Digital oscilloscope module with PC interface

Stephan Walter

<stephan@walter.name>

Department of Microengineering
Swiss Federal Institute of Technology Lausanne

Semester Project

January 14, 2008

Supervisor
Prof. Maher Kayal
EPFL / LEG

Assistant
Fabrizio Lo Conte
EPFL / LEG

Contents

Chapter 1

Introduction

The Master's project done by Fabrizio Lo Conte [10] presents a universal interface for the USB 2.0 bus. Various different modules can be connected to the interface card. The data received over the serial USB bus are converted to a parallel, 16-bit wide bus. The modules are addressed by a 16-bit parallel address bus. Data can be read from or written to modules in chunks of up to 1024 words.

This project builds upon this interface card to make a software-controlled DSO (digital sampling oscilloscope). It consists of the following elements:

- an analog circuit for amplifying the signal that is to be measured
- an analog-to-digital converter
- a digital circuit to temporarily store the digital data
- interface logic to communicate with the parallel data and address bus
- software running on a standard desktop PC that will collect and display the data in real-time

Figure 1.1: Universal USB interface (taken from [10])

Chapter 2

Previous work

2.1 Projects by others

The following list shows some existing work on designing a digital oscilloscope.

2.1.1 "Bitscope"

Author: BitScope Designs [3]
Performance: 2 channels – 40 MS/s

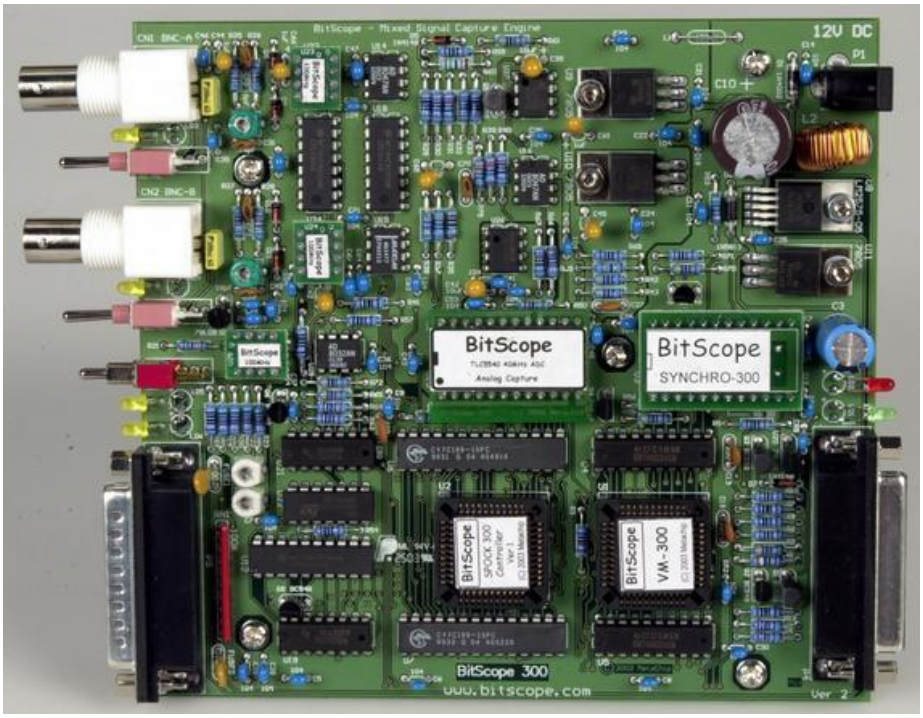


Figure 2.1: Bitscope 300

This oscilloscope is sold commercially in different variants for USD 550.– to 1600.–. The ADC is a TLC5540 by Texas Instruments, the sample buffer is a Micron MT 5C2568. These are controlled by a PIC microcontroller by Microchip. Interfaces to a PC by Ethernet or USB.

2.1.2 "PC-based DSO"

Author: "area26" [2]
Performance: 4 channels – 100 MS/s
This project consists of a main board with an Altera FPGA and up to four ADC boards with circuits of type MAX1449 by Maxim. The oscilloscope interfaces to a PC over Ethernet.

2.1.3 "DSOA Mk3"

Author: D. Jones [7]
Performance: 2 channels – 20 MS/s
The ADC is a Philips TDA8703, the sample buffer is an IDT 7202. There is a parallel (printer) port connection to a PC.

2.1.4 "Oscilloscopio digitale"

Author: L. Lutti [11]
The ADC is a Texas Instruments ADS830, the sample buffer is a CY7C199. These are controlled by a Xilinx CPLD.

2.1.5 "DSO"

Author: J. Glaser [5]
Performance: 4 channels – 80 MS/s
The ADC is a MAX1448 by Maxim, controlled by a Xilinx FPGA. Interfaces to a PC via USB. The analog input stage (see figure 2.2) is a good starting point for a new design and I have taken some inspiration from it.

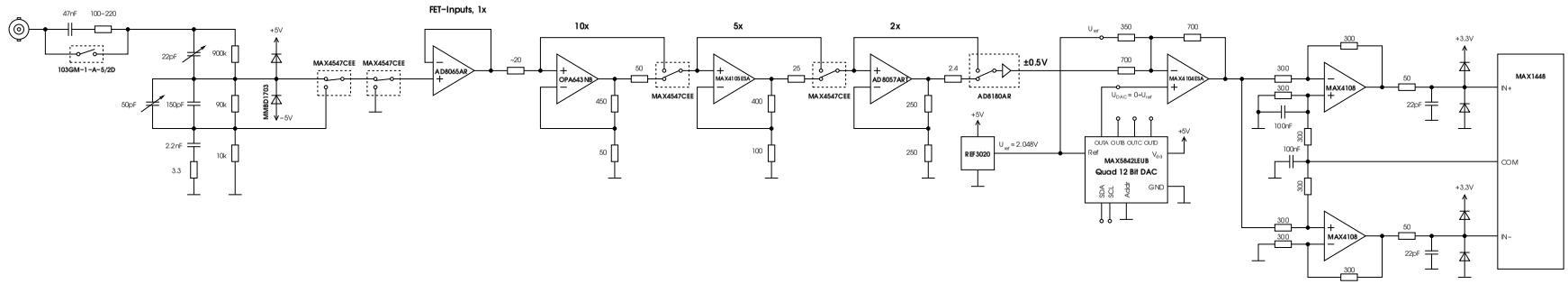


Figure 2.2: The analog stage of a digital oscilloscope designed by J. Glaser [5]

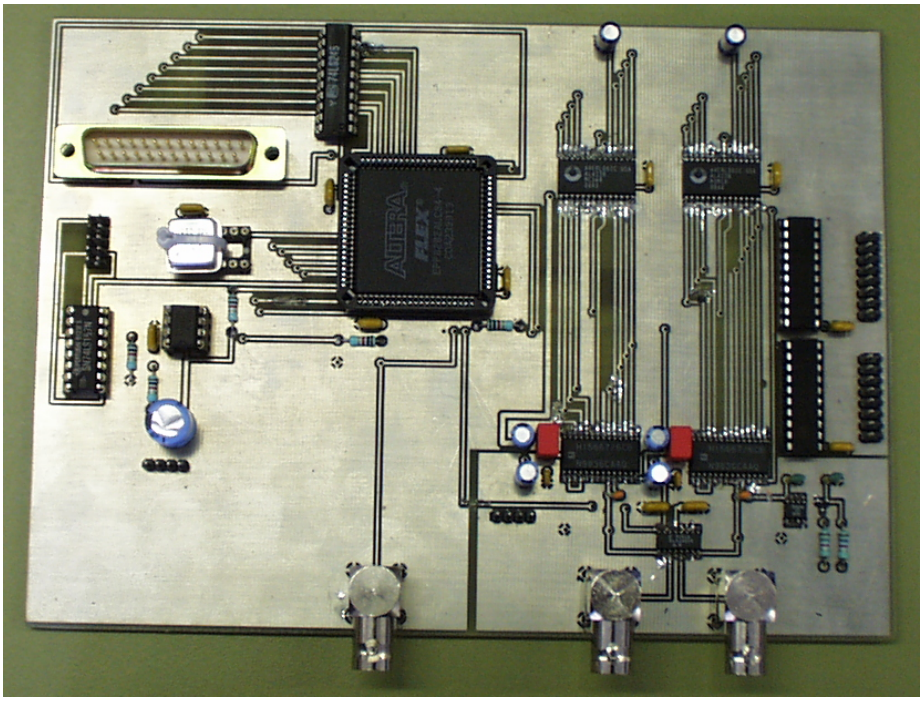


Figure 2.3: LDSO by T. Grocutt

2.1.6 "LSDO"

Author: T. Grocutt [6]
Performance: 2 channels – 50 MS/s
The ADC is an Intersil HI5667, which is controlled by an Altera FPGA.

2.1.7 "eOscope"

Author: "eosystems.ro" [4]
Performance: 1 channel – 40 MS/s
The ADC is a Texas Instruments ADS830, the sample buffer is an IDT 7201. An Atmel AVR microcontroller and a Xilinx CPLD are used for control. The device has an LCD screen and does not interface to a computer.

Figure 2.4: eOscope

Chapter 3

Specifications

3.1 Hardware specifications

After studying the performances of the designs shown in the last chapter, the following specifications have been established:

Sampling frequency	$\geq$	10MHz
Resolution	$\geq$	8 bit
Analog bandwidth (3dB)	$\geq$	100MHz
Input voltage range	$\geq$	$\pm$30V
Input resistance	=	1MΩ
Input capacitance	=	10 . . . 50pF
Coupling	selectable	AC / DC
Attenuation / amplification	selectable	
Supply voltage	=	3.3V or 5V
Power consumption	$\leq$	2W

Table 3.1: Hardware requirements

From these basic requirements, some others can be derived: for example, any operational amplifier acting on the signal must have a high enough slew rate so that it can follow the signal.

3.2 Software specifications

The software has two major functions: a) control the settings of the acquisition card such as frequency and attenuation and b) visualization of the samples in a diagram of time/voltage and possibly other information.

The detailed requirements will be defined later in section 5.1, as they depend on the hardware design.

Chapter 4

Hardware design

4.1 Analog section

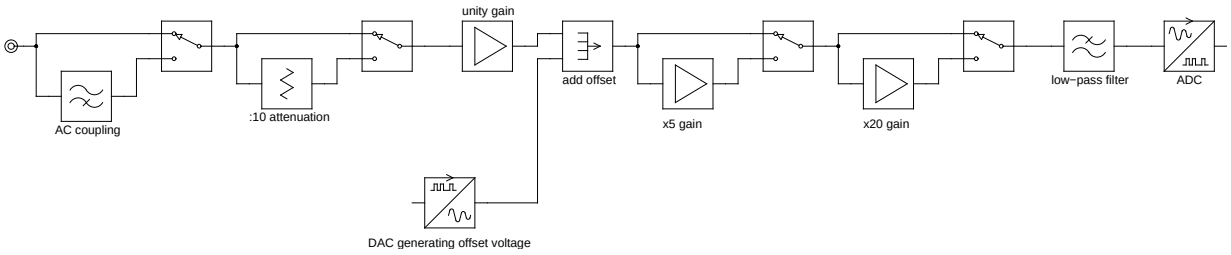


Figure 4.1: Overview of the analog section

An oscilloscope probe is connected to a BNC connector. From there, the signal to be measured goes through several different stages as shown on figure 4.1.

4.1.1 AC/DC coupling

The first stage of the oscilloscope is the AC or DC coupling. AC coupling is done by a capacitor C_{101} of 47nF. This capacitor can be by-passed by closing a relay (see figure 4.2). Although such a mechanical device can be bulky and consume a lot of current, it seems a better choice than a FET switch which would have to be driven at $\pm$30V.

For the relay, the model G6K by Omron was chosen for its small size, 3V operation and low coil current. This current of 30mA makes it possible to drive the relay directly from 74 logic ICs, provided the logic family has sufficient drive current.

When the relay is closed, the capacitor is discharged through the relay. In order to respect the relay's current limit, a resistor is placed in series with the coupling capacitor.

The coil of the relay ($R = 91\Omega$) is in series with a 20Ω resistor. This leads to a coil current of 30mA in the closed state. The Schottky diode is a protection against voltage spikes that can occur when removing the coil supply voltage.

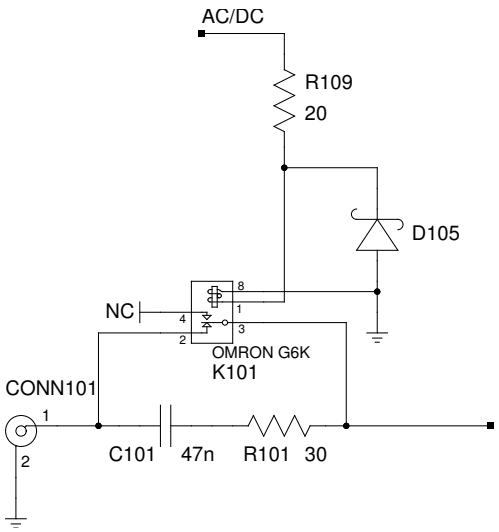


Figure 4.2: Coupling circuit

4.1.2 Attenuation and over-voltage protection

Now the signal is attenuated by a factor of 10, selected by a relay of the same type as above. The attenuation itself is achieved using a combined resistive and capacitive voltage divider for low and high frequencies, respectively. Figure 4.3 shows the attenuation circuit.

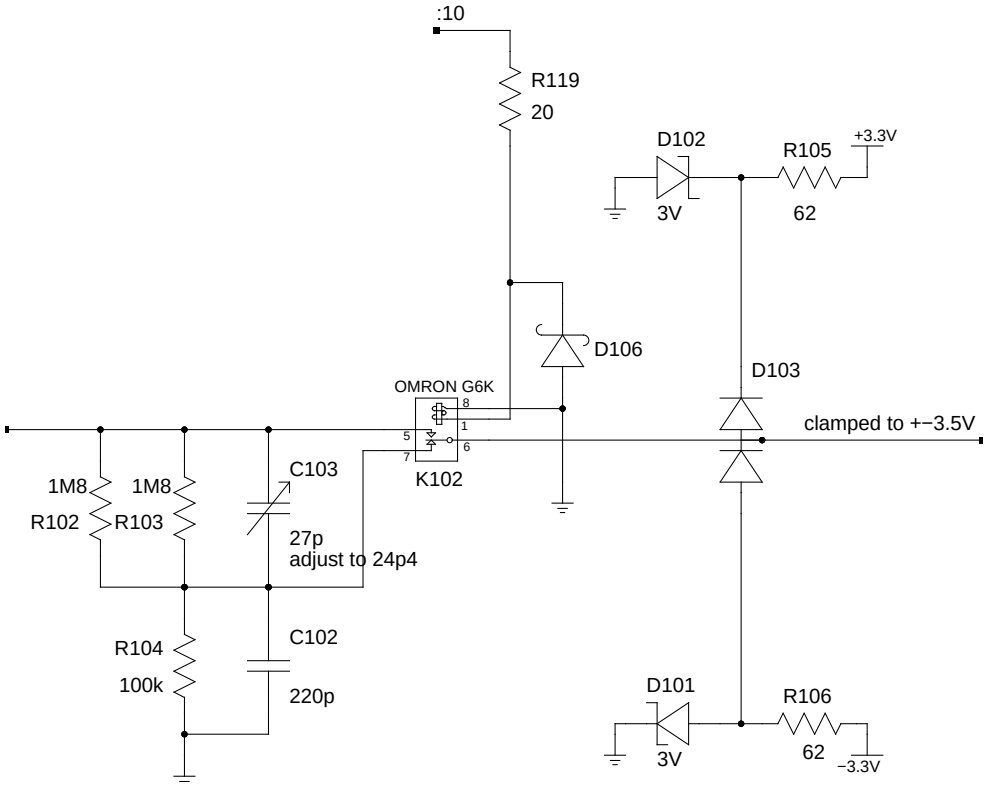


Figure 4.3: Attenuation and clamping circuit

Type	Bandwidth [MHz]	Amps/package	I_{ib}[pA]
AD8065	145	1	1
ADA4899	600	1	100000
LT1222	500	1	100000
LT1722	200	1, 2, 4	10000

Table 4.1: Input buffer op-amps

The values of $C_{102,103}$ and $R_{102...104}$ are chosen so that they result in a total input impedance of 1MΩ and about 25pF, values that are common in commercially available oscilloscopes.

The following stages must be protected against voltages exceeding ±3V. Sometimes, this is done by clamping the signal over two diodes to $\pm V_{supply}$. This means that the power supply must be able to compensate any over-voltage. In the present case, this would not be a good idea, as the power is supplied by a PC over the USB bus. A voltage surge might damage the components of the computer.

The solution adopted here is the use of Zener diodes. They are reverse biased with a constant small biasing current from the power supplies. The signal is then clamped by conventional diodes to the Zeners.

