the charge produced by internal photoelectric effect, but this is rarely used now. Instead, contemporary sensors contain an array of photodiodes, which are much more

[1] In one of the first designs, these signals were switched in two consequent instructions of the microcontroller and the delay of $\approx$ 80 ns probably caused the CCD to malfunction.

sensitive than the actual CCD register. When a *transfer gate* impulse comes, the charge is transferred to each corresponding pixel in the CCD register. This also eliminates blur of the image when the charges are shifted [4]. At the end of the analog register, the charge is coupled out to a pre-amplifier.

Fig. 3: Simplified diagram of waveforms driving a linear CCD. Input levels are TTL, corresponding output ranges approximately from 2 to 7 V. Grey areas contain each pixel's output voltage.



2.2.3 CCD output handling

Depending on the particular design, the two CCD phases and additional one to three inputs need to be activated by an impulse so that each pixel is read. After correct pixel driving signals are input, the CCD shifts the pixel's charge to its internal preamplifier and the corresponding voltage signal occurs at one pin. Sometimes is the output voltage stored in a sample-and-hold circuit.

The CCD detector output shall depend on the illumination linearly, but the signal is inverted, at least in the CCD detectors considered, e. g., for the CCD used in this project (NEC μPD3799): Under no illumination, the output is approximately $+7$ V, whereas under full saturation the output drops to $+2.4$ V. The influence of incident light on the CCD register itself and its supporting circuitry, along with thermal electrons, can interfere with measurement.

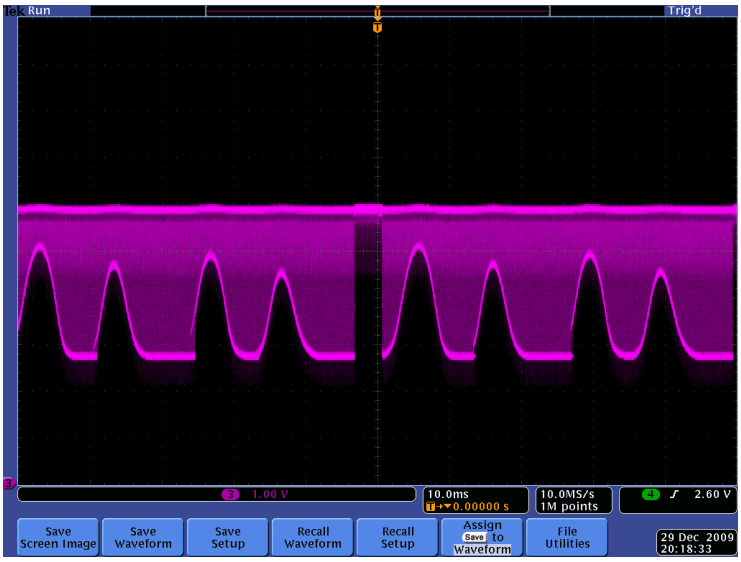
- **Blooming** occurs when excess charge concentrates in one cell and it leaks into neighbouring cells. It manifests as a flat, fully saturated regions with sharp edges, effectively obscuring the signal in the vicinity of the intensively illuminated spot. Most modern CCDs address this by electronic design, but the simple linear CCDs used in this project do not. Overexposed areas must be avoided by either reducing the light intensity or integration time. For instance, the saturation exposure of the used linear CCD is lower than 0.5 lx·s [4].

- **Charge volatility:** The stored charge decays over time. This is expected to be caused by both thermal electrons and incident light. At room temperature, it was experimentally proven that the CCD must be read in ≈ 0.5 s after *gate transfer* to avoid this. This presents a time constraint to the data acquisition and transmitting time. When the CCD is read in bursts, a sawtooth-like additive bias will occur.

- **Oversaturation:** When the CCD is illuminated by common daylight for a while (> 10 lx·s), the chip becomes so oversaturated that several full read-outs are needed to drain the integrated charge. Otherwise the output voltage may drop even under the saturation voltage.

  Sometimes this manifested as a step transition from saturation of one colour component in the middle of CCD reading, even when the CCD has been in dark for the last integration period.

Fig. 4: One of the first measurements of the CCD output on an oscilloscope. On the screen, there are two consecutive readouts of CCD. The detector window was covered by four thin paper stripes. Note the end-of-register signal in the middle of the screen and the blooming which slightly overlaps the less illuminated areas.



- **Thermal noise** is caused by random thermal electron-hole pairs. It is probably lower than electronic noise in the sampling circuit used.

Some of the aforementioned issues can be addressed by thermoelectrical cooling of the sensor, as a 20 °C temperature drop is expected to effectively suppress the thermal noise and conductivity. No cooling was used in this design because of rather high power requirements of a Peltier cell.

When a colour CCD is used, the red, green and blue signals must be averaged to obtain sensitivity covering the whole spectrum to be measured. This was done by inserting a 10 kΩ resistor at each channel and by connecting their second terminals together. The resulting spectral sensitivity is inhomogeneous, which was later addressed by amplitude calibration.

2.3 Description of the Atmega8 microcontroller

The proper CCD driving, analogue to digital conversion and consequent data transmission would require complicated circuitry if constructed from discrete components or TTL logic. Even such designs were constructed but most of these tasks can be easily handled by a common microcontroller of middle class. The most demanding task is the USB communication, which determines the minimum MCU computational power to approximately 12 MIPS at least. This topic is discussed later.

The microcontroller should be equipped by a sufficient internal analog-to-digital converter (ADC). Usage of an external ADC would present additional cost and complexity. Many contemporary microcontrollers have an internal 10-bit converter able to perform at least 10,000 samples per second, which is sufficient for this application.

Additionally, the need to avoid the mentioned sawtooth bias requires that the whole CCD is read at once and the spectral curve is stored in the memory until it is transmitted to the computer. Many microcontrollers come with fairly limited amount of internal RAM, which ranges from tens of bytes to 1 kB in the middle class. Taking into account that about 100 bytes are allocated by various program

data and that each pixel occupies 10 bits, there remains room for approximately 720 pixels if 1 kB of RAM was used. This severely limits the theoretical spectral resolution of 5300 pixels. The real spectral resolution in this case is, however, more limited by imprecise optics, and therefore the 1 kB of RAM was considered sufficient.

If higher optical resolution was achieved and consequently bigger data buffer was appropriate, microcontrollers with 2 kB or 4 kB of RAM are available for approximately 5 Euro and 8 Euro, respectively. A cheaper solution would be an external RAM. As of 2010, the market is saturated by cheap parallel static RAM circuits with capacity in tens of kB. However, most of these circuits use more than 20 pins for addressing and data transmission, which would occupy all available pins of the cheaper microcontroller. The SRAM circuits communicating over the serial line are much less available, although there are plenty of rather cheap serial EEPROM circuits, which have limited number of write cycles. Possible candidates could be *Microchip 23K640* and *23K256* models with 8 and 32 kB, respectively. Anyway, the use of external RAM would make the device more complicated and error prone. Therefore no external RAM was used in this design.

As 1 kB of RAM was considered sufficient, the *Atmel Atmega8* microcontroller was found to fullfill all aforementioned requirements. Moreover, this MCU supports *in system programming*, which was very handy to develop and debug communication with both CCD and computer. The *Atmega8-16PU* version has 28 pins, of which 4 were occupied by DC supply, 2 by USB, 2 by oscillator, 4 by CCD driving signals and one for ADC input. There remained enough pins and capacity for a 7-segment display and simple user controls on the device, but later it was considered impractical and only the computer control was left. As for 2010, *Atmega8-16PU* circuit is still one of the most cost effective microcontrollers on the market, with price of about 1.5 Euro.

2.4 Communication interface

2.4.1 Comparison of different interfaces

Several different interfaces are commonly used to transfer data between a computer and external instruments.

- The serial port (RS232) can use 3 wires only, one wire for each direction and a ground. The data rate must be set on both sides. Serial line is often used in industrial applications and for older computer peripherals. The microcontroller used in this project has hardware circuit for asynchronous serial communication, which could make this option favourable. However, RS232 begins to disappear from newer computers and it requires the user to set the serial port options manually, too.

- The parallel port would probably require writing an own driver for data transfer and it would be too bulky for this project.

- The GPIB interface is often used to communicate with scientific instruments, but is rarely available at standard computers, let alone notebooks. Moreover, it shares the disadvantages of parallel port.

- The ethernet interface uses complicated, high-level protocol, which probably could not be processed by the microcontroller unit (MCU) and would increase the device complexity.

The *Universal Serial Bus* interface remains to be the most advantageous option, providing decent data rate, excellent availability and sufficient power supply.

In the *low-speed* mode, the signalling speed is 1.5 Mb/s. This data rate can be processed by the MCU firmware, although the MCU has no hardware acceleration for USB communication.

For higher data rate in *full-speed* mode of 12 Mb/s, a dedicated USB to RS232 converter may be used. An example is the *Future Technology FT232BM* circuit. According to the datasheet, it should be able to communicate with computer via USB, handle all USB protocol messages and communicate with the MCU via serial port. In this case the maximum data rate over serial port should be 2 Mb/s. [9] This may not seem as an advantage over firmware USB driver until one realizes that it is the USB protocol in the *low-speed* mode what presents the bottleneck, limiting the average data rate to less than one tenth of the signalling speed. Likewise, when the MCU communicates using the USART (*Universal Synchronous and Asynchronous serial Receiver and Transmitter*) subcircuit, less time is spent by the driver routines used to handle the messages. The USB-RS232 converter was not used here in order to keep the device simple.

