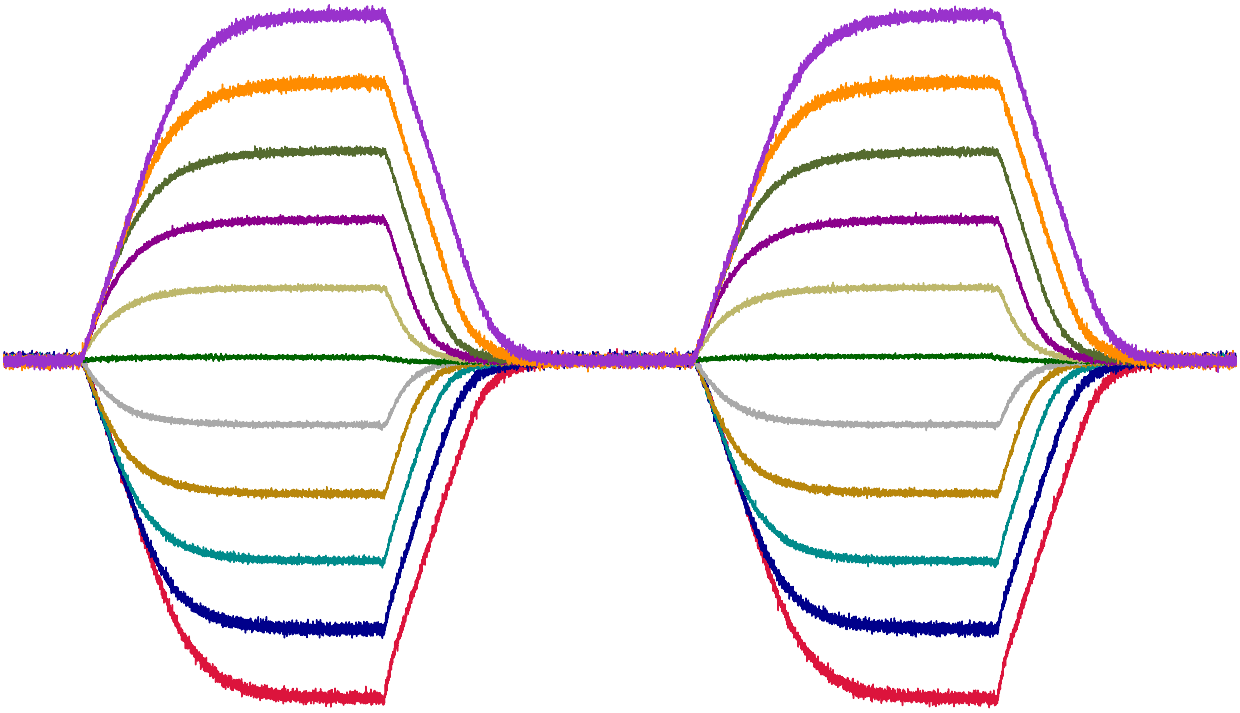
Fachhochschule Nordwestschweiz
Hochschule für Technik

Sensor Chip

Technical Report



Degree Program	Electrical Engineering and Information Technology
Course	Project 5
Advisors	Alex Huber, Hanspeter Schmid
Authors	Raphael Frey, Alex Murray
Date	January 20, 2017
Version	1.0

Created in fall semester 2016 at the FHNW School for Engineering.

This is the electronic version of this document. Hyperlinks are colored and clickable. For a version with non-colored hyperlinks, please contact rmfrey@runbox.com.

This document has been compiled 1210 times so far.

Version 0.1	10.10.2016	Creation
Version 0.2	04.12.2016	Raw Version
lersion 1.0	20.01.2017	as delivered to A.H. and H.S.

Abstract

As an ongoing project at the Institute of Microelectronics, a $\Delta\Sigma$ modulator has been in development over the past few years. Our project's objective was to develop a comprehensive test and data processing suite to efficiently and reliably assess the performance of various verions of these chips. In a second stage, the results of these measurements are to be used to further improve the $\Delta\Sigma$ modulator's design.

The test bench allows efficient measuring of multiple chips in a short amount of time and quickly evaluating the results. The measurement process has been largely automated with scripts, requiring little manual intervention. In our project, this has resulted in roughly 4000 measurements, representing about 500 million measurement points and requiring approximately 12 gigabytes of storage space. The setup has been documented so that future groups can rebuild it and reproduce our results.

For the chip design which has been evaluated, the slow rise and fall times of the pre-amplifier are the limiting factor in performance. The sampling frequency limit is around 256 kHz, which is where the OTA's gain starts decreasing drastically. Possible solutions include increasing the transconductance, resulting in a higher power consumption, decreasing the switched capacitor values, which increases susceptibility to parasitic factors, or decreasing the output load, perhaps by buffering the output instead of connecting it directly to one of the chip's leads.

Contents

1 Introduction

In his master thesis *Sensor Chip* [1], Tobias Burgherr developed a sensor chip for converting an analog input voltage to a digital output (bit stream). Broadly speaking, the chip consists of a pre-amplifier and an analog-to-digital converter.

Based on this work, two teams then each worked to improve the existing design; one focusing on the preamp [2], the other team on the ADC [3].

Our primary objective in this project will be to develop a test suite with which to conveniently and accurately measure the various versions of this chip. While some measurements have already been performed on the first chip, the amount of tests run was limited due to time constraints by the respective teams. We aim to expand on those measurements with our test suite, as well as to use them as a reference point against which we can compare our findings.

Once the two new chips become available, the methodology used in our test suite will allow efficient and accurate measuring of the two new chips in our next project.

This report focuses primarily on measurements and results analysis. For this reason, no extensive theoretical chapters are provided, and we refer the reader to the reports by the preceding teams [1], [2] and [3] as well as textbooks.

Chapter 2 presents an overview of the system hardware and software, and details how to put the setup into operation. Chapter 3 contains a description of measurement methodology and rationale, answering the questions *"What did we measure, how and why?"*. In Chapter 4, the results are presented. Finally, Chapter 5 presents our conclusions, which will form the basis for further work in our bachelor thesis.

2 Test Bench

This chapter presents a brief overview of the test bench and provides a guide on how to put it into operation.

A detailled list of all the devices, components and software which were used to obtain the measurements from Chapter 4 is given, along with information on how to make all those pieces work. This chapter therefore answers the questions *What did we use, and how does it all fit together?*, whereas the measurement methodology (*What did we do with it?*) will be answered in the next chapter.

Fundamentally, there are two kinds of measurements: Digital and analog. The digital measurements consist of the bit stream, whereas the analog measurements are concerned with measuring the pre-amplifier's characteristics. For the most part, the test benches for these two scenarios are identical. The primary difference is that for the digital measurements, the bit stream is measured on the GPIO pins of the Raspberry Pi, while for the analog measurements, data is obtained from oscilloscope. This can be done in multiple ways; the test bench as implemented in this project is based on remote-controlling the oscilloscope from a computer.

Unless otherwise noted, the information presented in the following sections applies to both types of measurements.

2.1 Overview

The complete experimental setup as used during our measurements is shown in figure 2.1. A schematic depiction of the various connections between the devices is shown in figure 2.2. The following sections describe how to configure each of the components. Table 2.1 contains a list of the devices components which are used.