4.1.3 Input buffer

Now the signal is put through an op-amp of unity gain. The requirements for this amplifier are as follows:

- Compatibility with the ±3.3V power supply.
- Input and output voltage range of ±3V.
- 3dB bandwidth of at least 100MHz at unity gain, in order to satisfy the requirements given in section 3.1.
- Slew rate $> 60\text{V}/\mu\text{s}$. (see below)
- Low input biasing current I_{ib}.

Besides the bandwidth, the slew rate is an important AC parameter for an amplifier. The signal must be allowed to swing along the whole voltage range (from -3 to $+3$V) during one sample period (100ns at 10Msps).

Input biasing current is important as this is the first amplifier, which is directly connected to the input. Any biasing current influences the circuit we want to measure. Operational amplifiers often have I_{ib} in the range of several microamperes. Table 4.1 lists some candidates.

For the first design of the analog stage, the AD8065 by Analog Devices was chosen. This chip turned out to be difficult to obtain. Also, it tends to amplify high frequencies to much, as will be shown in section 6.1.

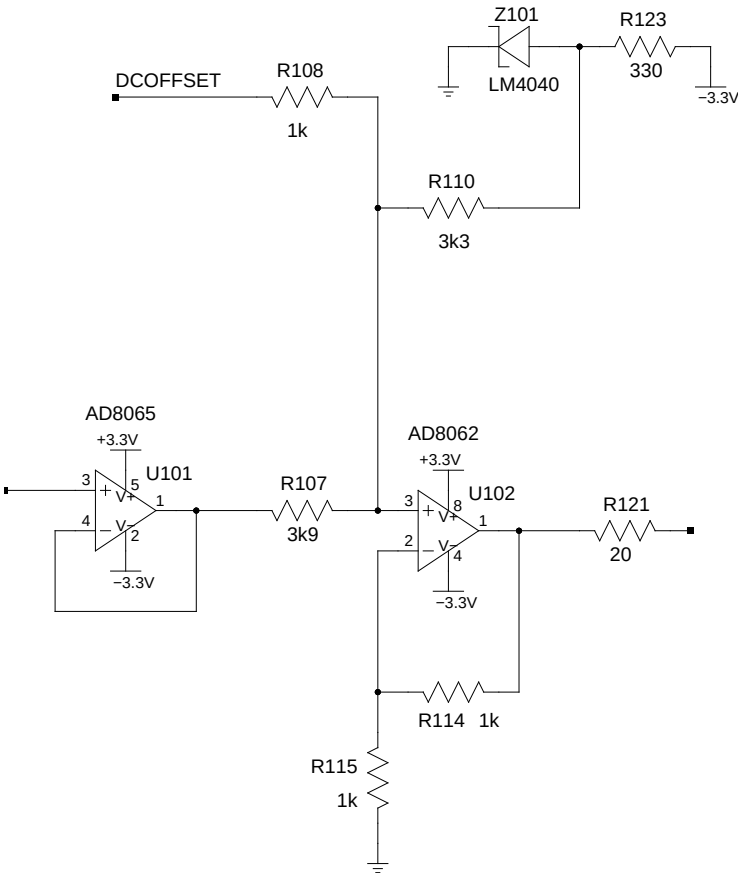


Figure 4.4: Buffering and offset circuit

4.1.4 DC offset

The second op-amp stage will add a selectable offset voltage to the signal, as well as limiting its swing for the ADC or the following gain stages. ADCs with a supply voltage in the order of 3V often have an input voltage swing of 2V centered around a reference voltage. This is also true for the ADC chosen here, which is presented in section 4.2.1. The offset voltage is generated by a DAC with an output range of 0 to 2.5V. This would make it impossible to have a negative offset, or "shift down" the signal. The solution is to use the amplifier in a summing configuration: the weighted addition of the signal, the DAC voltage and a negative reference voltage gives a great flexibility. The voltages are summed as follows:

$$V_{out} = 0.33V_{in} + 1.28V_{offset} + 0.39(-2.5\text{V})$$

The important requirements for the amplifier are: a gain bandwidth $>$ 200MHz and a slew rate $>$ 20V/µs. These parameters are obtained by the same reasoning shown for the first amplifier. Some candidates are listed in table 4.2

Type	Bandwidth [MHz]	Amps/package	Comments
AD8061		1	not rail-to-rail input
AD8062		2	not rail-to-rail input
LMH6609	900		not rail-to-rail
LMH6639	190		rail-to-rail
LMH6654	250		low noise, not rail-to-rail

Table 4.2: Op-amps for offset

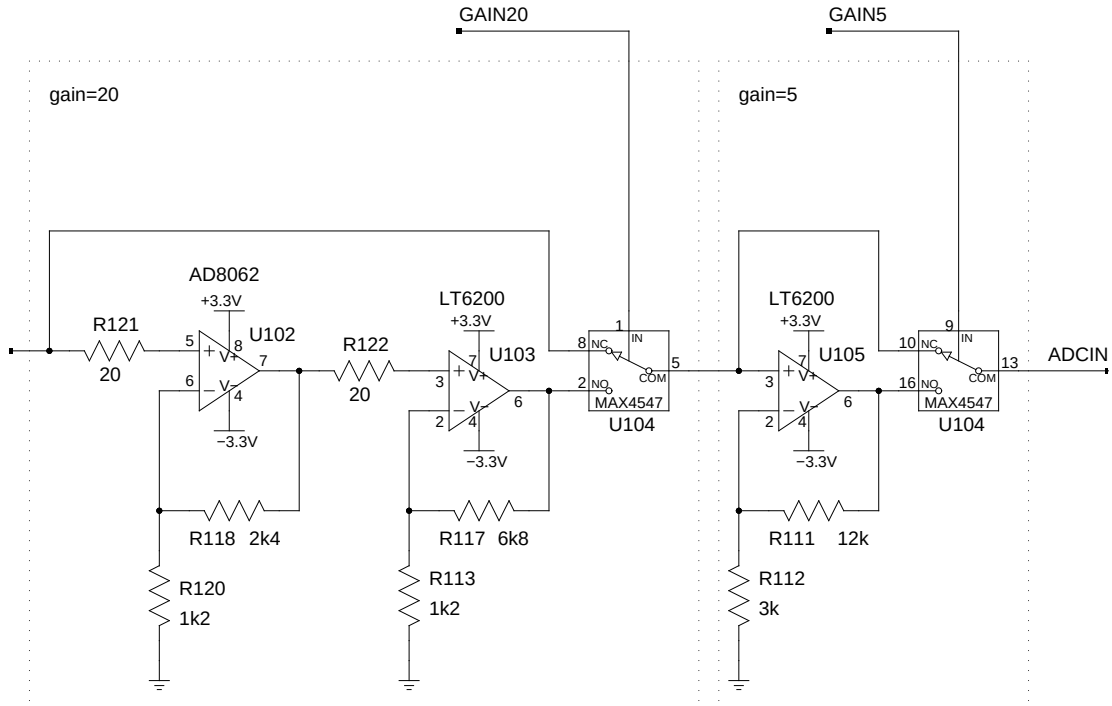


Figure 4.5: Gain circuit

4.1.5 Gain

The signal can be amplified using a multiplier of 5 or 20 or a combined multiplier of 100. The gain of 5 is achieved with a single operational amplifier with a gain bandwidth > 500MHz. A gain of 20 would be difficult for a single amplifier, so two operational amplifiers are used.

As can be seen on figure 4.5, the first amplifier has a gain of $1 + \frac{2400}{1200} = 3$ and the second $1 + \frac{6800}{1200} = 6.67$. For the lower gain, the type of the amplifier is the same as for the offset circuit.

For the two gain stages of 6.67 and 5, the LT6200-5 was used. It has a minimum stable gain of 5 (see table 4.3). Small resistors are placed in series with the op-amp outputs to isolate the outputs from the parasitic input capacitances of the next stage.

Type	Bandwidth [MHz]	Amps/package	Comments
LMH6733	1000	3	single supply, not rail-to-rail
LT1806	325	1, 2	unity-gain stable
LT1818	400	1, 2	unity-gain stable
LT6200-5	800	1	$G \geq 5$
OPA699	1000	1	limited voltage

Table 4.3: Op-amps for gain stage

4.2 Sampling

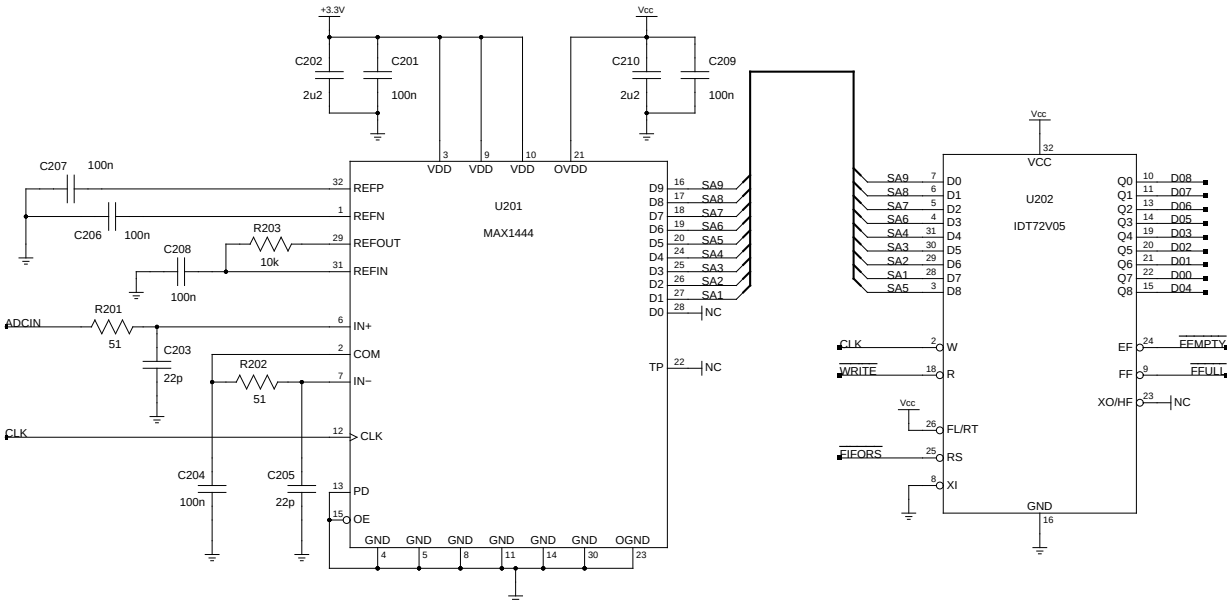


Figure 4.6: ADC and FIFO memory

4.2.1 ADC

Several companies offer analog to digital converters with the desired sampling rate. Important characteristics are the resolution and the maximum and minimum sampling rate (some converters have a minimum sampling rate of about 1MHz, which would be wasteful for acquiring signals of low frequency). Supply voltage is also an important concern as some circuits require different voltages for the analog and digital parts.

The converter chosen for this project is the Maxim MAX1444. It has a resolution of 10 bit and a sampling rate of up to 40MHz. Pin-compatible variants offer up to 80MHz.

4.2.2 Sample storage

The sample data is stored in a FIFO memory of type IDT72V06. This chip has a supply voltage of 3V, a capacity of 16k $\times$ 9 bit and an access time of 15ns. The high capacity allows for some leeway in the communication over USB. That is, a total of 16384 samples can be memorized before the FIFO memory is full and has to be read out.

The ADC will output a 10-bit value at each rising clock pulse plus $t_{DO} = 8$ns max, as shown in figure 4.7. Figure 4.8 show the timing diagram of the FIFO memory ($t_{DH} = 0$). If the same clock signal is applied to the ADC and to the $\overline{W}$ input of the FIFO, the output of the ADC changes between the periods DATA IN VALID of the FIFO.

On the FIFO, the write cycle starts on the falling edge of $\overline{W}$ if the full flag is not set. Data is not overwritten if the FIFO is full.

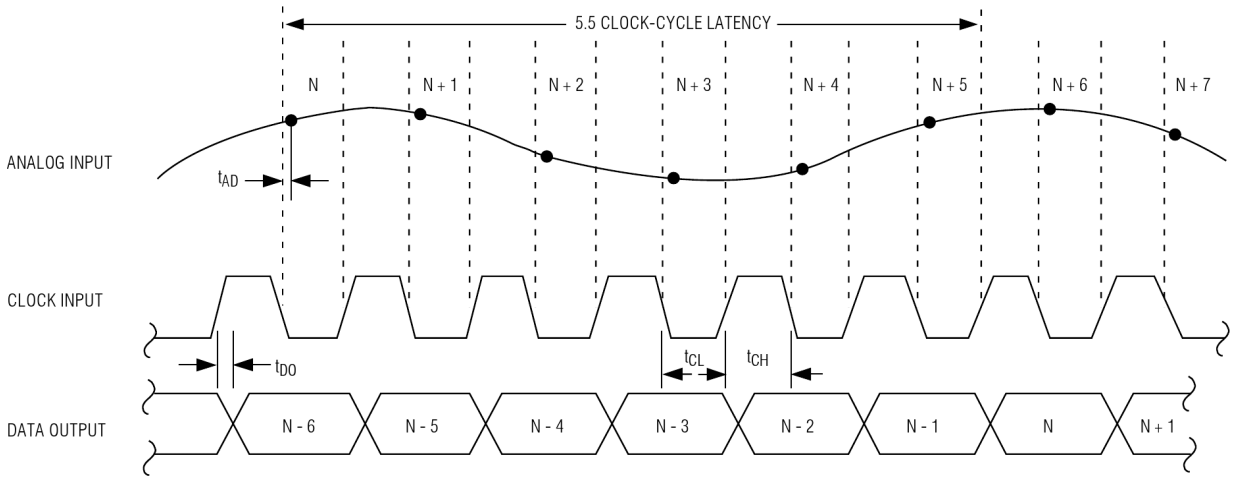


Figure 4.7: Sampling operation on the ADC. Excerpt from datasheet

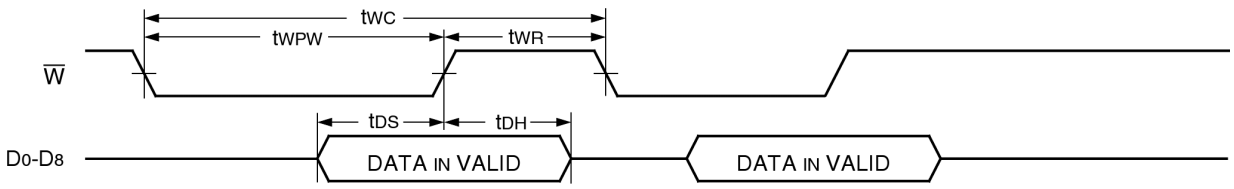


Figure 4.8: Write operation on the FIFO. Excerpt from datasheet

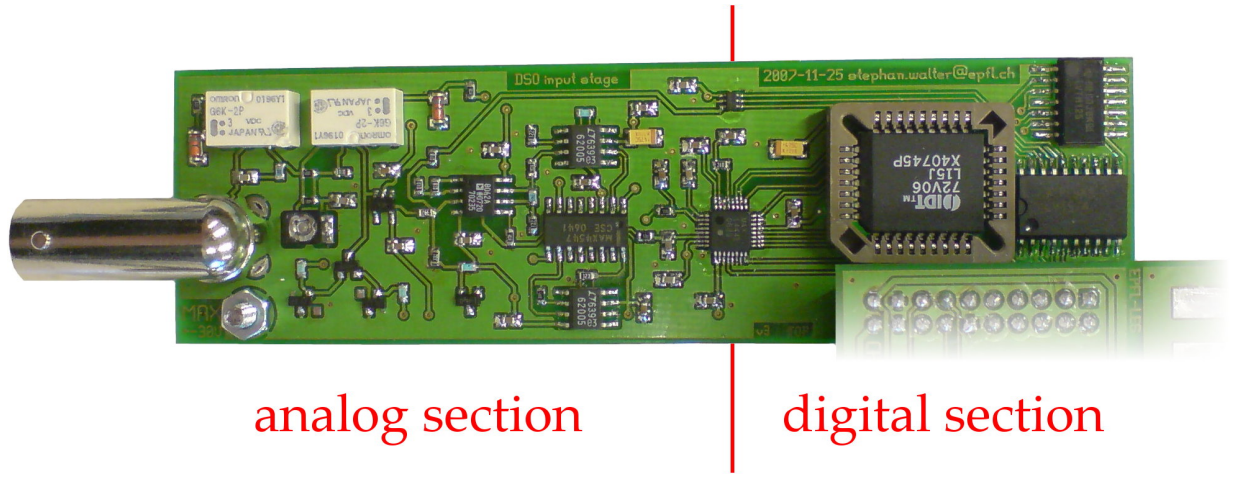


Figure 4.9: Acquisition circuit board

4.3 Clock

Both the ADC and the FIFO operate on the same clock. The clock source must be programmable by software in a range of up to 10MHz.