2.4.2 Hardware for USB communication

As no clock signal is sent over USB and the USB specification requires 1.5 % maximum tolerance for the timing [16], the MCU clock shall be timed precisely. Typically, the firmware drivers require oscillator whose frequency would be a multiple of 1.5 MHz. The minimum clock frequency is limited by the particular driver; mostly at least 8 processor steps per bit are required. A 12 MHz quartz crystal was used in this project.

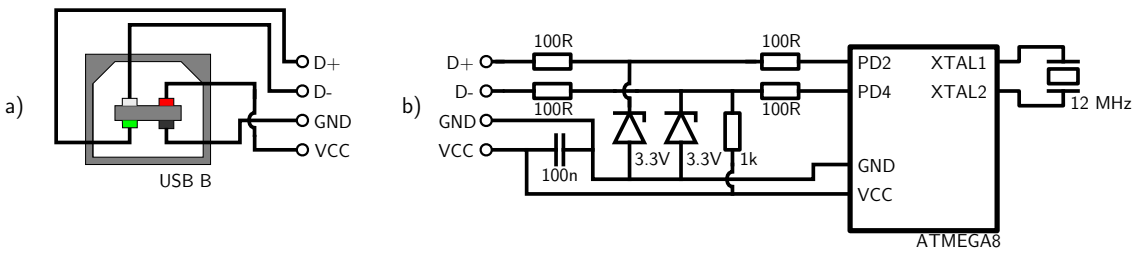
The USB cable contains 4 wires, which are listed in the table 1. The pins of the USB-B connector, used on the device side, are drawn in Fig. 5a. The *logical one* voltage should fall between 2.7 and 3.6 V [16] , therefore the 5 V TTL output of the MCU must be limited by Zener diodes. Additionally, the D− line must have logical one when the MCU is reset [16, p. 141], which is ensured by the 1 kΩ pull-up resistor (Fig. 5b).

Table 1: USB cable wires

Wire label	Usual colour	Purpose
GND	black	ground
U_{CC}	red	DC power (up to 500 mA at 5 V)
D+	green	half-duplex, differential data line
D–	white	*dtto*

The USB datasheet [16, p. 128] has some stricter requirements, e. g. for the device resistance, but the circuit from Fig. 5 worked flawlessly. Similar schematics are used in many other projects.

Fig. 5: a) Pins on the USB B connector; b) Minimal circuit for connecting ATMEGA8 to USB



2.4.3 Brief description of USB protocol

The data sent over USB are encoded using *non-return-to-zero* format: when a "0" bit is transmitted, the D– and D+ differential lines switch their state. When a "1" bit is transmitted, no change occurs. However, when more than 6 consecutive "1" bits are to be transmitted, one "0" bit is *stuffed* after them to enable the receiver to synchronize its clock at an edge of incoming signal. The data lines do not have to be always differential. For several low-level signals, e. g. end of packet, the D– and D+ lines may go both to logical "0" or "1".

All data are sent in *packets*. The packet contains an initial *sync* sequence, the packet identification number and optionally payload or control data along with its checksum. Some packets transmit payload data, while other packets are used for handshake and identification only. Three packets, *token*, *data* and *handshake*, usually form a *transaction* [5]. The USB protocol, mostly in a simplified, yet working form, is handled fully by the firmware driver.

After being requested by the host, each USB device identifies itself by a vendor number and an USB class. Most devices fall into the *human interface device* or *mass storage* classes. Each class is handled slightly differently by the computer. The *human interface device* class was chosen due to its simplicity and widespread universal drivers [16]. USB protocol is rather complicated and its more detailed description is beyond the scope of this document.

There are several software implementations of the USB protocol for AVR microcontrollers. For example, Igor Češko [2] and Jan Smrž [12] implemented their own. This project utilizes the *V-USB* driver by Christian Starkjohan [13] because of its good documentation and numerous practical examples. The driver is licensed under GNU GPL for noncommercial use.

All these drivers use a hardware interrupt, which is triggered by incoming USB packet. At 1.5 Mbit/s signalling speed and 12 MHz main oscillator, only 8 clock cycles remain to receive, convert and store each bit. This requires that the microcontroller completely halts all another operation when any USB packet is received or transmitted. Besides, the USB protocol has the *master-slave* topology, and accordingly the MCU stops to receive or transmit packets asynchronously with its operation, which has to be taken into account when programming real-time tasks.

3 Solution

3.1 Hardware

3.1.1 Optical setup

The optical components holders were made of a standard aluminium L-profile 25 mm high, 2 mm thick and with 12 mm long foot. The beam path axis was chosen to be 14 mm above the base surface. This provided optimal vertical room for the beam, maintaining enough room between the light path and the base for bolts etc.

- The input slit was made of two razor blade segments, 15 mm long. One of the segments was glued onto the inner side of the L-profile to cover half of the input hole. Its sharp edge was adjusted to be perpendicular to the base. The second segment was clamped next to it by a short screw with a spring bushing. This makes adjusting the slit width possible. In the final setup, the slit width of 50 μm – 100 μm appeared to be appropriate.

- The collimating lens had 14 mm diameter, focal length $f \approx 30$ mm, plano-convex geometry and standard anti-reflex coating. (Generally, any common small lens of similar f may be used.) It was glued at the edges to a hole of similar diameter, keeping its axis at the beam axis. At the base of the L-profile, a notch instead of a hole was grinded to enable focusing.

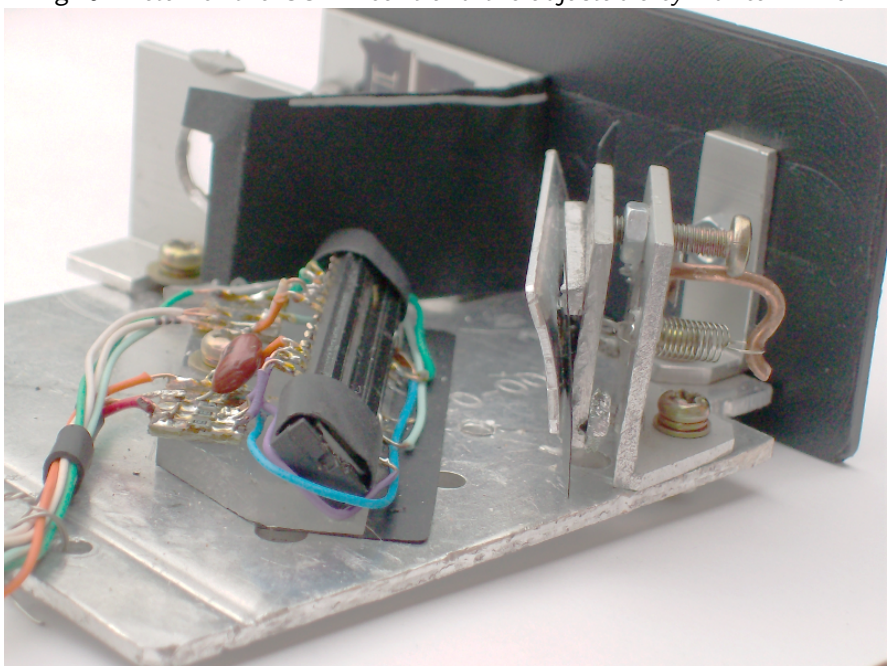
- The reflective diffraction grating was attached nearly perpendicular to the incident beam. For the first prototype, a 20×20 mm segment cut from the outer edge of a blank CD-R was used. Notes to this solution are discussed later. The grating was rotated so that the most desired spectral range in the first diffraction order is projected on the CCD.

- The cylindrical mirror was obtained from a scrap photocopier. It is a polished metal rectangle, 42×15 mm in size. Its focal length is approximately 30 mm. The mirror was attached to a metal chip, which had a groove at the bottom. The groove fitted to two spikes in the bottom of the L-profile. At the top, an M3 screw allows to adjust the vertical angle of the mirror.

- The CCD detector was glued to a low profile PMMA slab. Its pins were shortened not to obscure any passing light and soldered to the controller circuit. The slab had a notch for the bolt.

  The output signal from three colour channels was averaged by three 10 kΩ resistors near the CCD and it was connected using a thin shielded cable to eliminate noise from the digital wires. Between the supply voltage and ground, a 100 nF capacitor was inserted.

Fig. 6: Detail of the CCD mount and the adjustable cylindrical mirror



The optical setup was mounted (by M3 bolts and nuts) to a 2 mm thick aluminium plate, of approximately 70×90 mm in size. It provided a decently rigid, yet light base. There remained 6 degrees of freedom, that had to be precisely adjusted to obtain effective and narrow spectral lines projected on the CCD.

1. The input slit width: The second razor segment should limit the light entrance to an uniform, thin line.

2. The longitudinal position of the collimating lens should focus the slit onto the CCD in the horizontal plane.

3. The vertical grooves orientation of the diffraction grating should be parallel to the slit.

4. The horizontal angle of the diffraction grating: The whole desired spectral range should be projected onto the CCD.

5. The cylindrical mirror and CCD should be perpendicular to the beam coming from diffraction grating, which illuminates the center of CCD.