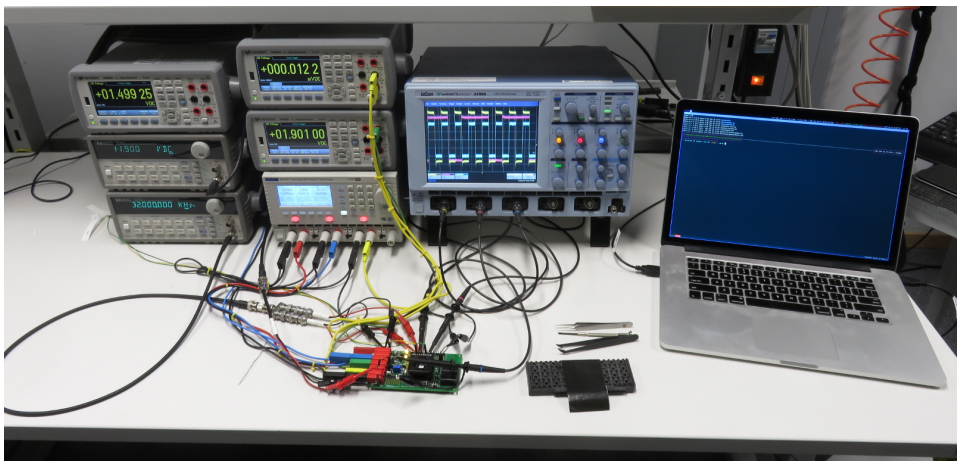


Figure 2.1: The test bench as used in our measurements. The Raspberry Pi is behind the waveform generators on the left-hand side.

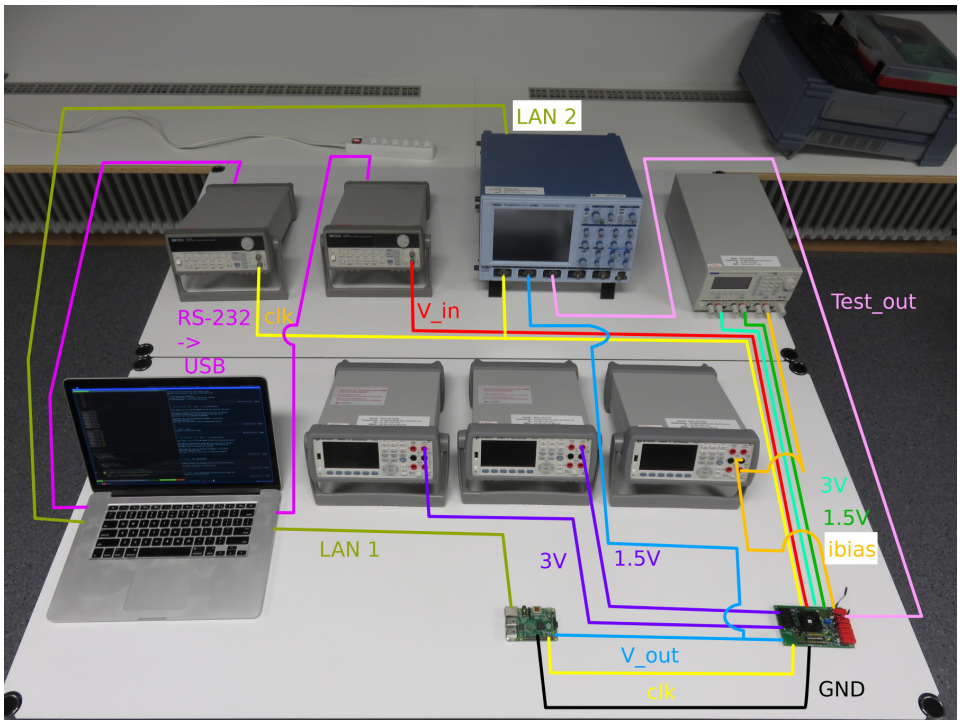


Figure 2.2: Schematic connections between the devices in the setup as used in figure 2.1. To reduce clutter, all ground connections except the one between the Raspberry Pi and the test board have been omitted.

Table 2.1: Component list

Item	Count	Notes
HP 33120A Arbitrary Waveform Generator	2	For generating input signal and clock
Keysight 34465A Tabletop Multimeter	3	Monitoring `VGNDA`, `VIN`, `IBIAS`
Aim TTi MX100TP DC Power Supply	1	Supply for `VDDD`, `VDDA`, `VGNDA`, `IBIAS`
LeCroy waveRunner 6100A 1 GHz Oscilloscope	1	Monitoring `CLK`, `TEST_OUT`, `SIGNAL_OUT`
Probes for LeCroy waveRunner 6100A	1	Measuring `CLK`, `TEST_OUT`, `SIGNAL_OUT`
Ethernet cable	1	Connecting the oscilloscope to a laptop
Raspberry Pi	1	reading and storing bit stream (`SIGNAL_OUT`), Operating System: Raspbian
Laboratory cables with industrial plugs	12	Various connections (see diagrams and pictures)
Coaxial Cables (50 Ω)	2	Connecting waveform generators to test board
Y-adapter for coaxial cable	1	
Breakout adapters for coaxial cables	3	
Measuring terminals, small	6	
Test board	1	
Sensor IC	10	Chips used: C1V1 through C1V10
USB → RS232 adapters	2	For controlling the waveform generators from a PC
USB cables	2	For controlling the waveform generators from a PC, USB A → USB-B
Computer	1	Remote-control for oscilloscope and waveform generators, Operating System: Linux or *BSD with a bash-compatible shell and the necessary Python libraries
Hand-held multimeter	1	For monitoring various parameters as needed.

Table 2.2: DIP switch assignments

Switch No.	Assignment
1	none
2	external reset for pre-amplifier
3	positive/negative sign for pre-amplifier (on: negative)
4	pre-amplifier bit 3
5	pre-amplifier bit 2
6	pre-amplifier bit 1
7	pre-amplifier bit 0
8	sample-and-hold enable
9	pre-amplifier enable
10	none

Table 2.3: Major gain settings for pre-amplifier

d_3	d_2	d_1	d_0	Gain
1	1	1	1	1
0	1	1	1	2
0	0	1	1	4
0	0	0	1	8
0	0	0	0	16

2.2 Sensor Test Board

The test board provides a platform for easy operation and configuration of the sensor chip. It has inputs for the various supply voltages, a socket for the chip itself and outputs various signals. The full list of the chip's pinout can be found in table 2.4.

The primary item to configured on the test board are the DIP switches. The assignment key for them is listed in table 2.2 . The major gains (also configurable via DIP switches) are listed in table 2.3.

The schematic for the test board can be found in Appendix B on page 55.

