University of Applied Sciences Northwestern Switzerland
School of Engineering

Using Android in Industrial Automation

Technical Report

A bachelor thesis at the

University of Applied Sciences Northwestern Switzerland for the Institute of Automation

Authors

Manuel Di Cerbo
Andreas Rudolf

Project principal

Matthias Meier

Acknowledgements

We owe our deepest gratitude to Matthias Meier who enabled us to create this project. With his deep knowledge in embedded system software and hardware design he was a major factor to the success of our work. If it was not for him, we would have never been able to progress to the point at which we are standing now.

We also want to thank Christoph Biel and Stefan Muhr who helped us during the process of creating our hardware layout and accompanied us through the process of print-ordering.

We especially want to thank our colleague Thomas “steady-hand” Schaub for soldering the tiny CPLD chip onto our custom front-end.

Abstract

The Android mobile operating system is constantly attaining more interest of developers throughout the globe. In this project we aim to further understand the Android platform and create a fast and powerful open source oscilloscope. An application that requires rapid data acquisition, hardware development, micro controller firmware and an Android application. Designing and building our own analogue to digital front-end we further develop our knowledge of the USB interface, CPLD firmware and system modeling. Using the Beagleboard OMAP 3530 arm platform as a basis for the Android operating system we benefit from experiences we have been gathering in the preceding project "Using Android in Industrial Automation: An Android Spectrum Analyzer". At the end of the semester we were able to successfully operate our self made open source Android oscilloscope which runs on an embedded system.

Contents

Part I

Introduction

Chapter 1

Assignment of Tasks

During the past semester the project team has gathered their first experiences with Android and real time data acquisition by building an Android Spectrum Analyzer. As a result the usability of Android in an industrial environment has been demonstrated.

As a goal for the project in this semester data throughput and real time features are to be further developed and hardened. An Android oscilloscope is to be designed. It should preferably be ready for distribution in the "Android-Market". As a further goal a sampling hardware is to be designed and built.

Specifications of the Application to be built:

- Dual channel data acquisition using 8 bit A/D Converters per channel. Variable sampling rate (range 1[Hz] - 40[MHz]). Signal input voltage of +/- 5[V]
- Continuous sampling rate around 1 [Msps].
- High resolution single shot sampling with 40 [Msps] resolution, using hardware triggering. Using only around 2k samples per channel.
- UI with common oscilloscope cursor and trigger functionality.
- Running on the Beagleboard or Devkit8000.
- Product specifications of Hardware and Software
- Documentation of the application, hardware and software

Chapter 2

Coverage of the project

2.1 Android Platform Setup

To create the desired project the team has to set up an appropriate Android Platform using either the Beagleboard or the Devkit8000. The team decided to use the Beagleboard since it was able to gather experience with the mentioned hardware platform in the preceding project ("Using Android In Industrial Automation: An Android Spectrum Analyzer"[1]. The setup consists of Kernel and Android Root File System configuration and compilation. Planing to use the USB interface there is also the need to compile the following tools for Android:

- libusb (USB user space library)
- lsusb (Command Line Interface tool to list usb devices)
- fxload (Tool to upload firmware on the FX2-USB Micro Controller)

The team decided to use the Kernel and Root File System of the "Rowboat" project[2] after evaluation. Key decision factor was the uncomplicated setup and compatibility of the kernel for the Beagleboard. Furthermore there is a support module in the "Rowboat" project for Open GL graphics acceleration.

The project team used the development environment of the preceding project and therefore was able to start implementation rapidly. For comprehension of how to set up helpful tools please refer to the preceding technical report "Using Android in Industrial Automation: An Android Spectrum Analyzer".

2.2 Android Software

Using the Android application framework the team has to develop a software oscilloscope program that offers nearly as much features as found on a generic hardware oscilloscopes. The application consists of an Android Activity that lets the user interact with the application and displays collected data samples tapped by a hardware front-end.

The team decided to once again use an Activity-Service approach as demonstrated in the preceding project. Thereby, the Activity is dealing with user input and graphical output, and the Service is handling and controlling real time data acquisition.

Furthermore, the Android Application has to make use of the USB host interface which requires a native program part written in C or C++, since there is no Android API that offers USB host mode access to attached devices.

[1] technical report available on http://android.serverbox.ch & http://web.fhnw.ch/technik/projekte/eit/Fruehling2010/DiCerRud/
[2] http://code.google.com/p/rowboat/

As a goal, the software should not “loose” collected samples and consider mechanics to enable “real time” acquisition. Also as a requirement the application needs to offer a “continuous” mode that constantly updates samples to the screen and a “single shot mode” that uses a hardware trigger and displays high resolution signal parts (sampling at 48 [MHz]).

A wish of the team was to be able to also use audio from a common mobile phone as data source. This lead to the idea of creating a developer API that enables any custom additional data source implementation (Audio, Usb, Bluetooth, Network Stream, File Stream, ...). Therefore, the oscilloscope framework can be universally used.

2.3 USB connectivity

To achieve the mentioned sampling rates the team decided to use USB as an interface between the custom front-end and the Android application. With a practical maximum throughput of 40 [MByte/s] the application is guaranteed to provide a reasonable data collection rate. The team has gathered experience with the Cypress FX2 USB micro controller during the preceding project and decided to use the same controller for this application. However, there are key differences to the spectrum analyzer application. Data acquisition rate in this project is 1000 times higher than in the previous application. The project team also decided to use the slave FIFO feature of the FX2 micro controller to be able to collect data quickly and have a simple interface towards a digital data source as for example an FPGA or CPLD.

2.4 Hardware

A custom printed circuit board for signal acquisition - the oscione board - has been developed in this project. The oscione board is USB-powered and consists of both an analogue and a digital front-end. In the analogue front-end, measured signals are adapted with variable amplifications so that they can be sampled with analogue-to-digital converters. Once the signals are sampled, they are passed to a CPLD, which acts a master for the FIFO queue of the FX2 USB micro controller.

2.5 Putting it all together

As you can seen in Illustration [2.1] we have decided to create our own analogue and digital front-end on one single board. Using a CPLD we send the digitalized data samples to a Cypress FX2 micro controller. We use USB to transmit collected data samples and the I2C interface to send setup data to the CPLD.

Using the same hardware platform (the Beagleboard) as in the preceding project we are running Android OS with our application built on top of it. The user sees the displayed application on a TFT screen and interacts with the application using a mouse.

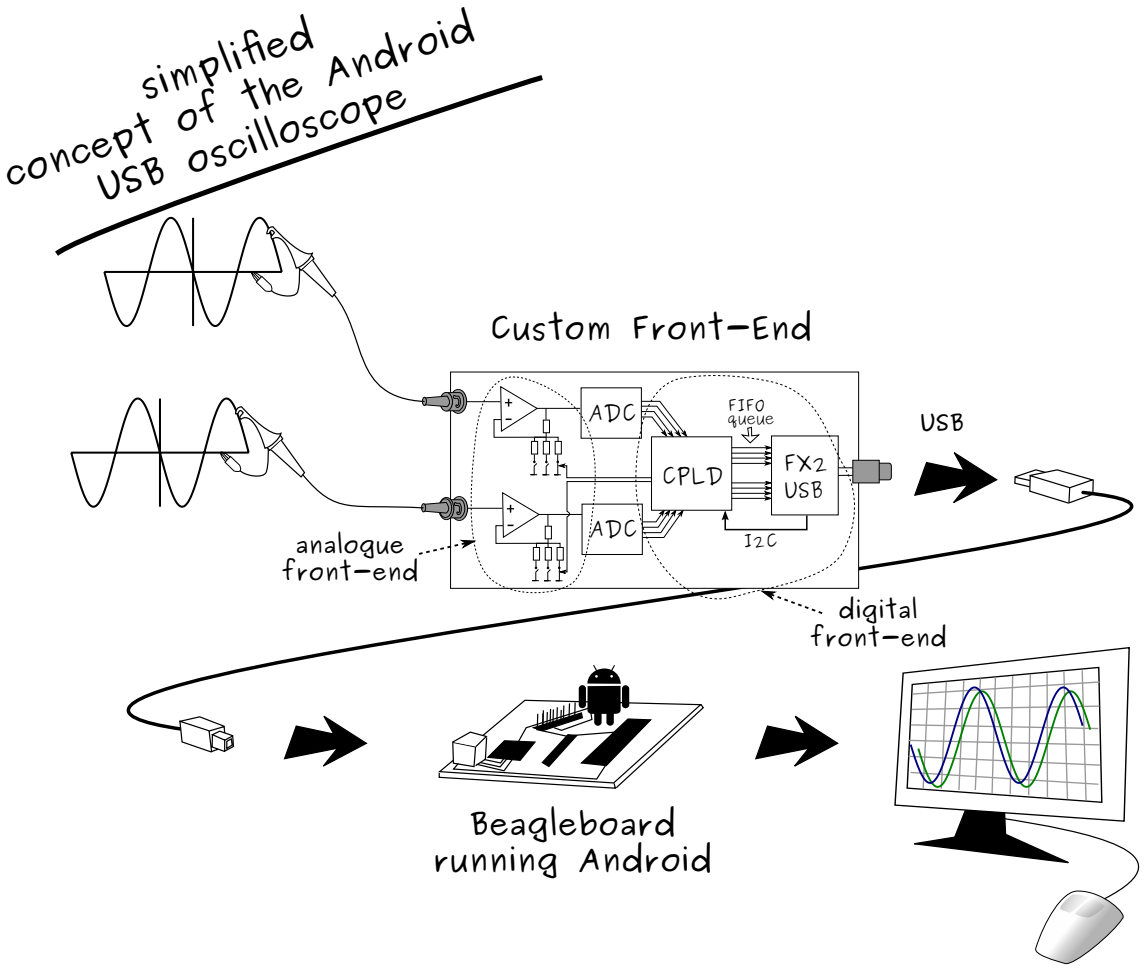

Figure 2.1: Simplified concept of the Android USB oscilloscope

Chapter 3

Result

3.1 Software

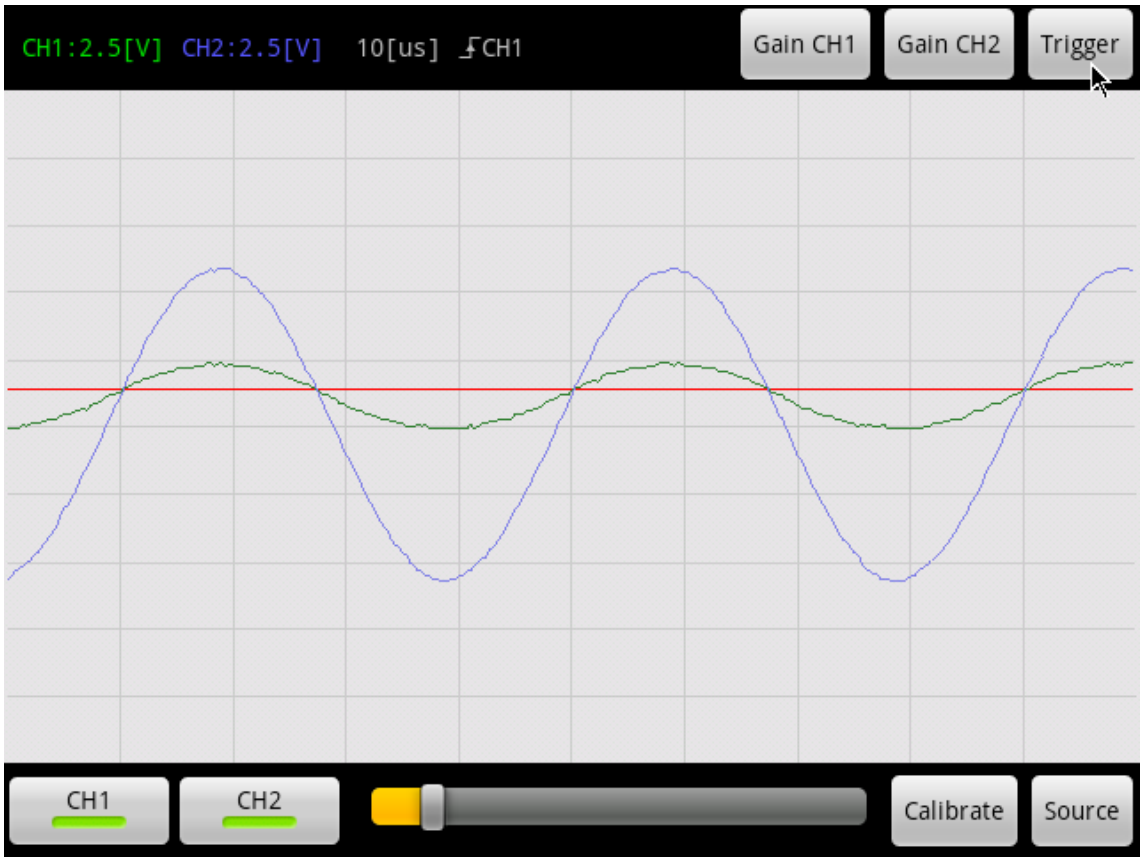

Figure 3.1: UI of the oscilloscope application

Features

- Oscilloscope User Interface
- Continuous mode
  - Max Resolution: 5[us] / division
  - Min Resolution: 2[ms] / division
  - 30 frames per second

- Single Shot mode
  - Max Resolution: 625[ns] / division
  - Min Resolution: 1.875[us]
- Two Channel Display
- Trigger
  - adjustable using the mouse or a touch screen
  - edge rising/falling, source channel 1 / channel 2
- Source Selection (USB, Audio, Generator)
- Hardware I/O (Gain control, Trigger control)

3.2 Hardware

- two separate channels
- designed for 10x oscilloscope probes
- AC/DC coupling
- 8 Bit analogue-to-digital conversion
- continuous sampling at a frequency of 6Mhz
- maximum sampling frequency of 48Mhz
- hardware trigger detection for single-shot mode
- Adjustable gain with 5 discrete values
- Analogue Bandwidth (3dB): minimum 3.3Mhz (depending on gain, see hardware specification [8])
- Maximum Input Voltage: 16V
- 5kHz square wave output for probe compensation
- USB-powered
- power consumption: $\approx$ 1W (see hardware specification [8])

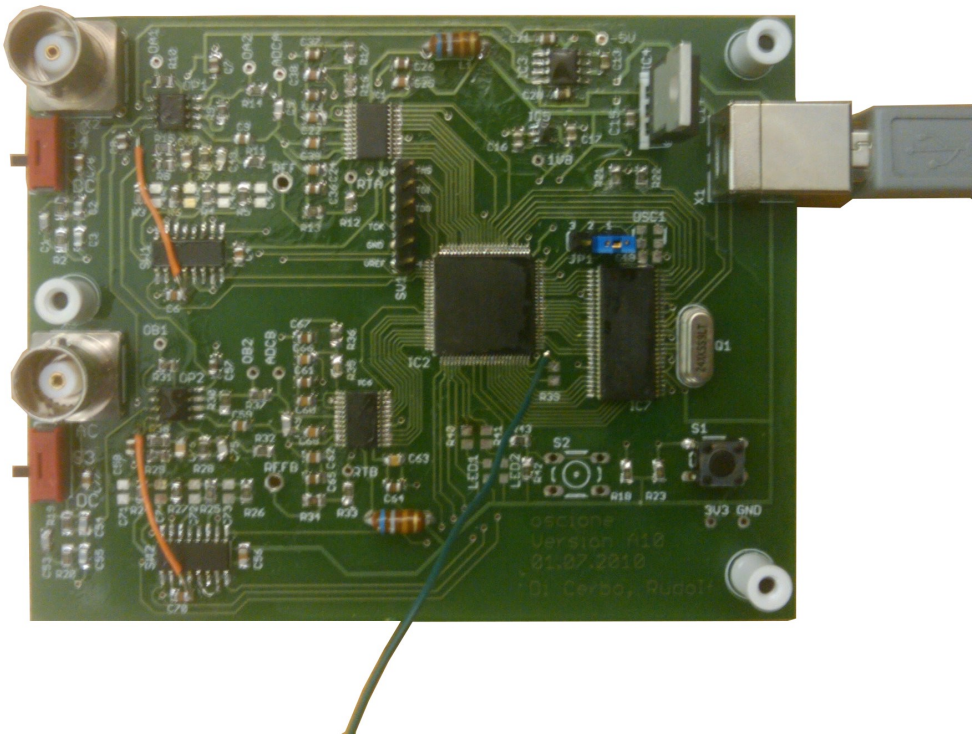

Figure 3.2: picture of the oscione board

Part II

Oscione Hardware

Chapter 4

Overview

The custom front-end - the oscione board - built during this project consists of both an analogue and a digital part. Illustration [4.1] provides a quick overview of the oscione board concept. Both the analogue front-end and digital front-end are discussed in detail in the following chapters.

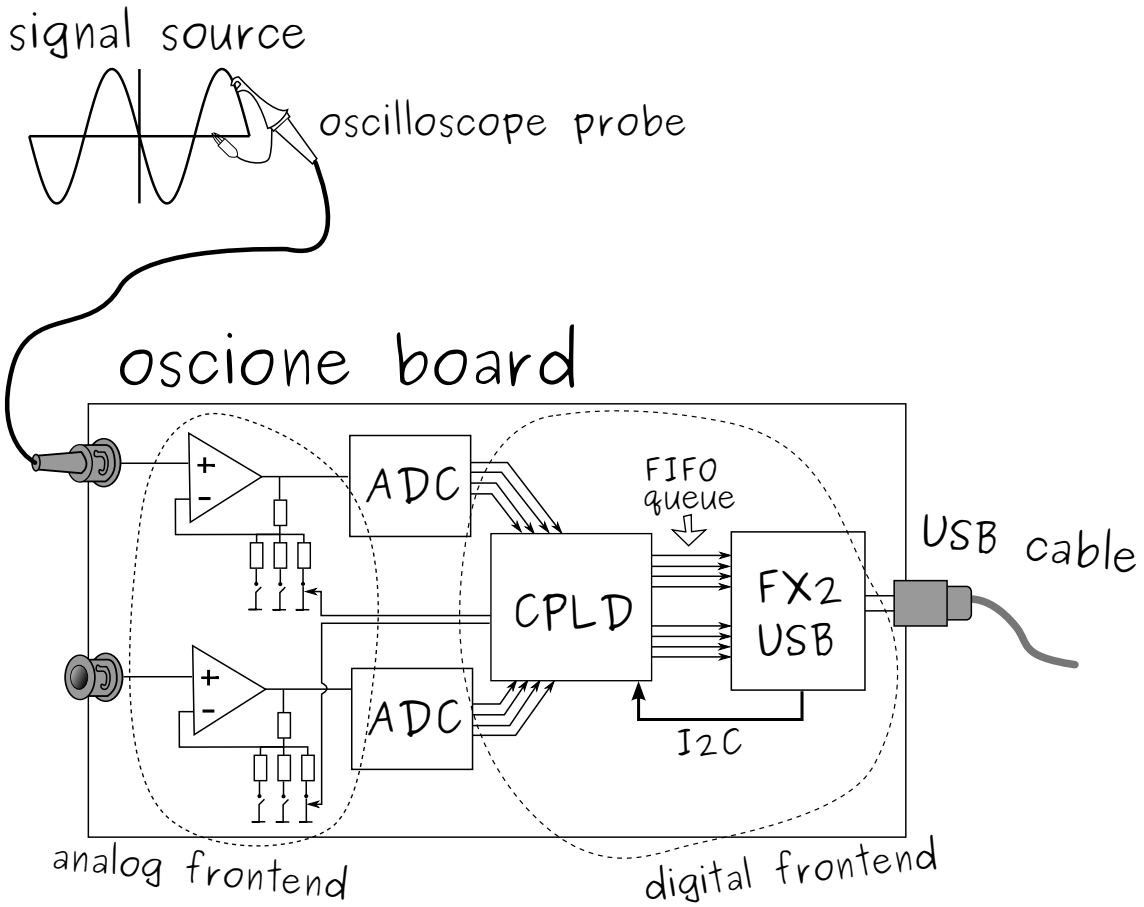

Figure 4.1: Overview of the oscione board

Chapter 5

Analogue front-end

5.1 Signal path

The analog front-end begins at the oscilloscope probe tip and ends at the input of the analog-to-digital converter (ADC). Therefore, signals of interest must be adapted through the analog front-end such that they can be optimally sampled at the ADC. We are using a unipolar ADC, meaning that it can only handle positive voltages. It goes without saying that we also want to measure bipolar AC-signals. Thus, the signals need to be level shifted at some point. But before going into details, have a look at figure [5.1] on the next page, which shows a brief summary of the complete signal path of the analog front-end. An explanation on this illustration is given in the following table [5.1].

1. Shows the original signal.
2. Our oscilloscope is designed to use 10x probes. Therefore the signal is always attenuated by a factor of 10.
3. Before amplifying the signal it is further attenuated by a factor of two. This will allow us to measure higher voltages.
4. At the first operational amplifier stage, the signal is either passed on and left without amplification as in a voltage follower circuit, or it is being amplified in a non-inverting amplifier circuit. Since the full scale input of our ADC is configured to be 2V, the amplified signal should not exceed 2V peak-to-peak.
5. In the second and last operational amplifier stage, the signal is being inverted and level shifted by +1V. The inversion has no practical use whatsoever! This inversion needs to be accounted for later on in the digital front-end (which is quite easy since you can simply negate the output of the ADC).
6. The adapted signal is then sampled at the input of the 8 Bit ADC. As already stated, the full scale input is 2V. Meaning that all inputs of 2V and above will result in a data output of 0xFF. Likewise, inputs of 0V and below result in 0x00. Voltages between 0V and 2V are linearly sampled into unsigned byte data representation.