6. The vertical angle of cylindrical mirror and the corresponding position of the CCD detector: Along with previous setting, it should focus the spectral lines to illuminate the narrow CCD line the most effectively.

The coarse setting was done visually using the light from a standard fluorescent lamp. The fine tuning of the lens position was performed when the device was connected to computer and the spectral lines were optimised to be as narrow as possible.

3.1.2 Diffraction grating source

The reflection diffraction grating of suitable parameters prove to be the component, that was the hardest to obtain for this project, maintaining the low price criterion. Ideally, a blazed diffraction grating with 500 – 1000 grooves/mm, with size at least 8×8 mm and with optimum efficiency in the visible range should be used. As of 2010, there is likely no shop in the Czech Republic where such gratings could be bought easily, according to the search on Internet.

Future models can use a professional grating, which would probably cost at least 50 Euro [8] . This price is not limiting, however it would spoil one of the objective of minimizing the device's cost. Maybe several gratings could be replicated after one professional grating is bought.

Alternatively, there is a proof that a reasonable diffraction grating may be manufactured using a nearly home-built ruling engine, taking into account the contemporary availability of MCUs, lasers and stepping motors [10]. Anyway, the precision requirements for such ruling engine are very tight. To achieve 1 nm spectrometer resolution at the 500 nm wavelength, the RMS error of groove spacing must be at most few nanometers. This limits the groove spacing error to less than 10 diameters of silver atoms!

For the first prototype, the CD-R slab worked quite well. The cheapest CD-R had unexpectedly low stray light and no apparent *Rowland ghosts* coming from low-frequency modulation of the grating. Apart from non-optimal efficiency and fragile metallic coating on the surface, the biggest problem was caused by the grooves curvature. A monochromatic beam limited by the slit is projected into a crescent shape in the focal plane of the collimating lens. Instead, a vertical line is required, as the cylindrical mirror focuses the beam in the vertical plane.

3.1.3 Microcontroller circuit

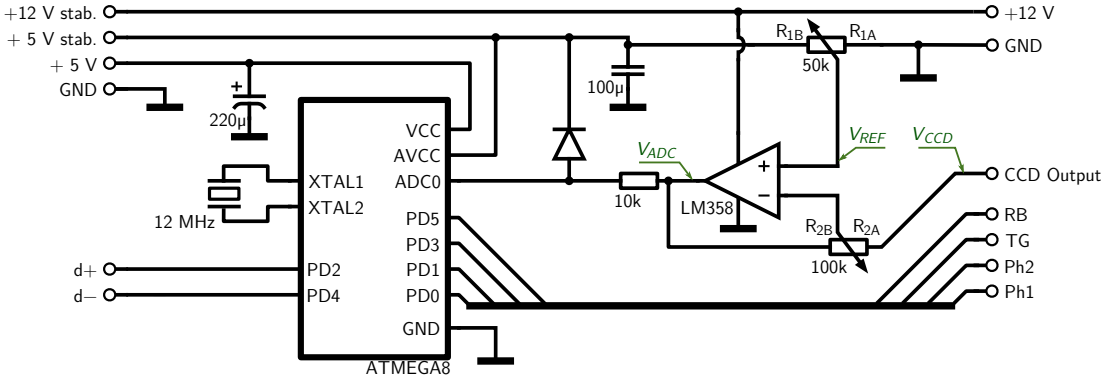
Aside from the CCD detector and the supply circuit mentioned later, the electronics do not require many components. The MCU is clocked by a 12.000 MHz quartz oscillator. Some designs suppose to couple each oscillator input to ground by two 30 pF capacitors, but this has proven not to be necessary.

The CCD driving signals use the TTL levels, hence the CCD inputs may be connected directly to the MCU pins. For the used *NEC μPD3799* CCD detector four signal wires are needed: the *phase 1* and *phase 2* signals, the *RB* (*reset gate*) and the *TG* (*transfer gate*) signals. When another CCD is used, additional driving signals may have to be added, e. g., for *NEC μPD3797*, additional *CLB* (*reset feed-through level clamp clock*) negative impulse must be included after the *RB* impulse. According to the particular datasheet, both the firmware routines and hardware wiring can be easily modified to accomplish this. Note that one MCU clock takes only 83 ns and some minimum pulse length and delay must be maintained according to the CCD datasheet.

As was noted in the 2.2.3 section, the output signal of CCD is too high to be directly connected to the analog-to-digital converter (ADC) of the MCU. The output must be linearly inverted, amplified and shifted before passing to the ADC, which measures in the range of 0 to 5 V. There is an older project of Paul Stoffregen, who accomplished this transformation using five operational amplifiers, of which two act as a sample-and-hold circuits, storing the two voltage levels of dark and saturated signal [14]. A circuit of two potentiometers and only one operational amplifier was used here instead, as both the dark and the saturated voltage levels both prove to be relatively stable.

If only one colour component becomes saturated, the output signal gets distorted and therefore unusable. Accordingly, the signal range should be stretched so that full saturation of single colour component corresponds to full saturation of the signal entering the ADC. In other words, the output signal must be amplified with gain enough to prevent the saturation of any colour component in CCD. The circuit with inverting preamplifier and microcontroller is shown in Fig. 7. Let us denote the CCD

Fig. 7: The microcontroller circuit. At its left side it is connected to the supply circuit, at right to the CCD detector.



output voltage as V_{ccd}, the voltage reference from $R_{1A,B}$ as $V_{ref} = \frac{R_{1A}}{R_{1A}+R_{1B}} \cdot 5$ V and the voltage at the output of the operational amplifier as $V_{adc} = V_{adc}(V_{ccd}, V_{ref})$. Taking into account that the operational amplifier inputs virtually no current, one obtains that

$$\frac{V_{ccd} - V_{ref}}{R_{2A}} = \frac{V_{ref} - V_{adc}}{R_{2B}}$$

and after simple arrangement

$$V_{adc} = \frac{R_{2A} + R_{2B}}{R_{2A}} \cdot V_{ref} - \frac{R_{2B}}{R_{2A}} \cdot V_{ccd}.$$

This proves that if not saturated, the operational amplifier performs the desired linear transformation. The voltages do not have to be known precisely, as the potentiometers R_1, R_2 may be tuned depending on the waveform in computer. The amplification factor should be between 3 and 5.

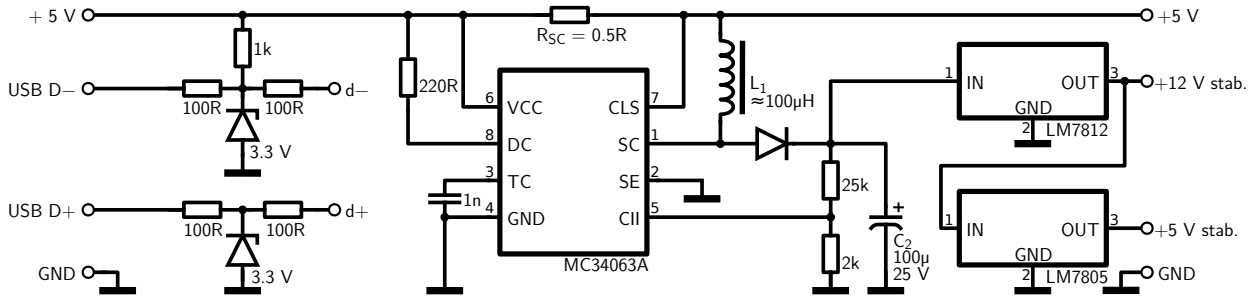
The operational amplifier supply voltage has to be 12 V, which presents a risk of destroying the ADC circuit by excess voltage, as the maximum input voltage of ADC is limited to 5.3 V in this case [3]. In order to prevent this, a 10 kΩ series resistor and a diode with low voltage drop limit the ADC input voltage, as shown at Fig. 7.

3.1.4 Supply circuit

The supply circuit converts the noisy 5 V USB voltage supply to stabilized 12 V required by the CCD detector and to additional stabilized 5 V for the ADC. Additionally, it limits the USB signalling levels to 3.3 V as required by the specification [16].

For the voltage conversion, a boost converter integrated circuit *MC34063* was used. In short, the circuit senses the output voltage at the *CII* (*comparator inverted input*). When the voltage drops below 1.25 V, the *SC* (*switch collector*) shorts to ground, letting the current through the L_1 coil and R_{sc} grow. The *CLS* (*current limit sense*) voltage drops as the current increases and when the voltage drop exceeds approximately 300 mV, the *SC* pin closes. The energy stored in the L_1 coil creates an impulse of higher voltage that charges the C_2 capacitor. The C_1 capacitor determines the time interval of operation. The

Fig. 8: The supply circuit. At its left side it is connected to the computer via USB cable.



output voltage is given by the voltage divider parallel to the C_2. The selected resistances of 25 kΩ and 2 kΩ set the minimum C_2 voltage to 16 or 17 V. This voltage is limited by the *LM7812* and *LM7805* stabilizers to 12 V and 5 V, respectively. Reducing the C_2 voltage would increase efficiency, but risk of negative ripple in the 12 V stabilized output would be present due to the voltage drop at *LM7812*.