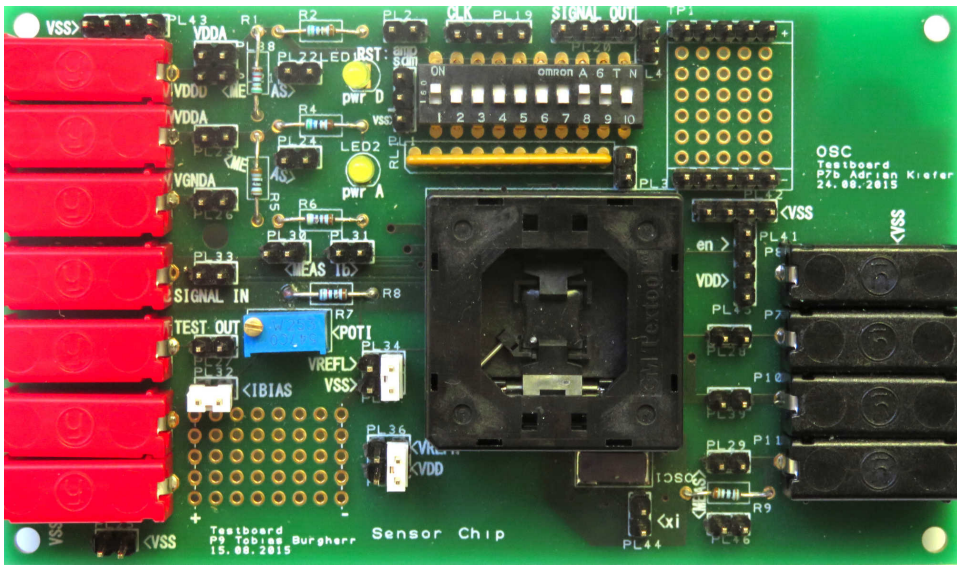


Figure 2.3: The test board.

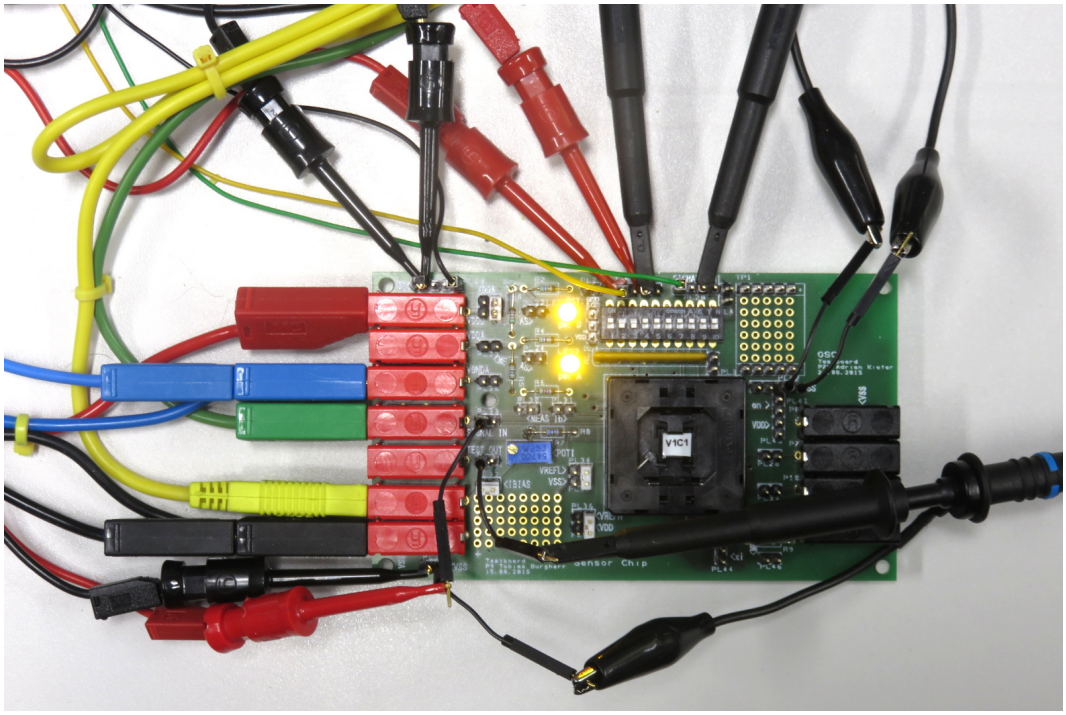


Figure 2.4: The test board with all connections as used in the configuration from figure 2.1.

Table 2.4: Sensor chip toplevel pins

Pin #	Name	Description	Value	Note
39	vin	analog input signal	0.5 V to 2.5 V	
25	bit_stream	digital output signal	0 V or 3 V	
34	clk	clock	0 V or 3 V	
36	sdm_rst	pulse to reset the $\Sigma\Delta M$	0 V or 3 V	short pulse is enough (min T_C)
41	vgnda	analog ground	1.5 V	
48	vrefh	high reference voltage	3 V	
47	vrefl	low reference voltage	0 V	
46	ibias	bias current	120 µA	$24 \cdot 5 \cdot 1$, internally reduced by 120
43	sc_amp_vout_test_en	enables test output after preamp	0 V or 3 V	0 V: off, 3 V: on
44	sc_amp_vout_test	test output after preamp	0.5 V to 2.5 V	
33	sc_amp_rst_ext_en	enables external reset input for the preamp	0 V or 3 V	0 V: off, 3 V: on
32	sc_amp_rst_ext	external reset input for the preamp	0 V or 3 V	Internally synced to **clk**. Needs **en** signal.
35	sc_amp_pos_neg_amp	positive or negative gain	0 V or 3 V	0 V: positive, 3 V: negative
28 - 31	sc_amp_csel<3:0>	set gain	0 V or 3 V	See table TODO
27	sc_amp_en	enable preamp	0 V or 3 V	0 V: off, 3 V: on
26	sah_sdm_en	enable $\Sigma\Delta M$ S/H bock	0 V or 3 V	0 V: off, 3 V: on
42	vdda	analog positive power supply	3 V	
37	vddd	digital positive power supply	3 V	
40	vss	analog and digital negative power supply	0 V	

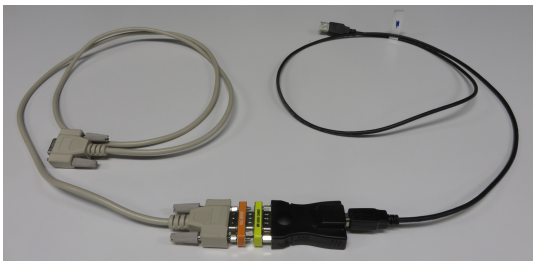


Figure 2.5: One of the two RS-232 to USB adapters with cables. In order for the adapter to properly connect with the cables, both a null modem and a gender changer are also used.

2.3 DC Power Supply

The MX100TP has three independent direct voltage outputs. They are used to supply `VDDD`, `VDDA`, `VREFH`, `VGNDA` and `IBIAS`[1]. Since the output current can only be limited, but not directly set, the bias current output is fed directly through one of the Keysight tabletop multimeters. The bias current is then monitored and the MX100TP's output voltage (and/or the variable resistor on the test board) adjusted as needed to achieve the desired bias current of 120 µA as per Table 2.4.

2.4 Waveform Generators

The HP 33120A Waveform Generators are controlled remotely via their RS232 interface from a computer (through RS232 → USB-A adapters, see Figure 2.5) and a Python script (`33120A.py`, see Appendix A on page 45).