Important parameters the frequency accuracy and jitter. The importance of the jitter of the ADC can be illustrated with the following example:

The relationship between signal-to-noise ratio and jitter delay t_j is given by the following equation (see [8]):

$$SNR_j = 20 \log_{10} \left[\frac{1}{2\pi f t_j} \right]$$

The effective number of bits (ENOB) of an ADC is typically defined as:

$$ENOB = \frac{SNR - 1.76}{6.02}$$

My design uses 9 bits of the ADC ($ENOB_{ADC}$ = 9). Jitter noise should not make things worse. Thus $ENOB_j$ (the equivalent limitation on the number of bits due to jitter) must be bigger than 9. Re-arranging the equation gives the following condition for the product of input frequency and jitter time:

$$f t_j < 2.5 \cdot 10^{-4}$$

Suppose we are measuring a sine wave signal of 100kHz at full swing. The jitter must be smaller than 2.5ns, which corresponds to 2.5% at a sampling frequency of 10MHz.

The chip chosen was an LTC6903 by Linear Technologies. It provides a clock signal between 1kHz and 68MHz and can be programmed via SPI (serial peripheral interface). This IC has a jitter of less that 0.4% for frequencies below 10MHz, as can be seen on figure 4.10.

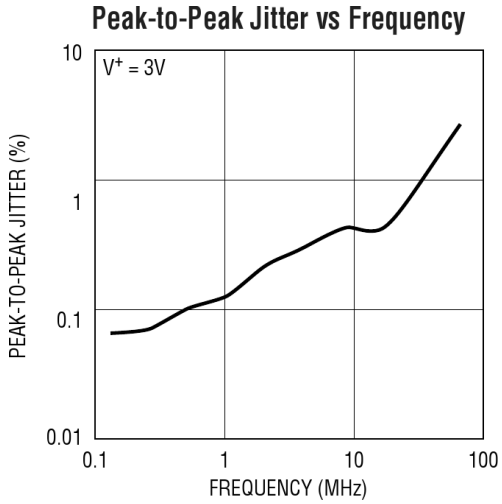


Figure 4.10: Jitter vs frequency for LTC6903. Excerpt from datasheet [9]

The clock signal is buffered by a 7404 inverter (in a single-gate variant). Such a buffer is suggested in the datasheet if more than two inputs are driven, or if the line is longer than 5cm. Both

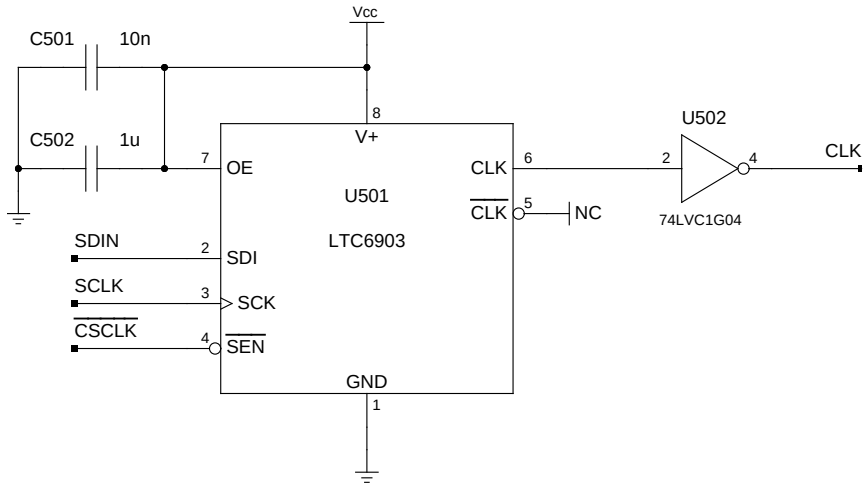


Figure 4.11: Clock generation circuit

are the case here. Also, the clock signal needs to stay high during a reset of the FIFO. Thanks to the inverting buffer, we can simply turn off the clock chip with the correct SPI command, and the clock line will stay high.

4.4 Bus interface

4.4.1 Bus timing requirements

Figure 4.12 shows the state of the bus when sending data from the PC to the oscilloscope module. The address on the bus must be compared to the module's address. If they match, the data is simply read at every rising edge of the EnableOut signal.

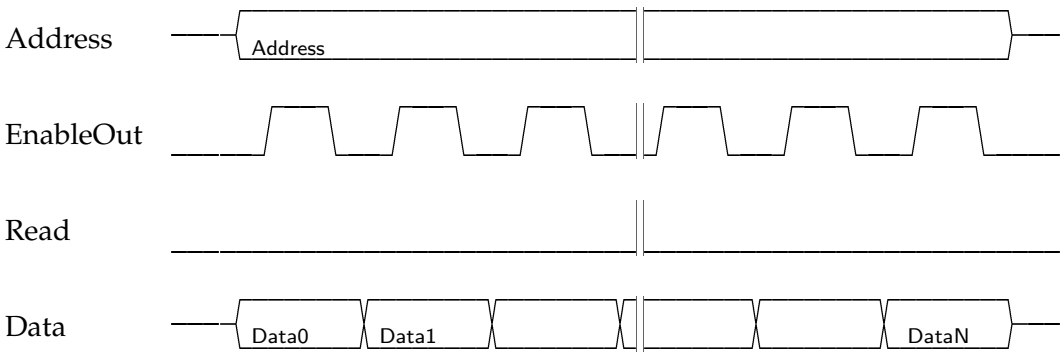


Figure 4.12: Timing diagram PC → oscilloscope

The bus timing for retrieving the data from the module is shown in figure 4.13. The Cypress chip will read the data at the middle of every pulse of EnableOut.

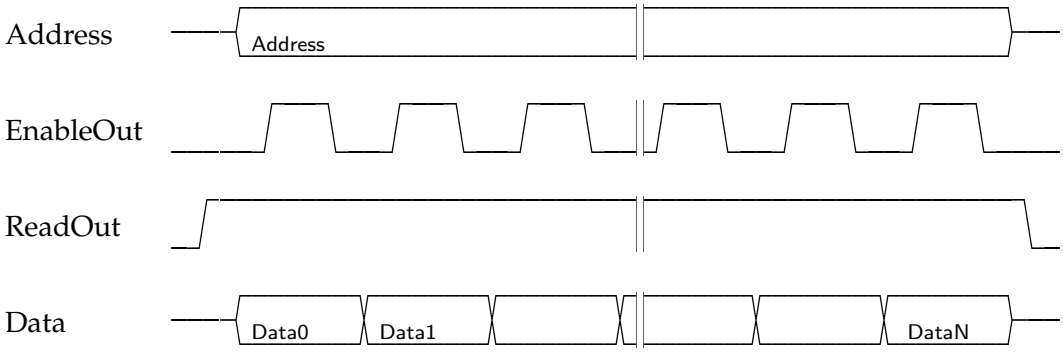


Figure 4.13: Timing diagram oscilloscope → PC

After a first design of the analog stage had been established, it became clear that multiple channels could be easily offered. This is done by making an interface card where multiple ADC cards could be plugged in. The overhead for this turned out to be minimal: only a multiplexer was needed to send the WRITE or READ signal to the correct ADC card.

4.4.2 Communication protocol

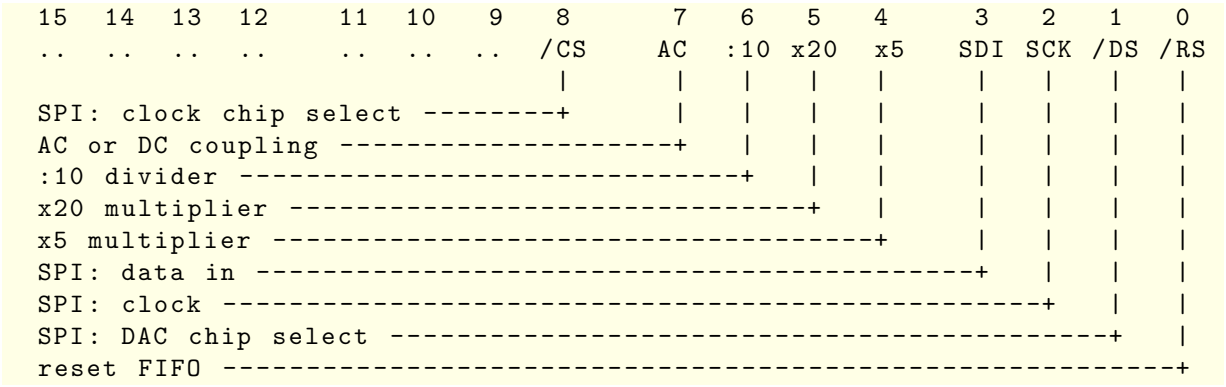
The communication protocol for this project has been designed to be simple. It can be implemented using standard logic circuits. For reading the samples, the 9-bit values are read directly from the FIFO. The data format is as follows:

```
 15  14  13  12    11  10   9   8      7   6   5   4     3   2   1   0
 ..  ..  ..  ..    ..  /FF /EF D8    D7  D6  D5  D4    D3  D2  D1  D0
                       |   |
 set if FIFO not full --+   |
 set if FIFO not empty -----+
```

The 9 lowest bits are the data bits. The flags indicating whether the FIFO is empty ($\overline{\text{EF}}$) or full ($\overline{\text{FF}}$) are also sent. The computer simply reads an amount of data and discards the words that have $\overline{\text{EF}}$ set to 0.

The data lines of the FIFO are connected directly to the bus. For the two flags, a buffer of type 74125 is used.

Changing the acquisition parameters is just as simple: one word is sent over the bus that contains all the control bits. The 8 lowest bits are kept separately for each channel in a flip-flop of type 74574. The following figure lists each control bit:



4.4.3 Logic circuitry

The digital logic is constructed using logic ICs from the 7400 series. The choice of the family is crucial: only a few both support low supply voltages and have low propagation delays. Additionally, for driving the relays, high output drive current is necessary.

Name	Supply [V]	Delay [ns]	Drive [mA]	Comments
HC	2–6	9	8	standard
AHC	2–6	5	8	
LVT	2.7–3.6	2	64	
LVC	1.2–3.6	4	24	

Table 4.4: Logic family parameters

For driving the relays, the LVT family will be used. The other chips will be selected by their availability and price.

4.4.4 Logic functions

The module must be able to detect its address on the bus. A 8-bit wide comparator (74521) is used. The address bus being 16 bit wide, in fact I use up 256 addresses of the 65536 possible ones.

Next we need to know whether we are supposed to read from or write to the bus. This is done by looking at the R/W line of the bus. These signals are now combined using NAND and inverter gates.

After the first prototype was built, it became apparent that some samples were lost. The problem was that after all valid values had been read from the FIFO, a pulse of the sampling clock could occur. This meant that one value was read from the ADC into the FIFO. As the FIFO was no longer empty now, another read operation would take place on the next bus clock pulse. Because

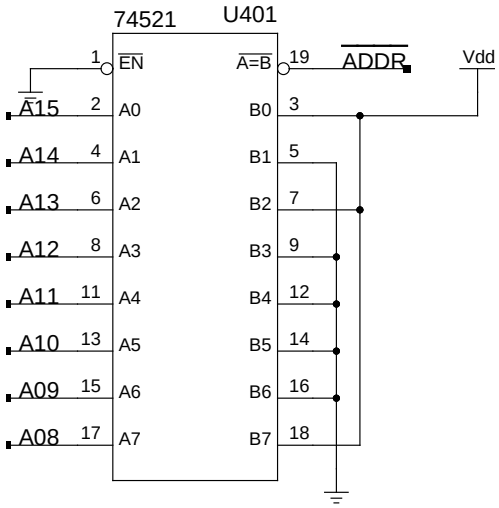


Figure 4.14: Address decoder circuit

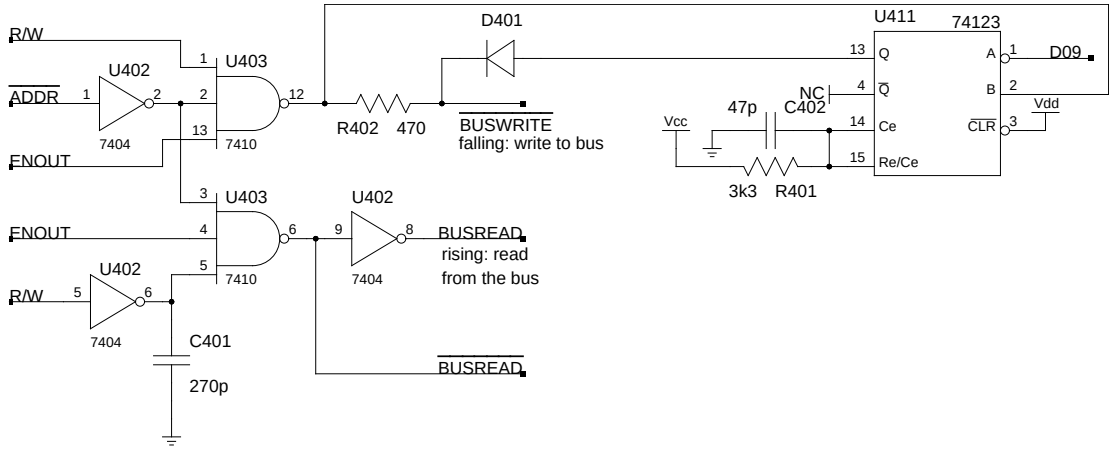


Figure 4.15: Bus read/write detection circuit

the ADC clock and the bus clock are not synchronized at all, this could lead to an incomplete write operation.

The solution was to inhibit further read attempts after the FIFO was empty. A 74123 monostable circuit is triggered by the empty flag. It is re-triggered at each pulse of the bus clock and resets itself after some time.

4.5 Power supply requirements

The analog part of the circuit needs +3.3V and −3.3V supplies. For the digital part, only a +3.3V supply is needed but it should be separated from the analog supply to avoid voltage spikes influencing the analog signals. Table 4.5 gives an estimation of power consumption when using one channel.

An attempt was made to design a power supply. However, the ripple voltage from the DC–

#	Component	Type	V_s [V]	I_s/device [mA]	Total power [mW]
2	Relay	Omron G6K	3.3	30	198
1	Op-amp	AD8065	6.6	7	46
1	Op-amp	AD8062	6.6	14	92
2	Op-amp	LT6200	6.6	20	132
1	Analog switch	MAX4547	6.6	≈ 0	≈ 0
1	ADC	MAX1444	3.3	19	63
1	FIFO	IDT72V06	3.3	60	198
1	DAC	LTC2630	3.3	≈ 0	≈ 0
1	Flip-flop	74574	3.3	5	17
1	Bus buffer	74125	3.3	5	17
1	Comparator	74688	3.3	5	17
1	Flip-flop	74574	3.3	5	17
1	Clock	LTC6903	3.3	3	10
1	AND	7408	3.3	5	17
2	Inverter	7404	3.3	5	17
1	Monostable	74123	3.3	5	17
				total	858

Table 4.5: Estimation of power consumption

DC converter turned out to be problematic. A commercially available dual power supply is currently used.

4.6 Board construction

Two acquisition boards and one bus interface board have been made and tested. Figure 4.16 shows the three circuit boards connected to the existing USB interface by Fabrizio Lo Conte (in gray).

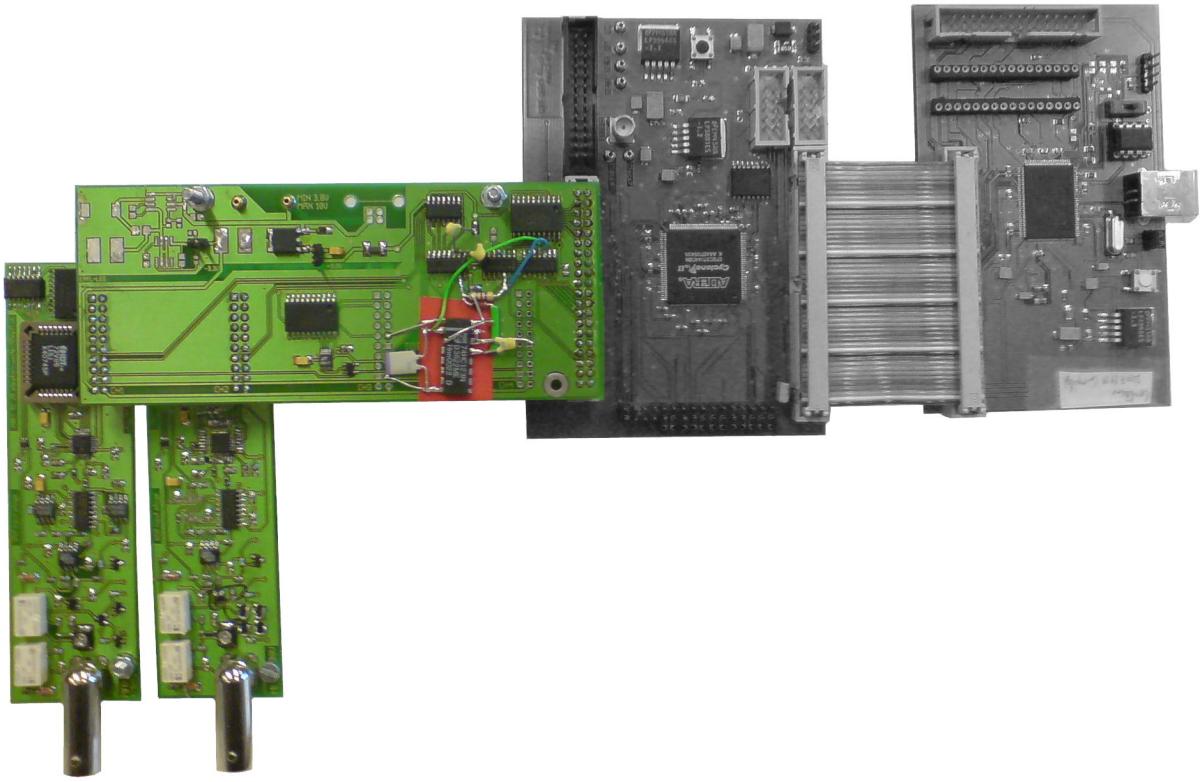


Figure 4.16: Oscilloscope with two channels.