The USB signalling wires *D+*, *D–* are connected according to the Fig. 5.

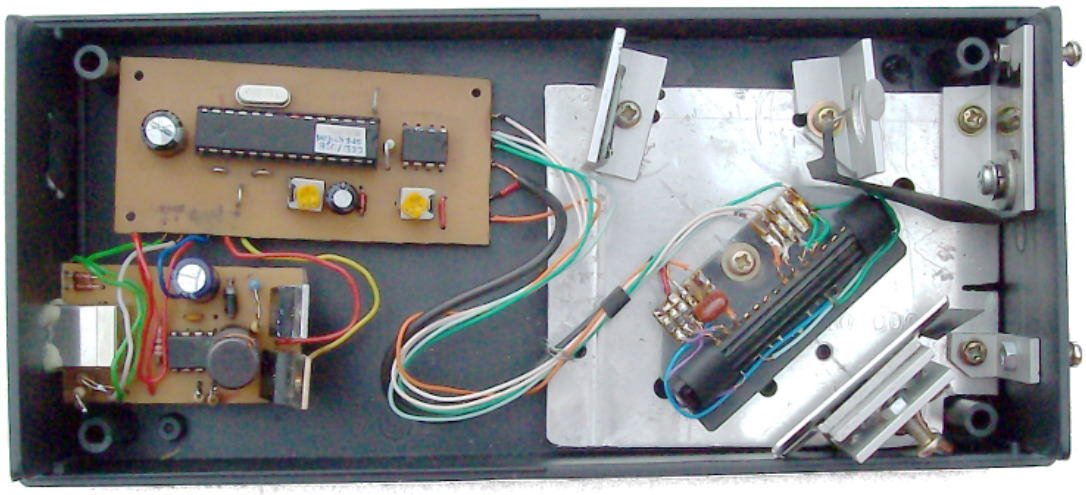
3.1.5 Spectrometer casing

The spectrometer was cased in a black PVC box with dimensions 190 × 90 × 50 mm. About a half of the box is occupied by the optical mount, the rest contains the two circuits. The optical mount is fixed to the front panel by two M3 bolts with head containing a nut, which are common e. g. in older

computer connectors. This enables the user to fix some light source or cuvette in front of the input slit (or vice versa, attach the spectrometer onto a bigger device output) easily.

These two bolts, as well as whole optical mount, were not connected to the USB ground to prevent accidental short-circuit or ground loop. Although the matt inner side of the plastic box has low reflection

Fig. 9: A view of the assembled spectrometer without top cover and paper light shielding. The light passes through the input slit at the top right and follows the path indicated at Fig. 1. At the left, there are the converter and microcontroller circuits and USB output.



itself, the cylindrical mirror and CCD were additionally covered by a folded sheet of black paper to minimize the stray light coming directly from the input slit or any tiny hole left in the plastic case.

3.2 Microcontroller firmware

In this section, the full microcontroller firmware written in the *C* language (except for the USB driver) will be described, as it well illustrates the device function. The code comments were mostly maintained.

The microcontroller was programmed via the serial programming interface using a *USBAsp* hardware programmer and the *avrdude* program. The firmware code was compiled on Linux (Ubuntu 10.04) in *avr-gcc* compiler and it was loaded into the microcontroller flash memory with commands:

```
avr-gcc -g -Wall -O1 -mmcu=atmega8 -c main.c -o main.o
avr-gcc -g -Wall -O1 -mmcu=atmega8    main.o -o main.elf
avr-objcopy -j .text -j .data -O ihex main.elf  main.hex
sudo avrdude -p m8 -c usbasp -U flash:w:main.hex
```

To enable the 12 MHz clock, two reserved bytes in the persistent microcontroller memory, called *fuses*, must be set to hexadecimal values of 0xff for *lfuse* and 0xd9 for *hfuse*.

```
sudo avrdude -p m8 -c usbasp -U lfuse:w:0xff:m -U hfuse:w:0xd9:m
```

During experiments, the fuses have been set wrong several times, which disabled also the serial programming needed to set fuses back. This was fixed by a parallel programming fuse resetting tool [6].

3.2.1 General constants and procedures

At the beginning, the essential header files from the *avr-libc* library are included.

```
#include <avr/io.h>
#include <avr/wdt.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <avr/eeprom.h>

#include <avr/pgmspace.h>
#include "usbdrv/usbdrv.h"
#include "usbdrv/oddebug.h"
```

To make the CCD driving routines clear, each CCD pin is referenced by its name from the datasheet instead of a simple port number. The CLB and SHB signals are predefined for the case that a CCD with more complicated driving signals is used. This definition enables to change the connection easily if needed.

```
// PORTD pins definition to comply with datasheet
#define PH1     _BV(PD0)
#define PH2     _BV(PD1)
#define TG123   _BV(PD3) // Transfer gates (R+G+B)
#define RB      _BV(PD5) // Reset gate clock
#define CLB     _BV(PD6) // Reset feed-through level clamp clock (disabled)
#define SHB     _BV(PD7) // Sample and hold clock (disabled)
```

The following part contains the USB report descriptor, which is sent when the device is recognized by the computer. Further USB settings are set in the *usbdrv.h* file which is provided with the USB driver.

```
PROGMEM char usbHidReportDescriptor[22] = {         /* USB report descriptor */
        0x06, 0x00, 0xff,              // USAGE_PAGE (Generic Desktop)
        0x09, 0x01,                    //   USAGE (Vendor Usage 1)
        0xa1, 0x01,                    //   COLLECTION (Application)
        0x15, 0x00,                    //     LOGICAL_MINIMUM (0)
        0x26, 0xff, 0x00,              //     LOGICAL_MAXIMUM (255)
        0x75, 0x08,                    //     REPORT_SIZE (8)
        0x95, 0x80,                    //     REPORT_COUNT (128)
        0x09, 0x00,                    //     USAGE (Undefined)
        0xb2, 0x02, 0x01,              //     FEATURE (Data,Var,Abs,Buf)
        0xc0                           //   END_COLLECTION
};
```

Here we define further constants: the number of all CCD pixels and the remaining space in memory to save the buffer. The full RAM size of *Atmega8* is 1 kB, of which some bytes are occupied by variables and stack and other bytes are required for the USB driver. Approximately 920 bytes remain to store the data buffer in this configuration.

The *BufMsrPos* and *BufBitPtr* denote the position in the buffer to write the 10 bits measured by ADC. The latter variable points to the *bit* at which the value shall be written, as is described later. The *CCDMsrPos* variable is incremented after measurement of each pixel to indicate when the CCD has been read out. The *BufReadPos* stores the position in the buffer which will the data transmission start from. Following variables will be described later.

```
# define PIXELS_COUNT 5400
# define MSR_BUF_SIZE 900
```

```
static unsigned char buf[MSR_BUF_SIZE];
static int  BufMsrPos;
static char BufBitPtr;
static int  CCDMsrPos;
static int  BufReadPos;
static unsigned int  IntegrTime;
static char    Status;  // indicates the status of measurement or transmission
```

This procedure creates a short delay of approximately 2 μs $\cdot$ *num*. It is needed to hold the minimum timings when driving the CCD.

```
void delay(unsigned int num)
{
  unsigned int i,j;
  for (j = 0; j < 2; j++)
   for (i = 0; i < num; i++)
     ;
}
```

3.2.2 CCD driving

Following three procedures are used to drive the CCD. They may need to be modified if different CCD is used. The first procedure performs one CCD register shift, as is drawn at Fig. 3. It both begins and ends in the *phase 1* state. A minimum delay of 110 ns is required for each phase [4].

```
void CCDShiftReg()          // expects the CCD to be in the phi2 state
{
   PORTD = (PH1);         // inverse RB strobe
   PORTD = (PH1 | RB);
   delay(0);
   PORTD = (PH2 | RB);
   delay(0);
}
```

The *CCDTransferGate* procedure initiates the CCD readout by transferring the charge from the photodiodes. When the *transfer gate* signal is activated, ten cycles of *reset gate* strobes are sent to the CCD, as is recommended [4].

```
void CCDTransferGate()
{
   int j;
   PORTD = (PH1);
   PORTD = (PH1 | RB);
   delay(2);
   for (j = 0; j < 10; j++) { // transfer gate
      PORTD = (PH1 | TG123);
      PORTD = (PH1 | TG123 | RB);
      delay(1);
   }
   PORTD = (PH1 | RB);
   delay(2);
   PORTD = (PH2 | RB);
   delay(1);
}
```

The *FlushCCD* procedure first performs several full readouts to drain all excess charge from the CCD (the more sensitivity was set, the more readouts are used). When the CCD was not used for a while, this does not drain the charge at once and several such cycles are needed.

After this, an integration period begins. Its duration is determined by the *IntegrTime* global variable, as the delay here is accomplished by running the ADC repeatedly each 32 CCD steps. The sampling speed is approximately 10,000 samples per second. Therefore, the *IntegrTime* value of 32 sets the integration time to half a second. Finally, the *CCDTransferGate* enables the measurement.