Table 5.1: Explanation on signal path in figure [5.1]

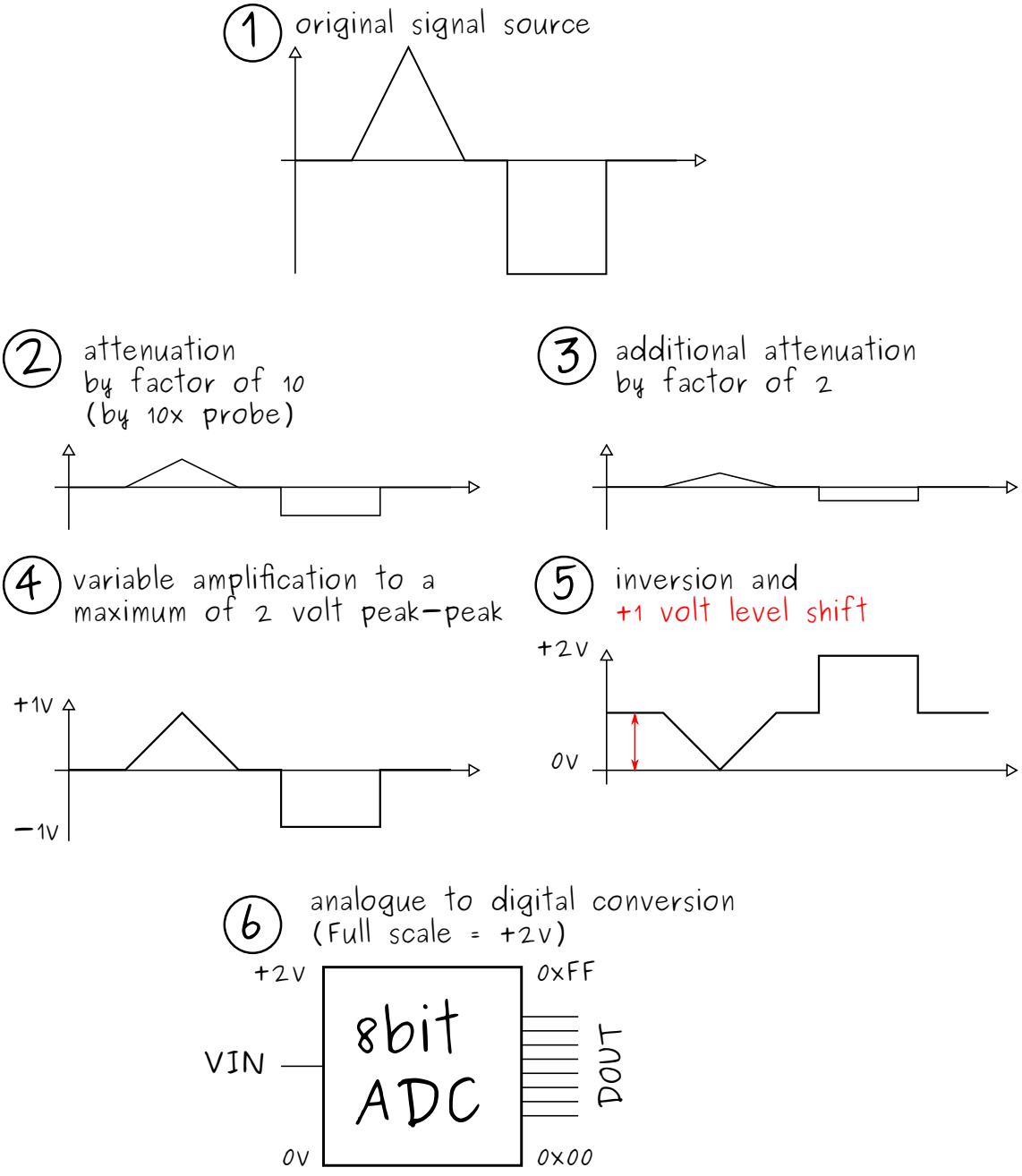

Figure 5.1: Signal path of the analogue front-end

5.2 Analysis

Now that we have a basic understanding of the signal path, let us see how this can be implemented in real hardware. For this purpose, have a look at figure [5.2], which shows a simulation schematic from LTspice [1]. If you are familiar with electrical circuits, this schematic will be pretty self-explanatory. Nevertheless, we will provide you with a quick explanation in the next paragraph, as well as in-depth analysis in the following sub-sections.

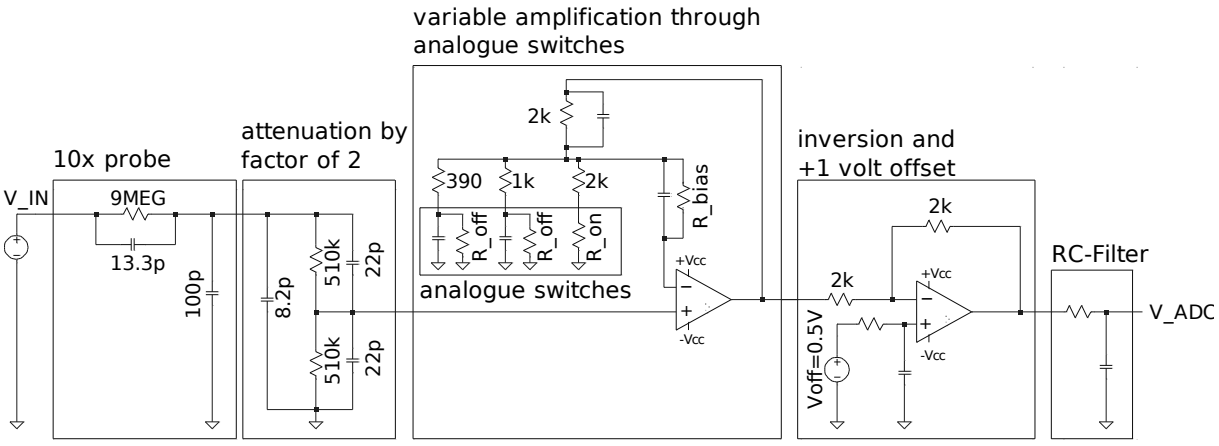

Figure 5.2: Simulation schematic of the analogue front-end

Again, have a look at figure [5.2]. In the first rectangle box to the left, we see the modeling of a 10x oscilloscope probe. The signal is then attenuated by a factor of two using a resistive and capacitive voltage divider. The following operational amplifier stage is basically a non-inverting amplifier circuit. Distinct analogue switches to ground are either open or closed, and thereby set the gain of the operational amplifier stage. Using an inverting operational amplifier circuit, the signal is being inverted and at the same time level shifted by +1V. At the end, there is an (optional) RC-Filter in front of the ADC input.

5.2.1 PROBES

When measuring electrical signals you will want to make sure not to alter the signal of interest by introducing additional load through your measurement setup. In other words, the input resistance of the measuring device should be chosen very high. Commonly used 10x oscilloscope probes feature a 9 mega ohm resistance in the probe tip. Together with an oscilloscope input resistance of 1 mega ohm this results in a total of 10 mega ohm input resistance. In a first approach, we can model this setup as a simple voltage divider as in figure [5.3]; the resistance of the coaxial cable is negligible. Note that only one tenth

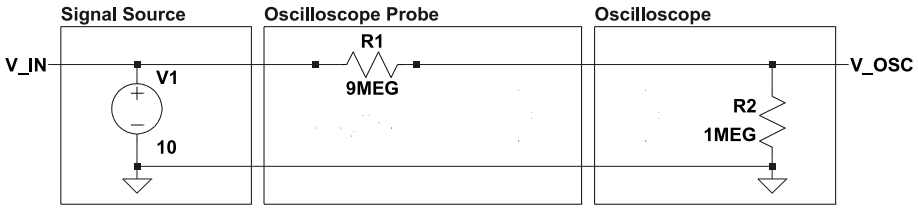

Figure 5.3: model of a 10x oscilloscope probe (1st approach)

of the signal amplitude is available at the oscilloscope input, hence the name 10x probe. However, the probe's coaxial cable also has a parasitic capacitance from signal to ground. This capacitance varies

[1] LTspice is a free spice simulator and runs fine under wine

from probe to probe. You may assume around 20-30pF capacitance per foot of cable. With a 4 foot cable this adds up to about 100pF. Inserting this parasitic capacitance in our model results in [5.4]. Now, if

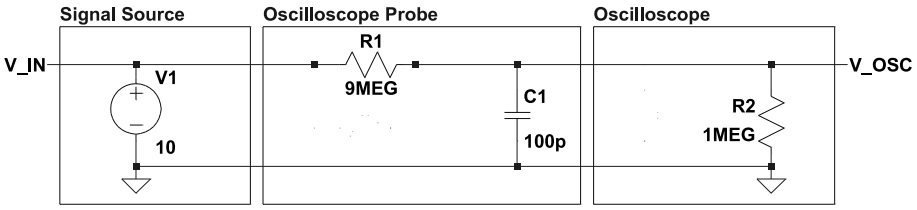

Figure 5.4: model of a 10x oscilloscope probe (2nd approach)

we only were to measure DC-signals, this capacitance would have no effect. But as we measure AC-signals as well, this capacitance acts as a low pass filter. The effects can best be seen when measuring a square wave as in [5.5]. To counter this low pass behavior, an oscilloscope probe includes an additional

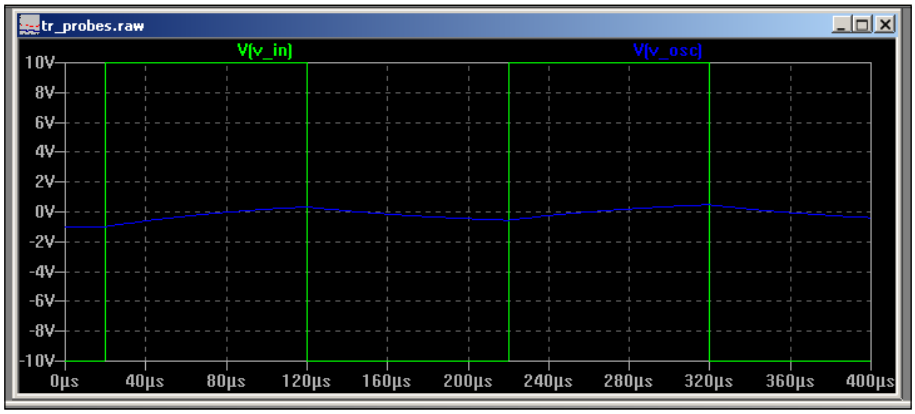

Figure 5.5: square wave input simulation of a 10x probe (2nd approach)

variable capacitor in parallel to the 9 mega ohm resistance in the probe tip, acting as a high pass filter. As we already mentioned, an oscilloscope has an input resistance of 1 mega ohm; furthermore, most oscilloscopes have a defined input capacitance of 20pF. Together with the 100pF parasitic capacitance this will add up to 120pF of total input capacitance. Thus, the variable capacitor in the probe tip must be tuned to cancel the effects of 120pF capacitance at the oscilloscope input. In our simulation the variable capacitor must therefore be tuned to $\frac{120}{9}$pF $= 13.3$pF. Figure [5.6] and [5.7] show the full model of a probe and oscilloscope input, with a simulated square wave. Of course, when practically using an

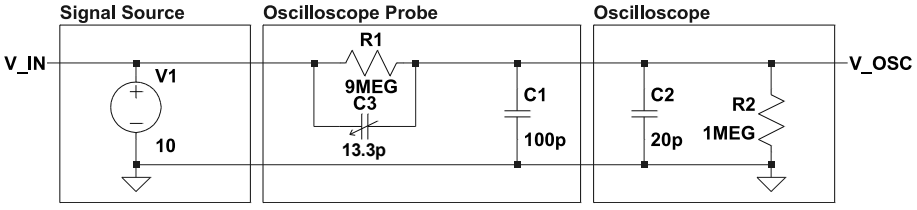

Figure 5.6: model of a 10x oscilloscope probe (final approach)

oscilloscope, one would never calculate the needed capacitance in the probe tip. Instead, you would measure a square wave and adjust the variable capacitor until the measured signal output looks like a square wave (as good as it gets). This procedure is called compensation.

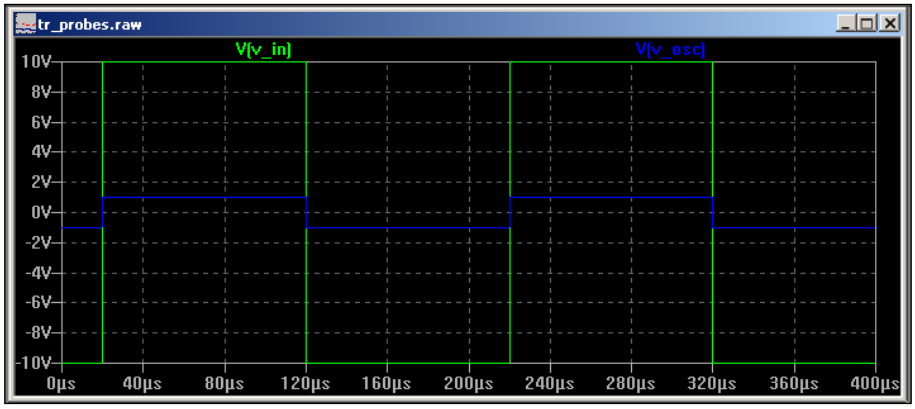

Figure 5.7: square wave input simulation of a 10x probe (final approach)

5.2.2 ATTENUATION BY 2

A resistive and capacitive voltage divider in parallel divide the signal by a factor of two (second box to the left in figure [5.2]). There really is not much to say here, except that you have to make sure that the total input impedance is 1 Megaohm in parallel to 20pF. The resistance seen at the input is the series of both 510k resistors, adding up to roughly 1 Megaohm. The effects of the following operational amplifier input resistance is negligible, since most operational amplifiers have very high input resistances at about 100 Megaohm or even more. The total input capacitance of this circuit is the series of both 22pF capacitors on the right in parallel to the 8.2pF capacitor on the left hand side. This will add up to 19.2pF input capacitance. However, you might have to take into account the input capacitance of the following operational amplifier. In our case, it is only 0.5pF, which is why we neglect it. If you choose an operational amplifier with higher input capacitance you could replace the bottom right capacitor with a variable capacitor, and fine tune it's value.

5.2.3 VARIABLE AMPLIFICATION

A non-inverting operational amplifier circuit, as seen in figure [5.8], will satisfy the following equation.

$$V_{out} = \left(1 + \frac{R1}{R2}\right) * V_{in}$$

How can this be explained? Simply put, an operational amplifier with feedback (that is, any signal path

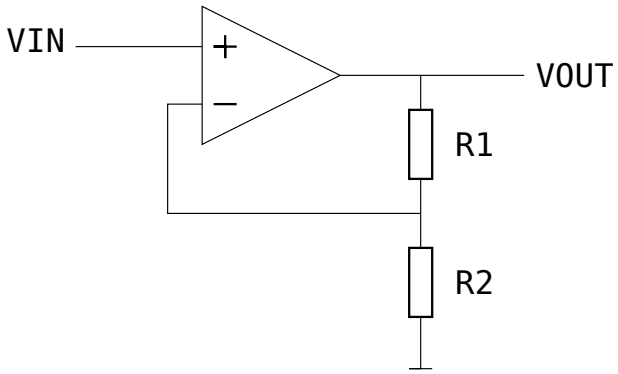

Figure 5.8: general non-inverting amplifier circuit

from output back to any inputs) will always try to set both it's positive and negative inputs to the same

voltage level. Since we have Vin at the positive input, the opamp will steer it's output so that in the end the negative input will be at the same voltage level. This is the case, when the mid-voltage between both resistors R1 and R2 is Vin. Since R1 and R2 form a voltage divider, the following equation must hold

$$\frac{R2}{R1+R2} * V_{out} = V_{in}$$

$$R2 * V_{out} = (R1+R2) * V_{in}$$

$$V_{out} = \left(\frac{R1}{R2}+1\right) * V_{in} = \left(1+\frac{R1}{R2}\right) * V_{in}$$

To get discrete variable gains, we use multiple resistors in parallel with analogue switches to ground as in figure [5.9]. An optimal analogue switch would have infinite resistance when opened, zero resistance

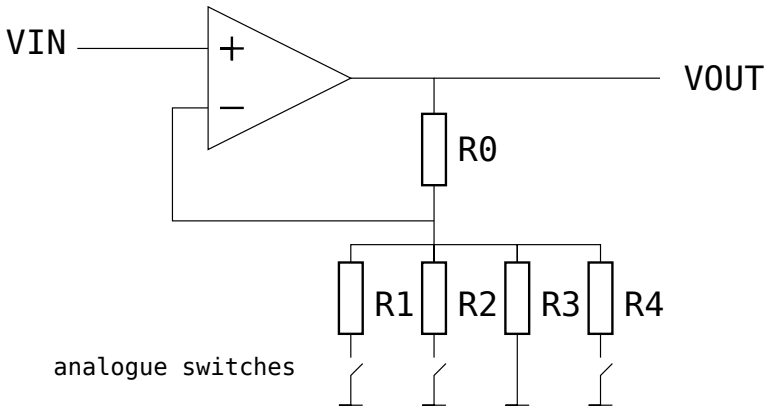

Figure 5.9: amplifier stage with analogue switches

when closed, and no parasitic capacitance at all. Real analogue switches do have very high resistance when opened; however, when closed, the switch resistance Ron is not negligible and usually about 100 Ohm. Furthermore, they all have parasitic capacitances which must be taken into account when modeling the analogue front-end.

In our implementation, as mentioned earlier, all analogue switches have one terminal to ground. An advantage of this implementation is that when all four switches are opened, the operational amplifier will act as a voltage follower, and therefore pass the signal without amplification. Closing any of the four switches results in four distinct gains. So all in all, we have five different gain values. A disadvantage of this implementation is, that the on-resistance Ron of a closed switch will have an effect on the effective gain. Furthermore, the parasitic capacitance of the analogue switches does have an impact on the signals, especially when used as a voltage follower as in figure [5.10]. For simplicity, we only look at one

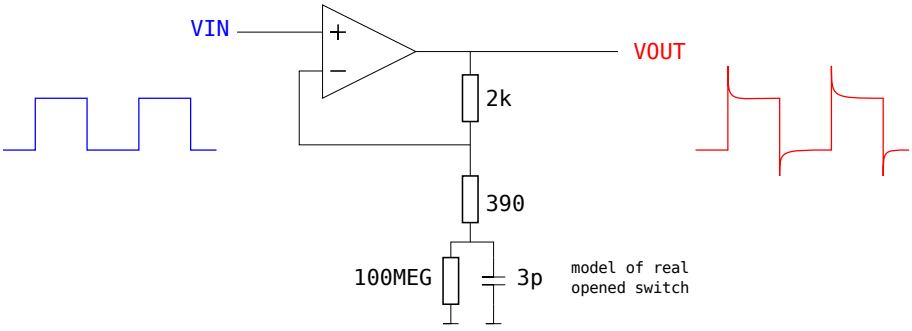

Figure 5.10: amplifier stage with a modeled open switch

analogue switch. The switch is modeled as opened, with a resistance of 100 Megaohm and a parasitic capacitance of 3pF. With a square wave input signal, the output signal is again a square wave of the same magnitude, but with remarkable overshot peaks. This overshot stems from the parasitic capacitance of the analog switch. When the square wave input toggles from low to high (and vice versa), it contains very high frequency components. At high frequencies however, the parasitic capacitance is almost a short circuit. Therefore, high frequency components are amplified as in a non-inverting amplifier circuit. To cancel the effects of the parasitic capacitance, one can insert an additional capacitor as in figure [5.11]. A

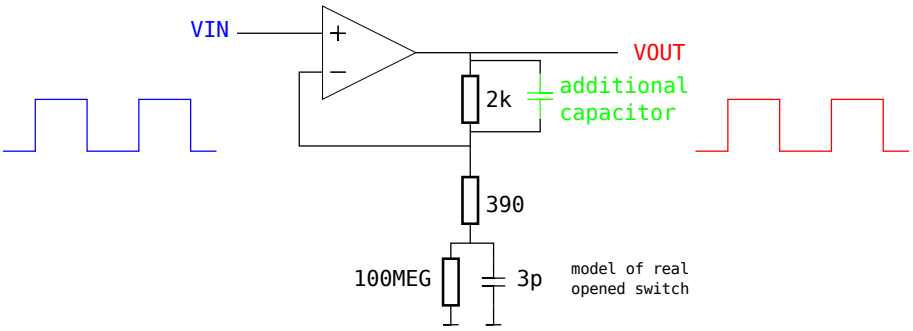

Figure 5.11: reduction of overshot spikes

value of 100pF will drastically reduce the overshot phenomenon. Then again, this additional capacitance will also reduce the bandwidth of the analog front-end, even with an active gain selected, as can be seen in figure [5.12]. To overcome this effect, yet another capacitor in parallel to the lower resistors could

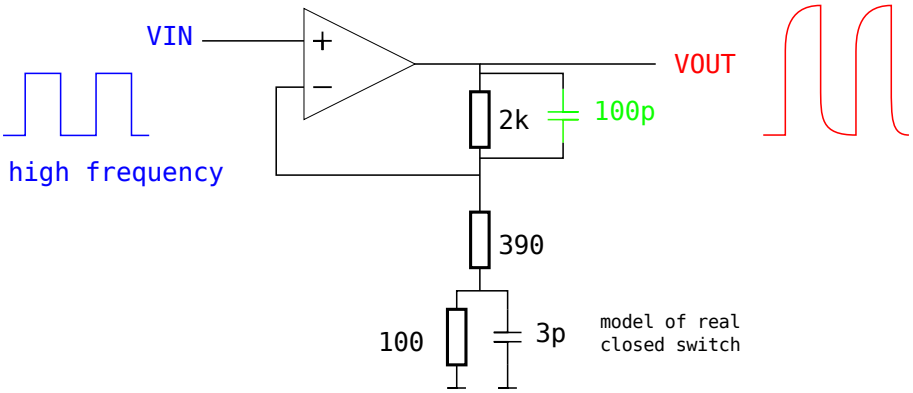

Figure 5.12: reduced bandwidth problem

be placed. This works well when simulating, and is qualitatively sketched in figure [5.13]. But as we tried this workaround with real hardware on the oscione board, these parallel capacitors did not show any effects at all. It needs to be said that the overshot phenomenon only occurs when the operational amplifier acts as a voltage follower and also to a lesser extent for small amplifications. Thus, we try to find a trade-off between overshot reduction for low gains and overall analogue bandwidth by placing an additional capacitor in parallel to the upper resistor.