Chapter 5

Software

5.1 Requirements

The program must first establish a connection to the Cypress card via USB. Then the acquisition parameters such as the gain must be sent to the card. The clock is halted so as not to have any difference in the sample memory of the different channels. After the FIFO memory of each channel has been reset, the clock can be programmed and started.

Now the program must periodically read the values from the FIFO chips. Apart from reading and storing the 9 data bits, the flags $\overline{\text{FEMPTY}}$ and $\overline{\text{FFULL}}$ must be checked for each word.

If the full flag is set, sample data has been lost. In that case we have no choice but to reset the FIFO and start the sampling anew. If the empty flag is set, the corresponding words are not valid samples. We simply ignore them and wait for data where the empty flag is not set.

The sample date is displayed as a diagram of voltage vs time. A trigger condition will define the first sample to show on the left edge of the screen. The display can be triggered by a rising or a falling edge of a channel and at a voltage both specified by the user.

5.2 Language and libraries

The Python programming language [13] has been chosen because of the following advantages over C++ (the traditional choice):

- it is an interpreted language, programs can be run on different platforms (Windows, Linux) without compiling
- memory is managed automatically, which reduces the work necessary as well as the potential for errors

For the communication over USB, the libusb library is used. This makes the code dealing with USB platform-independent.

Graphical user interfaces (GUI) for Python programs are traditionally implemented with the Tk toolkit. This toolkit is however rather old (the first version dates back to 1991) and was

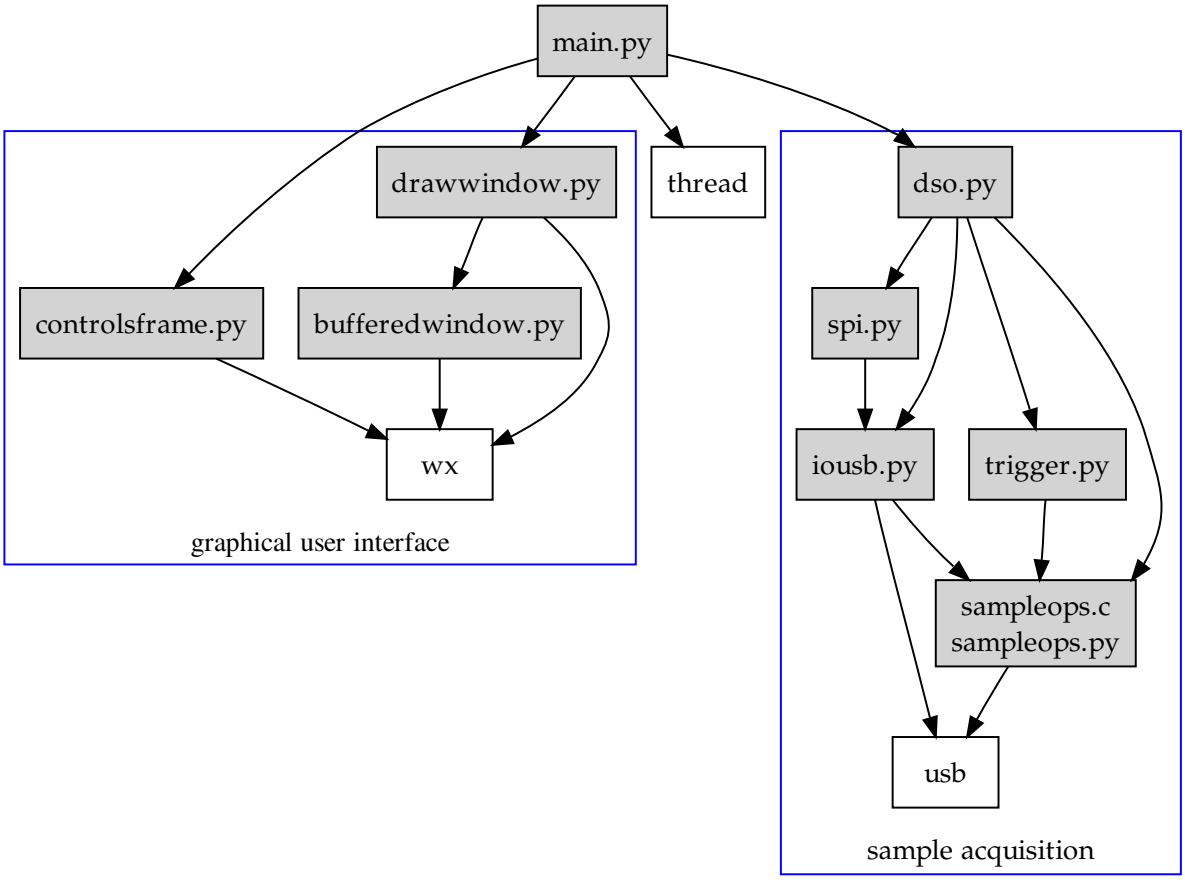


Figure 5.1: Simplified view of code dependencies. White boxes: existing libraries

developed for the X Window System used in Unix operating systems. WxWidgets [14] is a more modern toolkit running on all modern operating systems. It is used for this project via the WxPython interface [12].

5.3 Structure of implementation

5.3.1 `main.py`

This module creates the frames for displaying the data and the control elements. It then starts the data acquisition in `dso.py` as a seperate thread. A timer is set to re-draw the data 50 times per second.

5.3.2 `drawwindow.py` and `bufferedwindow.py`

The oscilloscope traces are drawn in this module. Also, a grid is drawn and the parameters such as voltage and time per grid unit are shown. The `DrawWindow` class is based on the `BufferedWindow` class, which implements double-buffering to avoid flickering.

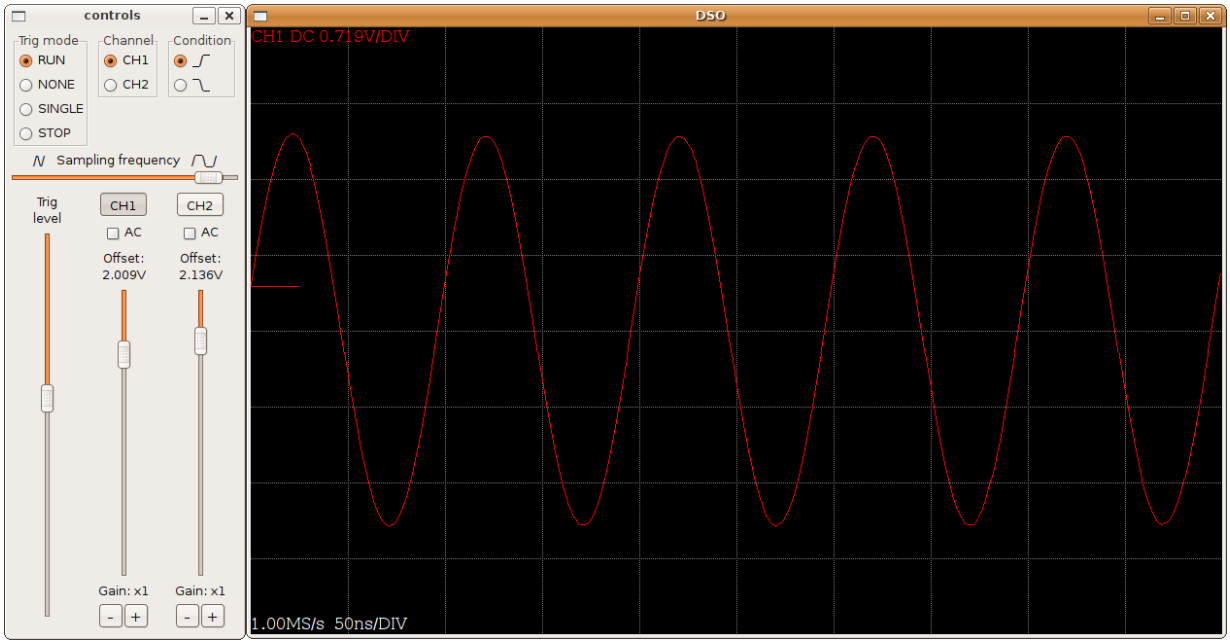


Figure 5.2: Screen-shot of the program showing a 10kHz sine wave

5.3.3 `controlsframe.py`

The control elements (buttons, sliders) are defined here. A screen-shot of the program is shown in figure 5.2. Note that the control elements are only shown for two channels. This is simply done to prevent the window from taking up a lot of screen space. All four channels can be activated by changing a single line in a configuration file.

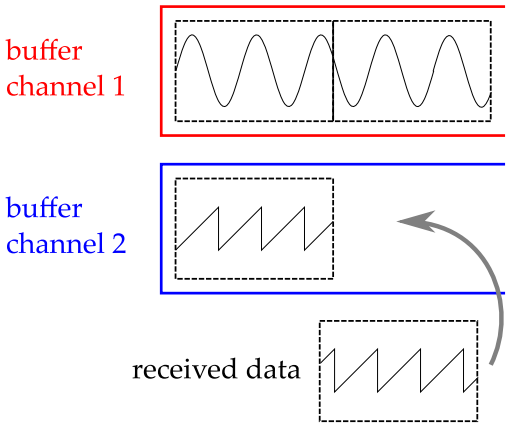
5.3.4 `dso.py`

Most of the actual work is done by this module. The method `start()` first establishes a connection with the USB interface. Then, the parameters such as gain, offset and sampling frequency are set. The FIFO memory chips are reset and the data acquisition begins.

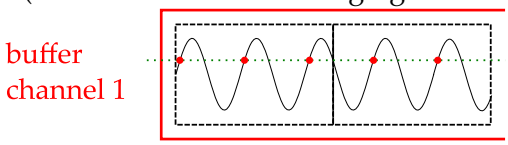
In an infinite loop, we first check if an operation by the user requires a reset of the FIFOs, for example if a channel has just been activated. Then the request for the sample data are sent to the USB interface. The samples received are then sent to the triggering module for storage and analysis (see next section). In the rest of the loop, we check if other parameters such as offset and gain have been modified. These do not require a reset, the new value is simply sent over the USB.

5.3.5 `trigger.py`

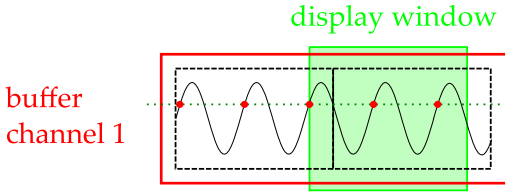
Sample data is received by this module in chunks of several hundred samples. First, it is put into a buffer according to the channel number.



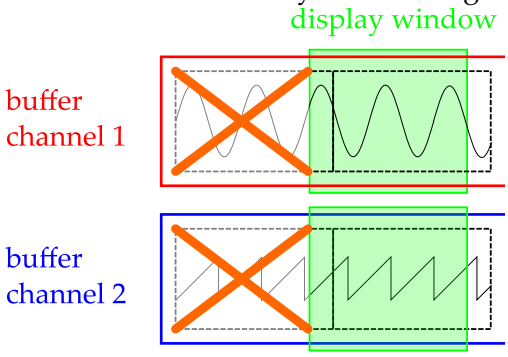
The channel that the user selected for triggering is then analyzed, sample for sample, to find the positions of the trigger event (indicated in the following figure as red dots).



For a periodic signal, there are usually several trigger points. We have to chose one that is followed by enough data to fill the screen.



The "old" data, that is samples which were received before the trigger point, can now be deleted. This has to be done for all channels in the same way in order to guarantee the synchronicity.



The sample values which are inside the display window are then sent to the `drawwindow.py` module, where all the channels are drawn in one frame.

5.3.6 `sampleops.py`

Some functions that deal with sample data in a time-consuming way have been separated into this file. This Python module can then be replaced by a module written in C, as shown later in the section on optimization.

5.3.7 `iousb.py` and `spi.py`

These modules contain the communication function for the USB bus and the SPI interface that is used for the DAC and the clock.

5.4 Optimization

The choice of the Python programming language made a rapid development possible. It is however not without disadvantages: Operations on byte strings are not as fast as they would be in C or C++.

Code execution time has been measured for an acquisition time of one minute. Table 5.1 shows the times for the function that use the most time in total. The cumulative time is the time of the function itself and all other functions that are called from it.

file	function	time/call [µs]	cumulative time/call [µs]
sampleops.py	**usb_bulk_read1**	614.4	–
trigger.py	callback	72.7	97.9
iousb.py	_write	62.8	–
sampleops.py	**checksamples**	31.3	–
iousb.py	_sortInData	43.5	312.5
sampleops.py	**find_trig_events**	23.3	–
iousb.py	sendData	22.2	1011.0
iousb.py	addDataToReadBuffer	5.6	–
dso.py	cb	4.5	134.5

Table 5.1: Time spent in functions (not optimized)

The functions highlighted with bold text have been chosen for optimization. These functions were re-written in C. The result is shown in Table 5.2. Measuring the time of the C function is not possible with the method used. But by comparing the cumulative time of some Python function, one can see that there it is indeed faster: The function `sendData` for example, which calls `write` and `sortInData`, uses 35% less time.

file	function	time/call [µs]	cumulative time/call [µs]
iousb.py	sendData	278.9	**652.7**
trigger.py	callback	72.8	**85.3**
iousb.py	_write	142.5	–
iousb.py	_sortInData	49.3	**231.0**
iousb.py	addDataToReadBuffer	5.7	–
dso.py	cb	5.7	**91.0**

Table 5.2: Time spent in functions (optimized)

Chapter 6

Results

6.1 Performance of analog stage

The transfer function of the input stage from the BNC connector to the input of the ADC has been measured. This measurement was done with DC coupling and has been repeated for all gain settings. A network analyzer of type Agilent 4395A with an active FET probe (750MHz) was used.

It should be noted that the values of the gain refer to the range of the ADC input. A gain of one means that the maximum input signal of ± 3V is amplified to the maximum input of the ADC, which is $+0.5\text{V} \ldots + 2.5\text{V}$. On the Bode diagrams, the marker denoted "0" indicates the 3dB cut-off frequency. Input power, scale and reference position have been adapted to best show each curve.