Before usage, the script needs to be configured. After connecting via the adapters, the two waveform generators will show up in the computer's device tree and their identities must be entered into the script in the `SETTINGS` section:

```
# ------------------------------------------------------------------------ #
# SETTINGS                                                                 #
# ------------------------------------------------------------------------ #
CLK_DEVICE = '/dev/ttyUSB1'
DC_DEVICE  = '/dev/ttyUSB0'
```

Whichever device registered with the PC first will have the lower `ttyUSB<N>` number. If other USB devices are connected to the computer, the numbers will of course be higher than `0` and `1`, respectively.

After successful configuration, the waveform generators can be controlled from the command line like so[2]:

```
$ ./33120A.py --clock=32e3
$ ./33120A.py --voltage=1.8
```

[1] All 3 V signals are fed from the same output and bridged with jumpers on the test board. This concerns `VDDD`, `VDDA`, `VREFH` and `VGNDA`.

The script will then select either the `CLK SIGNAL_IN` device and adjust its output as specified. In this case, the sampling rate for the test board is set to 32 kHz and the DC voltage on the other generator is set to 1.8 V. More information can be found in the script itself.

The `CLK` device will always output a square wave with voltage levels 0 V and 3 V, as per table 2.4.

If needed, the script can easily be extended to output any other arbitrary signal. The full list of available commands can be obtained from the 33120A operator's manual, which can be downloaded at [4].

2.5 Multimeters

Three *Keysight 34465A* multimeters are used. One for monitoring the bias current, a second for monitoring the voltage for the input signal, and a third one for monitoring the reference voltage (analog ground).

No further special configuration is needed on the multimeters besides setting them to the correct measurement modes (voltage and current, respectively). However, care must be taken when setting the range for the bias current measurement. 120 µA is right on the edge between two measurement ranges (µA and mA, respectively), and the measurement results between these two ranges are not identical. A difference of about 7 µA was typically observed in our experiments. In our measurements, the device is set to the millivolt range.

The user manual for the multimeter can be downloaded at [5].

2.6 Oscilloscope

The oscilloscope is used for monitoring the clock, the bit stream, and for acquiring measurment data for the preamp via the test output pin.

2.6.1 Clock and Bit Stream

Monitoring the clock and bit stream is straightforward: The oscilloscopes probes are connected to the corresponding pins on the test board as depicted in Figure 2.6.

If the clock and bit stream do not look as they should, common errors are a disabled sample-and-hold circuit, an incorrect clock voltage (particularly its DC offset) or an incorrect bias current.

2.6.2 Analog Measurements of Preamp

Performing analog measurements of the pre-amplifier is a noticeably more involved process. A measurement probe is connected to the test output pin on the test board as shown in Figure 2.7.

[2] This requires the executable bit to be set on the file: `chmod +x 33120A.py`.

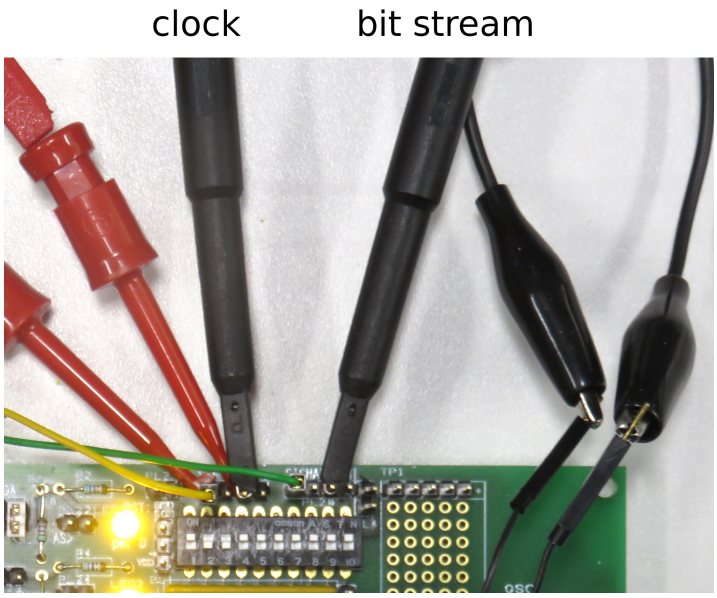


Figure 2.6: Clock and bit stream monitoring. *Note*: Since the clock input pins are not interconnected, and both the Raspberry Pi and the oscilloscope are connected to one clock pin each, two supply clamps (in red) are used on the clock input via a Y splitter from the coaxial cable coming from the signal generator. This is not strictly necessary, but makes connecting the probes, plugs and supply clamps easier without them slipping off the pins.

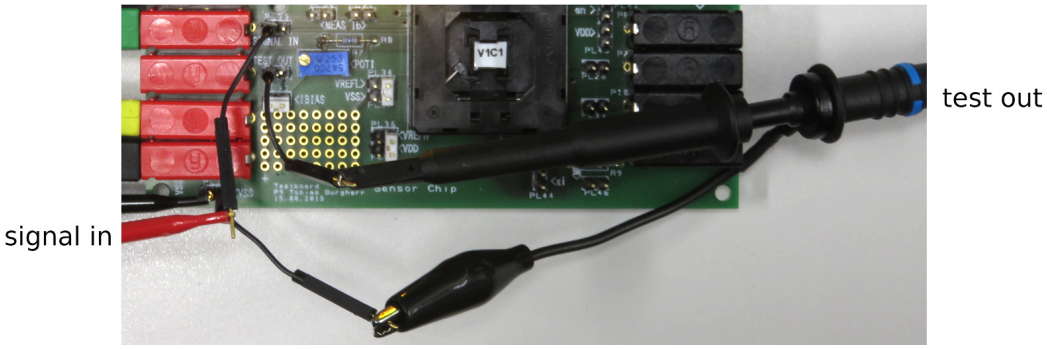


Figure 2.7: Measuring preamp output. On the left, the tip of the clamp for the input signal can be seen, coming from one of the waveform generators.

Besides the oscilloscope itself, a laptop is used to remotely control the oscilloscope and the two function generators (one for the clock and for the input signal, respectively).

The connection between the oscilloscope and the laptop is established via ethernet, with each device being assigned a fixed IP address. The laptop is configured to match the oscilloscope's network settings. How to configure the laptop is outlined in Section 2.8, starting on page 17.

After having successfully established a connection between laptop and oscilloscope, a collection of three scripts is used to configure the test setup and acquire the measurement data:

- A shell wrapper script,
- `33120A.py`: For controlling the waveform generators (see section 2.4)
- `initWaveRunner.py`: resets and initializes the oscilloscope with a given configuration.
- `configWaveRunner.py`: configures various settings on the oscilloscope.
- `acquireWaveRunnerData.py`: performs measurements via the oscilloscope.

The python scripts are mostly self-explanatory with aid of LeCroy's documentation found in [6], [7] and [8]. A detailled explanation of their implementation will therefore be ommitted, and instead a brief outline of each script's purpose and usage is provided. They can be found in Appendices A.2, A.3 and A.4, starting on page 47.

The initialization script resets the oscilloscope and configures the trigger channel, trigger level, trigger flank, time division, which traces to display and sets the DC coupling of the channels to a resistance of 1 MΩ.