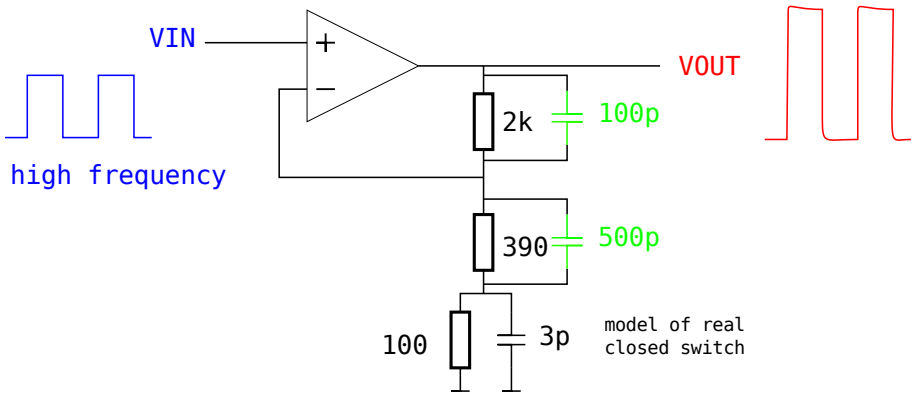

Figure 5.13: theoretical solution to overshot peaks and bandwidth reduction

Alternative solution: Instead of connecting one terminal of any analogue switch to ground, these switches could also be placed as in figure [5.14]. With this implementation, the on-resistance R_{on} of

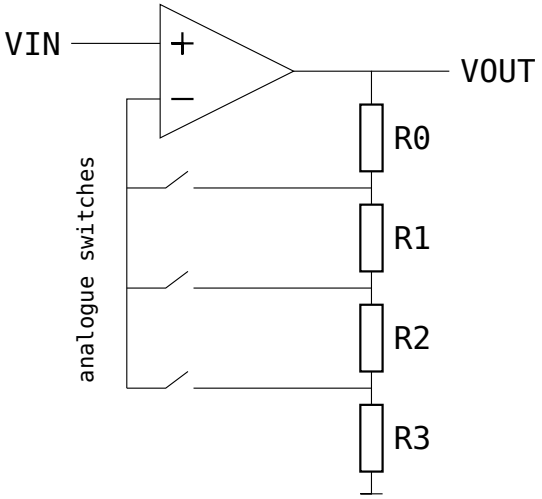

Figure 5.14: alternative switch placement

a closed switch would have no impact on the effective gain. Also, when simulating this scenario, no overshot spikes in a square wave occur. Therefore, we think that this implementation might be the better choice. However, to really make a profound decision, this setup would need to be tested in real hardware. Unfortunately, at the time we realized this probable solution, it was already too late in the project development.

5.2.4 INVERSION AND +1 VOLT OFFSET

An inverting operational amplifier circuit as in figure [5.15] will satisfy the following equation:

$$V_{out} = -\frac{R_2}{R_1} * V_{in}$$

When setting $R1 = R2$, the output is simply the inverse of the input voltage. The voltage shift is

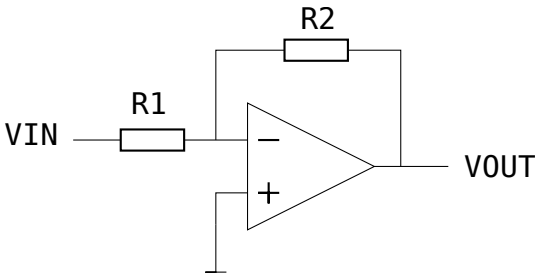

Figure 5.15: general inverting amplifier circuit

achieved by connecting the positive terminal to a certain voltage as in figure [5.16]. This circuit has two different voltage sources, namely V_{in} and V_{off}. Applying the concept of superposition, we first set V_{off} to zero, and determine the contribution of V_{in} to the output V_{out}. When V_{off} is zero, the circuit is the exact same as in figure [5.15]. And therefore

$$V_{out1} = -\frac{R2}{R1} * V_{in} \overset{R1=R2}{=} -V_{in}$$

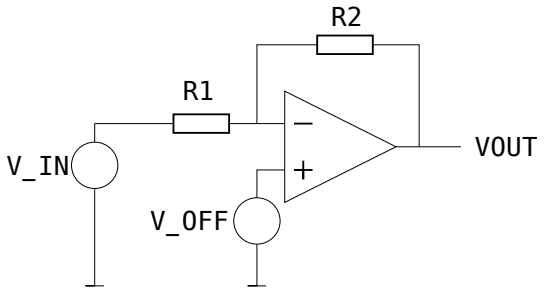

Figure 5.16: inversion and offset generating circuit

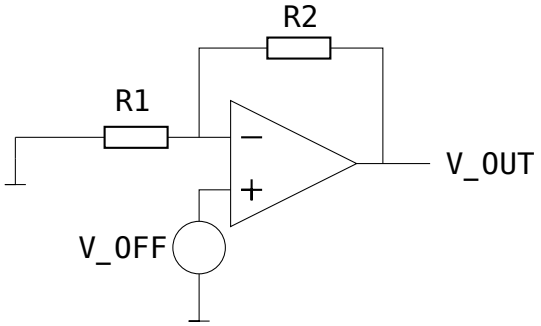

Figure 5.17: contribution of V_{off} to V_{out}

Now we set V_{in} to zero and determine the contribution of V_{off} to the output V_{out}. With V_{in} shorted to ground (figure [5.17]), the circuit becomes a non-inverting operational amplifier circuit, as explained in the last subchapter. Thus, the contribution of V_{off} to V_{out} is

$$V_{out2} = \left(1 + \frac{R2}{R1}\right) * V_{off} \overset{R1=R2}{=} 2 * V_{off}$$

Adding both contributions together results in

$$V_{out} = V_{out1} + V_{out2} = -V_{in} + 2 * V_{off}$$

To achieve a voltage shift by +1[V], V_{off} must be set to 0.5[V].

5.2.5 RC-FILTER

The RC-Filter at the end of the analogue front-end is not absolutely mandatory. The RC-Filter acts as a low-pass and helps to reduce the overshot spikes as described in the subsection “variable amplification” [5.2.3]. However, you could also remove the capacitor. The resistor on the other hand is necessary to limit the current flowing into the analogue-to-digital converter(ADC) in case the input voltage is too high (or too low) and the internal clamp diodes of the ADC are activated.

Chapter 6

Digital front-end

6.1 Overview

The digital front-end begins at the output of the analogue-to-digital converter and ends with an USB transfer to the host device. Figure [6.1] provides a quick overview of the digital front-end. The two main components are a Xilinx Coolrunner 2 CPLD (xc2c128) and a Cypress FX2 USB-Controller. Samples from the analogue-to-digital converters are passed to the CPLD, then queued in the FX2's FIFO interface, and ultimately sent over USB to the host device. The FX2 micro controller supports three different operating modes: Ports mode, general purpose interface master (GPIF) and Slave FIFO. For our application, Slave FIFO is the most suitable mode. As the name implies, the Slave FIFO mode requires an external master device, which is the CPLD in our application. The external master sets the FIFOADDR pins to address a certain endpoint buffer on the FX2. Now whenever the master asserts the SLWR pin, data on the FIFODATA pins is written into this particular endpoint buffer. As the FIFO interface is clocked at 48Mhz and FIFODATA is 16 Bits wide, one would need a throughput rate of 768Mbit/s in order not to loose any samples, however, the theoretical limit of high-speed USB is only 480Mbit/s. This is why there are two different sampling modes: continuous and hardware triggered. In continuous mode, the SLWR pin is only asserted at every eight clock cycle (or with a frequency of 6Mhz). This results in a data throughput of 96Mbit/s, which can be handled continuously over USB. In hardware triggered mode, the CPLD waits until a defined rising or falling edge in the samples occur, and then asserts SLWR for only 1024 cycles. Therefore, 2048 Bytes are written into the FIFO queue. This data is buffered in the internal memory of the FX2, and then sent to the host over USB, without data loss. All relevant parameters for the CPLD, such as continuous or triggered mode, rising or falling edge, trigger level etc. are transferred from the FX2 to the CPLD over I2C.

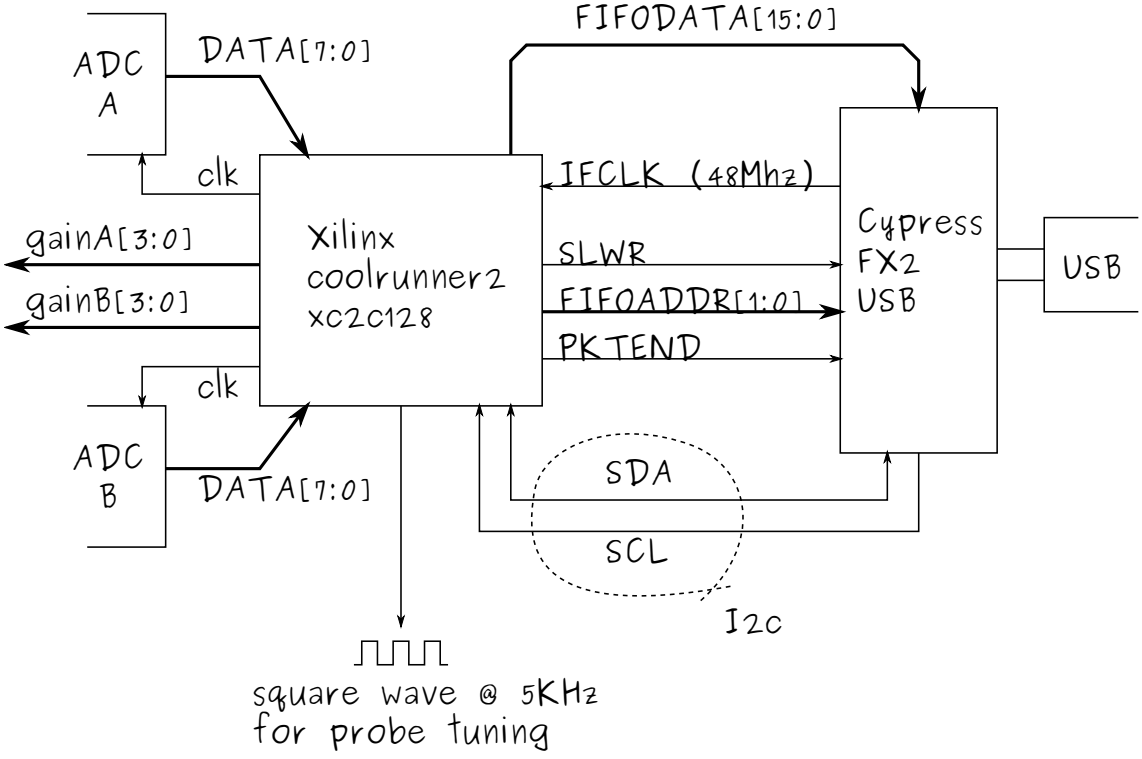

Figure 6.1: Overview of the digital front-end

6.2 CPLD firmware

The CPLD firmware is written in VHDL with the free Xilinx ISE WebPack Software. It consists of three distinct modules: I2C slave, controller and sampling. The I2C slave module receives data and informs the controller whenever a new Byte is available. This data is then read by the controller which will in turn set the external parameters, such as the gain of the analogue front-end, as well as the internal parameters for the sampling module. The sampling module consists of a continuous sampling mode and a hardware triggered mode with edge detection. In continuous mode, the CPLD addresses endpoint 2 on the FX2. In hardware triggered mode, endpoint 6 is used. Also, at the beginning of hardware triggered mode, the PKTEND signal is asserted. Asserting the PKTEND signal will initiate a short packet to make sure the buffers on the FX2 are empty before writing 2048 Bytes in the queue. To understand the basics of the VHDL code, a structured analysis is provided. With attention to the CPLD, again the previous figure [6.1] can be regarded as the main context diagram, with all important inputs and outputs to external devices. Figure [6.2] shows a data flow diagram with the internal VHDL modules on the CPLD. The signals of the data flow diagram are summarized in table [6.1]. A detailed explanation of the VHDL code is not provided, however, together with the data flow diagram and the data dictionary, the source code of both modules controller and sampling should be easily comprehensible. The I2C slave module may not be as readable, for a better understanding please see the I2C specification[1] from NXP (formerly Philips). This I2C slave module will only receive and acknoledge data, not send data back to the I2C master. To set the internal parameters on the CPLD, a received data byte has to fullfil a certain pattern. The first two bits (most significant bits) decide which parameters are changed by the following 6 payload bits. A summary of this data pattern is given in table [6.2]. You may wonder why the signal "probeTuning" stems from the module sampling, and not from the controller or a separate module. To generate the required square wave with a frequency of 5KHz a relatively large internal counter has to be used. Since the sampling module will write exactly 2048 Bytes in hardware trigger mode, and thus implements a counter, the same internal counter can be used for generating this square wave as well. Otherwise, the CPLD would run out of macrocells and can not synthesize.

VHDL source code Is available in the source file folder of this technical report, and can be downloaded from our website[2].

Testbenches Are also included in the VHDL source code. For a tutorial on how to generate graphical testbenches, have a look at chapter [E] in the appendix.

Timing report Minimum clock period is 14.600ns (68.493 MHz). Actual clock period is 20.833ns (48MHz).

CPLD Usage 112 of 128 Macrocells are used (95 macrocells by design)

Programming The CPLD is programmed via JTAG. We are using the Xilinx "Platform Cable USB" for a USB to JTAG connection. Unfortunately, this devce is pretty expensive, but there are also alternatives available such as the "XUP USB-JTAG Programming Cable"[3].

[1] http://www.nxp.com/acrobat/usermanuals/UM10204_3.pdf
[2] http://web.fhnw.ch/technik/projekte/eit/Fruehling2010/DiCerRud/
[3] http://www.digilentinc.com/

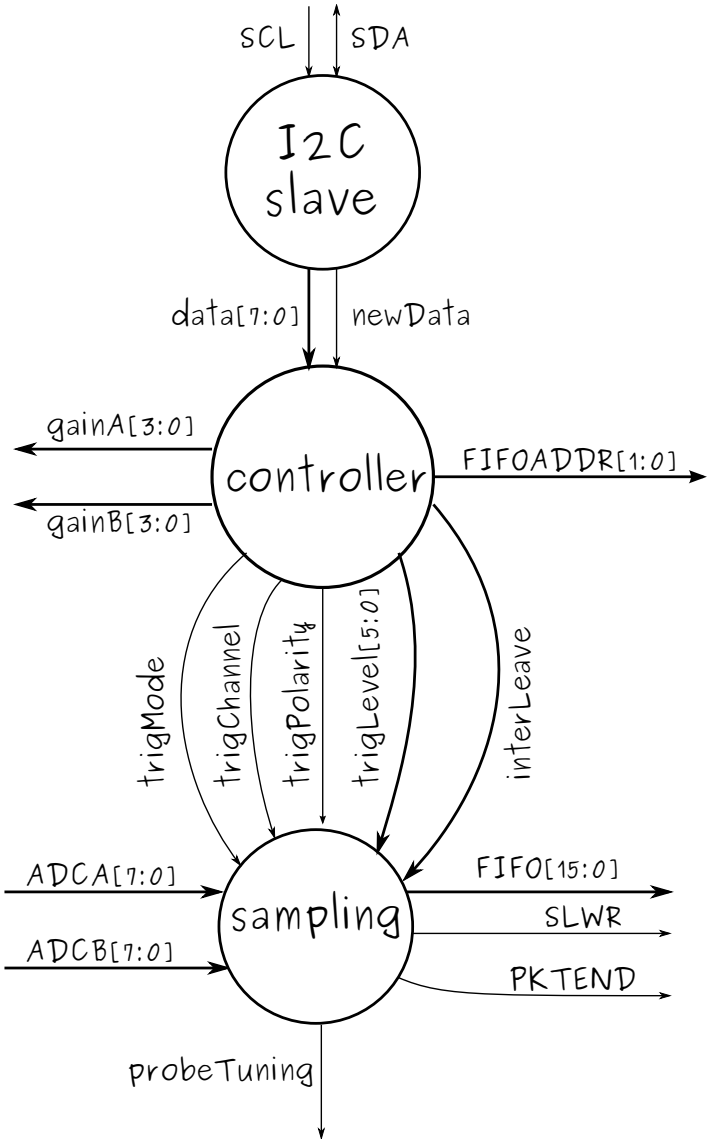

Figure 6.2: data flow diagram of the VHDL code

signal	type	source	description
ADCA	std_logic_vector(7:0)	ADC A	Output of analogue-to-digital converter A in unsigned byte data representation
ADCB	std_logic_vector(7:0)	ADC B	Output of analogue-to-digital converter B in unsigned byte data representation
data	std_logic_vector(7:0)	I2C slave	received byte over I2C
FIFO	std_logic_vector(15:0)	sampling	FX2 FIFO queue data
FIFOADDR	std_logic_vector(1:0)	controller	selects an endpoint on the FX2: ”00” endpoint 2 (continuous mode) ”10” endpoint 6 (triggered mode)
gainA	std_logic_vector(3:0)	controller	sets variable amplification on Channel A
gainB	std_logic_vector(3:0)	controller	sets variable amplification on Channel B
newData	std_logic	I2C slave	’1’ when new byte received over I2C ’0’ else
PKTEND	std_logic	sampling	sends a short packet to clear the endpoint 6 buffers on the FX2 in hardware triggered mode: ’1’ send a short packet ’0’ else
probeTuning	std_logic	sampling	square wave at a frequency of 5Khz for probe tuning
SCL	std_logic	FX2	I2C master clock line
SDA	std_logic	FX2	I2C data line
SLWR	std_logic	sampling	FIFO queue write enable: ’0’ enabled ’1’ disabled
interLeave	std_logic	controller	sets the SLWR assertion frequency in continuous mode ’0’ sampling at 6Mhz (default) ’1’ sampling at 48Khz
trigMode	std_logic	controller	’0’ continuous mode ’1’ hardware triggered mode
trigChannel	std_logic	controller	’0’ hardware trigger on channel A ’1’ hardware trigger on channel B
trigPolarity	std_logic	controller	’0’ hardware trigger on rising edge ’1’ hardware trigger on falling edge
trigLevel	std_logic_vector(5:0)	controller	(trigLevel<<2) sets the internal hardware trigger level. Example: with trigLevel=32 the internal trigger level is set to 32*4=128.

Table 6.1: data dictionary

CC | PPPPPP	received Byte over I2C CC = command (2Bit) PPPPPP = payload (6Bit)
CC=00 00 | GainB | GainA	**Gain/Amplification** GainA,GainB $= \begin{cases} 000 & \text{high gain} \\ 001 & \text{mid gain} \\ 010 & \text{low gain} \\ 011 & \text{highest gain} \\ \text{else} & \text{no gain} \end{cases}$
CC=01 01 | XXX | P | C | M	**Mode** X = don't care M = Mode $= \begin{cases} 0 & \text{continuous} \\ 1 & \text{triggered} \end{cases}$ C = Channel $= \begin{cases} 0 & \text{Channel A} \\ 1 & \text{Channel B} \end{cases}$ P = Polarity $= \begin{cases} 0 & \text{rising edge} \\ 1 & \text{falling edge} \end{cases}$
CC=10 10 | LLLLLL	**Triggerlevel** L = Triggerlevel/4 resp. L=(Triggerlevel>>2) Example: Triggerlevel = 128 $\Rightarrow$ L = 128/4 = 32
CC=11 11 | XXXXXi	**SLWR interleave (for continuous mode)** i = interleave for SLWR assertion $= \begin{cases} 0 & \text{sampling at 6Mhz} \\ 1 & \text{sampling at 48Khz} \end{cases}$

Table 6.2: I2C Byte pattern for controller module

6.3 FX2 micro controller firmware

The FX2 micro controller's role is to pass data received by the CPLD to the Android host if requested. It is also responsible for forwarding command data received from the host to the CPLD using the I2C interface.

The FX2 micro controller consists of helpful hardware that takes care of data submission (the Serial Interface Engine) and the 8051 micro controller that runs firmware. Featuring a slave FIFO mode the FX2 is able collect parallel data on 16 input pins and directly write the data to the desired endpoint buffer as soon as the write pin is asserted by the external hardware. In slave FIFO mode, the 8051 micro controller firmware can be bypassed and does therefore not need to take interaction.

To operate in the slave FIFO mode the firmware just has to configure the interface and endpoint-FIFOs on initialization.