Figure 6.1: Bode plot for $G = 1/10$

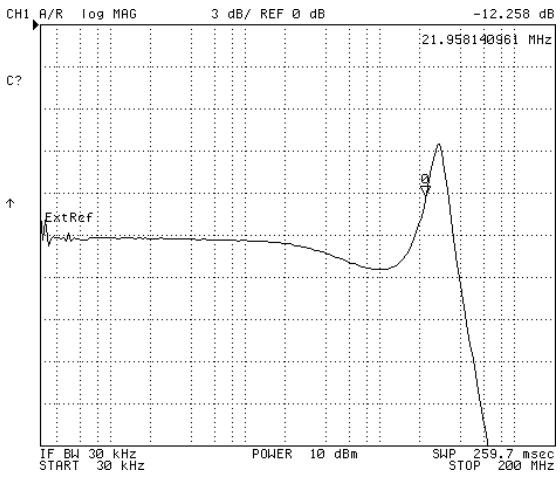


Figure 6.2: Bode plot for $G = 1/2$

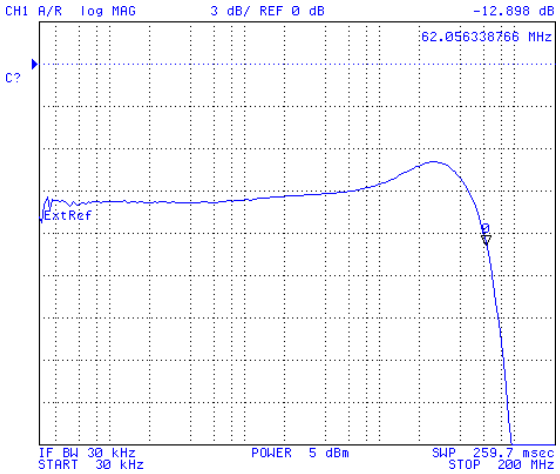


Figure 6.3: Bode plot for $G = 1$

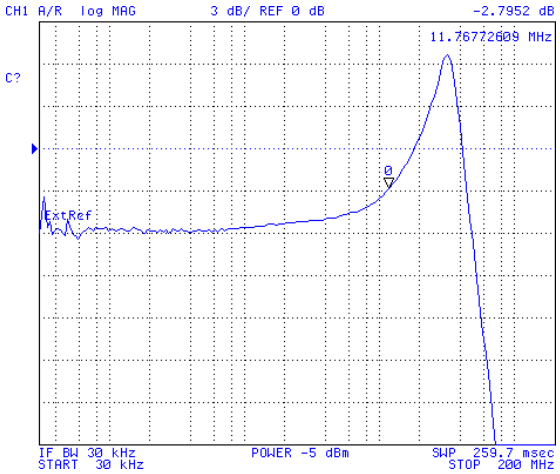


Figure 6.4: Bode plot for $G = 2$

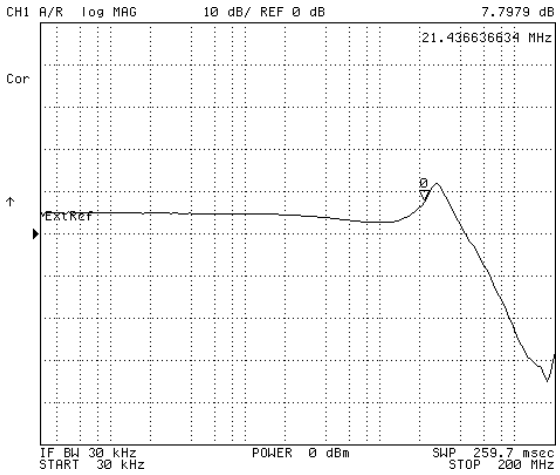


Figure 6.5: Bode plot for $G = 5$

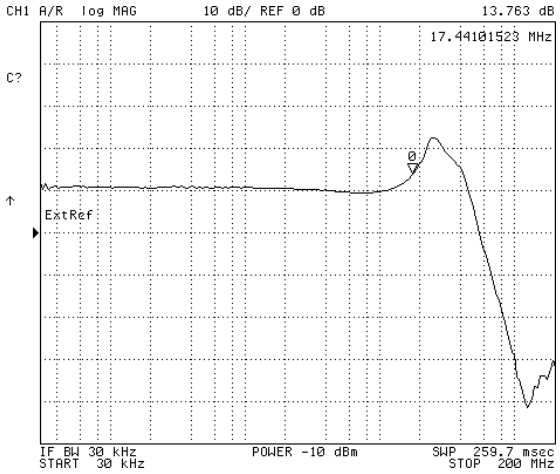


Figure 6.6: Bode plot for $G = 10$

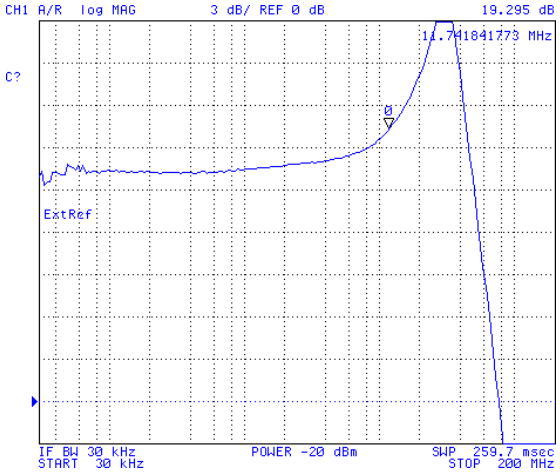


Figure 6.7: Bode plot for $G = 20$

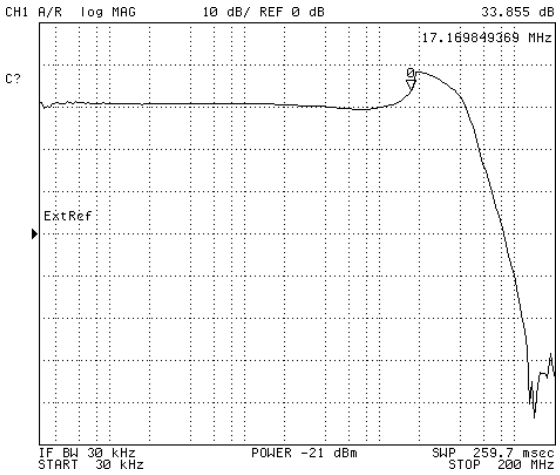


Figure 6.8: Bode plot for $G = 100$

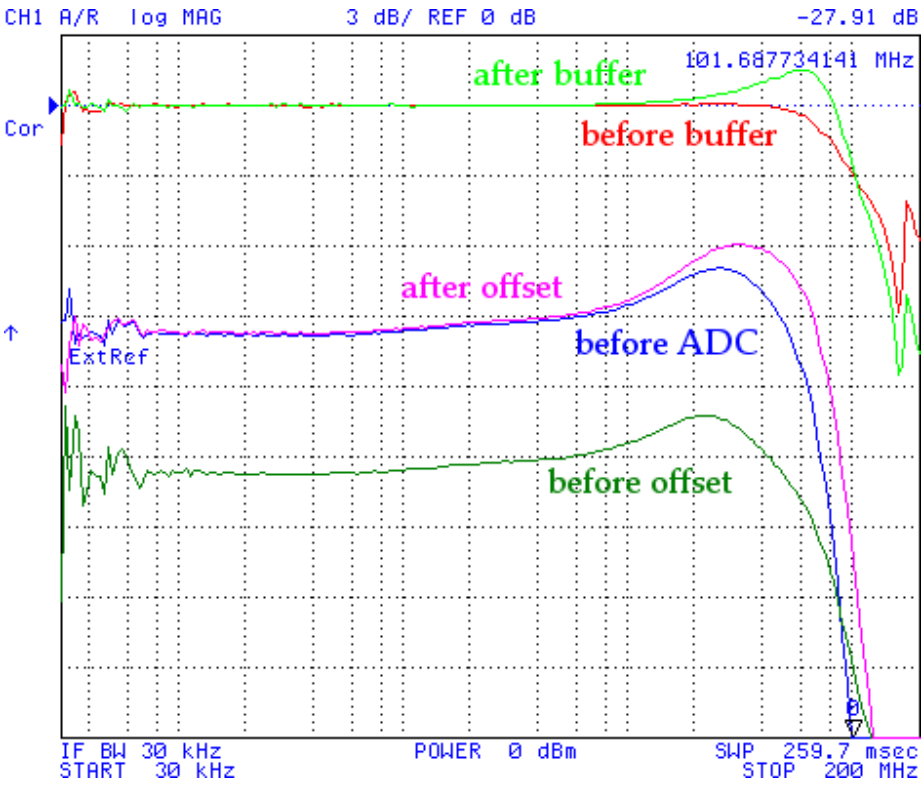


Figure 6.9: Bode plots for $G = 1$ at different nodes

The bandwidth obtained here is around 8MHz for some gain configurations and around 20MHz for others. This does not satisfy the specifications. An obvious problem is the spike at around 30MHz.

If we measure the frequency response at different nodes in the circuit (figure 6.9), it is clear that this spike is caused by the unity-gain buffer op-amp. In the bode diagram, the difference between the signals labeled "before buffer" and "after buffer" is caused by the AD8065.

The datasheet of this chip shows that it is indeed a shortcoming. In figure 6.11 it can be seen that the magnitude of this error depends on C_L, the load capacitance from output to ground.

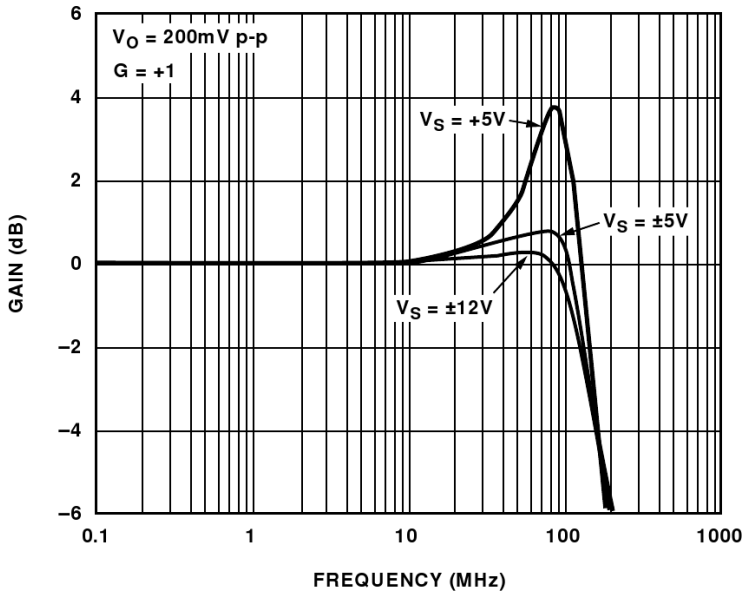


Figure 6.10: AD8065 small signal frequency response. Excerpt from datasheet [1]

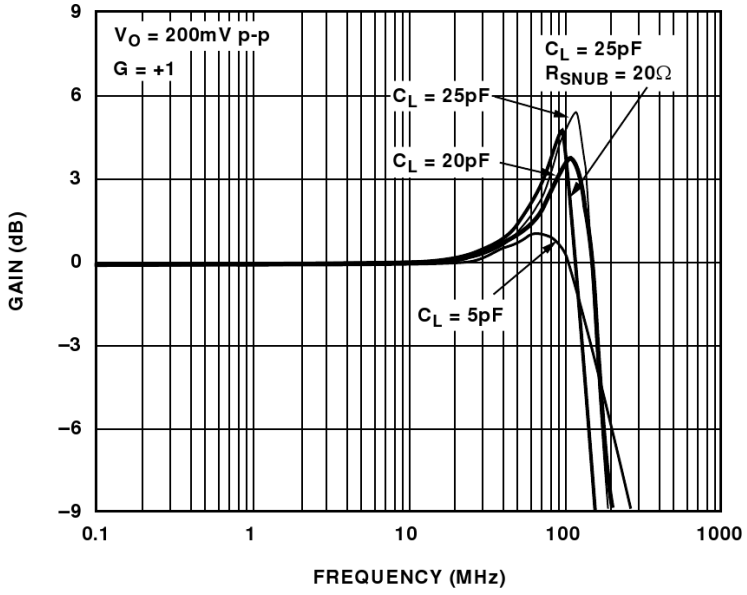


Figure 6.11: AD8065 frequency response for various C_L. Excerpt from datasheet [1]

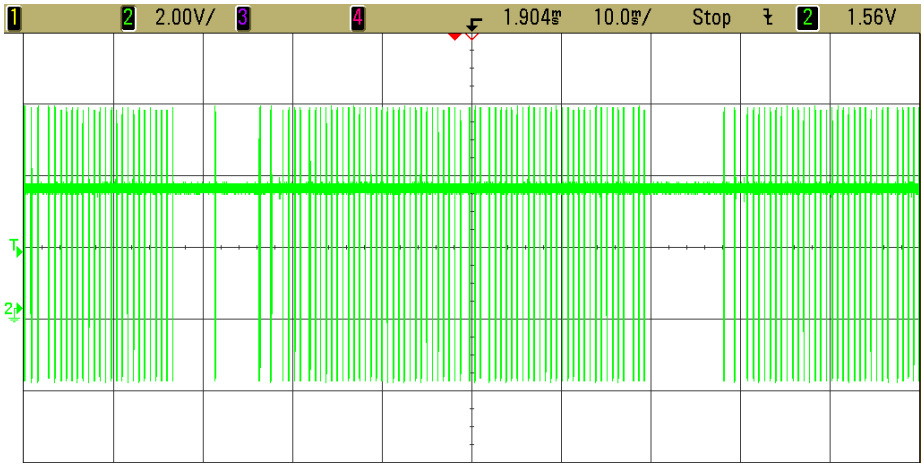


Figure 6.12: Bus communication

6.2 Software performance

The software runs on Linux, and with slightly worse performance on Windows. It can acquire the samples and display the traces graphically. All parameters (AC/DC coupling, attenuation, gain, offset) can be modified. The channels are shown in the same frame using different colors.

The sampling frequency is theoretically limited to about 3Msps. This is due to the USB interface which has a throughput of about 43Mbit/s (reading speed, see [10]), and the fact that each sample uses 16 bits. Improvements should be possible here, as the gross throughput for USB is 480Mbit/s.

Currently, the sampling frequency is limited by the latency of the operating system on the PC. At 2MHz for example, the FIFO memory would fill up completely in 8ms. Desktop operating systems such as Windows and Linux have a granularity of the task scheduler that is in the range of 1 to 10ms. The problem is apparent on figure 6.12, which shows the communication on the parallel bus when connected to a Linux PC. Each spike represents one packet of 1022 words. Interruptions of up to 10ms duration are occurring. On a reasonably fast PC running Linux, the highest sampling rate achieved using one channel was 1.5Msps.

One possible solution is to use a real-time operating system, such as RTLinux or RTAI. The drawback of such a system would be the loss of compatibility with any desktop PC.

Chapter 7

Summary and outlook

7.1 Conclusion

The project, in its current state, has the basic functionalities of a digital oscilloscope. It is capable of measuring voltage signals smaller than $\pm 30V$. The signal can be attenuated or amplified at various factors. The digital logic that processes the sample data has been designed to be simple. On the hardware side, no programming was involved.

PCB layouts for the circuits have been made and the oscilloscope was assembled with two channels. I characterized the frequency response of the analog stage.

The software allows the user to view the data in the same way as with any oscilloscope, and to modify the configuration of the hardware (gain, sampling rate, etc.). Written in Python with some functions optimized in C, the software runs on both the MS Windows and GNU/Linux operating systems.

The specifications are however not completely satisfied: real-time sampling faster than 1.5Msps is not possible. This is due to the latency of the operating system and is unlikely to be improved unless significant changes are made to the hardware or the software. Also, the desired analog bandwidth could not be reached.

7.2 Future prospects

The following improvements to this project are possible:

- resolve problems in the high frequency range by choosing a different operational amplifier (see section 6.1) and modifying the PCB layout to reduce parasitic capacitances.
- investigate the use of programmable logic (CPLD or FPGA) instead of standard 74 logic chips. This would likely result in higher flexibility and lower component count but also higher price.
- design, test and integrate a power supply that satisfies the requirements laid out in section 4.5. The 5V supply of the USB interface could be used for this, as power consumption is quite low ($<$ 1W for one channel).

- do trigger detection in hardware. This would allow for higher sampling rates if the data is not transmitted in real-time, but only after a trigger condition has occurred.
- extend the software to do various data visualization methods such as frequency analysis, X–Y curves or arithmetic operations on multiple channels.

* * *

Stephan Walter

Lausanne
January 14, 2008

Bibliography

[1] Analog Devices Inc.: *AD8065/AD8066 datasheet*.

[2] area26: *PC-based DSO*. http://beta.area26.no-ip.org/projects/dso.

[3] BitScope Designs: *The Bitscope hardware design*. http://www.bitscope.com/design/hardware/.

[4] eosystems.ro: *eOscope*. http://eosystems.ro/eoscope/eoscope_en.htm.

[5] Glaser, J.: *Digital sampling oscilloscope*. http://johann-glaser.at/projects/DSO/.

[6] Grocutt, T.: *Large storage depth oscilloscope*, 2000. http://dsoworld.co.uk/.

[7] Jones, D.L.: *Digital storage oscilloscope adapter Mk3*, 1998. http://alternatezone.com/electronics/dsoamk3.htm.

[8] Kester, W.: *Data Conversion Handbook*. Elsevier/Newnes, 2005, ISBN 0750678410. http://www.analog.com/library/analogDialogue/archives/39-06/data_conversion_handbook.html.

[9] Linear Technology Corporation: *LTC6903/LTC6904 datasheet*.

[10] Lo Conte, F.: *Interface I/O Universelle pour port USB*, 2006.

[11] Lutti, L.: *Autocostruzione di un oscilloscopio digitale*, 2003. http://www.enetsystems.com/~lorenzo/scope/.

[12] Rappin, N. and R. Dunn: *wxPython in Action*. Manning Publications Co., Greenwich, CT, USA, 2006, ISBN 1932394621.

[13] Rossum, G. van: *Python reference manual*, 2006. http://docs.python.org/ref/ref.html.

[14] Smart, J., R. Roebling, V. Zeitlin, R. Dunn, *et al.*: *wxWidgets 2.8.6: A portable C++ and Python GUI toolkit*, 2007. http://www.wxwidgets.org/manuals/stable/wx_contents.html.