`configWaveRunner.py` is used to set the horizontal (time) and vertical (voltage) divisions on the oscilloscope, as well as the vertical offset. This is necessary because the oscilloscope will only store measurement data for the section of any given trace which is being shown on the display. Any meaurement points which are not on the oscilloscope's display will not be stored in a measurement file. Although this is a bit cumbersome and counter-intuitive, it does have the advantage that one can see immediately if something has been incorrectly configured during the measurement process, as the display output will then be incorrect.

`acquireWaveRunnerData.py` executes the actual measurement on the oscilloscope for a given channel. The measurement results are stored in a text file, which is then downloaded to the laptop.

The entire process is controlled via a shell wrapper script. This script will be explained in more detail in the following paragraphs. There are various versions of the script for different measurements, but they all follow the same outline.

For each measurement run, the chip number, the gain (sign and absolute value) and the channel which is connected to the test output on the test board must be set. Furthermore, the target directory in which the measurement data will be stored on the PC is configured according to the chip number and gain settings. There is also an optional flag for resetting the oscilloscope before performing the measurements,

if so desired. In the following example, a gain of +1 will be measured for chip number 1:

```
CHIP='chip01'
SIGN='+'
CHANNEL='3'
GAIN='1'
DATA_DIR="data/${CHIP}Gain${SIGN}$(printf '%02d' $GAIN)"
RESET='0' # 1: reset and initialize the oscilloscope
```

Before actually performing any measurements, the script will display a series of messages to the user, accounting for a few quirks of the setup. This minimizes user error and ensures that time is not wasted on unusable measurements due to erroneous configurations.

The first message concerns the number of sample points which the oscilloscope is set to acquire. This must be set to 500 kilo samples per second, as shown in Figure 2.8. If this is not done, the measurement data will exhibit strong quantization errors, resulting in unusable data.

The next two messages concern the measurement files which are stored on the oscilloscope. They follow a naming scheme of the form `C<ChannelNo>Trace<FileNo>.txt`, starting at file zero (example: `C2Trace00002.txt` for the third file for channel 2). Each channel's counter is independent of the other channels' counters. The counter for these file names will keep incrementing, regardless of any remote commands, until they are manually reset via the Oscilloscope's graphical user interface.

To avoid needing to manually reset this counter between each measurement, the script maintains its own counter to match the oscilloscope's. Between each measurement run (i.e. each script execution), manual intervention via the oscilloscope's GUI is necessary to reset the oscrilloscope's counter, as shown in figure 2.9.

The next step in the script's execution is performing the actual measurements. Various settings are hard-coded into the script for this purpose. Firstly, the appropriate time divisions for configuring the oscilloscope's display[3]:

```
declare -A timeDivs
timeDivs[32e3]='5'
timeDivs[96e3]='2.5'
timeDivs[256e3]='1'
```

Additionally, the list of amplitudes to be used for the input voltages:

```
declare -a ampls=(0.5 0.7 0.9 1.1 1.3 1.5 1.7 1.9 2.1 2.3 2.5)
```

Each input voltage (including the gain's sign) is assigned a voltage division and a time division for the oscilloscope's display settings[4]:

[3] These values were obtained by trial and error to generate a sensible display output.

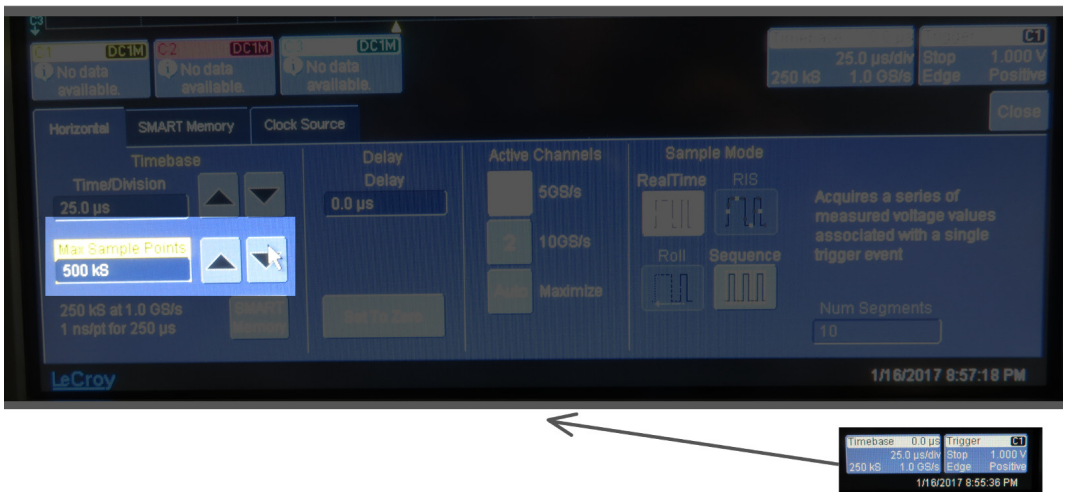


Figure 2.8: Set sample rate on oscilloscope. Click the *Timebase* icon on the bottom right of the screen, then adjust the *Max Sample Points* setting (highlighted) to 500 kilo samples in the dialog which opens.

Figure 2.9: Resetting the file counter on the oscilloscope. Click the *File* menu at the top of the screen, then select the *Save Waveform* dialog. If the oscilloscope has been reset, select the appropriate channel, DataFormat (ASCII) and the directory where the files are to be saved (which must correspond with the directory set in the wrapping shell script). After initial configuration, this dialog must be opened between each script run to reset the file name counter, but no further intervention is necessary.

```
declare -A voltDivs
voltDivs[+0.5]='200'
voltDivs[+0.7]='200'
voltDivs[+0.9]='100'
...
voltDivs[-2.1]='100'
voltDivs[-2.3]='200'
voltDivs[-2.5]='200'

declare -A offsets
offsets[+0.5]='-1000'
offsets[+0.7]='-1100'
offsets[+0.9]='-1200'
...
offsets[-2.1]='-1200'
offsets[-2.3]='-1100'
offsets[-2.5]='-1000'
```

Now, everything is ready to perform the measurements, tying together all the scripts and configurations which have been mentioned so far:

```
i=0
mkdir -p "$DATA_DIR"
for clk in 32e3 96e3 256e3;do
    ./33120A.py --clock=$clk
    sleep 0.25 # give the function generator time to settle
    ./configWaveRunner.py --tdiv=${timeDivs[$clk]}

    # Run through DC ramp
    for ampl in "${ampls[@]}";do
        ./33120A.py --voltage="$ampl"
        sleep 0.25
        ./configWaveRunner.py \
            --vdiv=${voltDivs[${SIGN}${ampl}]} \
            --offset=${offsets[${SIGN}${ampl}]}
        sleep 2 # Give the oscilloscope time to do configure
        printf 'Acquiring trace for %s Hz and %s V\n' $clk $ampl
        ./acquireWaveRunnerData.py \
            --channel="C${CHANNEL}" \
            --remotefile="C${CHANNEL}Trace$(printf '%05d' $i).txt" \
            --localfile="${DATA_DIR}/${CHIP}-gain${SIGN} \
                $(printf '%02d' ${GAIN})-${fsStrings[$clk]}-${ampl}V.txt"
        i=$((i+1))
    done
done
```

For each measurement series, one target directory is created, in which one file for each clock frequency and input voltage is generated. These files can then be processed as outlined in Chapter 4.