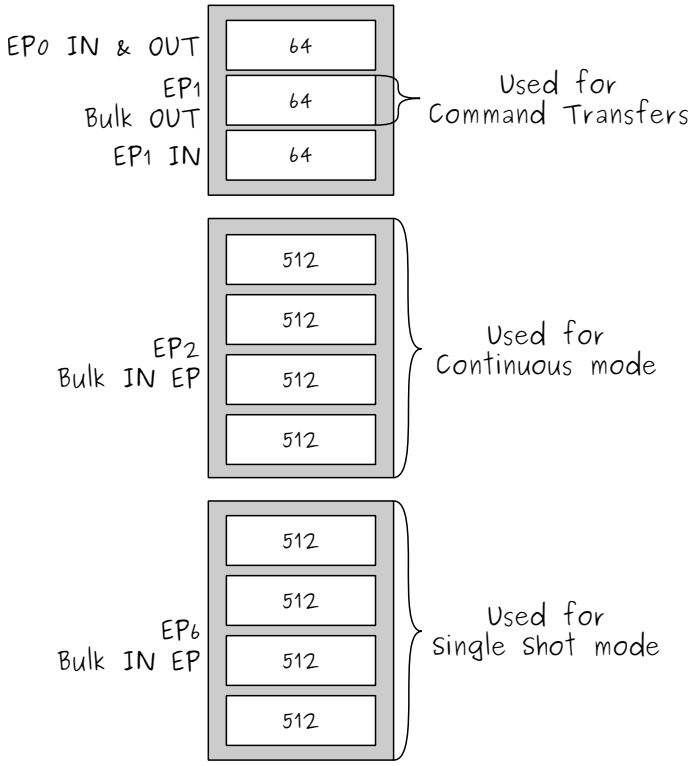

Figure 6.3: FX2 endpoint configuration

As shown in figure [6.3] we use endpoint 2 and endpoint 6 in quad-buffered configuration. Endpoint 1 OUT is used to send control data from the Android host to the CPLD. Therefore, the firmware checks if new data is available in the "POLL" loop, and if so, takes the first byte and submits it through the I2C interface to the CPLD.

As a basis we used the source code created in the preceding project. Since the main application for the FX2 is fairly dense the reader is able to find the source code in the Appendix [C].

Chapter 7

Layout

As a layout tool EAGLE[1] has been utilized. Great efforts have been made to minimize the size of the printed circuit board (PCB), so that the free EAGLE Light Version could be used; as the EAGLE Light Version is restricted to a PCB size of half a Euro card, with only two layers (bottom and top) available. A two layer design is probably not as complex as a four layer design, however, care must be taken to separate the analogue and digital parts of the design, as there are no designated analogue and digital ground or power plane layers. There are still many ways to prevent digital noise from interfering with analogue parts. We have decided to use the most simple one, which is local separation, with the bottom layer plane serving as both digital and analogue ground.

For a basic overview, figure [7.1] shows a screenshot of the EAGLE board layout. For more details, please have a look at the schematic and layout sources, which are available in the files collection folder of this technical report and can also be downloaded from our website[2]. There is also an EAGLE library "oscione.lbr" available, containing all relevant devices and footprints.

You may notice that there is a jumper on the board, which allows you to select an external clock source for the CPLD and the FX2 FIFO interface. This jumper is there for historical reasons. In the beginning of the project the highest sampling frequency was limited to 40Mhz due to an analogue-to-digital converter. Thus an external oscillator was thought as a clock source for both the CPLD and the FIFO interface. With the currently used ADC, the IFCLK pin of the FX2 at a frequency of 48Mhz is used as the overall clock source. Yet the jumper was still left in the design as a backup solution, in case the IFCLK clock source was not able to clock the CPLD properly, which is not the case.

The layout was printed at PCB-POOL Beta LAYOUT[3]. The finished printed circuit board is illustrated in figure [7.2].

As a side note, we want to point out that this printed circuit is the first layout we have ever created. Therefore, we know that it is not perfect, on the other hand, we are also happy that it all worked out.

[1] http://www.cadsoft.de/
[2] http://web.fhnw.ch/technik/projekte/eit/Fruehling2010/DiCerRud/
[3] http://www.pcb-pool.com

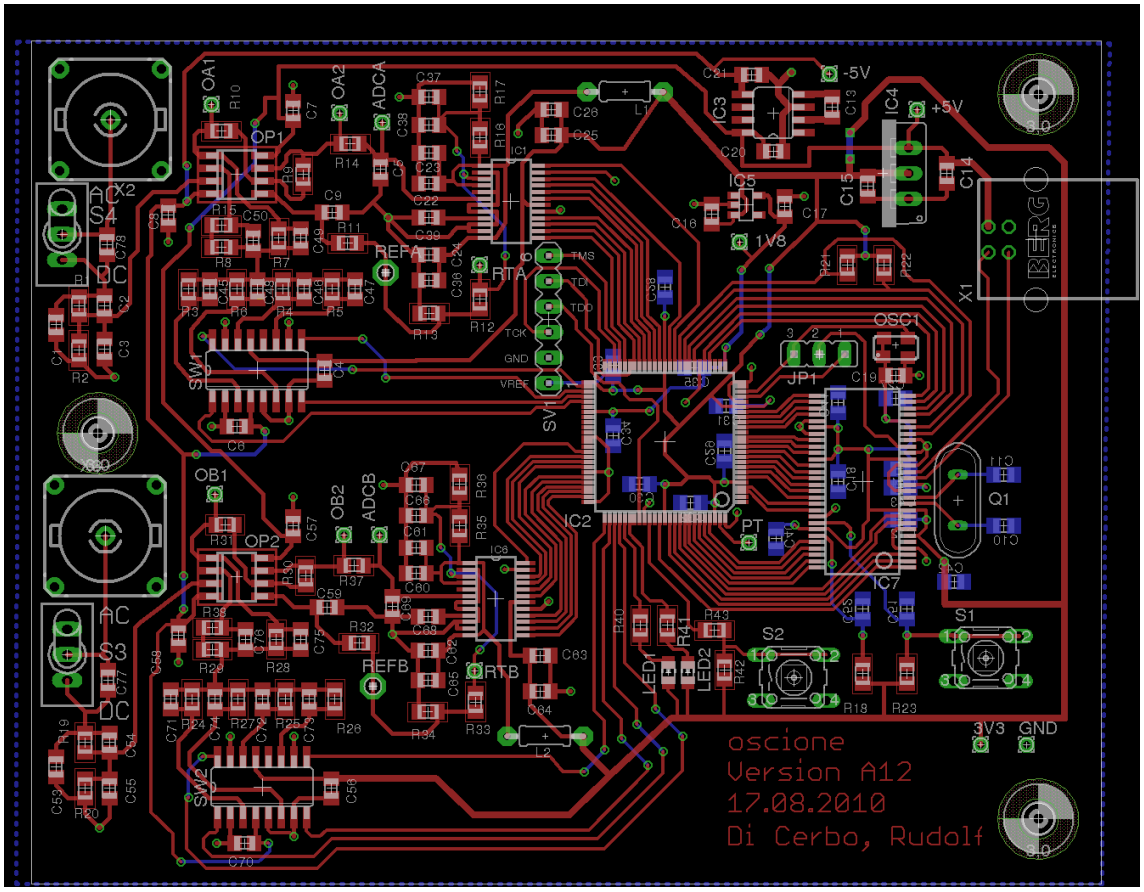

Figure 7.1: EAGLE layout of the oscione board

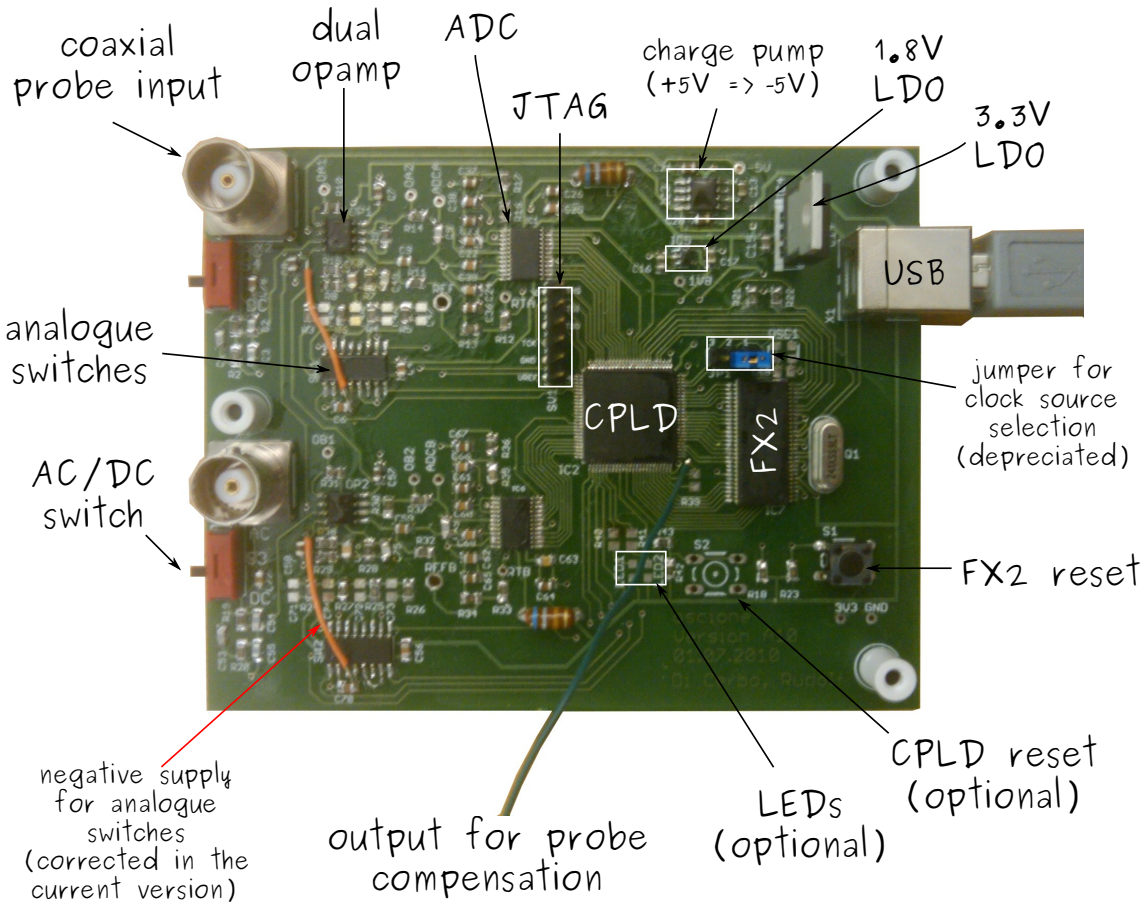

Figure 7.2: Picture of the printed circuit board

Chapter 8

Specification

- Two separate channels
- Designed for 10x oscilloscope probes
- AC/DC coupling
- 8 Bit analogue-to-digital conversion
- Continuous sampling at a frequency of 6Mhz
- Maximum sampling frequency of 48Mhz
- Hardware trigger detection for single-shot mode
- Overall system clock of 48Mhz
- Adjustable gain with 5 discrete values
- Analogue Bandwidth (3dB): minimum 3.3Mhz (depending on gain)
- Maximum Input Voltage: 16V
- 5kHz square wave output for probe compensation
- USB-powered
- Power consumption: ≈1W

The power consumption of 1W is an estimated value. The following table [8] sums up the relevant contributions in the power estimation.

2x ADC reference	100mW $2 * 18\text{mA} * 3\text{V} \approx 100\text{mW}$
2x ADC	130mW (data sheet value)
2x opamp	20mW 1mA per opamp, $\pm$5V supply voltage
CPLD	50mW calculated with coolrunner II "Power Evaluation Equation"[1]
FX2	280mW (data sheet value)
3.3V LDO	300mW $(5\text{V} - 3.3\text{V}) * I_{tot} = 1.7\text{V} * 180\text{mA}$
TOTAL	**880mW**

Table 8.1: estimation of power consumption

[1] http://www.xilinx.com/support/documentation/application_notes/xapp317.pdf

The analogue bandwidth has been measured with a sine wave input and is dependant on the active gain setting.

Gain setting	-3dB bandwidth
0.25 V/div	3.3MHz
0.5 V/div	4MHz
1 V/div	6MHz
2 V/div	8MHz

In 4V/div gain setting, there is no active amplification and the first operational amplifier is basically a voltage follower. However, as explained in [5.2.3], the signal may be altered due to parasitic capacitances of the analogue switches. Therefore, the frequency response gets distorted as well. This frequency response is plotted in figure [8.1]. Thus, with this gain setting it is not encouraged to measure signals with a frequency of more than 1Mhz, as the resulting values will be erratic.

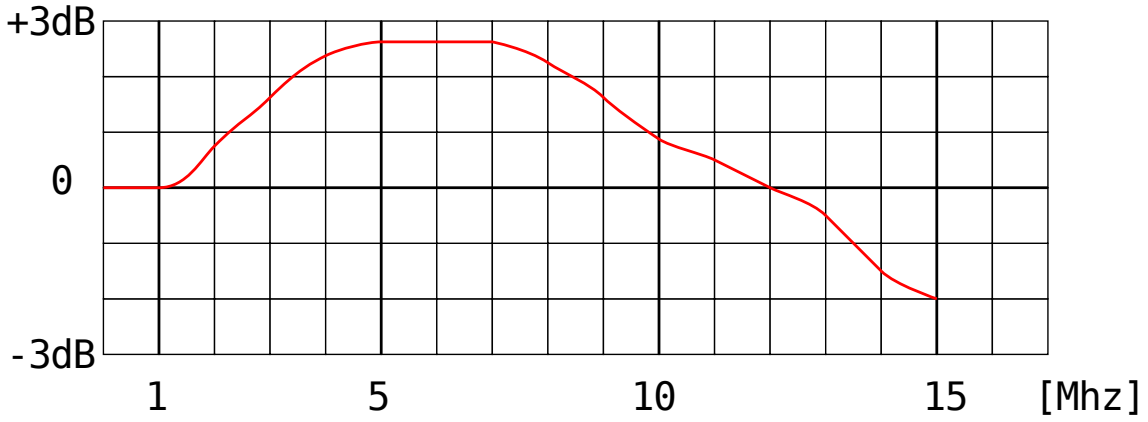

Figure 8.1: frequency response of 4V/div gain setting

Chapter 9

Components evaluation

As in almost every components evaluation, the two hardest constraints are power consumption and pricing. The oscione board is USB powered, therefore the maximal power consumption is 5V*500mA=2.5W. But of course, we try to achieve even lower consumption, as it could be used as a mobile device. It needs to be said that all components were searched and purchased at Farnell[1].

9.1 Operational amplifier

9.1.1 Bipolar supply

We have come to the conclusion that the operational amplifiers need a bipolar supply. In the beginning of the project we tried to realize an analogue front-end with only a single power supply. For that purpose, theoretically, a virtual ground circuit or a rail splitter such as the Texas Instruments' TLE2426 could be used. The USB power supply would provide 5V, and the virtual ground of the analog front-end would be the mid-voltage at 2.5V. However - if the USB host has a galvanic connection to ground - when measuring signals with this setup and clamping the (virtual) ground of the oscilloscope probe to another galvanically connected ground potential, the virtual ground of 2.5V would be short circuited! See figure [9.1] for an illustration. Thus, a DC-DC converter could be used, but they are also pretty costly. So we have decided to use an inverse charge pump. The ICL7660 from Maxim was already in stock in our laboratory, so we use this component to create a dual power supply. The only problem with the ICL7660 is that it can only provide relatively small currents. When drawing 10mA the output voltage will already drop by 0.5V.

9.1.2 Power consumption

A low quiescent current is needed, as the inverse charge pump ICL7660 can not deliver much current. Furthermore, a bipolar supply of $\pm$5V (better $\pm$4.5V as the USB voltage may vary) must be enough to power the opamp.

$\Rightarrow$ OPA2889 (Texas Instruments): quiescent current 460μA/channel, minimum bipolar supply $\pm$1.3V.

9.1.3 Gain-bandwidth product

The fastest sampling rate is 48Mhz. Following the Nyquist-Shannon sampling theorem, frequencies up to 24Mhz could be identified. Therefore the analog bandwidth of the analogue front-end should be at

[1] http://www.farnell.com

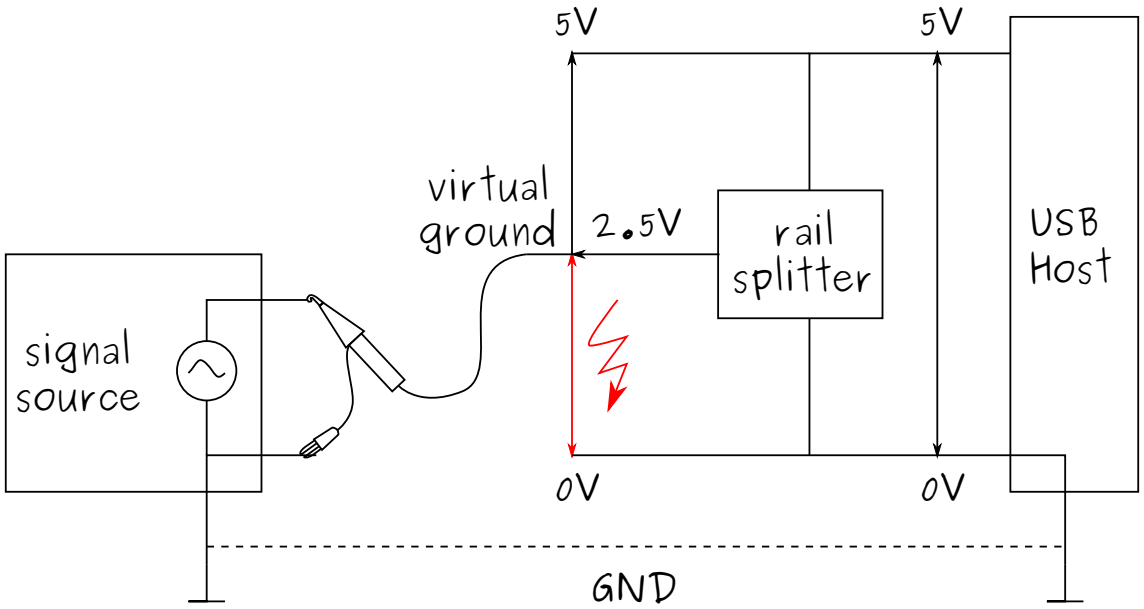

Figure 9.1: short-circuit problem without bipolar supply

least 24Mhz as well. But on the other hand, an oscilloscope graph where only 2 samples per period of a 24Mhz signal are plotted will not be very useful either. For evaluation purposes, we set the desired analogue bandwidth at 10Mhz, which is still rather high. With a maximal amplification of +12V/V, this results in a minimum gain-bandwidth product of 120Mhz.

⇒ OPA2889: bandwidth of 60Mhz at a gain of +2 V/V (gain-bandwidth product ≈ 120Mhz).

9.1.4 Slew rate

The full scale input of the ADC is 2V. With a sampling frequency of 48Mhz, the slew rate needs to be at least

$$\frac{2\text{V}}{(1/48\text{Mhz})} = 95\text{V}/\mu s$$

In the best case scenario, an opamp with 95V/μs slew rate can toggle between the lowest and highest ADC-input within one sampling period. Of course, an even higher slew rate is desired.

⇒ OPA2889: slew rate 250V/μs.

9.1.5 Packaging

the OPA2889 is a dual opamp in one package, which comes in handy, since each oscilloscope channel needs exactly two operational amplifiers.

9.2 Analogue switches

Power consumption of analogue switches is not an issue, as they all draw very low currents in the range of 1uA and below. Also, for our application switching speed is not important. We have been looking for an analogue switch with low parasitic capacitance to reduce overshot peaks as explained in the analog-front end section [5.2.3]. Also, we were looking for a component with multiple switches within one package. Switches should be normally open.

⇒ MAX4522 (Maxim): 2pF capacitance Off-Capacitance, 4 switches in one package, normally open.

9.3 Analogue-to-digital converter (ADC)

At first, the TLC5540 (Texas Instruments) was intended to be used in the design, as it is a low power device (≈85mW) and has also been successfully implemented in other Oscilloscope projects[2]. However, this ADC has a few drawbacks as well. The maximum sampling frequency is only 40Mhz, and the voltage of it's digital data output pins is 5V. This would imply using an external oscillator instead of the FX2's 48Mhz IFCLK as a clock source. Furthermore, a voltage converter would have to be used, since the input pins of the Coolrunner 2 CPLD are not 5V tolerant. Therefore, we have decided to dismiss this particular ADC.

The highest desirable sampling frequency is 48Mhz. When looking for an appropriate ADC do not make the same mistake as we did by limiting the online search filters to ADC's with sampling frequencies ranging from 50Mhz to 70Mhz. One potential ADC we have found this way is the ADS830 from Texas Instruments with a maximum sampling frequency of 60Mhz. However this ADC has a relatively high power consumption of around 170mW. Later on, by extending our search to higher sampling frequencies, we have found the ADC08100 from National Semiconductor. This ADC08100 has a power consumption of only 130mW at a sampling frequency of 100Mhz; and the power consumption is almost half at 48Mhz. Interfacing to the CPLD is no problem either, as it's supply voltage is only 3.3V.

⇒ ADC08100 (National Semiconductor): power consumption≈65mW at a sampling rate of 48Mhz. Single ended analog input. +2.7V to +3.6V supply voltage.