The sensitivity is roughly proportional to the integration time. However, two effects limit the integration time to approximately one second: first, the spectrometer does not respond to USB messages, causing USB timeouts when integrating too long, and secondly, the CCD charge volatility creates a significant background signal. To improve the sensitivity, the CCD would probably have to be cooled by a Peltier element and the integration should be controlled by asynchronous timer. The best way to detect weak signals in the current setup is to set moderate sensitivity and average multiple spectral measurements.

```
void FlushCCD() {
   int j, i, k;
   PORTB |= _BV(PB0);
   CCDTransferGate();    // read the CCD to drain all charge
   for (i = 0; i < IntegrTime; i++) {
      for (j = 0; j < PIXELS_COUNT; j++) { CCDShiftReg(); };
      CCDTransferGate();
   }
   for (j = 0; j < PIXELS_COUNT; j++) {
      for (i = 0; i < IntegrTime; i++) {  // delay loop
         k++;
         if (k%32 == 0) {
            ADCSRA |= _BV(ADSC);
            while  (!(ADCSRA & _BV(ADIF)));
            ADCSRA |= _BV(ADIF);
            k~= 0;
         }
      }
      CCDShiftReg();
      };
   CCDTransferGate();
   PORTB &= ~(_BV(PB0));
}
```

3.2.3 USB communication

As noted above, the USB communication is ensured by a software driver by Hans Starkjohann (Objective Development GmBH) [13] . The USB driver has relatively low requirements, as it occupies no more than 1.5 kB of program code.

These definition will be be used later in the following routines.

```
   // Spectrometer commands received via USB
#define COMMAND_SET_MEASURE 100
#define COMMAND_READ_CALIB_AMPLI 101
#define COMMAND_WRITE_CALIB_AMPLI 111
#define COMMAND_READ_CALIB_LAMBDA 102
#define COMMAND_WRITE_CALIB_LAMBDA 112
```

```
7     // Status
8  #define STATUS_IDLE 0
9  #define STATUS_CCD_PENDING 1
10 #define STATUS_CCD_INTEGRATING 2
11 #define STATUS_SAMPLING 3
12 #define STATUS_BUFFER_READY 4
13 #define STATUS_CALIB_READING 10
14 #define STATUS_CALIB_WRITING 11
15    // Room for calibration data in EEPROM
16 #define CALIB_BYTES_PER_POINT 4
17 // EEPROM bytes 9 ... 410 reserved for amplitude calibration data
18 #define CALIB_AMPLI_POINT_NUMBER_MAX 100
19 #define CALIB_AMPLI_BUFPTR  9
20 // EEPROM bytes 411 .. 511 reserved for lambda calibration data
21 #define CALIB_LAMBDA_POINT_NUMBER_MAX 25
22 #define CALIB_LAMBDA_BUFPTR  411
```

The *usbFunctionSetup* procedure is called by the driver always when a read or write packet arrives. The type of packet can be obtained from the *rq->bRequest* structure.

```
1  usbMsgLen_t usbFunctionSetup(uchar data[8])
2  {
3  usbRequest_t    *rq = (void *)data;
4     if((rq->bmRequestType & USBRQ_TYPE_MASK) == USBRQ_TYPE_CLASS){
5        if(rq->bRequest == USBRQ_HID_GET_REPORT){
6           BufReadPos = 0;
7           return USB_NO_MSG;  // use usbFunctionRead() to obtain data
8        }
9        else if(rq->bRequest == USBRQ_HID_SET_REPORT){
10          return USB_NO_MSG;  // use usbFunctionWrite() to receive data from host
11       }
12    }else{
13       // ignore vendor type requests, we don't use any
14    }
15    return 0;
16 }
```

The *write* packet contains data transmitted from the computer. Whereas the *usbFunctionSetup* is called once the packet arrives, the driver calls the following function *usbFunctionWrite* repeatedly, supplying the payload data in 8-byte chunks.

All actions of the spectrometer are initiated by an incoming USB message from the computer. The first byte of the message describes the *command*, as is defined above in the *#define COMMAND...* block. Possible commands are to start spectral measurement, to read or write the wavelength or amplitude calibration.

Once a *COMMAND_SET_MEASURE* message arrives, the spectrometer sets the integration time, flushes the CCD, starts the integration time, samples the data read from the CCD and stores the data in internal buffer. These steps can take up to several seconds and would cause USB timeout error if they were all done within the *usbFunctionWrite* procedure. Instead, only the *Status* variable is set here to *STATUS_CCD_PENDING* and all mentioned tasks are performed later in the main loop.

The *COMMAND_READ_CALIB_AMPLI* and *COMMAND_READ_CALIB_LAMBDA* both set the *Status* variable into the *STATUS_CALIB_READING*. The difference is in the initial EEPROM pointer value only.

```
1  static uchar    EEPROMPtr;
2  static uchar    CalibDataPointNumber;
3  static uchar   bytesRemaining;
4  uchar   usbFunctionWrite(uchar *data, uchar len)
5  {  // Longer tasks may not be handled here. Instead, set the Status variable and
       perform them in the main routine later.
6     if (Status == STATUS_CALIB_WRITING) {
7        if(bytesRemaining == 0)
8           return 1;               // end of transfer
9        if(len > bytesRemaining) {
10          len = bytesRemaining;
11          Status = STATUS_IDLE;
12       }
13       eeprom_write_block(data, (uchar *)0 + EEPROMPtr, len);
14       EEPROMPtr += len;
15       bytesRemaining -= len;
16       if (bytesRemaining == 0) {
17          Status = STATUS_IDLE;
18          return 1;   // return 1 if this was the last chunk
19       } else return 0;
20
21    }
22    // if first chunk of USB message, decide the command from the 1st byte
23    if (Status == STATUS_IDLE)
24       switch (data[0]) {
25          case COMMAND_SET_MEASURE: {
26             // if not zero, the two bytes set the sensitivity (integration time):
27             if (data[1]+data[2] != 0) {IntegrTime = data[1]*256 + data[2];};
28             Status = STATUS_CCD_PENDING;
29             return 1;          // no additional data required
30             break;
31          }
32          case COMMAND_READ_CALIB_LAMBDA:
33          case COMMAND_READ_CALIB_AMPLI: {
34             // decide which memory location will be read
35             if (data[0] == COMMAND_READ_CALIB_AMPLI) {
36                EEPROMPtr = CALIB_AMPLI_BUFPTR;
37             } else {
38                EEPROMPtr = CALIB_LAMBDA_BUFPTR;
39             }
40             // number of points to be read
41             eeprom_read_block(&bytesRemaining, EEPROMPtr, 1);
42             bytesRemaining=bytesRemaining*CALIB_BYTES_PER_POINT+1;
43             Status = STATUS_CALIB_READING;
44             // no additional data required:
45             return 1;
46             break;
47          }
48          case COMMAND_WRITE_CALIB_LAMBDA:
49          case COMMAND_WRITE_CALIB_AMPLI: {
50             // number of points to be written
51             bytesRemaining = data[1]*CALIB_BYTES_PER_POINT;
52             if (data[0] == COMMAND_WRITE_CALIB_AMPLI) {
53                EEPROMPtr = CALIB_AMPLI_BUFPTR;
54             } else {
```

```
55                  EEPROMPtr = CALIB_LAMBDA_BUFPTR;
56              }
57              // 1st byte of EEPROM block stores number of points:
58              eeprom_write_block(data+1, EEPROMPtr, 1);
59              EEPROMPtr++;
60              Status = STATUS_CALIB_WRITING;
61              // last 6 B of this chunk discarded, next chunk will contain data
62              return 0;
63              break;
64          }
65      }
66  }
```

The *usbFunctionRead* function is called when the computer asks the spectrometer for measured data. Naturally, the action depends on the *Status* variable: if the spectrometer finished the measurement recently, the spectral data are sent; if the microcontroller was requested for its calibration data, the corresponding EEPROM block is sent. Otherwise, no action is performed and the function returns 0 to indicate that no more data are available.

The USB specification limits the payload data length to 8 bytes for *USB low-speed* devices; however this would make the communication slow. Therefore, an extended length of packet up to the buffer size is used. This bigger size of packets is common for *high-speed* devices and it has been tested to work flawlessly, enabling to quickly transport the whole buffer in a single USB message.

```
1  uchar   usbFunctionRead(uchar *data, uchar len)
2  {
3      if (Status == STATUS_IDLE) {            // no command set
4          return 0;
5      }
6      if (Status == STATUS_BUFFER_READY) {          // Transmit the buffer with
           spectrum to the computer.
7          if (len > (BufMsrPos - BufReadPos - 1)) { // buffer will be depleted, read
               the left bytes
8              len = (BufMsrPos - BufReadPos);
9              if (CCDMsrPos < PIXELS_COUNT) {Status = STATUS_SAMPLING; BufBitPtr = 0;
                   BufMsrPos = 0;}
10                 else {Status = STATUS_IDLE;};
11             BufMsrPos = 0;
12         }
13         int DataPos;
14         for (DataPos=0; DataPos<(len); DataPos++) {
15             data[DataPos] = buf[BufReadPos + DataPos];
16             buf[BufReadPos + DataPos] = SENT_BUF_STUFF;
17         };
18         BufReadPos += len;
19         return len;
20     }
21     if (Status == STATUS_CALIB_READING) {         // transmit the calibration data
           from EEPROM
22         if(len > bytesRemaining) {
23             len = bytesRemaining;
24             Status = STATUS_IDLE;
25         }
26         eeprom_read_block(data, (uchar *)0 + EEPROMPtr, len);
27         EEPROMPtr += len;
```

```
28          bytesRemaining -= len;
29          return len;
30      }
31      return 0;
32  }
```

3.2.4 The main loop

This loop is called after the microcontroller starts and its clock stabilizes.