[4] Also determined by trial and error.

2.7 Raspberry Pi

Installation of the operating system is fairly straight forward. We recommend using the NOOBS [9] OS installer.

The first thing to do is to set up SSHD so that a remote connection with the Raspberry Pican be established. This isn't strictly required, if preferred, a monitor and keyboard connected directly to the Raspberry Pi can also be used. The scripts used measuring measuring data and transferring the data to our computers do rely on SSH, though.

cmake and gcc need to be installed, if they're not already. To check whether the programs exist, type the following into a terminal:

```
$ cmake --version
$ gcc --version
```

If either return a "command not found" then you will need to install them with:

```
$ sudo apt-get install cmake
$ sudo apt-get install gcc
```

The Raspberry Pi has dhcpcd running by default, which means it will automatically lease an IP as soon as it is connected to the network. The Raspberry Pi will need to be whitelisted by the IT department in order to be able to establish a remote connection. This involves finding out its MAC address. This can be done with the following command:

```
$ sudo ifconfig
eth0      Link encap:Ethernet  HWaddr b8:27:eb:e2:d2:c6
          inet addr:10.84.130.16  Bcast:10.84.130.255  Mask:255.255.255.0
```

`HWaddr b8:27:eb:e2:d2:c6` is the information needed.

The software for recording the bit stream can be found in the svn repository. It can be copied to the Raspberry Pi using SSH:

```
$ scp -r path/to/software pi@10.84.130.16:/home/pi
```

This will place the software folder into the home folder on the Raspberry Pi. Next, open a terminal and navigate into the software folder. Execute the following commands to configure and compile the software:

```
$ mkdir build
$ cd build
$ cmake -DCMAKE_BUILD_TYPE=Release ../
$ make
```

This will produce an executable *bitstream*. You can initiate a measurement by executing:

```
$ sudo ./bitstream
```

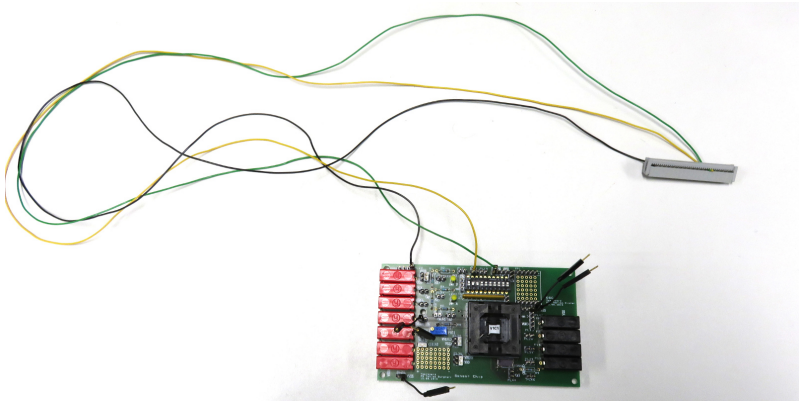


Figure 2.10: Raspberry Pi cable with wires for clock, ground and bit stream, connected to the test board

If the chip is connected and running properly, the program will exit after a few seconds and a new file *bit_stream.txt* will exist in the folder.

On the hardware side, not much configuration is needed. A dedicated cable with a 40-pin plug has been created to easily connect the Raspberry Pi with the required pins on the test board (bit stream, clock and ground). This cable is shown in Figure 2.10, connected to the test board.

2.8 Laptop

This section outlines the prerequisites, configuration and usage of a computer which is used to control the measurement process. A laptop is used in our case, but any computer with a network and at least two USB ports (for the waveform generators) can be used.

2.8.1 Prerequisites

For controlling the test bench, a laptop running Linux is used. Fundamentally, other configurations are likely also possible, but have not been tested. Any Bash version 4.0 and newer should be able to execute the wrapper scripts (associated arrays must be supported). Even Windows 10 has Bash as well these days [10], for the adventure-minded.

Besides Bash $\geq$ 4.0 support, Python 3 must be installed, along with the following modules:

- `serial`: For communicating with the waveform generators
- `vxi11`: For communicating with the oscilloscope
- `sigmadelta`: For processing the bit stream

These modules can either be installed via pip or via your distribution's package manager.

The Raspberry Pi is remote-controlled via SSH from the laptop. In order to avoid having to type in a password for each SSH connection attempt (which would become cumbersome very quickly), SSH key files are used. For information on how to generate and install SSH keys, there are many online resources, for example the Arch Linux Wiki [13].

2.8.2 Ethernet Configuration

For communicating with the oscilloscope via ethernet, a static network connection is used, corresponding to the settings on the oscilloscope (which can be checked via the network dialog in the oscilloscope's Windows XP operating system).

On the machine used in our measurements (a Macbook Pro running Arch Linux), this is done via `netctl` as follows[5]:

```
Description='WaveRunner Oscilloscope Ethernet Connection'
Interface=ens9
Connection=ethernet
IP=static
Address=('169.254.14.1/16')
```

The `Interface` parameter needs to be adjusted to the machine which is being used. A list of available interfaces can be displayed with `ip addr` on systems which have the `iproute2` utility suite [11] installed, or with `ifconfig` [12] on other *nix operating systems. For further information, consult your distribution's manpages.

[5] Make sure *not* to set a `Gateway` parameter in this configuration, or the laptop will try to access the internet via the oscilloscope, which will obviously fail.

3

Measurements and Data Processing

This chapter elaborates on what measurements were performed and what special considerations, if any, had to be taken into account. Additionally, the work flow for the processing of the obtained data is explained.

The goal in this project is to create an efficient, streamlined workflow for measuring multiple sensor chips and correlating the results in meaningful ways. The conclusions drawn in this report mostly originate from DC measurements; A number of known voltages are applied to the chip's input and its output is observed. There are multiple configurations to do this with, namely in function of the preamp gain and the sampling frequency fs, which in turn can be done again as a whole for just the preamp, for just the $\Delta\Sigma$ modulator or for both combined (the whole chip). These measurements were performed on 10 different chips that all originated from the same wafer. This gives us some statistical reinforcement in the results, assuming the whole wafer is not biased of course.

Figure 3.1 shows a rough overview of the chip's components.

The preamp is responsible for amplifying the difference of an input voltage to a 1.5 V reference voltage by an externally configurable factor (either by 1, 2, 4, 8, or 16). Additionally, the amplifier can be set to invert the gain.

The SAH (Sample and Hold) block works together with the $\Delta\Sigma$ modulator to convert the analog voltage from the preamp into a digital bit-stream.

There are two different kinds of measurements: The results from measurements involving the $\Delta\Sigma$ modulator, which is recorded using a GPIO pin on a Raspberry Pi. Here, the input signal is either applied to the preamp (when measuring the whole device) or to the `TEST OUT` pin (when measuring the $\Delta\Sigma$ modulator alone). The output from the preamp on the other hand is analog, so it needed to be measured using an oscilloscope via the `TEST OUT` pin.

The sample size is 10 chips, unless indicated otherwise.