9.4 CPLD - Xilinx or Altera?

Both Xilinx and Altera offer free versions of their integrated development environment running natively on Linux (Xilinx ISE WebPack and Altera Quartus II Web Edition). Since we already had a Xilinx coolrunner 2 evaluation board (Digilent XC2-XL[3] & [4]) for testing at our laboratory, we decided to stick with Xilinx, it is as simple as that. We have purchased the 128 Macrocell version of the Coolrunner 2 CPLD. By the time of this writing, 112 Macrocells are occupied.

[2] http://www.bitscope.com
[3] http://www.digilentinc.com/Data/Products/XC2XL/XC2XL-rm.pdf
[4] http://www.digilentinc.com/Data/Products/XC2XL/XC2XL-sch.pdf

Part III

Android Application

Chapter 10

Android Platform

During the preceding project experiences in configuring and compiling the Android root file system (RFS) have been gathered. Furthermore, knowledge of kernel configuration and compiling has been acquired. The "Beagleboard" OMAP 3530 ARM platform again serves as a hardware platform in this project, as it did in the preceding project.

While we have been creating our own Android root file system build in the last project we decided to adjust the popular "Android Rowboat Project" to our needs. This enabled us to use the Android version 2.1, code name "Eclair".

10.1 Operating System

10.1.1 Kernel

The Rowboat project links the common OMAP Kernel tree owned by Tony Lindgren. The compiling process has been identical to the one in our last project. We configured the Kernel to implement our USB-to-Ethernet controller driver.

Build instructions for the kernel can be found here:

`http://code.google.com/p/rowboat/wiki/ConfigureAndBuild`

10.1.2 Root File System

We used the Android sources of the Rowboat project[1] to build our application platform. Since most of the standard android applications that are delivered with the distribution are not of interest for our project, we built the system only with core functionality.

To do so, we removed all applications found in the file

`build/target/product/generic.mk`

For further information about the Android RFS layout please consider the technical report of our preceding project. It can be found online under http://android.serverbox.ch

[1] http://code.google.com/p/rowboat/

Chapter 11

Application Overview: OsciOne - an Android Oscilloscope

The task of the Android application is to display data samples to the user. Furthermore, the user is then able to interact with the application in a similar fashion as with a hardware oscilloscope.

The application controls data sources such as USB payload or Audio data. It therefore consists of an Activity which handles UI-Requests and graphical display, and a Service which handles data sources and flow control. Details regarding Android "Activity - Service Model" can be found in our previous technical report[1].

To display data samples to the user a data source is of necessity. As pointed out in part [II] an essential task of the project was to create a hardware front-end. The front-end taps a signal source and passes the resulting samples to the Android platform using USB Transfers and therefore serves as a principal source of data.

Although we enable and encourage the developer to implement his or her own data sources this report covers primarily USB data as source unless pointed out otherwise. USB data is the most time critical kind of source in this project and therefore needs to be handled more sensitively than other sources.

Illustration [11.1] shows the steps that have to be taken to display a choice of samples to the user in continuous mode. Since the application altogether serves as an oscilloscope, and the display limits the number of data points that can be shown, we also need to make a reasonable selection of data samples. This selection is the combination of a trigger which detects a rising or falling edge in the samples and a time scale that defines which samples around the trigger have to be selected and displayed (see chapter "Triggering and Sample Selection" [14]). In single shot mode, the hardware takes care of trigger selection and only sends relevant samples to the application. Thus, software triggering is not of necessity in this case.

As shown in [11.1] the application generally consists of a cyclic process. The hardware front-end sends data samples. The application issues USB transfers. Those transfer structures hold data buffers that are filled with data acquired by the front-end. Once acquired by the host application the data will be processed so the desired samples can be selected and displayed to the user. The processing (steps 2,3) consist of finding a trigger and selecting the data samples around that trigger for display. In step 2, the selected data samples are displayed to the user. At the end of this process the initial USB transfer has to be re-queued so the next buffer of samples can be transmitted.

As pointed out in step 4 in [11.1] the application uses Open GL to display the selected samples to the user. Open GL enables speedy rendering, therefore, a decent frame redraw rate can be achieved.

[1] available under http://android.serverbox.ch

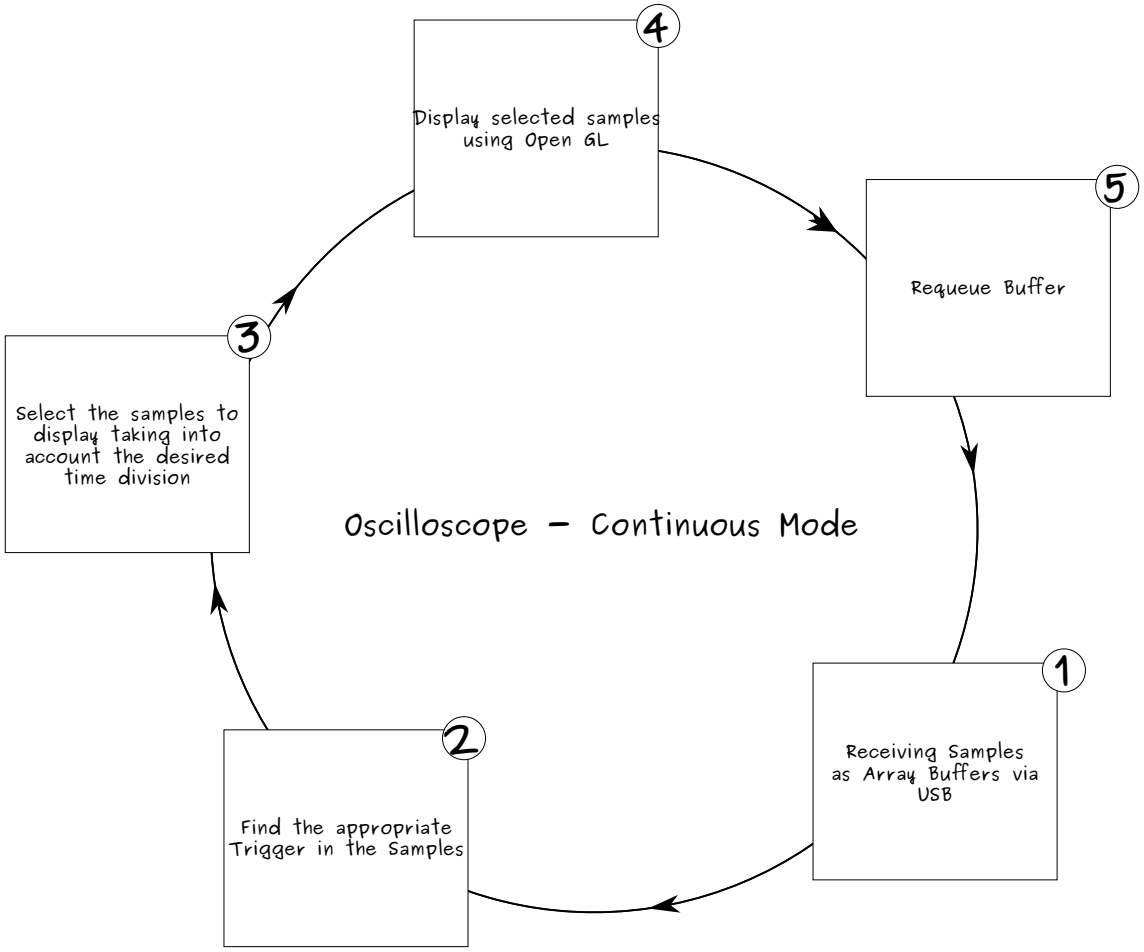

Figure 11.1: Cyclic application process for continuous mode

There are two major challenges in this cyclic process:

- **Timing:** The average time to operate from point 1 to 5 must not take longer than the time between two transfers occurring.
  ⇒ Solution: implement time consuming operations in native C.
- **Flow Control:** If using exactly one data buffer, data that has been collected while operating from step 1 to 5 is going be lost since there are no USB packets available for new data. Thus some kind of "Multi-Buffering" is mandatory so collection of USB packets can continue during processing steps 1 to 5.
  ⇒ Solution: use multiple transfer structures and data buffers.

The upper limit of data resolution is solely defined by the hardware front-end. Whereas the lower limit of data resolution is defined by the application. Using High Speed USB Bulk transfers, the practical limit of continuous throughput lies around 40 MByte / second.

In single shot mode, the application displays only a set of samples that have been collected after a certain trigger level has been passed. The front-end detects such a trigger occurrence and sends the data to the Android application after the trigger event. When inspecting figure [11.2] and comparing it to illustration [11.1] you can see that there is no need to find a trigger in the samples. The trigger is implemented in hardware using the CPLD.

The main reason to implement such a "hardware" trigger is to achieve superior sample resolution up to 48 [Msps] per channel over a short period of time. It is not possible to keep up a constant data throughput

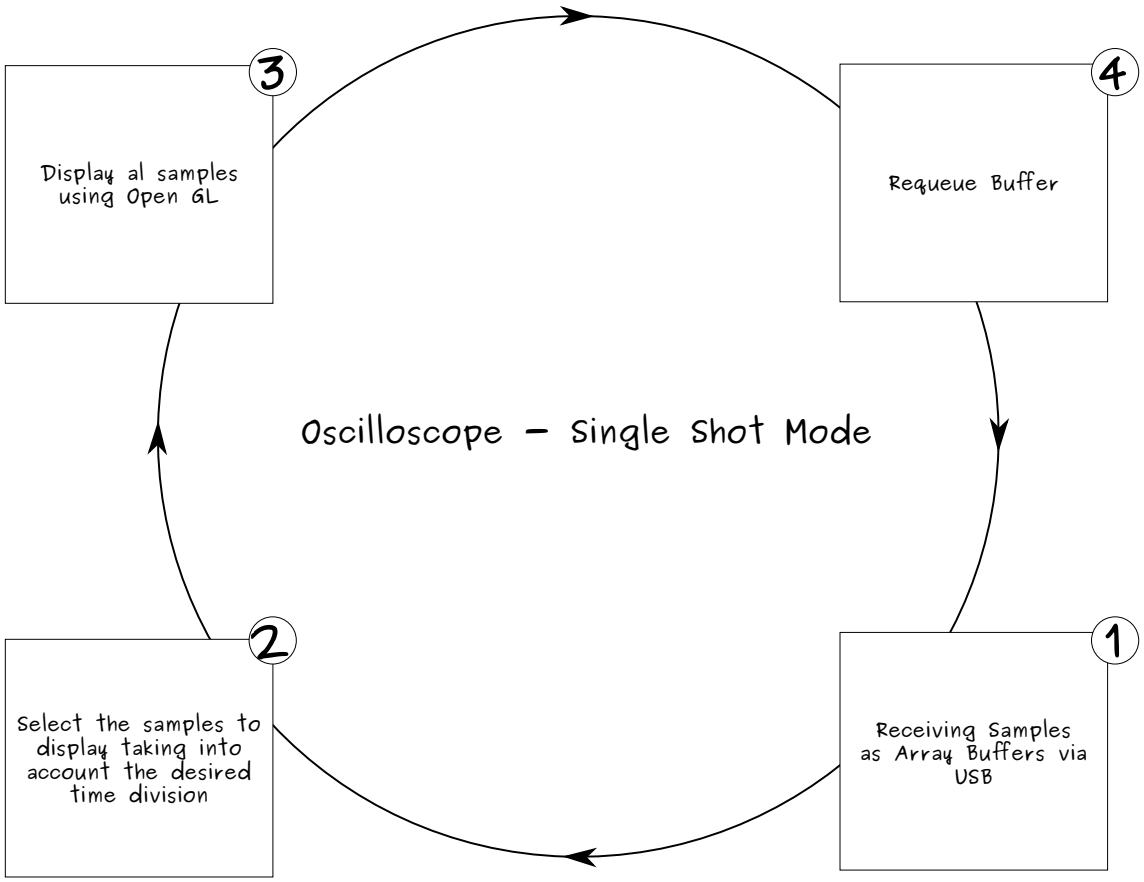

Figure 11.2: Cyclic application process for single-shot mode

of 2x48[Msps] = 96 [MB/s] over USB. Using the endpoint buffers of the USB micro controller such a high sampling rate can be achieved over a short period of time.

There is however a serious challenge to overcome when implementing a single shot.

Pre-Trigger: Since we do not use RAM for the CPLD a pre-trigger implementation is not trivial. Samples are saved in the FX2 endpoint buffer. Since the endpoint buffers are not meant to be driven as ring buffers it is not possible to create a constant number of pre-trigger samples. However, it is possible to write samples continuously to another endpoint and dismiss the data as soon as the endpoint buffer is full and then restart writing samples to the endpoint. This will lead to a pre-trigger of undefined size. At the point of this writing the team has implemented post trigger data acquisition only.

Chapter 12

Application mechanics

As mentioned earlier, the Android application consists of two essential parts: An Android Activity and an Android Service.

While the Service is responsible for its data sources, data collection and data processing, the Activity handles input events from the user and data visualization.

Figure [12.1] illustrates how the Activity is built on top of the service and how the service handles data sources.

The reason why the implementation is not solely based on an Activity is that a Service offers the ability to run even when the Activity is not visible for the user. Furthermore it is possible to re-use the Service for another Activity or application. Another benefit is the logical distinction between the user interface layer and the data layer.

Native implementation is accessed using Java Native Interface. Thus native functionality is achieved by loading shared library objects during runtime.

The main anchor point for service level implementation is the OsciService class that extends the android Service class.

The Service class holds references to data sources. Data sources all extend at least the interface "OsciSource" whereas data sources that also handle input implement the interface "OsciIoSource". During project development the Service was gaining complexity and started lacking tidiness. As a solution those interfaces have been created. Not only did they support tidiness of the source code but they as also offer a way to let a developer implement a fully customized data source in a clear and easy fashion. Thus, the way the oscilloscope is programmed offers great modularity. Data sources can be "plugged in" as the developer desires.

Native method access is handled by the data source classes and therefore do not rely or interfere with the service or the application logic.

In order to create the previously described application there is a requirement for a USB-API. In the preceding Project "Using Android in Industrial Automation An Android Spectrum Analyzer"[1] libusb, an open source USB library, has already been used to implement USB transfers. The library has been ported to Android as a part of the preceding project and is used for this oscilloscope application.

Native C/C++ implementation is therefore a requirement to be able to create this Android application. Since the Android application framework is designed to be used with Java, the project also makes use of the Java Native Interface (JNI) to access data acquired by libusb.

[1] http://android.serverbox.ch

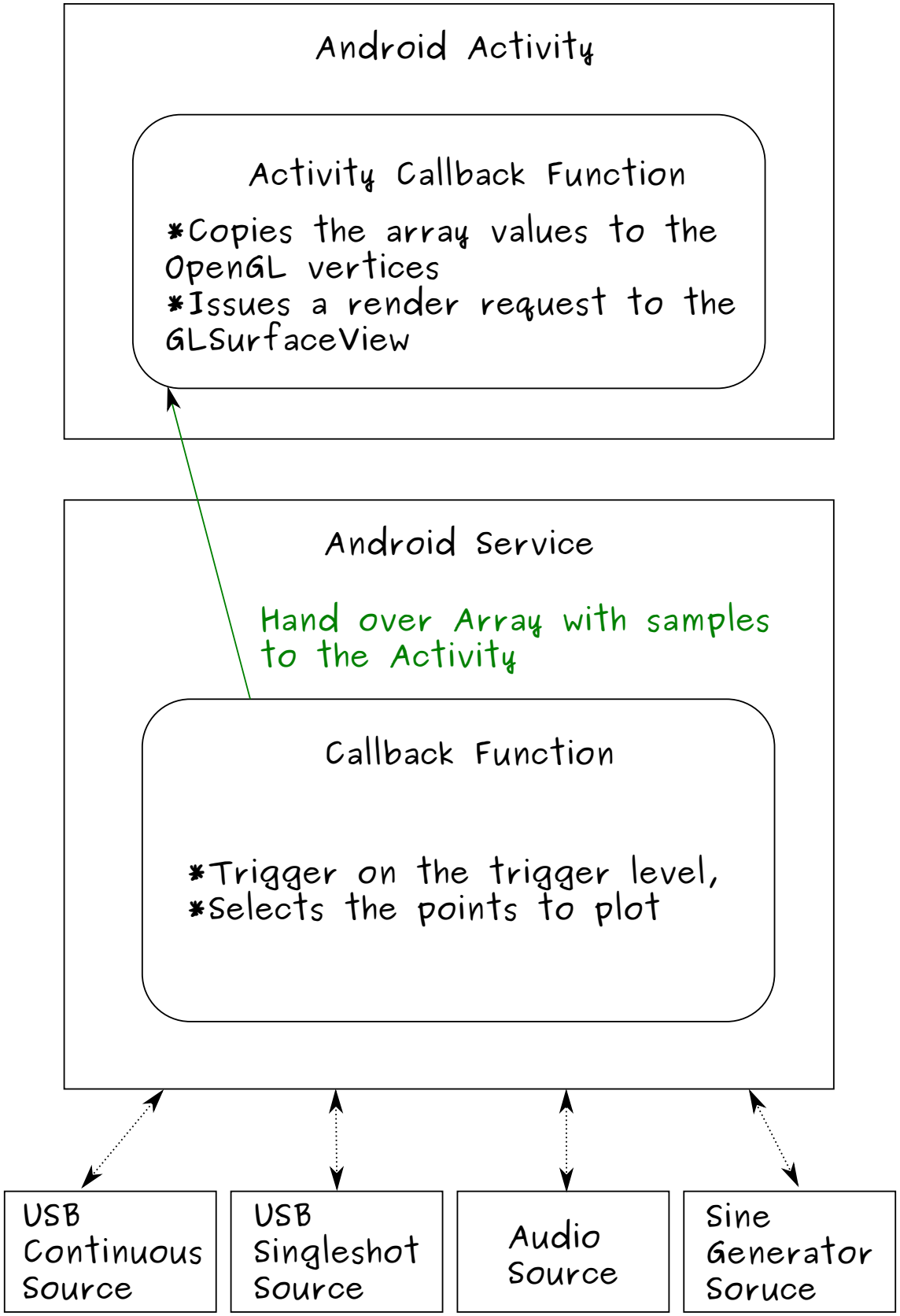

Figure 12.1: overview of application layers

For ease of implementation the project followed the rule “Use Java where ever possible, use native implementation when ever necessary”. There is a number of functions where native implementation is required to improve performance. Especially functions that loop through arrays or copy array regions suffer from performance issues in Java. Therefore most of these functions have been written natively and were compiled with the Android Native Development Kit to ensure a maximum of portability.

There are two parts of the application that are programmed natively in order to make them execute faster. One part is the trigger and sample selection which can be found in the project’s sources in the module “jni/trigger.c”. The other part consists of copying data points into the OpenGL vertex buffer. It can be found in the module “jni/vertexcopy.c”.

The libusb application part has been compiled within the Root File System since it is not possible to compile this part of the application against an arbitrary shared library using the Android NDK. However, since this part of the application requires changes to the Root File System and Kernel any how this has no particular downside. It resides in the modules found under the external/liboscione folder of the root file system sources.

Another consideration that has been made concerns the application’s threading. The application consists of three threads. One is the main Activity thread that is used by the system for UI-event handling. Another Thread is the rendering thread of the GLSurfaceView. Both of those threads are not created explicitly but rather come with object creation. The third thread, which is created intentionally, is a thread in the service class. It is used by the data sources and all data collection and handling is done by this dedicated thread. This thread is even used when executing methods of the Activity through the Remote Interface (see section “Communication with the Activity” [13.2]). Using this approach, many potential issues of threading and thread synchronization are avoided.

Chapter 13

Service and Data Sources

The following sections explain how the service is designed and implemented. To illustrate the patterns and schemes that were designed for the application it is not possible to avoid Java code snippets. Therefore, the reader is required to either have experience in Java programming or have basic understanding of Object Oriented programming.

13.1 Service

OsciService is the Service class that is used for the project. It extends the android Service class. It also implements the Interface “Callable”.

“Callable” exposes the function “doCallback(OsciSource src)”. To make adding new data sources easy, it is possible to create a new class that implements OsciSource and holds a reference to a “Callable”-object. At this point a new source only needs to call “doCallback” of the given “Callable”-object.

Illustration [13.1] shows how the service class is designed.

The essential advantage this programming scheme has is that the developer does not have to care about rendering, threading and Service-Activity communication.