During this loop the microcontroller continuously polls the USB driver for incoming packets, which is required to ensure reliable communication, and performs the action the *Status* variable describes. After the CCD readout is finished, the *Status* variable is finally set to the *STATUS_BUFFER_READY* value. In a consequent read request from computer, the spectrometer sends the data in one or more USB packets and *Status* is set back to *STATUS_IDLE*.

```
1   int main(void)
2   {
3       DDRB |= _BV(PB0) | _BV(PB1); DDRD |= (TG123 | PH1 | PH2);   DDRD |= (RB | CLB |
            SHB); // enable output pins
4       ADMUX = 0; ADCSRA |= _BV(ADEN) | _BV(ADPS2) | _BV(ADPS2);      // A/D converter
            settings
5       odDebugInit(); usbInit(); usbDeviceDisconnect();          // enforce re-
            enumeration
6       unsigned char i; i = 0; while(--i){wdt_reset(); _delay_ms(5); }      // fake USB
             disconnect for > 250 ms
7       usbDeviceConnect(); sei();
8       IntegrTime = 1;
9       BufMsrPos = 0;
10      CCDMsrPos = 0;
11      Status = STATUS_IDLE;
12      for(;;){
13          usbPoll();
14          int j;
15          int MeasPerPoll = 1;
16          int PixPerMeas = 8;      // buffer capacity: 900 B = 7200 b = 720 px; 5400 pix
                /8 = 675 pix
17          if (Status == STATUS_CCD_PENDING) {
18              Status = STATUS_CCD_INTEGRATING;
19              // erase buffer
20              int i; for (i=0; i<MSR_BUF_SIZE; i++) {buf[i] = EMPTY_BUF_STUFF;};
21              FlushCCD();
22              BufBitPtr = 0;
23              CCDMsrPos = 0;
24              BufMsrPos = 0;
25              Status = STATUS_SAMPLING;
26          }
27          if (Status == STATUS_SAMPLING)
28              for (j=0; j<MeasPerPoll; j++)
29              if (CCDMsrPos < PIXELS_COUNT) {
30                    if (BufMsrPos < (MSR_BUF_SIZE-1)) {
31                        unsigned int value = 0;
32                        for (i=0; i<PixPerMeas; i++) {
33                            CCDShiftReg();  CCDMsrPos += 1;
```

```
34                      ADCSRA |= _BV(ADSC);
35                      while   (!(ADCSRA & _BV(ADIF)));
36                      ADCSRA |= _BV(ADIF);
37                      // 10 bits ADC input, always read ADCL first
38                      value += ADCL + 256*(ADCH&3);
39                  }
40                  // average from several measurements
41                  value = value / PixPerMeas;
42                  char BitNumber  = 10;
43
44                  // remove leading zeros
45                  unsigned int svalue = value << (16 - BitNumber);
46                  int ValBitPtr = 0;
47                  // compresses 4 pixels * 10 bits to 5 bytes
48                  while (ValBitPtr < BitNumber) {
49                      int ChunkLen = 8 - BufBitPtr;
50                      if (ChunkLen > BitNumber-ValBitPtr) {ChunkLen = BitNumber-
                            ValBitPtr;};
51                      unsigned char Chunk = ((svalue << ValBitPtr)  >> BufBitPtr) >>
                             8;
52                      unsigned char Mask  = ~(0xFF     >> BufBitPtr);
53                      buf[BufMsrPos] = (buf[BufMsrPos] & Mask) | (Chunk & 0xFF);
54                      ValBitPtr += ChunkLen; BufBitPtr += ChunkLen;
55                      if (BufBitPtr >= 8) {BufBitPtr -= 8; BufMsrPos++;}
56                  }
57              } else {
58                  BufBitPtr = 0;
59                  Status = STATUS_BUFFER_READY; // buffer full, wait until the
                        computer reads the data
60              }
61          } else {
62              Status = STATUS_BUFFER_READY;        // whole CCD read, wait
63          }
64      }
65      return 0;
66  }
```

The used CCD detector had much more pixels than the microcontroller could hold in the buffer at once. Although full CCD data could be read and transmitted sequentially, this was problematic due to the sawtooth bias. The need to fit the full CCD readout into the 7200 bits long buffer has lead to these two measures:

1. As the microcontroller has a 10-bit analog-to-digital converter, the measured values are compressed on the fly so that each value does not occupy 16 bits as in an integer, but only the significant 10 bits. Theoretically, the 7200 bit buffer can hold 720 measured points.

2. The total pixel number was 5400. Dividing this by 8 we obtain 675 measured points, which fits the buffer the best. Accordingly, eight pixels were read sequentially and then averaged.

   This reduced the noise significantly, but did not spoil the spectral accuracy much, because the optical imprecision manifested even in the reduced pixel number.

3.3 Computer software

The software in computer was split into two programs:

1. the *backend* ensures communication with the spectrometer and transforms the data according to the wavelength and amplitude calibration,

2. and the *frontend*, which provides graphical user interface and, calling the backend repeatedly, presents the measured spectra in real time.

3.3.1 Command line backend for USB communication

The backend is written in *C*, compiled using the *Gnu Compiler Collection* (*gcc*, version 4.4.3). It was developed and tested under Linux 2.6, but it is designed to compile and run under Windows and other operating systems, too. Under Linux, the backend depends on *libusb*, on Windows it depends on the *Driver Development Kit* (DDK), which should be shipped along with *MinGW* compiler [13].

The full code is included in the attached CD and it will not be listed here. The commandline backend accepts several commands. The first parameter determines the action performed and optional parameters are in square brackets:

1. `set_measure [integration time]` starts the measurement, integration time ranges from 1 to approx. 200

2. `get_spectrum` receives, transforms and writes out the spectrum to the stdout

3. `write_calib_ampli lambda1:amplitude1,[lambda2:amplitude2,[...]]` write amplitude calibration to the EEPROM

4. `write_calib_lambda pixel1:lambda1,[pixel2:lambda2,[...]]` write wavelength calibration

5. `read_calib_ampli [--short]` reads amplitude calibration from the EEPROM

6. `read_calib_lambda [--short]` write wavelength calibration

Usually the *get_spectrum* command is called soon after the *set_measure* was called. (These commands do never have to be joint together into one command, as the spectrometer requires about half a second to measure the spectrum and much longer wait periods can occur if longer integration time is needed. Such single command would halt the backend from responding. This would consequently either cause the frontend not to respond for a while or it would require to implement multiple threads.)

The calibration *write* commands use the second parameter to set the calibration points in the EEPROM. Each calibration point occupies 4 bytes in the EEPROM. See the `README` file for more information.

For wavelength calibration, pairs of pixel number and corresponding wavelength in picometers have to be input in the format of pairs of integers related by colon and separated by comma. First two bytes are the integer position of the pixel. Following two bytes are the wavelength divided by 32 pm. Therefore, the wavelength of $2^{21} = 2{,}097{,}152$ pm is encoded as the maximum integer value $2^{16} = 65{,}536$.

For amplitude (i. e. sensitivity) calibration, the pairs of wavelength and corresponding coefficient are used, as the sensitivity curve mostly depends not on the optical geometry, but rather on the averaged spectral sensitivity of the three CCD channels. If the projected image moves slightly, only the wavelength

calibration would have to be repeated. The wavelength and the coefficient are calculated similarly as above.

3.3.2 Data preprocessing according to calibration

Once the data are received in the compressed raw form, they are read sequentially to convert each 10 bits to a new integer that can be dealt with easier. For each measured point, the wavelength is calculated with spline interpolation of the calibration points.

Then, the calculated wavelength is input to the second spline interpolation, which calculates the sensitivity coefficient for the given wavelength and divides the measured data by this coefficient.

The spectrum is output to *stdout* with two columns of tab-separated numbers, and usually this output is redirected into a file. This form of communication does not cause much holdup, because the file is stored in the disk cache and possibly does not get written to the disk until it is read by frontend and rewritten by new measurement.

3.3.3 Python module backend

Two versions of the backend were written: one as a standalone binary operated from command line, the second is a python module, which can be loaded dynamically into the running python program. The graphical user interface uses this module.

The functions provided by the backend module are nearly the same, except for not existing equivalent for the `get_spectrum` parameter. The `SetMeasure` function not only starts the measurement, but also receives the spectrum, performs calibration transformation and return it as two lists of wavelenghts and corresponding values.

3.3.4 Graphical user interface

Unlike the firmware and backend, the graphical frontend was written in the *Python* language with *Gtk* widgets. Python is an interpreted language with simple syntax and many advanced features. Its great advantage is the number of additional modules, which facilitate various tasks. Among others, the *numpy* and *scipy* modules were included to perform mathematical operations, *matplotlib* to plot the interactive, antialiased graphs and *gtk* python module to build the graphical user interface.