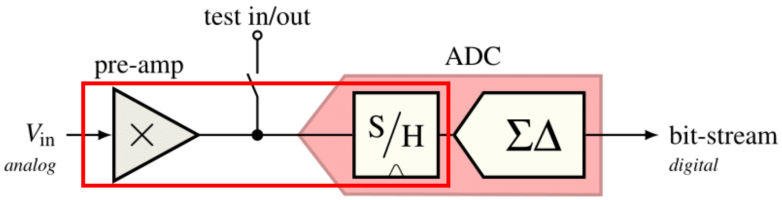


Figure 3.1: Rough block diagram of the sensor chip

3.1 Measurements

In total, ten chips have been evaluated. While this number is far below the 1027 samples which would be needed to obtain a range which contains, with a probability as low as 75 %, all but 0.27 % of the chips coming from the same manufacturing batch (see [14]), it does give at least some additional statistical confidence in our results beyond measuring simply a single sample.

The measurements were conducted for a range of DC input signals. This allows to assess linearity, correctness of gain, offset and time constants for the pre-amplifier.

The differential amplifier outputs a signal which oscillates between the reference voltage (1.5 V) and the difference between the input voltage and the reference voltage, multiplied by the signed gain, as per equation 3.1:

$$V_{\text{out,preamp}} = (V_{\text{in}} - 1.5\,\text{V}) \cdot \text{Gain} \tag{3.1}$$

Below 0.5 V and above 2.5 V, the circuit goes into saturation, so keeping $V_{\text{out,preamp}}$ inside that range is desired. Consequently, the range of sensible input voltages changes with the gain. Table 3.1 presents an excerpt of the test suite as it was performed on the pre-amplifier of a single chip for a single sampling frequency and positive gains.

The process of performing the measurements is based on the scripts documented in the previous chapter.

3.2 Data Processing

Condensing the large amount of measurement data into useful, manageable information is achieved via a two-stage approach.

The first stage is performed by the python script `parse_all_measurements.py`. It goes through all bit stream files, applies the sinc/decimate filter to each one, and recovers the measured DC voltage. Additionally, it goes through all pre-amplifier measurements and fits a first order low pass filter model to the data. It also determines the voltage that would be passed to the sample-and-hold block based on the fitted model. The recovered DC values, various fit parameters, and standard deviations are written to the XML file `processed_measurements.xml`.

Table 3.1: Measurement suite for a single chip for the pre-amplifier, with positive gain and a sampling frequency of 32 kHz. This was repeated for sampling frequencies 96 kHz and 256 kHz, respectively, for negative gains of identical magnitude, for the $\Delta\Sigma$ modulator and for both elements, for nine more chips, resulting in roughly 4000 measurements.

Gain	Input Voltage	Element Measured
+1	0.5 V	Preamp
+1	0.7 V	Preamp
...		
+1	2.3 V	Preamp
+1	2.5 V	Preamp
+2	1.0 V	Preamp
+2	1.1 V	Preamp
...		
+2	1.9 V	Preamp
+2	2.0 V	Preamp
+4	1.25 V	Preamp
+4	1.30 V	Preamp
...		
+4	1.70 V	Preamp
+4	1.75 V	Preamp
+8	1.375 V	Preamp
+8	1.425 V	Preamp
+8	1.475 V	Preamp
+8	1.500 V	Preamp
+8	1.525 V	Preamp
+8	1.575 V	Preamp
+8	1.625 V	Preamp
+16	1.450 V	Preamp
+16	1.475 V	Preamp
+16	1.500 V	Preamp
+16	1.525 V	Preamp
+16	1.550 V	Preamp

The first stage is responsible for reducing all of the raw data, so it takes quite a while to process (approximately 1 hour on an Intel i5-2410M).

The second stage is performed by the python script `calculate_fits.py`, which does some higher-level processing on the data resulting from the first stage. This includes a linear fit on the expected and measured DC voltages as well as forwarding/preparing other data for presentation. The results from this stage are stored in `fitted_data.xml`.

The last script, `display_results.py`, uses the fitted data from the second stage to create plots.

4

Results

This chapter presents the measurement results, processed data and some analyisis of the results.

4.1 Pre-Amplifier: DC Measurements

The preamplifier was directly measured, isolated from the $\Delta\Sigma$ modulator, by applying a number of constant voltages on the input and recording the output voltage with an oscilloscope on the `TEST OUT` pin.

Since the preamp uses a switching capacitor implementation, the output waveform is of course a square wave of sorts, as can be seen in figure 4.1.

A first observation to make are the slow rise and fall times. Indeed, this is one of the major limiting factors of the chip's performance; If the target voltage can't be reached within half of the clock period (before it is reset again), the $\Delta\Sigma$ modulator will end up converting a lower voltage than expected.

For every chip, for every sampling frequency, for every gain, inverted and non-inverted, 11 different input voltages were applied and the output was measured. All of the data was then fitted using a simple first order RC low-pass charge/discharge model, described by the following python code:

```
def preamp_curve(t, amp, amp_offset, period, t_offset, duty_cycle, tau1, tau2):
    def discharge_segment(t, amp, tau):
        return amp * np.exp(-t / tau)
    def charge_segment(t, amp, tau):
        return amp * (1.0 - np.exp(-t / tau))
    t_local = (t+t_offset) % period
    t_switch = period * duty_cycle
    def preamp_curve_single():
        for t_value in t_local:
            if t_value < t_switch:
                yield charge_segment(t_value, amp, tau1) + amp_offset
```

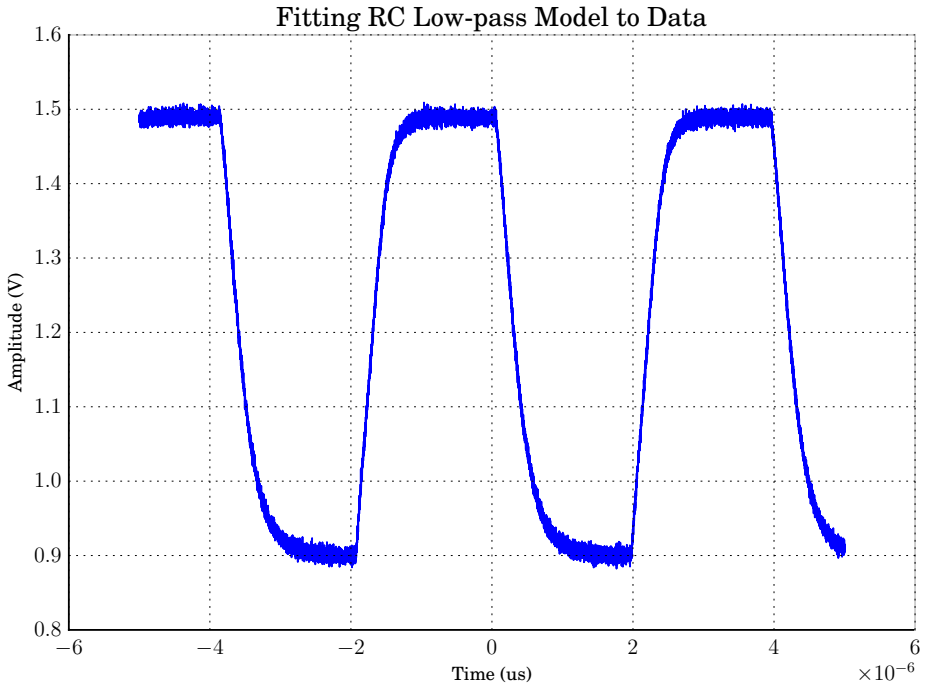


Figure 4.1: Typical preamp output signal as measured on the `TEST OUT` pin.

```
        else:
            yield discharge_segment(t_value - t_switch, amp, tau2) + amp_offset
return np.array([x for x in preamp_curve_single()])
```

The initial parameters for the fit were obtained by first smoothing the data (using a Savitzky-Golay filter) and finding the transition points. This process is illustrated in figure 4.2. This was necessary because there are a lot of local minima the fit could fall into. The initial period, duty cycle, and phase shift can be extracted from the distance between the transition lines. The initial amplitude and offset can be determined with a min/max search.