An example illustrates the usage:

```
public class SpiSource implements OsciSource{

    private ByteBuffer mCh1 = ByteBuffer.allocateDirect(512);
    private ByteBuffer mCh2 = ByteBuffer.allocateDirect(512);
    private boolean mStop = false;
    ...

    @override
    public void start(Callable call){
        while(readFromSpi(mCh1, mCh2) && !mStop){
            //let's assume readFromSpi takes around 33[ms]
            //to assure 30 fps display
            call.doCallback(this);
        }
    }

    @override
```

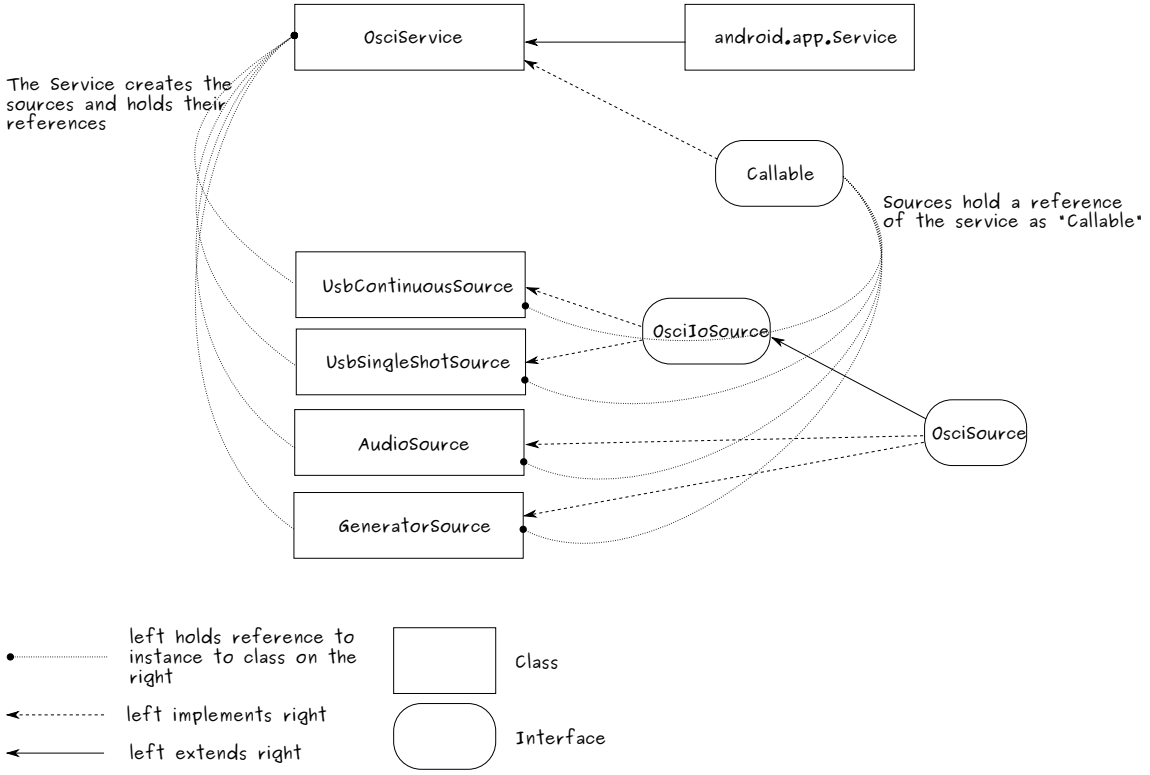

Figure 13.1: relational diagram of the service

```
    public void stop(){
        this.mStop = true;
    }
    ...
    // let's assume the developer has a native implementation
    // of the spi read access
    private native boolean readFromSpi(ByteBuffer ch1, ByteBuffer ch2);
}
```

To further help the developer with source implementation we created an example source called "GeneratorSource" that creates a sine wave. Using the NDK we also provided detailed code examples of how to use the Java Native Interface for such a purpose.

OsciSource Interface in detail

```
public interface OsciSource {
    public static final int DATATYPE_UNSIGNED_BYTE = 0;
    public static final int DATATYPE_SIGNED_BYTE = 1;
    public static final int DATATYPE_UNSIGNED_SHORT = 2;
    public static final int DATATYPE_SIGNED_SHORT = 3;

    public int getDataType();
    public ByteBuffer getCh1();
    public ByteBuffer getCh2();
    public void start(Callable call);
    public int[] getSkipSamples();
    public String[] getSkipSampleLabels();
```

```
    public int getNumPointsPerPLot();
    public void stop();
    public void forceUpdate();
    public boolean isAvailable();
    public boolean isAlreadyTriggered();
}
```

By implementing this interface and creating the code for all methods a developer is able to create a new data source. The developer does not have to worry about threading, as this is managed by the service.

The Service ("OsciService" class) will issue calls to the defined interfacing methods.

public int getDataType()

The developer tells the service what data to expect. Therefore the developer returns one of the DATATYPE_ values. This is necessary for the Service to interpret the values found in the source's ByteBuffer.

public ByteBuffer getCh1(), public ByteBuffer getCh2()

The user returns the latest collected data as a ByteBuffer-Type. The method getCh1() refers to the data of Channel 1, getCh2() refers to Channel 2.

public void start(Callable call)

This is the most essential function of the data source. The Service will call the start function as soon as the corresponding data source is selected by the user.

The start function needs to collect data and execute "call.doCallback(this)" as soon as an update should be issued. The service will then take care of triggering and sample selection.

public int getNumPointsPerPlot()

Return the number of samples you want to display on the screen.

public int[] getSkipSamples(), public String[] getSkipSampleLabels()

These two functions return name-value pairs for sample selection. The function getSkipSamples() returns an array with values which are interpreted as the number of samples that can be skipped. See the chapter "Triggering and Sample Selection" for details. The function getSkipSampleLabels returns the label to be displayed in the time division part of the user interface.

Depending on how many samples you want to display on the screen, the time division labels should be adjusted.

Say you have a sampling rate of 1 [kHz] per channel and you are displaying 250 samples on the screen. Assume your interleaving values are $\{1,2,4\}$. The screen is divided by the grid into 10 divisions.

In the case that every sample gets displayed the amount of time t displayed by the whole screen would be:

$$t = \frac{1}{f} * 250 = \frac{250}{1000} = 0.25[\text{s}]$$

This would mean that the initial time division is 0.25[s] / 10 = 25 [ms]. The function getSkipSampleLabels() therefore needs to return the values {"25[ms]", "50[ms]", "100[ms]"}.

public void stop()

This function is called by the service upon changing the data source or leaving the activity. The sampling has to be stopped here.

public void forceUpdate()

This method is designed for a single shot source and called upon changing the time division in the UI. Therefore a new callback request has to be issued in order to update sample selection.

public boolean isAvailable()

Decide in the method constructor or in a static block if the source is available and return false if not or true if so. When true, the source will be selectable for the user.

public boolean isAlreadyTriggered()

When using a hardware trigger and your source has already been triggered then return true here. Otherwise return false. As of this writing, the trigger index is assumed to be zero; better implementation could return the actual trigger index instead of just true or false.

13.2 Communication with the Activity

When the user changes the trigger level, the time scale or simply shuts down the application the Service has to be informed of relevant changes. On the other hand, the Service needs to send processed data to the Activity so it can be rendered and displayed to the user. For this purpose, communication between Service and Activity in both directions is mandatory.

Service and Activity communicate through the Android Binder. In order to do so, a "Remote Interface" has to be created using Android's "Android Interface Definition Language" (AIDL). Please refer to the Android developer guide found under

`http://developer.android.com/guide/developing/tools/aidl.html`

for a detailed description of AIDL. Also in our preceding technical report[1] you will be able to find helpful information about Android inter process communication.

Figure [13.2] illustrates I/O communication of Activity and Service.

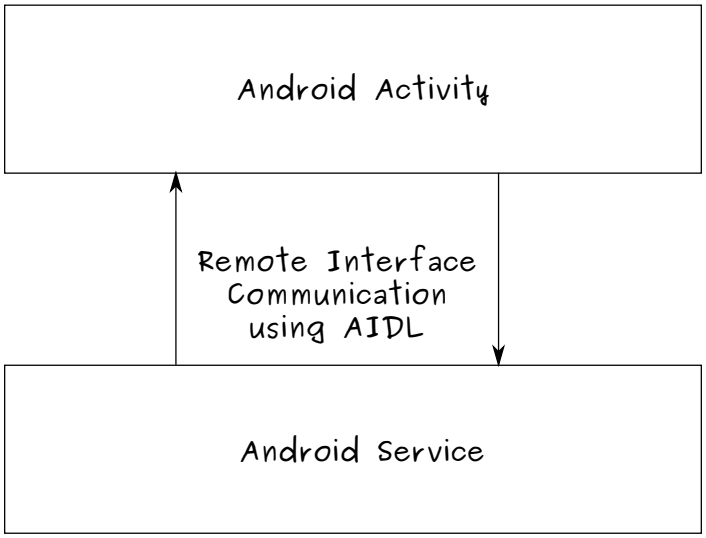

Figure 13.2: Android inter process communication using remote interfaces

[1] available online http://web.fhnw.ch/technik/projekte/eit/Fruehling2010/DiCerRud/

Chapter 14

Triggering and Sample Selection

14.1 Concept

A basic feature that any hardware oscilloscope contains is a trigger. When using a trigger the user wants to display a certain selection of data points. When the source data signal has passed a certain level of voltage a trigger is found. The user then likes to center the view around the region of that trigger. Using a time division shifter the user is able to see data samples that are closer or further away from that particular trigger.

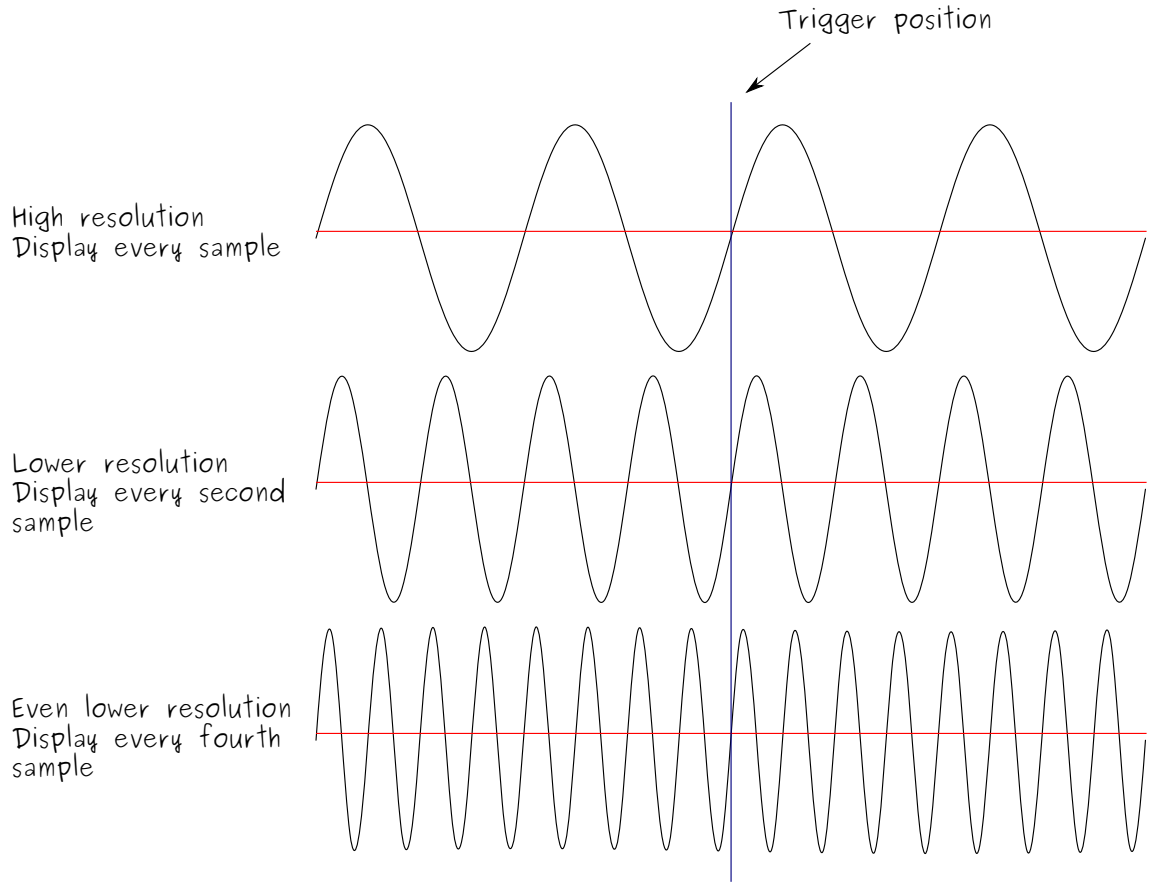

Figure 14.1: signal resolution depending on the amount of samples that are skipped

By changing the time division the resolution alters and therefore the user differs the display window.

Illustration [14.1] shows the changes to resolution when changing the selection of samples.

When data acquisition is block-based the sample selection can be achieved using the following algorithm:

- Find all potential triggers in a block.
- Select the trigger that is closest to the median index.
- Depending on the time scale, select every n-th point around the trigger.

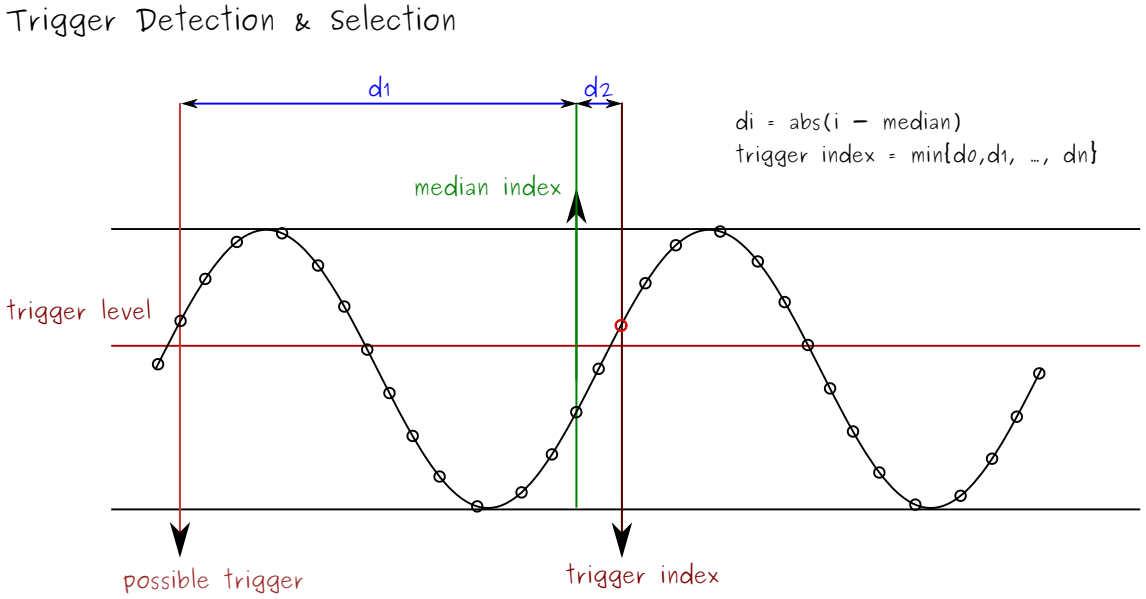

Figure 14.2: concept of trigger detection and selection

Figure [14.2] illustrates how a trigger is being detected.

Assume M samples are to be displayed on the screen. Therefore the Block length B needs to be greater or equal M.

For instance, in our application we collect samples with a sample rate of 6 [Msps] per channel in continuous mode. To display 30 frames per second we collect 2*B=396288 during one frame. We then display M=300 samples per channel to the user. Depending on the time scale we adjust the selection of samples and therefore alter the display "window". By selecting not every single sample but rather every "n-th" sample we are able to change the resolution and therefore the time scale of the mentioned window.

Illustration [14.3] demonstrates an example.

There are however some restriction using this model. Since processing is block-based it is not trivial to reasonably handle cases where the "display window" escapes the current block.

$$\text{Trigger Index} - \frac{M}{2} * n < 0 \quad \text{or} \quad \text{Trigger Index} + \frac{M}{2} * n > \text{B}$$

Figure [14.4] illustrates the exceptional cases.

While case one can be handled saving the last block, example two requires to "look into the future" to enable correct display.

In our implementation we solved this with a rather rigorous approach by setting the start of the window to zero in both of the above cases.

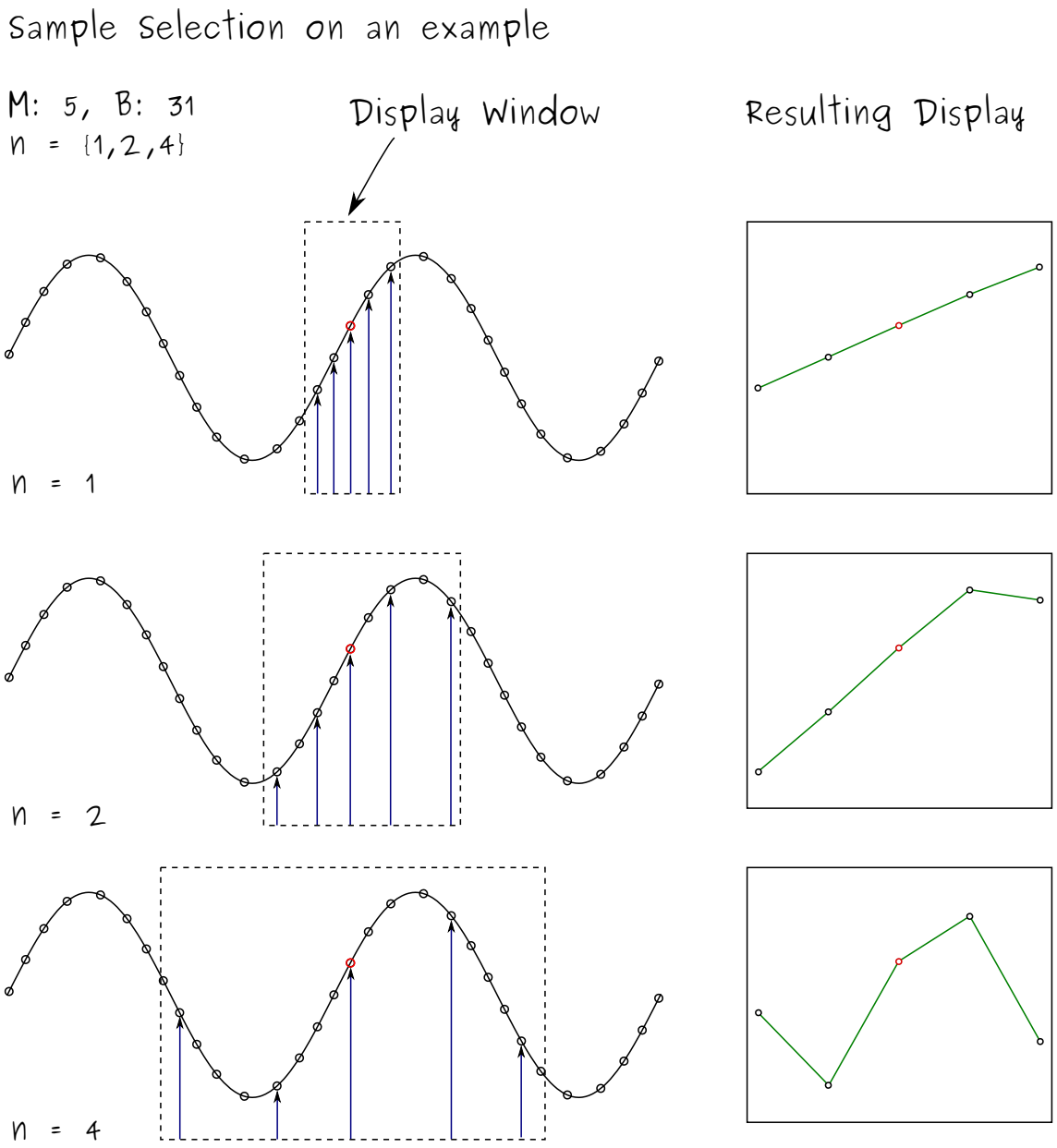

Figure 14.3: sample selection resulting in display windows

14.2 Implementation

At first we implemented the trigger detection and sample selection in Java. However, performance was very low and it consumed a large amount of time to detect a trigger and select the desired samples. By implementing functionality in native C and accessing it through the Java Native Interface a performance boost of a factor larger than five has been achieved. To compile the native library we used the Android NDK so the resulting library can be used on other devices as well.

The implementation can be found in the project folder “libs” and the module is called “trigger.c”. The following pseudo-code describes implementation of the trigger detection.

```
Iterate through all samples
    if the preceding sample was smaller than the trigger level
    and the current sample is greater
        if the index of the current sample is closer to the median
```

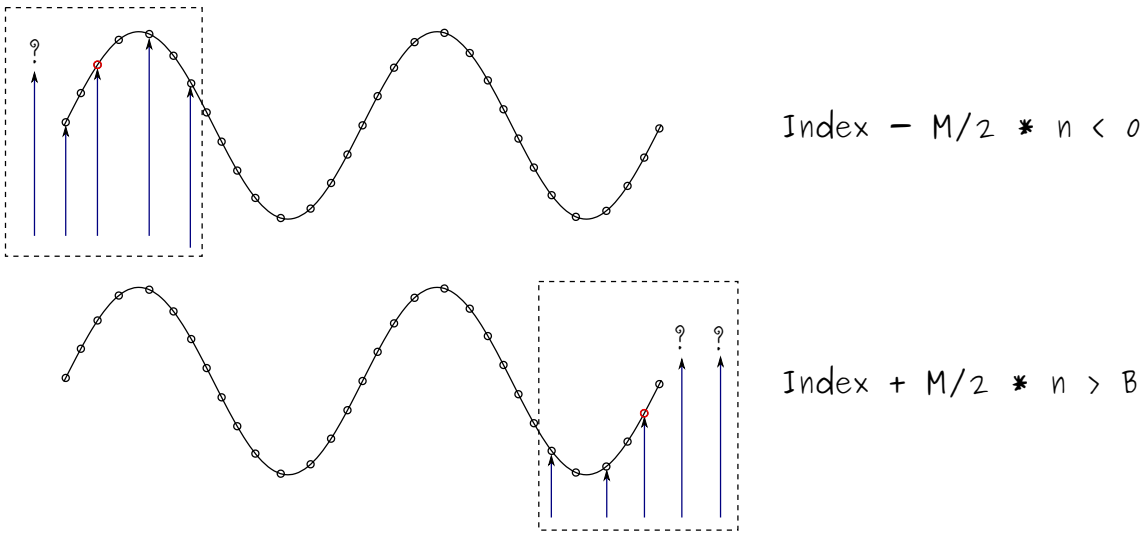

Figure 14.4: exceptional cases: display window overlapping a block

```
than the best trigger found
    the new best trigger is the index of the current sample
```

Sample selection is done fairly simple:

M = Number of samples to display
n = (How many samples to skip each time)+1

Find the start position (i.e. trigger index - $\frac{M}{2}$*n)
Find the stop position (i.e. trigger index + $\frac{M}{2}$*n)

```
if the start is below zero
    set start to zero
if the stop is greater the block length
    set start to zero

Iterate over the samples from start to stop incrementing by n
    copy sample to display array
```

Note that depending on the data type of the source (char, short, signed, unsigned) the interpretation of the comparison operation matter and therefore there are four different implementations of the trigger.