The *gtk* libraries are a widget toolkit often used to create multiplatform graphical applications. The *gtk* libraries are commonly installed on Linux desktops by default, under Windows, one has to download and install them. Both *python* and *gtk* are released under free licenses compatible with GPL [7], [15]. Thanks to this, all the software equipment of the spectrometer can be distributed without any license costs.

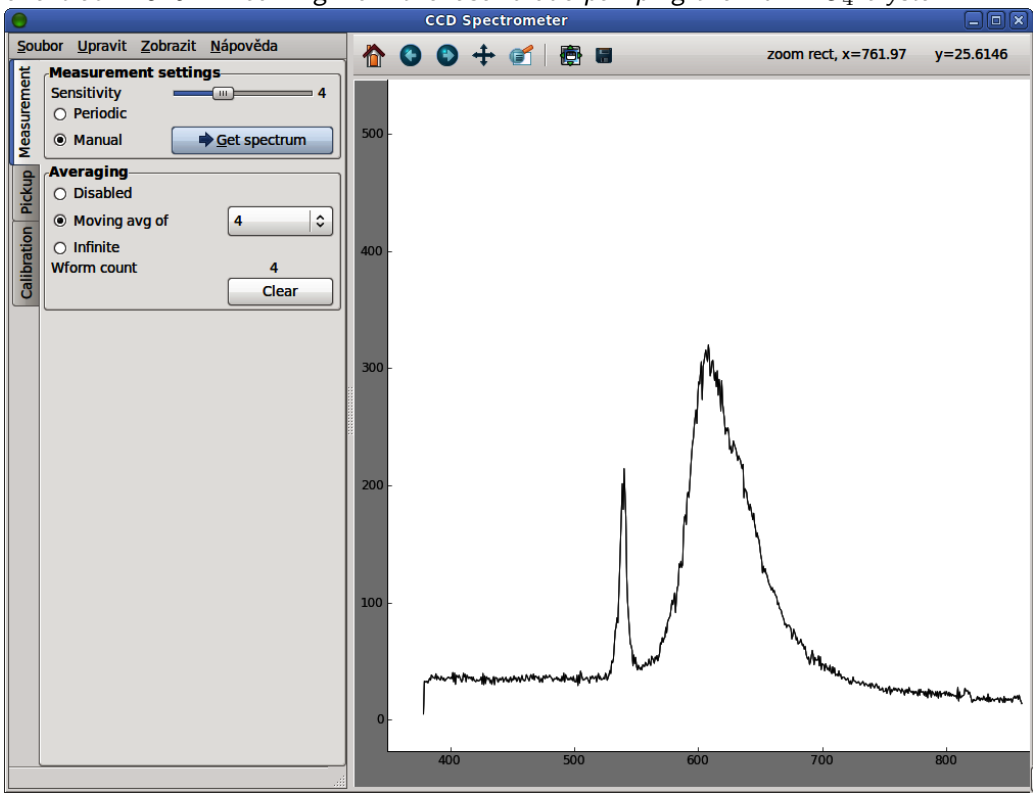
The graphical user interface provides only basic functions as for now (see Fig. 10). It allows to switch between manual spectral measurement and automatic measurement, which runs the spectrometer repeatedly when the previous spectrum was acquired and displayed.

The Sensitivity slider at the top sets the integration time to be two on the power of the value set. The most practical values are between 2 and 5.

Averaging can be set either to the moving average of 2 to 64 waveforms, or to average infinitely to suppress the sampling noise.

There is an option to calibrate the amplitude, which is the most important for measurement of a transmittance $T(\lambda)$ of a sample. The calibration curves are stored in the computer memory. The spectrometer should be recalibrated always when the light source changes. It is recommended that after

Fig. 10: A screenshot of the graphical user interface, showing the orange-red fluorescence of a highlighter pen illuminated by a green laser pointer, whose reflection manifests as the narrow peak at 532 nm. Note also the weak component at ≈ 810 nm coming from the laser diode pumping the Nd:YVO_4 crystal.



several minutes of CCD warm up, the source is turned off and the dark signal calibration curve $I_D(\lambda)$ is set, then the source is turned on again without the sample and the full signal curve $I_F(\lambda)$ is set. Long averaging time improves accuracy. When the amplitude calibration is applied to the measured signal $I_M(\lambda)$, the transmission $T(\lambda)$ is calculated the following way:

$$T(\lambda) = \frac{I_M(\lambda) - I_D(\lambda)}{I_F(\lambda) - I_D(\lambda)},$$

i. e., the dark signal curve I_D is subtracted first, then the remaining extra signal is divided by the source intensity at given wavelength. To suppress noise, an intensive and broadband source, such as focused halogen bulb, is needed.

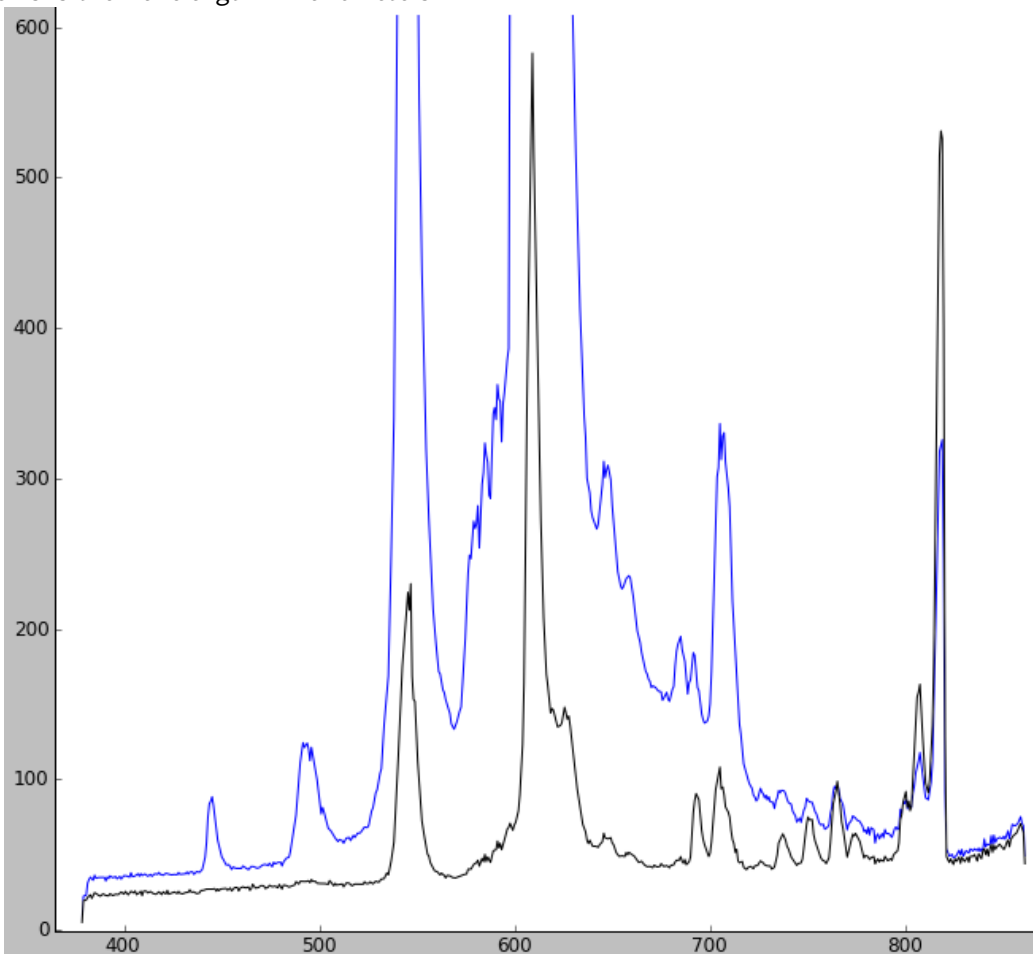
Thanks to the interactive interface of the *matplotlib* package, the graph may be zoomed and panned, as well as easily exported into a bitmap file.

3.3.5 Calibration

The calibration procedure consists of two steps, that shall be done in the following order: the wavelength and the amplitude calibration.

For the wavelength calibration, a common compact fluorescent lamp (CFL) may be used. Such lamp contains diluted inert gas and mercury vapor and at the inner surface, a thin fluorescent layer of europium and terbium salts is sputtered. The output spectrum consists of many spectral lines with

Fig. 11: Emission of a compact fluorescent lamp, 5 s after cold start (black) and ≈ 60 s later (blue). The horizontal axis is the wavelength in nanometers.



different intensity and width, of which the most intensive are listed in the Table 2. The spectrum was first acquired without calibration, then the peaks were easily resolved and corresponding calibration points were saved. As noted above, the backend uses spline interpolation/extrapolation and accordingly even for three distant enough calibration points it was possible to obtain reasonable spectral accuracy.

The amplitude calibration requires a source of a broad and known spectrum. Possible candidates are a halogen bulb or common daylight. Similarly as for the wavelength calibration, several calibration points shall be calculated from the shape of uncalibrated spectrum. The value of each point is obtained as a ratio of the measured intensity to the expected real intensity of the source.

It must be noted that the amplitude calibration does not take into account the linear background. This background depends on CCD temperature and sensitivity. Therefore, the background transforms to a curve that may not be easily visually subtracted without previous observation of its shape for zero input light.