Appendix A

Code

main.py

```
#!/usr/bin/env python

import logging, thread, time, sys, wx

import controlsframe, drawwindow, dso, settings, trigger
from commondefs import *
from config import *

class TracesFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, parent=None, title="DSO", size=(1000,512))
        self.window = drawwindow.DrawWindow(self, settings)

        self.timer = wx.Timer(self, id=1)
        wx.EVT_TIMER(self, 1, self.newDrawing)
        self.timer.Start(50) # 20 images / second

    def newDrawing(self,event):
        trigger.get_samples()
        self.window.updateDrawing()

class App(wx.App):
    def OnInit(self):
        frameTraces = TracesFrame()
        frameTraces.Show()

        frameControls = controlsframe.Frame(settings)
        frameControls.Show()

        frameTraces.Bind(wx.EVT_CLOSE, self.OnExit)
        frameControls.Bind(wx.EVT_CLOSE, self.OnExit)

        return True

    def OnExit(self, event=None):
        settings.halt.acquire()
        logging.info("Waiting for acquisition thread to reset device...")
        time.sleep(0.5)
        logging.info("Good-bye!")
        sys.exit()

settings = settings.Settings()
settings.channels[0].active = True
settings.channels[1].active = False
settings.trig_channel = 0
```

```
trigger = trigger.Trigger(settings)

logging.basicConfig(level=getattr(logging,LOGLEVEL))

def main():
    thread.start_new_thread(dso.DSO().start, (trigger.callback, settings))
    App().MainLoop()

main()
```

drawwindow.py

```
import wx, bufferedwindow
from commondefs import *

T_DIVS = 10
V_DIVS = 8

class DrawWindow(bufferedwindow.Window):

    """Window showing the oscilloscope traces."""

    channelColors = [
            wx.Colour(0xff,0,0),
            wx.Colour(0,0,0xff),
            wx.Colour(0,0xff,0),
            wx.Colour(0xff,0,0xff) ]

    def __init__(self, parent, settings, id=-1):
        self.settings = settings
        bufferedwindow.Window.__init__(self, parent)

    def onSize(self, event):
        self.w, self.h = self.GetClientSizeTuple()

        # pre-calculate list of x-coordinates of samples
        self.xpoints = [ i*self.w/self.settings.time_width
                                for i in xrange(self.settings.time_width) ]

        # pre-calculate grid coordinates
        self.grid = [(x*self.w/T_DIVS, 0, x*self.w/T_DIVS, self.h)
                                for x in range(1,T_DIVS) ] + \
                    [(0, y*self.h/V_DIVS, self.w, y*self.h/V_DIVS)
                                for y in range(1,V_DIVS) ]

        bufferedwindow.Window.onSize(self, event)

    def draw(self, dc):
        def scalepoint(s): return (512 - s) * self.h / 512

        dc.SetBackground(wx.Brush('black'))
        dc.Clear()

        # draw grid
        dc.SetPen(wx.Pen('#808080', 1, wx.DOT))
        dc.DrawLineList(self.grid)

        # show sampling frequency and time per grid division
        dc.SetTextForeground('#FFFFFF')
        f_khz = self.settings.frequency / 1000.0
        tdiv = self.settings.time_width / f_khz / float(T_DIVS)
        if tdiv > 10.0:
            freq_text = "%.3fkS/s %.1fus/DIV" % (f_khz,tdiv)
        elif tdiv > 1.0:
            freq_text = "%.2fkS/s %.2fus/DIV" % (f_khz,tdiv)
        elif tdiv > 0.1:
            freq_text = "%.1fkS/s %.3fus/DIV" % (f_khz,tdiv)
        else:
            freq_text = "%.2fMS/s %.0fns/DIV" % (f_khz/1000.0,tdiv*1000.0)
        dc.DrawText(freq_text, 0, self.h-20)

        for chan in self.settings.active_chans:
            ch_data = self.settings.channels[chan]

            dc.SetPen(wx.Pen(self.channelColors[chan], 1, wx.SOLID))

            # calculate y pixel coordinates from sample values
            ypoints = map(scalepoint, ch_data.samples)

            dc.DrawLines(zip(self.xpoints, ypoints))

            text = "CH%d %s %.3fV/DIV" % (chan+1,
```

```
                    coupling_names[ch_data.coupling],
                    (5.75 / gain_factors[ch_data.gain] / V_DIVS))
            dc.SetTextForeground(self.channelColors[chan])
            dc.DrawText(text, chan*self.w/4, 0)

            if chan == self.settings.trig_channel:
                tlevel = scalepoint(self.settings.trig_level)
                dc.DrawLine(0, tlevel, self.w/20, tlevel)
```

bufferedwindow.py

```
import wx

class Window(wx.Window):

    """Double-buffered window"""

    def __init__(self, parent, pos = wx.DefaultPosition, size = wx.DefaultSize):
        wx.Window.__init__(self, parent, pos=pos, size=size)

        wx.EVT_SIZE(self, self.onSize)

        self.onSize(None)

    def draw(self,dc):
        """Place holder. Override this method"""
        pass

    def onSize(self,event):
        """Make a new offscreen bitmap buffer."""
        self.buffer = wx.EmptyBitmap(*self.GetClientSizeTuple())
        self.updateDrawing()

    def saveToFile(self,FileName,FileType):
        """Save a screenshot to a file."""
        self.buffer.SaveFile(FileName,FileType)

    def updateDrawing(self):
        """Create buffer for new drawing"""
        dc = wx.BufferedDC(wx.ClientDC(self), self.buffer)
        self.draw(dc)
```

controlsframe.py

```
import wx, dso
from commondefs import *
from config import *

class Frame(wx.Frame):

    """Frame with buttons and other widgets for acquision parameters."""

    offset_label = {}
    gain_label = {}

    def __init__(self, settings):
        self.settings = settings

        # create frame and divide space
        wx.Frame.__init__(self, parent=None, id=-1, title='controls',
            style=wx.MINIMIZE_BOX|wx.SYSTEM_MENU|wx.CAPTION|wx.CLOSE_BOX)
        self.panel = wx.Panel(self)
        sizer = wx.BoxSizer(wx.VERTICAL)
        trigsizer = wx.BoxSizer(wx.HORIZONTAL)

        def make_radio_box(label, choices, selection, callback):
            box = wx.RadioBox(self.panel, label=label,
                      style=wx.RA_VERTICAL, choices=choices)
            box.SetSelection(selection)
            self.Bind(wx.EVT_RADIOBOX, callback, box)
            trigsizer.Add(box, 0, wx.ALL, border=5)

        # create trig mode radiobox
        make_radio_box("Trig mode", trig_mode_names, self.settings.trig_mode,
                self.trig_mode)

        # create trig channel radiobox
        make_radio_box("Channel",
                ["CH"+str(i+1) for i in range(NUM_CHANNELS)],
                self.settings.trig_channel, self.trig_channel)

        # create trig condition radiobox
        make_radio_box("Condition", trig_condition_names,
```

```
            self.settings.trig_condition, self.trig_condition)

        sizer.Add(trigsizer, 0, wx.EXPAND)

        freq_label = wx.StaticText(self.panel,
            label=u"/\/ Sampling frequency /^\_/",
            style=wx.ALIGN_CENTRE)
        freq_slider = wx.Slider(self.panel,
            value=self.settings.freq_setting/FREQ_DIV,
            minValue=0, maxValue=MAX_FREQUENCY/FREQ_DIV)
        self.Bind(wx.EVT_SLIDER, self.frequency, freq_slider)
        sizer.Add(freq_label, 0, wx.EXPAND)
        sizer.Add(freq_slider, 0, wx.EXPAND)

        chansizer = wx.BoxSizer(wx.HORIZONTAL)

        # trig level
        tsizer = wx.BoxSizer(wx.VERTICAL)
        tlevel_label = wx.StaticText(self.panel,
            label="Trig\nlevel", style=wx.ALIGN_CENTRE)
        tsizer.Add(tlevel_label, 0, wx.ALL|wx.ALIGN_CENTER, 2)
        tlevel = wx.Slider(self.panel,
            value=MAX_SAMPLE_VALUE-self.settings.trig_level,
            minValue=0, maxValue=MAX_SAMPLE_VALUE, size=(5,400),
            style=wx.SL_VERTICAL)
        self.Bind(wx.EVT_SLIDER, self.trig_level, tlevel)
        tsizer.Add(tlevel, 1, wx.ALL|wx.EXPAND|wx.ALIGN_CENTER, 2)
        chansizer.Add(tsizer, 1, wx.ALL, border=5)

        for c in range(NUM_CHANNELS):
            # channel toggle button
            csizer = wx.BoxSizer(wx.VERTICAL)
            cbutton = wx.ToggleButton(self.panel, id=c,
                label=channel_names[c], size=(50,26))
            if c in self.settings.active_chans:
                cbutton.SetValue(True)
            self.Bind(wx.EVT_TOGGLEBUTTON, self.toggle_channel, cbutton)
            csizer.Add(cbutton, 0, wx.ALL|wx.ALIGN_CENTER, border=0)

            # AC/DC coupling checkbox
            box = wx.CheckBox(self.panel, id=c, label=coupling_names[1])
            self.Bind(wx.EVT_CHECKBOX, self.coupling, box)
            csizer.Add(box, 0, wx.ALL|wx.ALIGN_CENTER, border=5)

            # offset slider
            self.offset_label[c] = wx.StaticText(self.panel, label="",
                style=wx.ALIGN_CENTRE)
            self.set_offset_text(c)
            csizer.Add(self.offset_label[c], 0, wx.ALL|wx.ALIGN_CENTER,
                border=2)
            coffset = wx.Slider(self.panel, id=c,
                value=MAX_OFFSET - self.settings.channels[c].offset,
                minValue=0, maxValue=MAX_OFFSET, size=(5,300),
                style=wx.SL_VERTICAL)
            self.Bind(wx.EVT_SLIDER, self.offset, coffset)
            csizer.Add(coffset, 1, wx.ALL|wx.EXPAND|wx.ALIGN_CENTER,
                border=2)

            # gain increase/decrease buttons
            self.gain_label[c] = wx.StaticText(self.panel, label="Gain: x1")
            csizer.Add(self.gain_label[c], 0, wx.ALL|wx.ALIGN_CENTER,
                border=2)

            gainsizer = wx.BoxSizer(wx.HORIZONTAL)
            minusbutton = wx.Button(self.panel, id=c, label="-", size=(26,26))
            self.Bind(wx.EVT_BUTTON, self.decrease_gain, minusbutton)
            plusbutton = wx.Button(self.panel, id=c+10, label="+", size=(26,26))
            self.Bind(wx.EVT_BUTTON, self.increase_gain, plusbutton)
            gainsizer.Add(minusbutton, 0, wx.ALL, border=0)
            gainsizer.Add(plusbutton, 0, wx.ALL, border=0)
            csizer.Add(gainsizer, 0, wx.ALL|wx.ALIGN_CENTER, border=2)

            chansizer.Add(csizer, 1, wx.ALL, border=5)

        sizer.Add(chansizer,1, wx.EXPAND)
        self.panel.SetSizer(sizer)
        sizer.Fit(self)

    def trig_mode(self, event):
        self.settings.trig_mode = event.Selection

    def trig_channel(self, event):
```

```
        self.settings.trig_channel = event.Selection

    def trig_condition(self, event):
        self.settings.trig_condition = event.Selection

    def trig_level(self, event):
        self.settings.trig_level = MAX_SAMPLE_VALUE - event.Int

    def toggle_channel(self, event):
        # first remove channel from list of active channels
        self.settings.active_chans = [ c for c in self.settings.active_chans
                                       if c != event.Id ]
        # then add it again if it is active
        if event.Selection:
            self.settings.active_chans.append(event.Id)

    def coupling(self, event):
        self.settings.channels[event.Id].coupling = event.Selection

    def set_offset_text(self, c):
        self.offset_label[c].SetLabel("Offset:\n%.3fV" % (
            self.settings.channels[c].offset * \
            MAX_OFFSET_VOLTAGE / (MAX_OFFSET+1) ))

    def offset(self, event):
        o = MAX_OFFSET - event.Int
        self.settings.channels[event.Id].offset = o
        self.set_offset_text(event.Id)

    def frequency(self, event):
        self.settings.set_freq(event.Int * FREQ_DIV)

    def decrease_gain(self, event):
        g = self.settings.channels[event.Id].gain - 1
        if g<0: g = 0
        self.settings.channels[event.Id].gain = g
        self.gain_label[event.Id].SetLabel("Gain: "+gain_names[g])

    def increase_gain(self, event):
        g = self.settings.channels[event.Id-10].gain + 1
        if g>MAX_GAIN: g = MAX_GAIN
        self.settings.channels[event.Id-10].gain = g
        self.gain_label[event.Id-10].SetLabel("Gain: "+gain_names[g])
```

dso.py

```
import sys, logging, time
from optparse import OptionParser

import iousb, settings, spi
from commondefs import *
from config import *
from sampleops import checksamples

class DSO(object):
    def __init__(self):
        logging.info("opening USB device...")
        try:
            self.card = iousb.IODevice()
        except iousb.NoInterfaceError:
            logging.error("No Cypress USB interface found")
            sys.exit(1)
        except iousb.InterfacePermissionsError:
            logging.error("Claiming USB interface not permitted. " +
                          "Change permissions or run program as root user.")
            sys.exit(1)

        spi.init(self.card)

    def _reset_fifos(self):
        """Stop oscillator, reset FIFO of each channel, restart oscillator."""
        spi.sendOscillator(0,0,3)

        for channel in self.settings.active_chans:
            self.card.clearBit(BASE_ADDRESS + channel, FIFO_RS)
            self.card.setBit(BASE_ADDRESS + channel, FIFO_RS)

        #TODO halt and check fifo state
        spi.sendOscillator((self.settings.freq_setting >> 10),
                (self.settings.freq_setting & 0x3ff))
        self.card.sendData()
```

```
    def _set_coupling(self, channel, coupling):
        if coupling:
            self.card.setBit(BASE_ADDRESS + channel, ACDC)
        else:
            self.card.clearBit(BASE_ADDRESS + channel, ACDC)

    def _set_offset(self, channel, offset):
        spi.sendDAC(channel, offset)

    def _set_gain(self, ch, gain):
        """Combine divider and multiplier circuits to obtain correct gain."""

        if gain in [G_0_1, G_0_5, G_2, G_10]:
            # enable /10 divider
            self.card.clearBit(BASE_ADDRESS + ch, ATTEN10)
        else:
            self.card.setBit(BASE_ADDRESS + ch, ATTEN10)

        if gain in [G_0_5, G_5, G_10, G_100]:
            # enable x5 multiplier
            self.card.setBit(BASE_ADDRESS + ch, GAIN5)
        else:
            self.card.clearBit(BASE_ADDRESS + ch, GAIN5)

        if gain in [G_2, G_10, G_20, G_100]:
            # enable x20 multiplier
            self.card.setBit(BASE_ADDRESS + ch, GAIN20)
        else:
            self.card.clearBit(BASE_ADDRESS + ch, GAIN20)

    def start(self, callback, settings, format="dec", channels="0"):
        """Send configuration to module and start receiving samples."""

        logging.info("parameters: format=%s channels=%s frequency=%d",
            format, channels, settings.frequency)

        self.settings = settings
        resend = True

        def cb(addr, data):
            channel = addr - BASE_ADDRESS
            try:
                result = checksamples(data)
                callback(channel, result)
            except RuntimeError:
                logging.warning("FIFO FULL channel:%d", channel)
                callback(-1, None)
                self._reset_fifos()

        logging.info("set coupling, attenuation and gain...")
        for ch in range(NUM_CHANNELS):
            self.card.addDataToWriteBuffer(BASE_ADDRESS + ch, [0xffff])
            self.card.clearBit(BASE_ADDRESS + ch, GAIN5)
            self.card.clearBit(BASE_ADDRESS + ch, GAIN20)
            self._set_coupling(ch, 0)
            self._set_offset(ch, settings.channels[ch].offset)
        self.card.sendData()

        while not settings.halt.locked():
            if settings.active_chans_modified or settings.frequency_modified:
                callback(-1, None)
                self._reset_fifos()
                settings.active_chans_modified = False
                settings.frequency_modified = False

            READ_SAMPLES = 1020 / len(settings.active_chans)
            for ch in settings.active_chans:
                self.card.addDataToReadBuffer(READ_SAMPLES,
                                              BASE_ADDRESS+ch, cb)
            self.card.sendData()

            for ch in settings.active_chans:
                chan_data = settings.channels[ch]
                if chan_data.coupling_modified:
                    self._set_coupling(ch, chan_data.coupling)
                    chan_data.coupling_modified = False
                    resend = True

                if chan_data.offset_modified:
                    self._set_offset(ch, chan_data.offset)
                    chan_data.offset_modified = False
                    resend = True
```

```
                if chan_data.gain_modified:
                    self._set_gain(ch, chan_data.gain)
                    chan_data.gain_modified = False
                    resend = True

            if resend:
                self.card.sendData()
                resend = False

            if self.settings.frequency < 5000:
                time.sleep(1/self.settings.frequency)

        logging.info("Stopping oscillator...")
        spi.sendOscillator(0,0)
```