Using AIDL for Service-Activity communication there is another restriction. Apparently AIDL does not support the type short. To avoid this problem we used “int” instead of “short”. Since the display array is not that large the memory overhead for using int instead of short is minimal.

Trigger detection and sample selection are found in the module “trigger.c” which resides in the “jni” folder of the Android project.

Chapter 15

USB Data Sources

15.1 Overview of Data Sources

An USB data source is a data source that transmits payload over the USB interface. In our project we have two distinct modes that require a USB data source.

A continuous mode that displays data samples to the user in a fluid 30 frames per second fashion. Samples are continuously acquired even during processing and rendering.

We also implemented a single shot data source that resolutes at 48[Msps] per channel. The data source only sends new samples when a certain trigger value is reached and then only sends 1024 data samples per channel (restriction due to FX2 memory size).

Both the continuous and the single shot source share common native functionality and are linked against libusb. Implementation differs slightly in transfer buffer size and multi-buffering. As mentioned earlier, the continuous source requires several buffers in order to not lose any samples. Buffer size depends on the desired frame rate of the display. The single shot source requires only one buffer with the exact size of 2048 bytes (since the FX2 Endpoints are exactly 2048 Bytes in the used configuration).

Either source communicates back to the CPLD through the FX2 to issue commands for the hardware (gain, trigger mode, trigger polarity, trigger level, trigger channel). Communication from the Android application to the CPLD is described later on in this section.

As illustration [15.1] shows, there are two libraries for both type of acquisition modes. The library "liboscicontinuous.so" holds functionality to drive the oscilloscope in continuous mode. Whereas "liboscisingleshot.so" provides Java with the functionality for single shot data acquisition.

Through the Java Native Interface (JNI) the native methods written in C are accessed. The native methods also need to issue a callback into Java when new samples are available which is also implemented through JNI.

Figure [15.2] illustrates the architecture of the two USB data sources and their communication through JNI. It also illustrates the placement of the sources relative to the entire application.

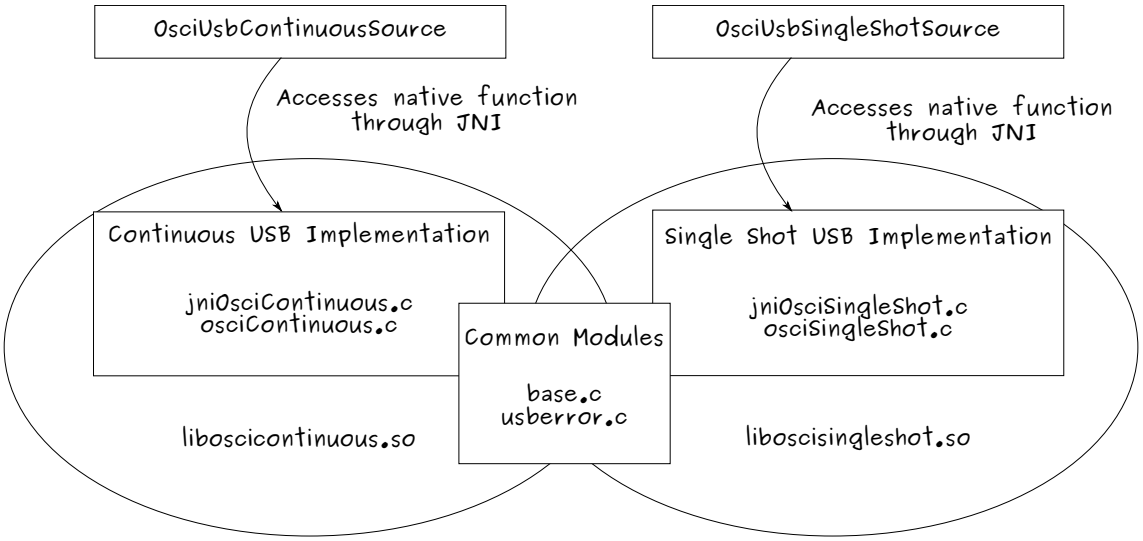

Figure 15.1: overview of native modules

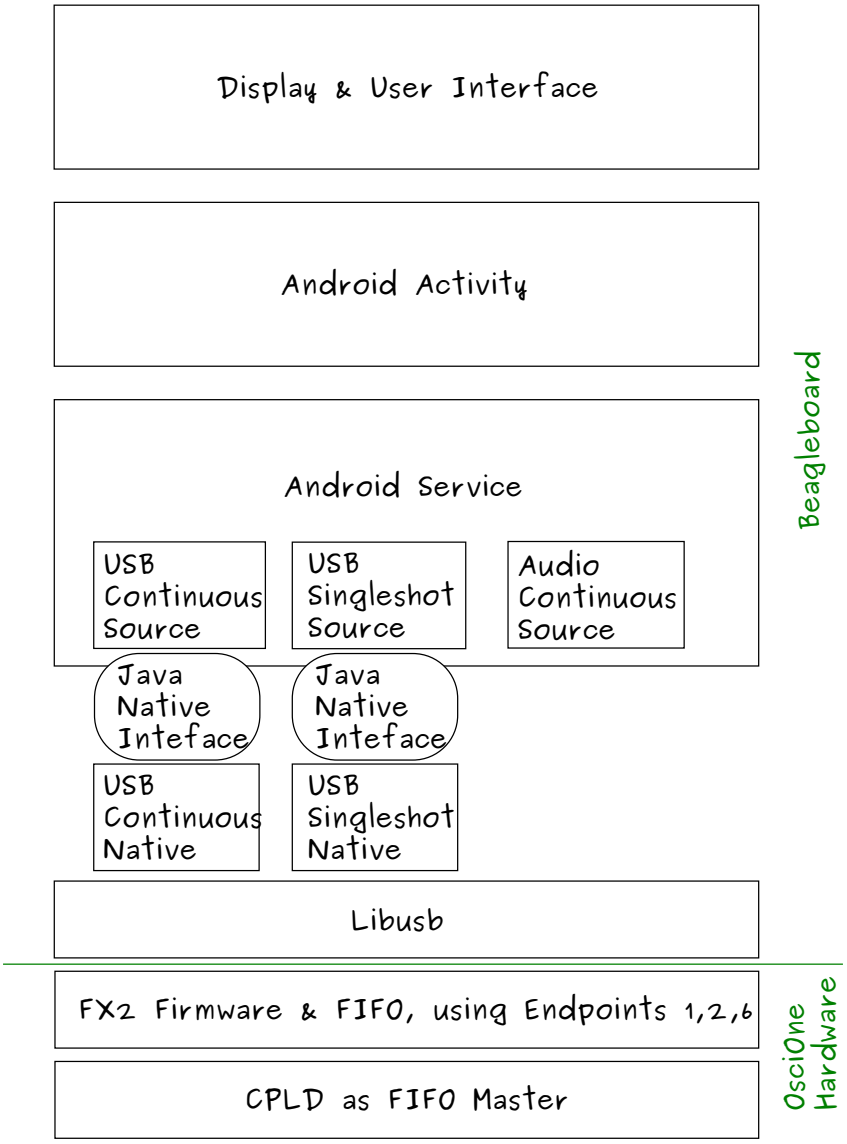

Figure 15.2: data sources in relation to the entire application

In the following two sections the native implementation of the two USB data sources are discussed in detail. However, it makes sense to discuss common functionality of the two types of data sources at first.

Both data sources need to initialize libusb, acquire a handle for the FX2 device and set up CPLD communication.

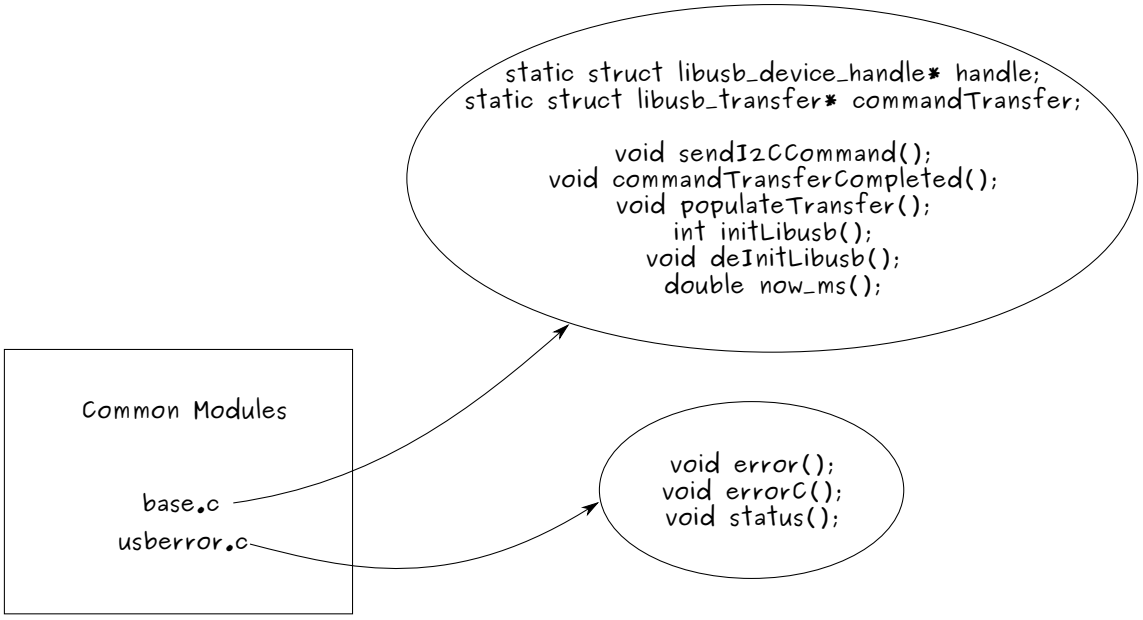

Figure 15.3: common usb modules

In figure [15.3] you can see two modules that are used by either data source. The module “error.c” has been created to ease debugging. It offers methods that convert specific libusb status messages into human readable text.

Whereas the module “base.c” offers functionality for libusb initialization an de-initialization. The function initLibusb() acquires a handle for the FX2 device. The helper function “populateTransfer” fills a bulk transfer for the provided endpoint, buffer and callback function and uses the previously acquired handle.

The CPLD handles certain commands to operate in different modes, change trigger channels, change trigger levels and polarity. Commands are constantly one byte long and transmitted by the FX2 I2C interface. Furthermore, the command is transmitted by the Android application using libusb. The FX2 firmware repeatedly checks endpoint 1 OUT for new command data and “routes” it through the I2C bus upon reception.

Illustration [15.4] shows the data flow of a command originating from the Android application.

Since both data sources, continuous and single shot, need to at least access to gain configuration of the front-end, the functionality for this feature has been placed in the shared module “base.c”.

The function “sendI2Ccommand(unsigned char command)” populates the transfer “commandTransfer” which previously has been allocated in “initLibusb”. The populated transfer is sent to OUT endpoint 1 of the FX2 device. It is not encouraged to call any function of the base module before “initLibusb” has been called. However, there are checks if libusb has been initialized in all other functions that need previous initialization of libusb.

The function “deInitLibusb()” de-initialzies libusb and frees the handle structure. The function “now_ms()” is a convenience-function that returns the system time in milliseconds in order to measure delay for debugging.

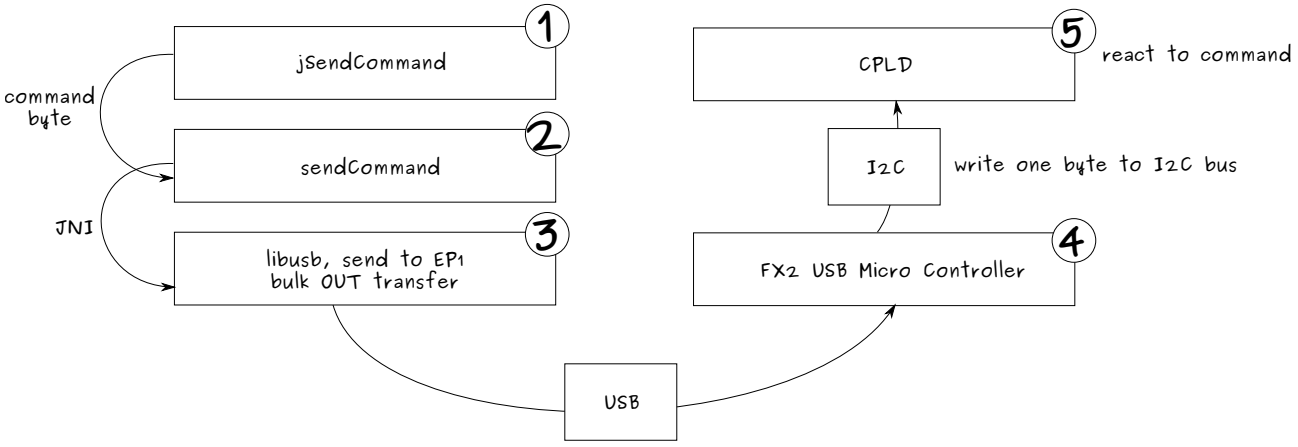

Figure 15.4: data flow of CPLD control commands

15.2 USB Continuous Data Source

In continuous mode the application constantly collects data samples from the front-end. The samples are being triggered, selected and displayed. After displaying, new samples are requested.

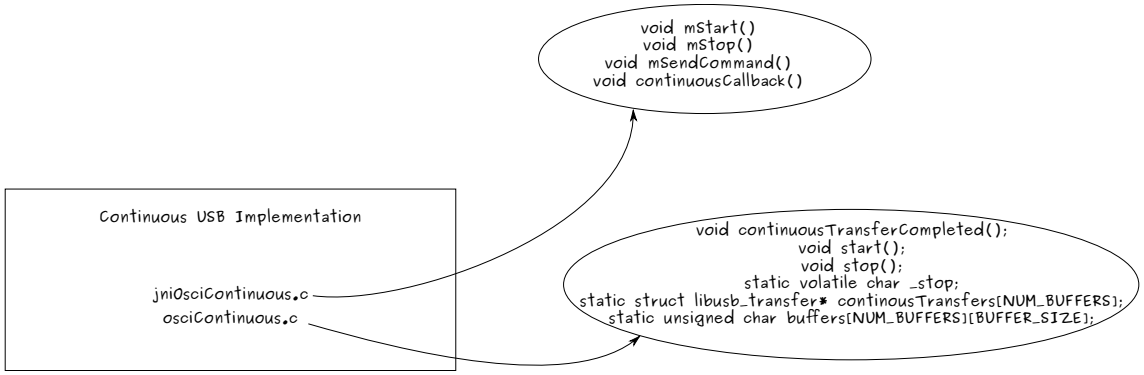

Figure 15.5: specific modules for continuous mode

Figure [15.5] shows the contents of the two C-modules.

Detailed information about the mechanics of libusb can be found in our previous technical report[1] as well as on the libusb website

`http://libusb.sourceforge.net/api-1.0/`

All modules we are using are written in C. As done in the preceding project we are using bulk transfers to transmit data over USB. With libusb this is done by first allocating, populating and finally submitting a "libusb_transfer" structure. During runtime, the application is in a blocking state and handles USB events. As soon as an event is triggered, a transfers callback function will be executed.

As mentioned in chapter [11] it is important to consider timing to not loose samples. As you can see in Illustration [15.5] there are several buffers allocated for transmission. The amount of buffers is defined by "NUM_BUFFERS" and is as of this writing set to four. Libusb is able to have several transfers at the same time in a queue. By issuing a transfer using the libusb function "libusb_submit_bulk_transfer(struct libusb_transfer* transfer)" a new item will be queued to libusb.

[1] available on http://android.serverbox.ch

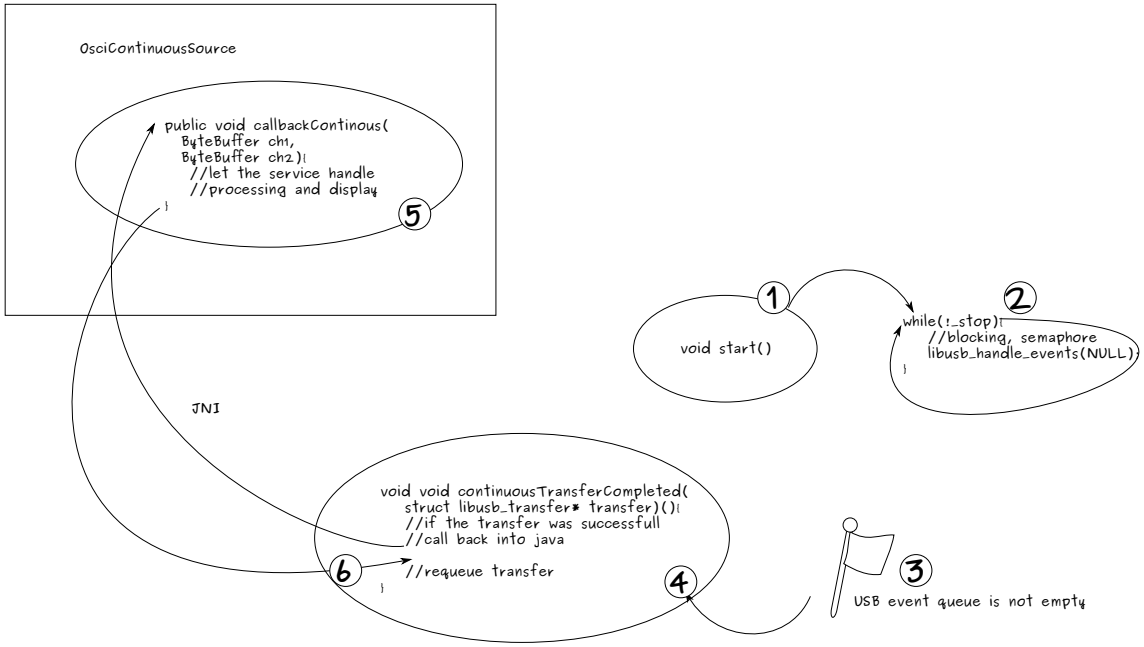

Figure 15.6: program flow for usb data handling

Illustration [15.6] shows the application flow.

The application is started by letting the Android service call “start()” through JNI. “start()” queues multiple transfers holding equally sized buffers. The main loop (2) handles USB events and lets the transfers call their callbacks upon completion or error (3). The transfers buffer is then passed to the Java callback function in the source class (4)+(5). In this case the Java-callback function is located in the “OsciContinuousUsbSource” class. The buffer is processed (triggering, selection) (5) and the method jumps back into the initial USB callback function (6) where the handled transfer is re-submitted. Until the service issues a stop request this process is repeated over and over again.

By queuing multiple transfers at the start (1) of the application we are able to continue triggering, selecting and displaying a buffer while at the same time the next libusb transfer is already being populated.

As long as we guarantee that the average time to process a buffer is smaller than the time it takes between two buffers being ready to use we rest assured that no samples are lost.

The CPLD writes 16 bit values to the slave FIFO of the FX2 whereas the high 8 bit are samples from channel 1 and the low 8 bit are samples of channel 2. Therefore, channel 1 and channel 2 have to be separated and copied into buffers each of the size BUFFER_SIZE/2. This is done in the “continuousTransferCompleted()” function (4).

15.3 USB Single Shot Data Source

Figure [15.7] shows the contents of the two native C-modules.

The USB single shot source is implemented in a very similar manner as the continuous data source with two exceptions.

First, there is only one transfer queued with the exact same buffer size as the FX2’s endpoint buffers. It may take an undefined amount of time until the CPLD actually registers a trigger and answers to the submitted IN bulk transfer.

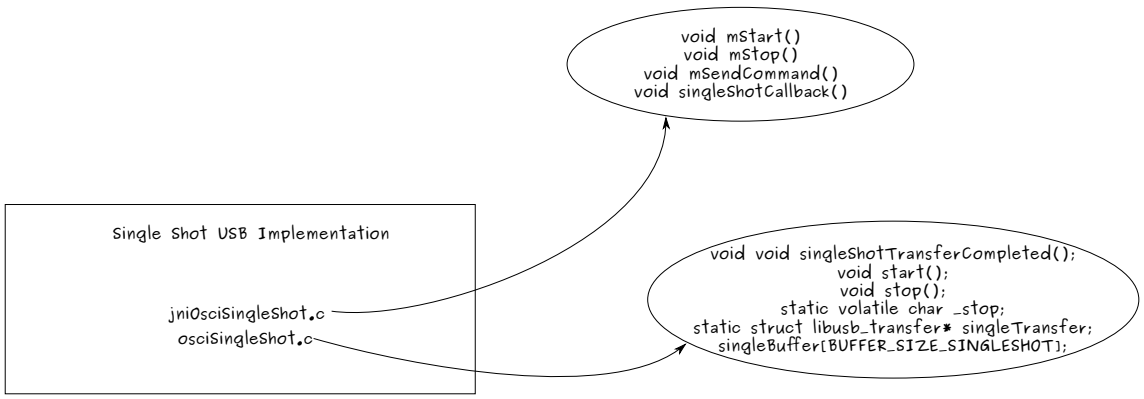

Figure 15.7: specific modules for single-shot mode

Second, the CPLD uses the slave FIFO to send a short packet to assure that there are no more samples in the endpoint buffer. This has to be checked in the "singleShotTransferCompleted()" callback function accessing the transfer's "actual_length" field. If it is not exactly "BUFFER_SIZE_SINGLE_SHOT" long, we know that a short packet has been submitted. Therefore the transfer is re-queued. When the transfer is completed, the next time the "actual_length" field will contain the value of the expected length or otherwise something went wrong.