Additionally, it was observed that the spectrometer sensitivity varies by several percent from pixel to pixel. This can be accounted to irregularities of the cylindrical mirror or the CCD detector. Unfortunately, the corresponding compensation curves would not fit into the internal EEPROM of the microcontroller. Both unwanted effects must be compensated in the graphical user interface as the amplitude calibration curves for a particular device and ligth source.

Table 2: List of the CFL spectral lines used for the wavelength calibration [17]

Color	Origin	Peak wavelength [nm]
Dim violet	Mercury	404
Dim blue	Mercury	436
Turquoise	Terbium	485 to 490
Green doublet	Terbium and Mercury	544, 546
Red	Europium	611
Near infrared	Argon	811

4 Results

In this section, several measured spectra will be presented to illustrate the performance and shortcomings of the current design. In the Figure 11, the spectra of a starting 15 W *IKEA* compact fluorescent lamp is shown. This lamp was used to calibrate the spectrometer. The spectrum has been averaged from 16 measurements to suppress the noise and no amplitude calibration was used here.

Initially, the argon filling emits mostly in the near infrared region. The slow start is probably caused by slow evaporation of the mercury filling, which manifests as growth of the spectral lines in visible region. Note that on the contrary, some lines intensity drops during the start.

With longer averaging and proper zoom, several other lines may be resolved, such as the weak violet mercury line at 406 nm.

Both the blue mercury line at 436 nm and the infrared argon line at 811 nm are emitted by diluted gas and are therefore narrow enough to assess the spectral resolution. The spectrometer measured their width at half maximum as 5 to 8 nm, which can be considered to be the maximum spectral resolution of the device. However, with proper zoom, the green peak at the 544 to 546 nm has a shallow, yet stable notch in its center. This may be accounted either to the nonuniformity of CCD response, or to the fine resolution of spectra. (Both is possible as the convolution function is probably not gaussian, but has a sharp peak instead.)

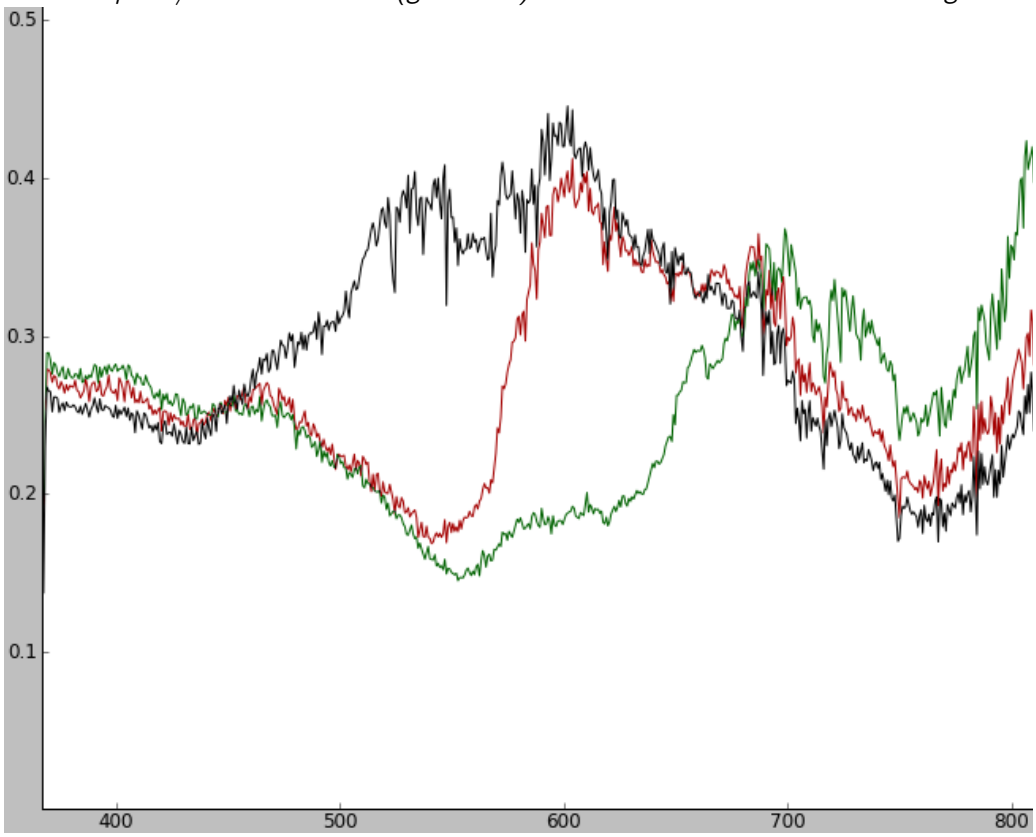
The Figure 12 depicts three spectral curves measured by passing an incandescent bulb light through a 10 mm long cuvette with pure water and two different concentrations of potassium permanganate ($KMnO_4$). In low concentration, the very strong absorption in the green spectral region causes the pink tint of the solution. In higher concentration, broader spectral region from 450 to 650 nm gets absorbed. This explains the visible hue difference of different concentrations. The amplitude growth in unabsorbed regions is caused by the cuvette displacement.

In this measurement, the amplitude calibration did not correct the pixel-to-pixel nonuniformity, although the spectrum was averaged from 16 measurements again. For practical application, the proper calibration for the particular source and fixed cuvette position is important.

5 Conclusion

In the above sections, a concept of a cheap, compact digital spectrometer was presented. If a CCD detector from an old scanner is used, the components cost less than 20 Euro. (A new suitable CCD can be obtained for under 10 Euro if a bigger package is requested.) The work per each next piece can be expected to be less than 5 hours, since the first prototype is tested and both the software and firmware are ready.

Fig. 12: Transmission of pure water (black line); of water with ≈ 10 μmol/l solution of $KMnO_4$ (red line) and of water with ≈ 100 μmol/l solution thereof (green line). The horizontal axis is the wavelength in nanometers.



In the Results section it was shown that such device works with sensitivity and resolution acceptable for many tasks both in education and research. The graphical user interface makes the device relatively easy to work with.

The main conclusion of this work is that it *is* possible to build a relatively usable, cheap and portable digital spectrometer.

6 Appendix

A CD with the software and source code is attached to this project. The graphical user interface can be found as `cdrom:///gui/CCD_spectrometer_GUI.py`. See the `cdrom:///gui/README.txt` file for installation notes and more information.

The spectrometer firmware source codes reside in `cdrom:///firmware/` and the source of the command line backend in `cdrom:///commandline/`.

A copy of this PDF is included on the CD.

Fig. 13: Front view of the assembled spectrometer

References

[1] Amit Bhagwat and Ankur Kumar. CCDImager – the use of the Atmel Mega32 microcontroller for imaging, 2007. URL <http://courses.cit.cornell.edu/ee476/FinalProjects/s2007/arb66_ak364/arb66_ak364/Index.html>.

[2] Igor Češko. Implementation USB into microcontroller: Igorplug-USB. URL <http://www.cesko.host.sk/IgorPlugUSB/IgorPlug-USB\%20(AVR)_eng.htm>.

[3] Atmel Corporation. ATmega8(L) data sheet, 2003. URL <www.datasheetarchive.com>.

[4] NEC Corporation. μPD3799 data sheet, 1999. URL <www.datasheetarchive.com>.

[5] Kamil Eckhardt. USB - univerzální sériová sběrnice, 2002. URL <http://home.zcu.cz/~eckhardt/popis.html>.

[6] Filip Dominec et al. Programujeme jednočipy, czech textbook at Wikibooks, 2009. URL <http://cs.wikibooks.org/wiki/Programujeme_jedno\%C4\%8Dipy>.

[7] Python Software Foundation. Python license, 2010. URL <http://www.python.org/psf/license/>.

[8] Edmund Optics Inc. Reflective ruled diffraction gratings, 2010. URL <http://www.edmundoptics.com/onlinecatalog/displayproduct.cfm?productid=1896>.

[9] Future Technology Devices Intl. Ltd. FT232BL datasheet, 2005. URL <http://www.datasheetarchive.com/>.

[10] Brian Manning. A grating ruling engine. *Scientific American*, 4:232, April 1975.

[11] Courtney Peterson. How It Works: The Charged-Coupled Device, or CCD, 2001. URL <http://www.jyi.org/volumes/volume3/issue1/features/peterson.html>.

[12] Jan Smrž. Implementace USB rozhraní AVR mikrokontrolérem. Master thesis, ČVUT, Praha, 2008. URL <http://smrz.chrudim.cz/>.

[13] Christian Starkjohann. Virtual USB port for AVR microcontrollers, 2010. URL <http://www.obdev.at/products/vusb/index.html>.

[14] Paul Stoffregen. CCD Array Reader Project, 1992. URL <http://www.pjrc.com/tech/ccd/>.

[15] The GTK+ Team. The gtk+ license, 2008. URL <http://www.gtk.org/>.

[16] The USB Implementers Forum Inc. (USB-IF). Universal Serial Bus Revision 2.0 specification, 2010. URL <http://www.usb.org/developers/docs/usb_20_052510.zip>.

[17] Wikipedia user Deglr6328. Fluorescent lighting spectrum, 2009. URL <http://en.wikipedia.org/wiki/File:Fluorescent_lighting_spectrum_peaks_labelled.png>.