trigger.py

```
import thread
from commondefs import *
from sampleops import find_trig_events

class Trigger(object):

    """Receive samples and apply trigger checking."""

    samples = {0: [], 1: []}
    trig_events = []
    lock = thread.allocate_lock()

    def __init__(self, settings):
        self.settings = settings

    def callback(self, channel, data):
        if channel == -1:
            # reset sample buffer
            for i in self.samples.keys():
                self.samples[i] = []
            self.trig_events = []
            return
        if self.settings.trig_mode == T_STOP: return

        trig_ch = self.settings.trig_channel
        old_trig_len = len(self.samples[trig_ch])

        self.samples[channel].extend(data)

        if channel == trig_ch and self.settings.trig_mode in (T_RUN, T_SINGLE):
            # if we have new data for the triggered channel, analyze it

            self.trig_events.extend(
                find_trig_events(
                    data,
                    old_trig_len,
                    self.settings.trig_level,
                    self.settings.trig_condition))

            trig_sample = 0

            min_len = min([ len(self.samples[c])
                                for c in self.settings.active_chans ])

            for i, e in enumerate(self.trig_events):
                if (min_len - e) < self.settings.time_width:
                    for ee in range(i, len(self.trig_events)):
                        self.trig_events[ee] -= trig_sample
                    del self.trig_events[0:i]
                    break
                trig_sample = e

            if len(self.samples[trig_ch]) > 10*self.settings.time_width:
                # no trig event found, delete old samples
                print "del old samples"
                trig_sample=len(self.samples[trig_ch])-self.settings.time_width
                self.trig_events = []

            for c in self.settings.active_chans:
                del self.samples[c][:trig_sample]

        elif self.settings.trig_mode == T_NONE:
            for ch in data.keys():
```

```
                del self.samples[ch][:-self.settings.time_width]

    def get_samples(self):
        for c in self.settings.active_chans:
            self._get_channel_samples(c)

    def _get_channel_samples(self, ch):
        # trigger type
        ttype = self.settings.trig_mode
        swidth = self.settings.time_width

        if ttype == T_NONE:
            self.settings.channels[ch].samples = self.samples[ch]
        else:
            if ttype == T_RUN:
                self.settings.channels[ch].samples = self.samples[ch][:swidth]
            elif ttype == T_STOP:
                self.settings.channels[ch].samples = self.samples[ch][-swidth:]
                del self.samples[ch][:]
```

spi.py

```
import logging, time
from config import *

card = None

def init(handle):
    global card
    card = handle

def sendOscillator(oct, dac, cnf = 2):
    """Send data to oscillator chip.

    f = 2^oct * 2078[Hz] / (2 - dac/1024)

    See LTC6903 datasheet for detailed meaning of parameters.
    Does not call sendData()."""

    card.clearBit(BASE_ADDRESS, SPI_CLK)
    card.clearBit(BASE_ADDRESS, SPI_OSCI_CS)

    _shiftBits(0, oct, 4)
    _shiftBits(0, dac, 10)
    _shiftBits(0, cnf, 2)

    card.setBit(BASE_ADDRESS, SPI_OSCI_CS)

def sendDAC(channel, data, command = 3):
    """Send data to DAC.

    Default command = 3 is to write voltage data and power up DAC.
    See LTC2630 datasheet. Does not call sendData()."""

    card.clearBit(BASE_ADDRESS+channel, SPI_CLK)
    card.clearBit(BASE_ADDRESS+channel, SPI_DAC_CS)

    _shiftBits(channel, command, 4)
    _shiftBits(channel, 0, 4)
    _shiftBits(channel, data, 12)
    _shiftBits(channel, 0, 4)

    card.setBit(BASE_ADDRESS+channel, SPI_DAC_CS)

def _shiftBits(channel, value, bits):
    for i in range(bits):
        if value&(2**(bits-1)):
            card.setBit(BASE_ADDRESS+channel, SPI_DIN)
        else:
            card.clearBit(BASE_ADDRESS+channel, SPI_DIN)
        _clkToggle(channel)
        value <<= 1

def _clkToggle(channel):
    card.clearBit(BASE_ADDRESS + channel, SPI_CLK)
    card.setBit(BASE_ADDRESS + channel, SPI_CLK)
```

iousb.py

```
import array, time, usb, logging
from config import *
from sampleops import usb_bulk_read1
```

```
class IODevice(object):

    """Communicate with the USB card developed by Fabrizio Lo Conte.

    The names of the methods correspond mostly to those in Fabrizio's
    CUSBControl.cpp. Buffer lengths are not given in arguments as the length
    of the list is defined implicitly in Python.
    """

    def __init__(self):
        self.dev = self._findDevice()

        if not self.dev:
            raise NoInterfaceError()

        self.handle = self.dev.open()
        try:
            self.handle.claimInterface(self.intf)
        except usb.USBError:
            raise InterfacePermissionsError()

        self.bulkIn = BULK_IN_EP
        self.bulkOut = BULK_OUT_EP
        logging.info("Device opened OK")

        self.writeBuffer = []
        self.readBuffer = []
        self.readRequests = []
        self.currentGPIFCount = 0
        self.last_written = {}

    def __del__(self):
        try:
            logging.info("closing USB interface...")
            self.handle.releaseInterface(self.intf)
            del self.handle
        except:
            pass

    def _findDevice(self):
        """Return interface handle of USB card"""
        for bus in usb.busses():
            for dev in bus.devices:
                if dev.idVendor==VEND_ID and dev.idProduct==PROD_ID:
                    logging.info("IO card found, trying to configure...")
                    self.intf = dev.configurations[0].interfaces[0][0]
                    logging.debug(
                        "Interface nr:%s class: %s subclass:%s protocol:%s",
                        self.intf.interfaceNumber,
                        self.intf.interfaceClass,
                        self.intf.interfaceSubClass,
                        self.intf.interfaceProtocol)
                    return dev

    def addDataToWriteBuffer(self, address, data):
        """Queue data for writing"""
        self.writeBuffer.append(len(data))
        self.writeBuffer.append(address)
        self.writeBuffer.extend(data)
        self.last_written[address] = data[-1]

    def addDataToReadBuffer(self, count, address, callback):
        """Queue read requests"""

        req = Request()
        req.count = count
        req.address = address
        req.callback = callback
        self.readRequests.append(req)

        self.readBuffer.append(count)
        self.readBuffer.append(address)
        self.currentGPIFCount += count+2

    def sendData(self):
        """Send data in queue"""
        tosend = []

        read_len = len(self.readBuffer)
        if read_len > 0:
            tosend.append(read_len)
```

```
            tosend.append(0xFFFF)
            tosend.extend(self.readBuffer)

        tosend.extend(self.writeBuffer)

        self._write(tosend)

        if read_len > 0:
            self.inData = usb_bulk_read1(self.handle,
                                         2*self.currentGPIFCount)
            self._sortInData()

        self.writeBuffer = []
        self.readBuffer = []
        self.currentGPIFCount = 0

    def _sortInData(self):
        """Call callback function for every received data block"""
        for req in self.readRequests:
            # first two words are not used, ignore them
            req.callback(req.address, self.inData[2:req.count+2])
            del self.inData[:req.count+2]
        self.readRequests = []

    def _write(self, buffer, timeout = 200):
        # buffer is a 16bit array but we need to write 8bit
        # therefore we convert it to a string
        b = array.array('H', buffer).tostring()
        self.handle.bulkWrite(self.bulkOut, b, timeout)

    def setBit(self, address, bit):
        """Set a specific bit."""
        word = self.last_written[address] | (1<<bit)
        self.addDataToWriteBuffer(address, [word])

    def clearBit(self, address, bit):
        """Clear a specific bit."""
        word = self.last_written[address] & ~(1<<bit)
        self.addDataToWriteBuffer(address, [word])

    def repeat(self, address):
        """Re-send the last value."""
        self.addDataToWriteBuffer(address, [self.last_written[address]])

class Request(object): pass

class NoInterfaceError(Exception): pass
class InterfacePermissionsError(Exception): pass
```

main.py

```
#!/usr/bin/env python

import logging, thread, time, sys, wx

import controlsframe, drawwindow, dso, settings, trigger
from commondefs import *
from config import *

class TracesFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, parent=None, title="DSO", size=(1000,512))
        self.window = drawwindow.DrawWindow(self, settings)

        self.timer = wx.Timer(self, id=1)
        wx.EVT_TIMER(self, 1, self.newDrawing)
        self.timer.Start(50) # 20 images / second

    def newDrawing(self,event):
        trigger.get_samples()
        self.window.updateDrawing()

class App(wx.App):
    def OnInit(self):
        frameTraces = TracesFrame()
        frameTraces.Show()

        frameControls = controlsframe.Frame(settings)
        frameControls.Show()

        frameTraces.Bind(wx.EVT_CLOSE, self.OnExit)
        frameControls.Bind(wx.EVT_CLOSE, self.OnExit)
```

```
            return True

    def OnExit(self, event=None):
        settings.halt.acquire()
        logging.info("Waiting for acquisition thread to reset device...")
        time.sleep(0.5)
        logging.info("Good-bye!")
        sys.exit()

settings = settings.Settings()
settings.channels[0].active = True
settings.channels[1].active = False
settings.trig_channel = 0

trigger = trigger.Trigger(settings)

logging.basicConfig(level=getattr(logging,LOGLEVEL))

def main():
    thread.start_new_thread(dso.DSO().start, (trigger.callback, settings))
    App().MainLoop()

main()
```

sampleops.c

```
#include <stdio.h>
#include <stdlib.h>
#include <usb.h>
#include "Python.h"

typedef struct _Py_usb_DeviceHandle {
    PyObject_HEAD
    usb_dev_handle *deviceHandle;
    int interfaceClaimed;
} Py_usb_DeviceHandle;

#define FIFO_EMPTY_MASK (1<<9)
#define FIFO_FULL_MASK (1<<10)
#define USB_READ_TIMEOUT 100
#define BULK_IN_EP 0x86

#define MAX_WRITE_BUFFER 1024

static PyObject *usb_bulk_read1(PyObject *self, PyObject *args)
{
    int size = 0;
    Py_usb_DeviceHandle *handle;

    if (!PyArg_ParseTuple(args, "Oi", &handle, &size))
        return NULL;

    char *buffer = malloc(size);
    if (!buffer) return NULL;

    Py_BEGIN_ALLOW_THREADS
    size = usb_bulk_read(handle->deviceHandle, BULK_IN_EP, buffer,
            size, USB_READ_TIMEOUT);
    Py_END_ALLOW_THREADS

    PyObject *ret = NULL;
    if (size < 0) {
        PyErr_SetString(PyExc_RuntimeError, "USB read error");
    } else {
        int i;
        u_int16_t *bufptr = (u_int16_t *) buffer;
        ret = PyList_New(size/2);
        for (i=0; i<size/2; i++)
            PyList_SET_ITEM(ret, i, PyInt_FromLong((int)bufptr[i]));
    }
    free(buffer);
    return ret;
}

PyObject *result;

static PyObject * checksamples(PyObject *self, PyObject *args)
{
    PyObject *data;
    if(!PyArg_ParseTuple(args, "O", &data))
        return NULL;
```

```
    unsigned int i, s;
    int len = PyList_Size(data);

    for(i=0; i<len; i++) {
        s = PyInt_AS_LONG(PyList_GET_ITEM(data, i));
        if (!(s & FIFO_FULL_MASK)) {
            PyErr_SetString(PyExc_RuntimeError, "FIFO full");
            return NULL;
        }
        PyList_SetItem(result, i, PyInt_FromLong(s & 0x1ff));
        if(!(s & FIFO_EMPTY_MASK)) {
            i++;
            break;
        }
    }
    Py_DECREF(data);
    return PyList_GetSlice(result, 0, i);
}

static PyObject * find_trig_events(PyObject *self, PyObject *args)
{
    static int prev = 10000;

    PyObject *data;
    int old_trig_len, level, tcond;
    PyArg_ParseTuple(args, "Oiii", &data, &old_trig_len, &level, &tcond);

    PyObject *result = PyList_New(0);

    int len = PyList_Size(data), i, s;

    for (i=0; i<len; i++) {
        s = PyInt_AS_LONG(PyList_GET_ITEM(data, i));

        if(tcond ?
            ((prev>level) && (s<=level)) : //falling edge?
            ((prev<level) && (s>=level)) ) //rising edge?
            PyList_Append(result, PyInt_FromLong(i+old_trig_len));

        prev = s;
    }
    return result;
}

static PyMethodDef sampleops_methods[] = {
    {"checksamples", checksamples, METH_VARARGS, NULL},
    {"find_trig_events", find_trig_events, METH_VARARGS, NULL},
    {"usb_bulk_read1", usb_bulk_read1, METH_VARARGS, NULL},
    {NULL, NULL} /* sentinel */
};

PyMODINIT_FUNC initsampleops(void)
{
    fputs("Using C sample analysis functions\n", stderr);

    result = PyList_New(1020);

    Py_InitModule3("sampleops", sampleops_methods, NULL);
}
```

config.py

```
# USB device vendor ID
VEND_ID = 0x04b4

# USB device product ID
PROD_ID = 0x1004

# USB device bulk in endpoint
BULK_IN_EP = 0x86

# USB device bulk out endpoint
BULK_OUT_EP = 0x02

USB_READ_TIMEOUT = 100

BASE_ADDRESS = 0xA100
READ_BUFFER_SIZE = 4096
WRITE_BUFFER_SIZE = 4096
MAX_GPIF_COUNTER = 65535
```

```
# I/O pins
FIFO_RS = 0
GAIN5 = 4
GAIN20 = 5
ATTEN10 = 6
ATTEN10_MASK = 2**ATTEN10
ACDC = 7
FIFO_EMPTY = 9
FIFO_EMPTY_MASK = 2**FIFO_EMPTY
FIFO_FULL = 10
FIFO_FULL_MASK = 2**FIFO_FULL

# SPI pins
SPI_CLK=2
SPI_DIN=3
SPI_OSCI_CS=8
SPI_DAC_CS=1

TIME_WIDTH = 500

MAX_OFFSET = 4095
MAX_OFFSET_VOLTAGE = 2.5
MAX_SAMPLE_VALUE = 511
MAX_FREQUENCY = 11000
FREQ_DIV = 16

# log level
LOGLEVEL = "DEBUG"
```

Appendix B

Software user's guide

B.1 GNU/Linux

The following programs and libraries are required to run the program:

- Python 2.4 or higher (http://python.org)
- wxPython 2.6.* or 2.8.* (http://wxpython.org)
- wxWidgets (http://wxwidgets.org)
- pyusb (http://pyusb.berlios.de)
- libusb (http://libusb.sf.net)

On a Debian or Ubuntu system, these can be installed by issuing the following command:

```
sudo apt-get install python-pyusb python-wxgtk2.8
```

By default, a Linux system does not permit the direct usage of a USB device for any user. The program will abort with an error message. There are two solutions: log in as the "root" user, or allow USB access for normal users. The latter can be done by copying the file `software/linux/90-cypressio.rules` from the CD to the directory `/etc/udev/rules.d`.

B.2 Microsoft Windows

The use of the software requires a driver which is incompatible with the driver normally used with Cypress USB devices. The Cypress driver must be completely removed before installing the new driver, which is located on the CD in the `software/windows/` directory.