A callback into Java will be issued through JNI passing the received single shot samples. In this case the overlaying class "OsciUsbSingleShotSource" will handle the callback.

15.4 Audio Source

To be able to use the application on a mobile device we implemented an Audio Source. With the Android "AudioRecord" class the implementation was fairly simple.

Refer to the "OsciAudioSource" class in the sources. It is located in the package "fhnw.oscione.service".

15.5 Generator Source

To test the application we have also implemented a sine waveform source. The sample values are generated in native C using JNI. The generator source example also helps a developer to easily understand how to implement a source that needs native components. You can find the Java part of the source code in the class "OsciGeneratorSource" in the "fhnw.oscione.service" package. The native part is found in the jni folder of the application's source code and is called "generator.c". The native library is compiled using the Android Native Development Kit (NDK).

Chapter 16

Oscione Activity

The Activity displays data samples to the user and holds user interface components. It aims to give the user the experience as he or she is using a commonly found oscilloscope.

16.1 Overview

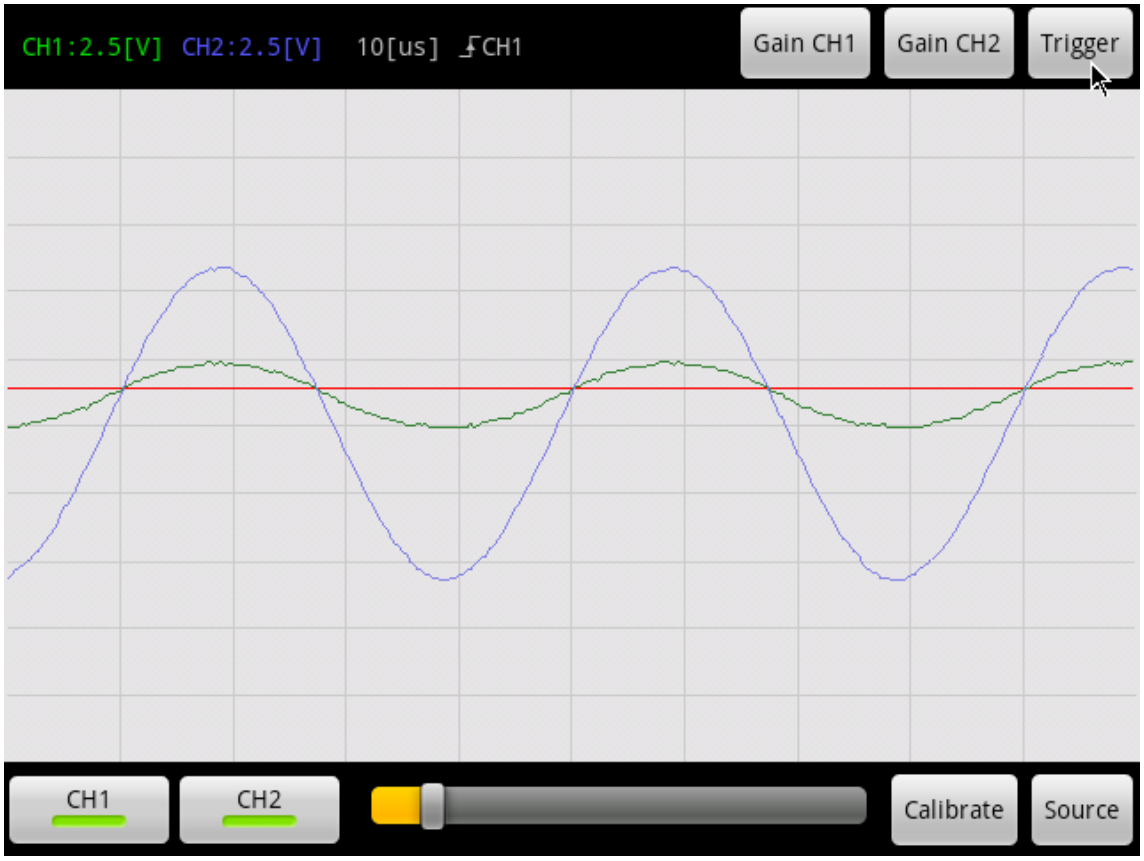

Figure 16.1: screenshot of the activity

The Activity implementation is, thanks to the excellent Android framework, a very straight forward piece of Java code. The layout editor in Eclipse is of great help for placing and positioning components as for instance buttons or images.

However, the OpenGL display requires further understanding of graphical programming and therefore requires some experience of the developer. To offer the user a fast updating display we have decided to use the features of OpenGL. After all, we aim to give the user a constant frame rate of 30 frames per second so that the UI feels fluid and responsive. Similar as in the preceding project[1] we use vertices to draw lines between acquired data samples.

As you see in [16.1] there is also a grid implemented using OpenGL.

Furthermore the Activity displays Pop-Up menus to enable source selection, gain configuration and trigger configuration. The "Seekbar" element on the bottom of the application changes the time division of the oscilloscope. The "Toggle-Buttons" CH1 and CH2 selectively disable or enable display of the corresponding channel.

The button "calibrate" performs a software calibration of the data source. Since it is not entirely possible to create a perfect analogue front-end there is usually an offset found in the sampled signal. In order to correct this offset and therefore "calibrate" the source the average value over the samples can be calculated, saved and subtracted from the signal before it gets plotted. When "calibrate" is pressed the average value of the next arriving samples to display is calculated for each channel and is subtracted.

By clicking on the OpenGL surface view the trigger level is adjusted. Since only coordinates of the View can be acquired and not actual values, the level has to undergo a series of operations in order to be used as an actual trigger level value for the service. The simplest operation is a projection. Since the domain of the actual value is know the desired level can be calculated using a linear operation that includes the GLSurfaceView's actual height. However, if calibration is used, the mean offset has to be added to the trigger level again.

Upon creation the Activity binds to the service. As soon as the service has registered the Activity it will find available data sources and select the first available source. It then calls back into the Activity and initializes the following UI specific settings that are dependent on the data source selected:

- The number of points that are to be plotted each time
- The dimensions of the y-axis depending on the data type
  -2^{15} to $2^{15} - 1$, if short type
  -2^{7} to $2^{7} - 1$, if byte type
- An array of values that contain possible amounts of samples to skip for time division setup
- An array of Strings that contain names for the time division labels; these are corresponding to the value array above.

If the data source is of the kind "OsciIoSource" then the following additional values have to be taken into account:

- An array of commands that can be sent to the CPLD to adjust the current gain
- The labels for the selection pop-up that describe the gain values given by the previous array
- For both channels, an array of float values that contain correction factors for each gain setting

There is a certain number of features that would be very nice to be implemented in the Activity. For instance a cursor and a user defined offset can very much enrich the application. Cursor calculation would not be that difficult to implement. However, since we are using the mouse input to set the trigger level the question arises how to let the user set a cursor in an uncomplicated manner. One possibility would be to create a mechanism of estimating and evaluating mouse pointer proximity to a component inside the GLSurfaceView. Using some visual indication of the currently selected Component by "highlighting" would be a perfect way to make the oscilloscope feel intuitive. Also user-offset implementation is straight

[1] http://android.serverbox.ch

forward on the logical level (just add another offset value similar to the calibration feature). But again the real challenge for this feature is the user interaction.

Another common feature found in oscilloscopes is a hold-off. Implementation of logic would rest in the service and only involve one more variable which is simply subtracted from the trigger index before sample selection is executed ("doCallback()" method of the service). Thus difficulty is not found in logical implementation but rather in smart user interface integration.

Furthermore, a very nice feature would be data sinks. Instead of only showing samples to the user, a logging functionality could be implemented. Using a similar approach as done for data sources the data sink integration could be based on an "OsciSink" interface. Possible sinks could be file streams, TCP/IP, Bluetooth, etc... The service would then again take care of calling the user selected sink and issue commands to the current source that implemented the OsciSink interface. With such a functionality, it would be possible to use the OsciOne also as data logger which could be utilized even in an environment where UI is not of necessity.

Part IV

Appendix

Appendix A

License Overview

The entire source code as well as the hardware layout of the developed front-end are available under the GPL (General Public License). Further development of the project by any interested developer or engineer is highly desired.

As authors of the project we wanted to create a project that is freely available and free of any proprietary licenses. We believe that the best applications originate from open source projects.

During development we used the following tools:

- Android SDK
- Android NDK
- Eclipse IDE
- g++, gpp compiler
- Android arm compiler (bionic libc)
- Ubuntu OS (as development environment)
- NFS Server
- GIT
- Keil U-Vision (Arm IDE for Cypress FX2)
- Xilinx ISE WebPACK Edition
- GHDL (VHDL simulation)
- GTK Wave (Waveform viewer)
- EAGLE (Layout tool, Light version)
- LTspice (Spice simulation)

While not every tool is open source, all of those tools are available free of charge. EAGLE Light limits use only for non-profit projects.

Appendix B

Android Project Layout

B.1 Java Classes

B.1.1 package fhnw.oscione

OsciOne This is the Activity class. It handles UI and communication with the Service. It holds references for instances of the classes found in fhnw.oscione.UI.

B.1.2 package fhnw.oscione.core

Trigger The class "Trigger" loads the native triggering library and offers access to it to other classes in the project.

B.1.3 package fhnw.oscione.service

Callable This is an interface that is implemented by OsciService. There is only one function to be implemented: "doCallback(OsciSource src)", see OsciService for detail.

FxLoader This is a helper class to load the FX2 hex image to the USB micro controller.

OsciSource This is an interface that defines a set of function a data source has to provide in order to be able to let the oscilloscope interact with the given source.

OsciIoSource This interface extends OsciSource and defines a couple of functions that are only specific to data sources that communicate back to the hardware source. For example the USB continuous source needs so send commands to the CPLD to adjust gain settings.

OsciAudioSource, OsciUsbContinuousSource, OsciUsbSingleShotSource, OsciGeneratorSource All these classes are data sample sources and all of them at least implement OsciSource. Responsible for initializing, de-initializing and controlling the data source below. Hold a reference to a Callable Class (in this case OsciService). Have to continuously call the "doCallback" method of the referenced Callable class as soon as there are new samples.

OsciService Extends the Service class and implements Callable. Holds references to a set of possible sources. Is bound to the OsciOne Activity and processes samples held by all referenced OsciSources. Calls triggering functions from fhnw.core.Trigger.

IFullBufferCallback.aidl AIDL file that describes the interfacing functions of the Activity which can then be called by the Service. Used to retrieve user input such as the trigger level. Also used to transmit data samples to the Activity to be drawn.

ISamplingService.aidl AIDL file that describes the interfacing functions of the Service. Can be called by the Activity. Usually used to inform OsciSources about updates or events.

B.2 Native Helper Functions

trigger.c Holds the native implementation of trigger finding and sample selection. Implementation is discussed in the chapter "Trigger and Sample Selection"

vertexcopy.c Holds a simple function to copy the sample array to the vertex buffer used by OpenGL

generator.c An example source that generates a sine wave.

Appendix C

FX2 Firmware

```
void TD_Init(void){// Called once at startup
  // set the CPU clock to 48MHz
  CPUCS = ((CPUCS & ~bmCLKSPD) | bmCLKSPD1) ;
  SYNCDELAY;

  // set the slave FIFO interface to 48MHz, default is 0x80; int. clocked
  IFCONFIG = 0xE3;//INTERNAL 48 MHZ IFCLK
  SYNCDELAY;

  REVCTL = 0x03;   //Revision Control, enable operations
  SYNCDELAY;

  //invalidate unused eps
  EP4CFG = 0x00;
  EP8CFG = 0x00;

  //reset routine
  FIFORESET = 0x80; //nak incomming requests
  SYNCDELAY;
  FIFORESET = 0x02; //reset ep2
  SYNCDELAY;
  FIFORESET = 0x04; //reset ep4
  SYNCDELAY;
  FIFORESET = 0x06; //reset ep6
  SYNCDELAY;
  FIFORESET = 0x08; //reset ep8
  SYNCDELAY;
  FIFORESET = 0x00; //resume
  SYNCDELAY;

  //EP2 FIFO Config
  EP2FIFOCFG = 0x0D; //AUTOIN, ZEROLENIN, WORDWIDE 1
  SYNCDELAY;

  //EP6 same
  EP6FIFOCFG = 0x0D; //AUTOIN, ZEROLENIN, WORDWIDE 1
  SYNCDELAY;

  //EP2 Config Quad Bulk IN 512
  EP2CFG = 0xE0;
  SYNCDELAY;
```

```
  //EP6 same cfg
  EP6CFG = 0xE0;
  SYNCDELAY;

  //EP1 for command messages
  EP1INCFG = 0xA0;
  SYNCDELAY;

  EP2AUTOINLENH = 0x02; // Auto-commit 512-byte packets
  SYNCDELAY;
  EP2AUTOINLENL = 0x00;
  SYNCDELAY;

  EP6AUTOINLENH = 0x02; // same as ep2
  SYNCDELAY;
  EP6AUTOINLENL = 0x00;
  SYNCDELAY;

  //set polarity of FIFO Control Pins
  FIFOPINPOLAR = 0x38; //inv PKTEND, SLOE and SLRD
  SYNCDELAY;

  EP1INBC = 64;
  SYNCDELAY;

  EP2BCH = 0x02;
  SYNCDELAY;
  EP2BCL = 0x00;
  SYNCDELAY;

  EP6BCH = 0x02;
  SYNCDELAY;
  EP6BCL = 0x00;
  SYNCDELAY;

  EZUSB_InitI2C();//init i2c
 }

void TD_Poll(void){// Called repeatedly while the device is idle
    //Handle EP1
    //incomming
    if(!(EP1OUTCS & bmBIT1)){
        EZUSB_WriteI2C(I2C_ADC_ADDR, 1, EP1OUTBUF);
        SYNCDELAY;
        EP1OUTBC = 0;//re-arm out
        SYNCDELAY;
    }
}
```

Appendix D

Hardware part list

Description	Farnell Ordering Number	Price (CHF)
Xilinx coolrunner 2 CPLD, 128 macrocells, vq100 package	1605829	12.10
Cypress FX2 USB micro controller (CY7C68013A), 56 pin package	1269134	25.90
2x analogue-to-digital converter: ADC08100 (National Semiconductor)	8180679	24.2
2x operational amplifier: OPA2889 (Texas Instruments)	1603400	3.65
2x analogue switch: MAX4522 (Maxim)	1422328	6.20
inverse charge pump, ICL7660 (Maxim)	9724710	5.15
1.8V voltage regulator	1564761	0.88
3.3V voltage regulator	9485813	3.55
quartz crystal (24Mhz)	1368794	1.7
USB-plug (type B)	1642035	2.45
	total:	**85.80 CHF**

Furthermore,

- 2x female coaxial plugs
- 2x three pin DIP switches (for AC/DC coupling)
- 2x choke/inductor (around 680uH)

As well as 70 SMD capacitors (package 0805), and 43 SMD resistors (package 0805).

The whole part list is also available in HTML format in the EAGLE Layout sources[1].

[1] available on http://web.fhnw.ch/technik/projekte/eit/Fruehling2010/DiCerRud/

Appendix E

Testbench

The Xilinx WebPack ISE on Linux does not ship with a ModelSim[1] Simulator. However, you can still plot signal waveforms from your VHDL code under Linux. This tutorial will show you how:

First of all, make sure that you have Xilinx ISE WebPack installed on your system. The free ISE WebPack Edition can be downloaded from the Xilinx website[2].

Download and install GHDL from the offical homepage[3]. GHDL is a free and complete VHDL simulator.

Also install gtkwave with

```
sudo apt-get install gtkwave
```

gtkwave will be used for signal plotting.

Create a new project in the Xilinx ISE, and write your VHDL code. For this tutorial we will be using a full adder entity (with an additional pwm output port) and name it adder.vhd. The source code of adder.vhd is as follows:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.All;

     entity adder is
       port (i0, i1 : in std_logic;
             ci     : in std_logic;
             clk    : in std_logic;
             reset  : in std_logic;
             sum    : out std_logic;
             co     : out std_logic;
             pwm    : out std_logic
            );
     end adder;

     architecture rtl of adder is
     signal counter : unsigned(7 DOWNTO 0) := (OTHERS=>'0');
     begin
```

[1] ModelSim is a popular HDL simulation tool, with the ability to plot signal waveforms

[2] http://www.xilinx.com/support/download/index.htm

[3] http://ghdl.free.fr/download.html

```
    clocked: PROCESS(reset, clk)
    BEGIN
    if reset='0' then
     sum <= '0';
     co <= '0';
     counter <= (OTHERS =>'0');
    elsif rising_edge(clk) then
     sum <= i0 xor i1 xor ci;
     co <= (i0 and i1) or (i0 and ci) or (i1 and ci);
     counter <= counter + 1;
    end if;
    END PROCESS clocked;

    pwm <= '0' WHEN counter<128 ELSE
           '1';
  end rtl;
```

In the project Hierarchy window, right click and choose "Create New Source". Select "VHDL Test Bench", name it adder_tb.vhd or similar. Click next and then assosciate it with the previously created entity (adder in our case). An automatically generated testbench has been added to the project. We need to make a few changes to this testbench, in order to simulate it with GHDL.

You will see that there is already a clock generating process in the testbench. However, this process has no defined ending and when simulating with GHDL, it will never halt. A simple trick is to define a boolean signal "run", and make the clock process dependant on the value of this signal. So, insert the following line in the signal declaration section of adder_tb.vhd

```
signal run : boolean := true;
```

. . . and change the clock process, so that it reads . . .

```
-- Clock process definitions
clk_process :process
begin
 while run loop
  clk <= '0';
  wait for clk_period/2;
  clk <= '1';
  wait for clk_period/2;
 end loop;
 wait;
end process;
```

The default clk_period is defined as "100us". Insert a blank space between the number and literal, otherwise GHDL will produce an error message. You may also want to change the duration of a clock period to 1 us.

```
 -- Clock period definitions
constant clk_period : time := 1 us;
```

Finally, uncomment the std_logic_unsigned library in the heading of adder_tb.vhd, as GHDL can not use it. Do not worry, you can still use unsigned types.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
--USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;
```

Now we can insert our stimulus in the already existing stimulus process ...

```
    -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 1 ms.
      reset <= '0';
      wait for 1 ms;
      reset <= '1';

      i0 <= '1';
      i1 <= '0';
      wait for 1 ms;
      i0 <= '1';
      i1 <= '1';
      wait for 1 ms;
      run <= false;
      wait;
   end process;
```

Do not forget to set the run signal to false, followed by a wait statement, at the end of the stimulus process. Now we are ready to begin the simulation with GHDL and gtkwave by issuing the following commands

```
ghdl -a adder.vhd
ghdl -e adder

ghdl -a adder_tb.vhd
ghdl -e adder_tb

ghdl -r adder_tb --vcd=wave.vcd

gtkwave wave.vcd
```

As soon as gtkwave starts, click on the "zoom to fit" icon, and drag signals of interest (like pwm) into the waveform. In the top left, you can select "uut", the instantiated unit under test, to see the internal signals of the tested entity. Right clicking on a signal let's you change it's data format. For instance, you might want to alter the data format of the counter signal by choosing data format → analog → interpolated. The counter signal is then plotted as an upcounting ramp. See figure [E.1] for a screenshot.

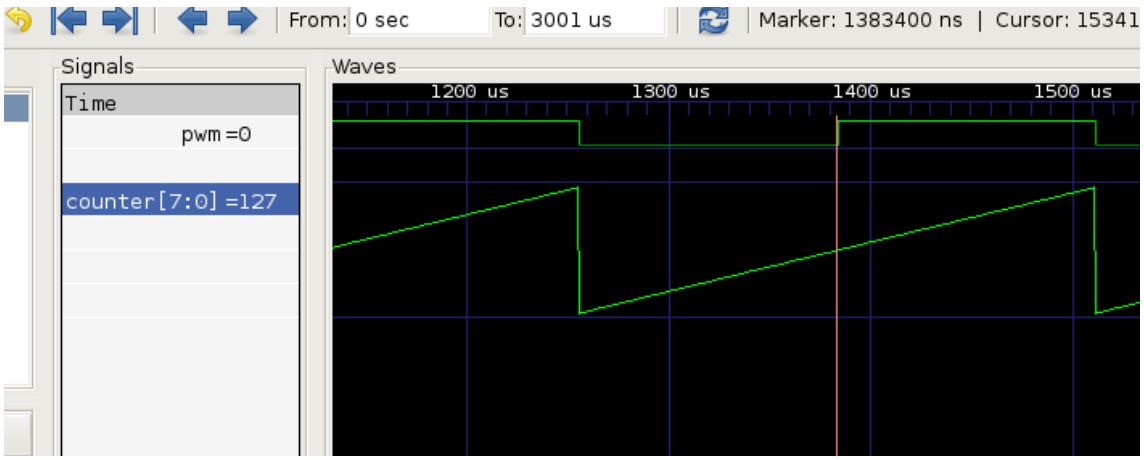

Figure E.1: plotting vhdl signal waveforms with gtkwave