Appendix C

Schemata

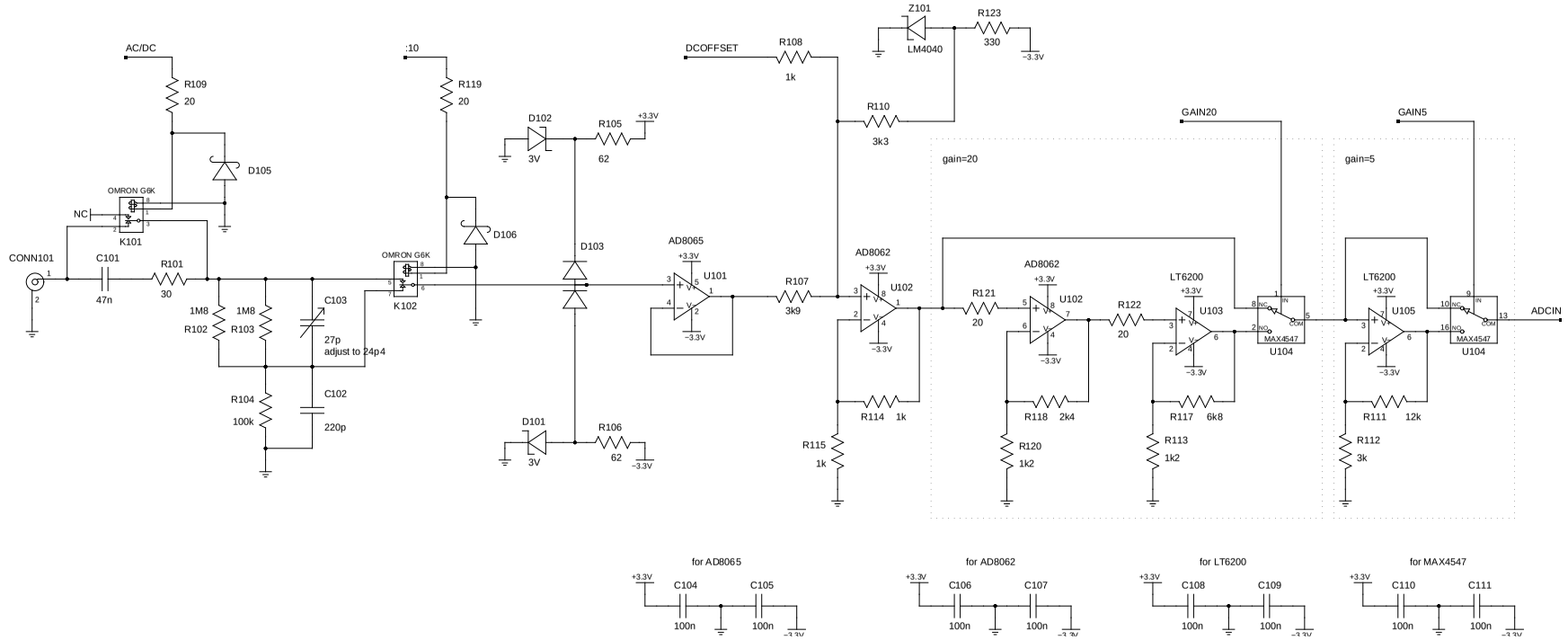


Figure C.1: Analog stage

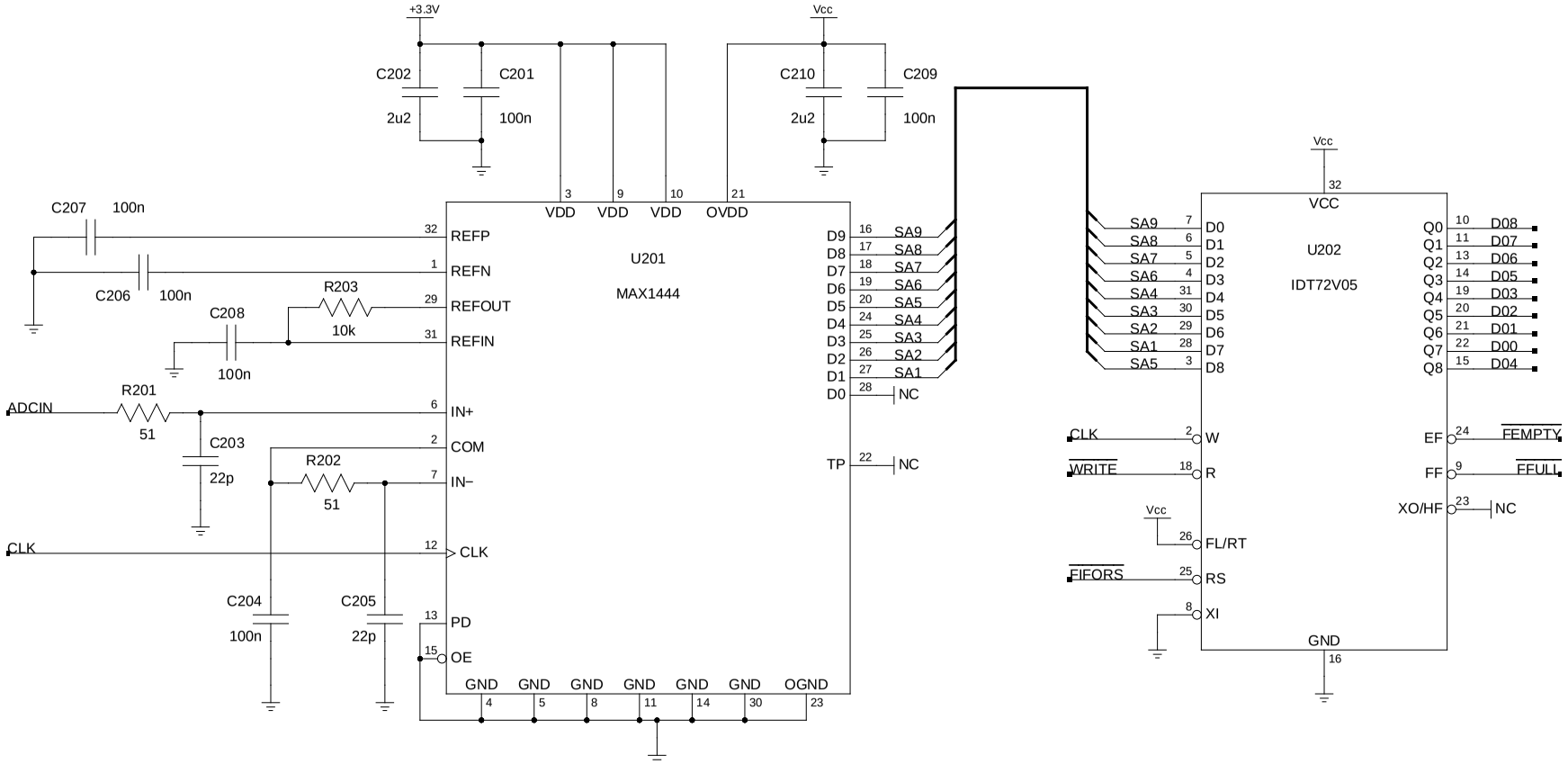


Figure C.2: ADC and FIFO

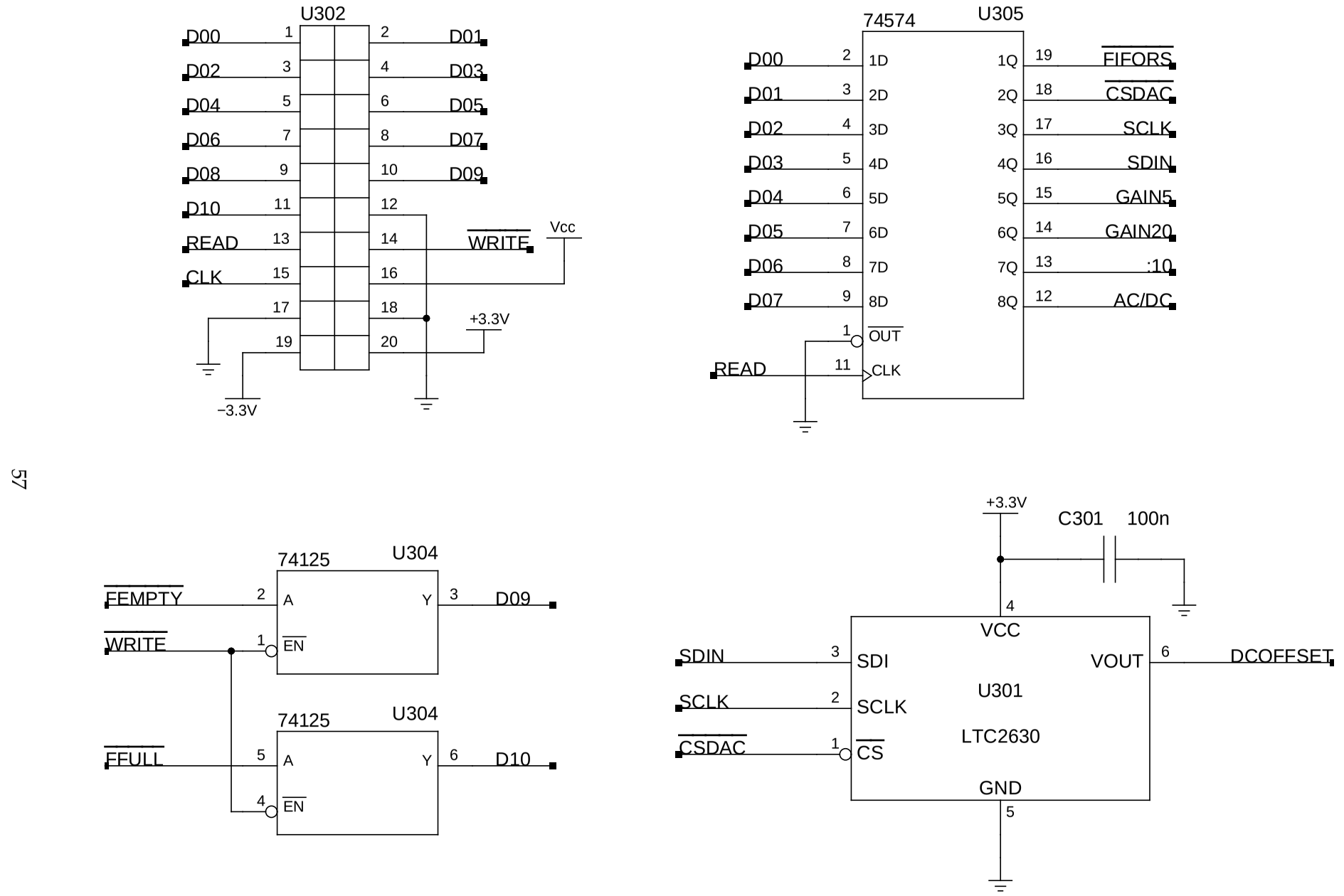


Figure C.3: Per-channel bus logic and DAC

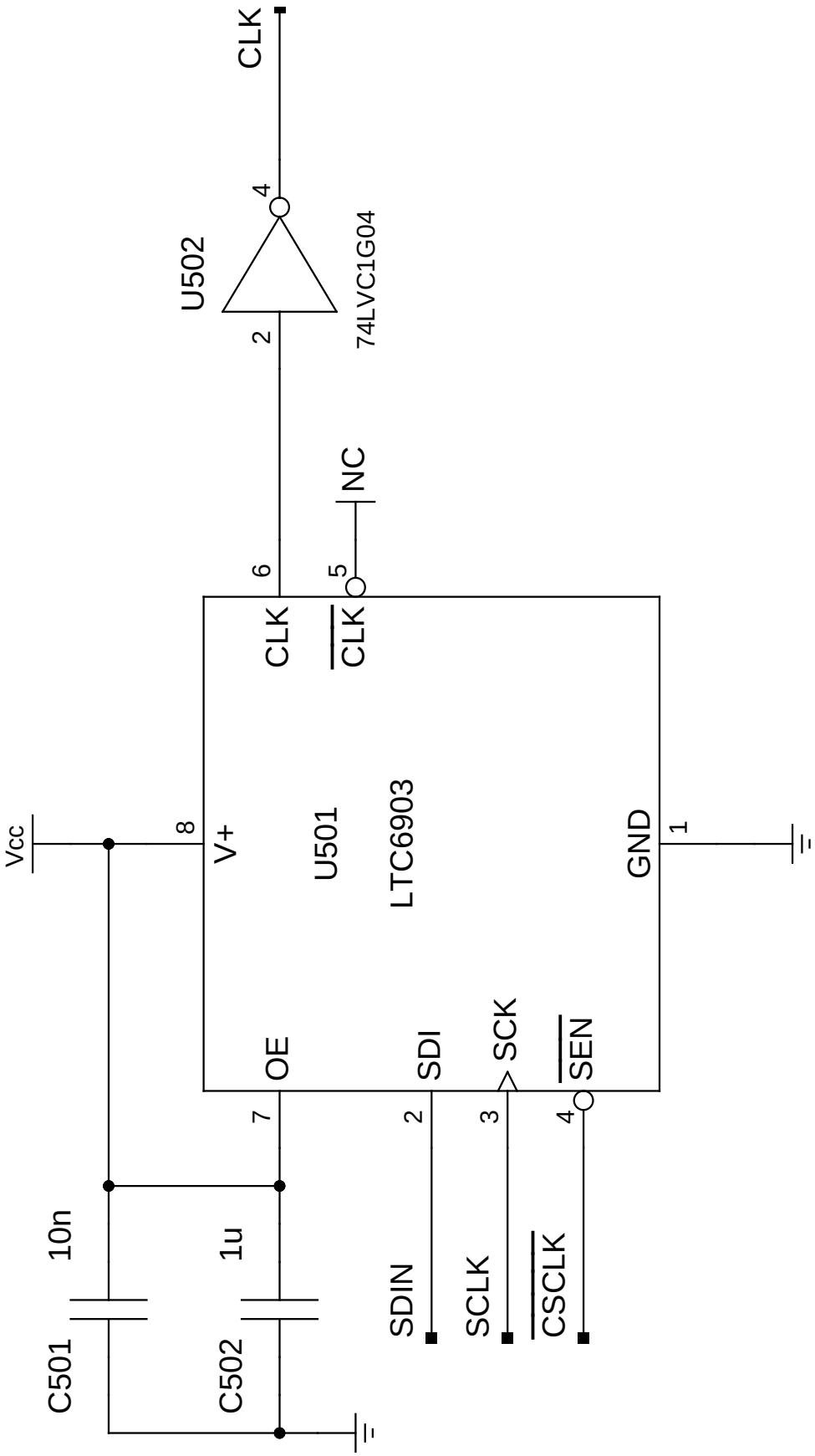


Figure C.4: Clock generation

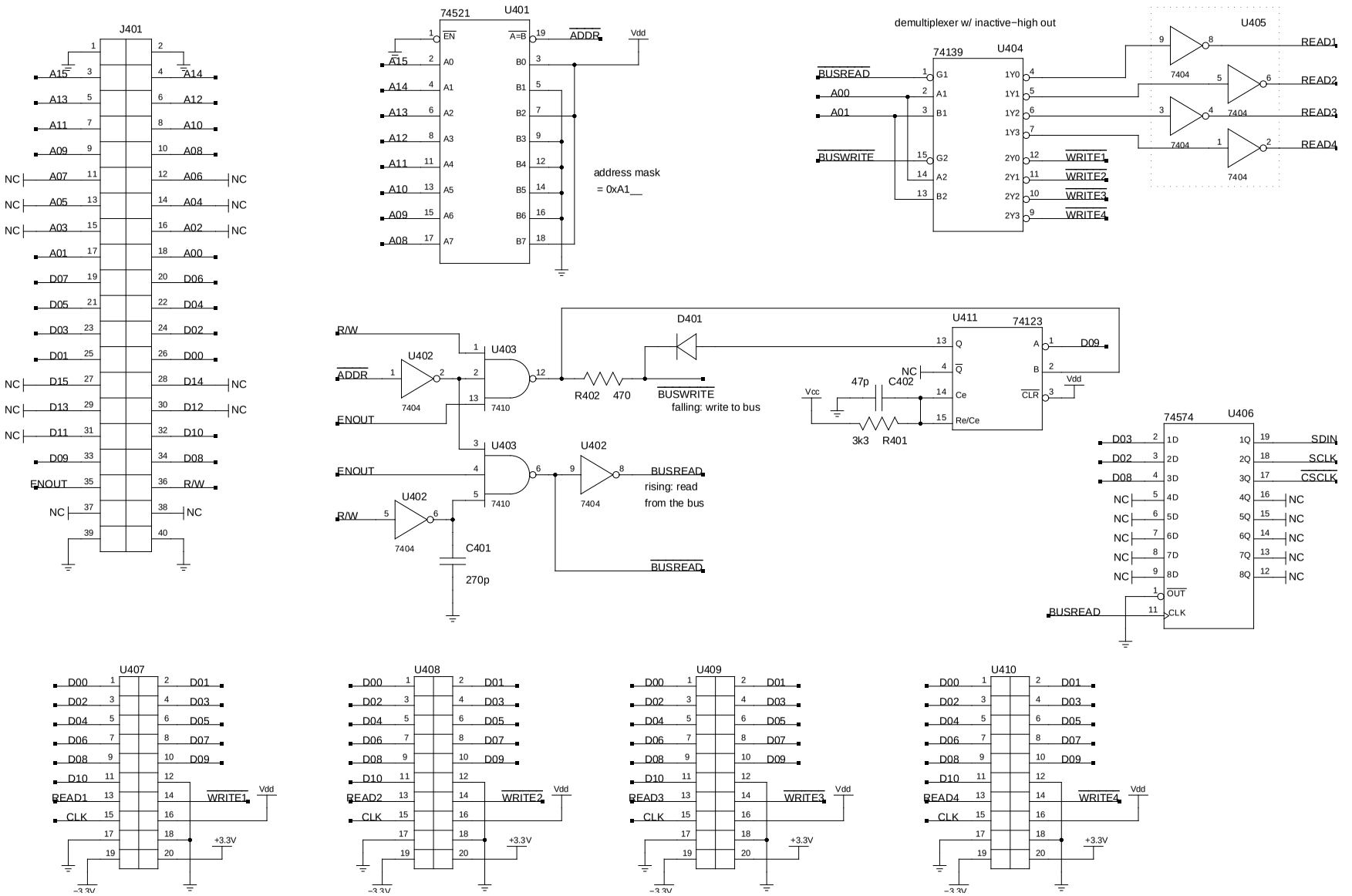


Figure C.5: Bus interface