The most interesting parameters yielded from the fit are τ_1 and τ_2, which give us a rough idea of the transconductance g_m of the OTA. The gain can also be easily extracted after the fit by performing a min/max search on the fitted curve. This is illustrated in figure 4.3 as the horizontal line.

It can be observed that the transconductance g_m is largest when near the reference voltage $V_{ref} = 1.5\,\mathrm{V}$ and decreases with higher input amplitudes. This effect can be seen in figure 4.4. Furthermore, in the same figure, one can observe slewing on the larger signals. This effect could be included in the fit model above in future evaluations.

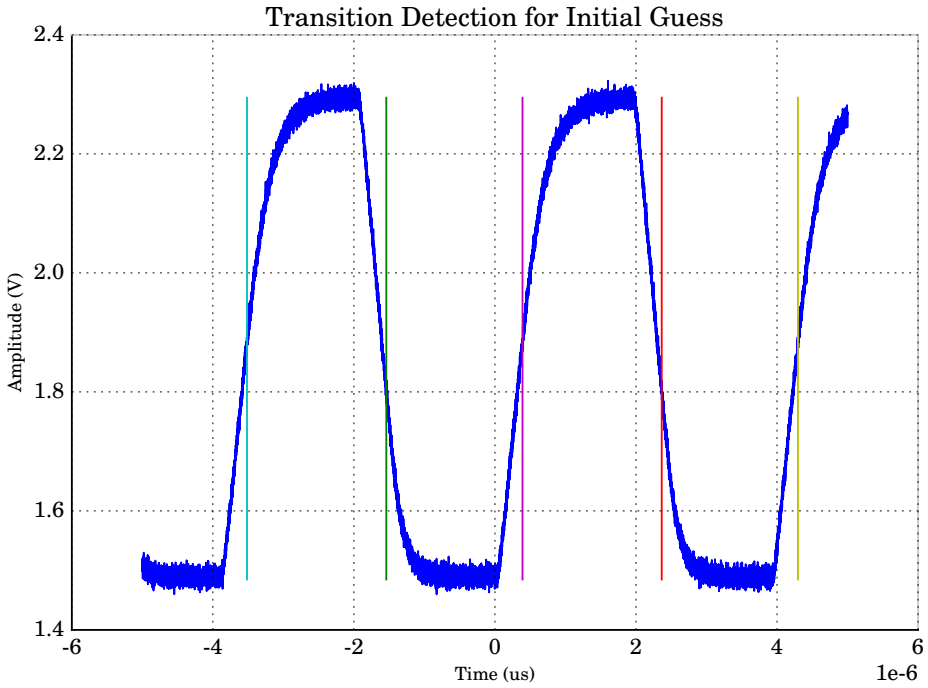


Figure 4.2: Finding the transition points as a method to accurately estimate initial parameters for duty cycle, period and phase.

By combining the individual τ constants and plotting them in function of the input voltage, we see the effect of g_m decreasing more clearly (figure 4.5). Here, the results from 10 chips were statistically averaged to get a better result.

The τ constants appear to be frequency dependent. This doesn't make much sense and is very likely a direct result of using a very rough model to fit the data.

By plotting the output voltage from the preamp in function of the input voltage and fitting a linear function $y = mx+q$ to the data, we expect a linear function whose incline is equal to that of the configured gain. Additionally, we expect a small offset to exist there, due to non-symmetry in a non-ideal (i.e. real) differential amplifier. This process is visualised in figure 4.6.

The upper plots in Figures 4.7, 4.8, 4.9, 4.10 and 4.11 show all positive gains (1, 2, 4, 8, 16) for every chip (x axis) in function of fs. There appeared to be a correlation between sampling frequency fs and gain, which made sense at the time, since at higher sampling frequencies, the output of the preamp no longer manages to properly reach the target voltage in time. The lower plots in these figures show the gain in function of sampling frequency.

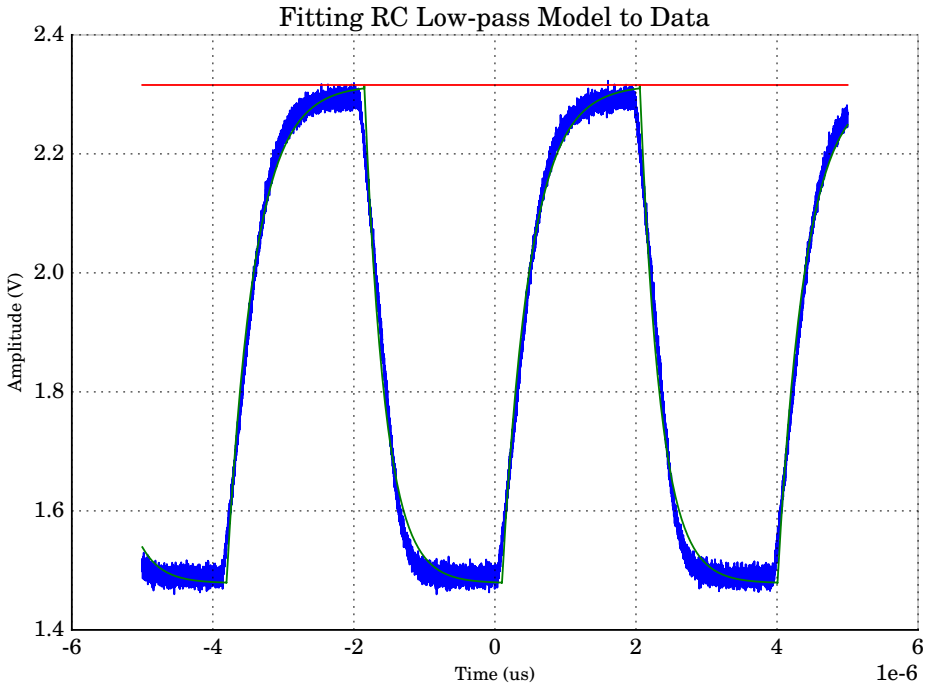


Figure 4.3: Fitting the RC model to the raw data and determining the preamp output voltage.

It is interesting that as the configurable gain is increased, the measurements with a higher sampling frequency decrease in effective gain. There is currently no explanation as to why this is the case, it is open to further investigation.

When the gain is inverted, we see more of the same behaviour. The same measurements but for negative gains can be seen in figures 4.12, 4.13, 4.14, 4.15 and 4.16.

Another interesting figure to look at is be the offset of the pre-amplifier. In much the same way, the offsets are presented in function of gain and sampling frequency in Figure 4.17.

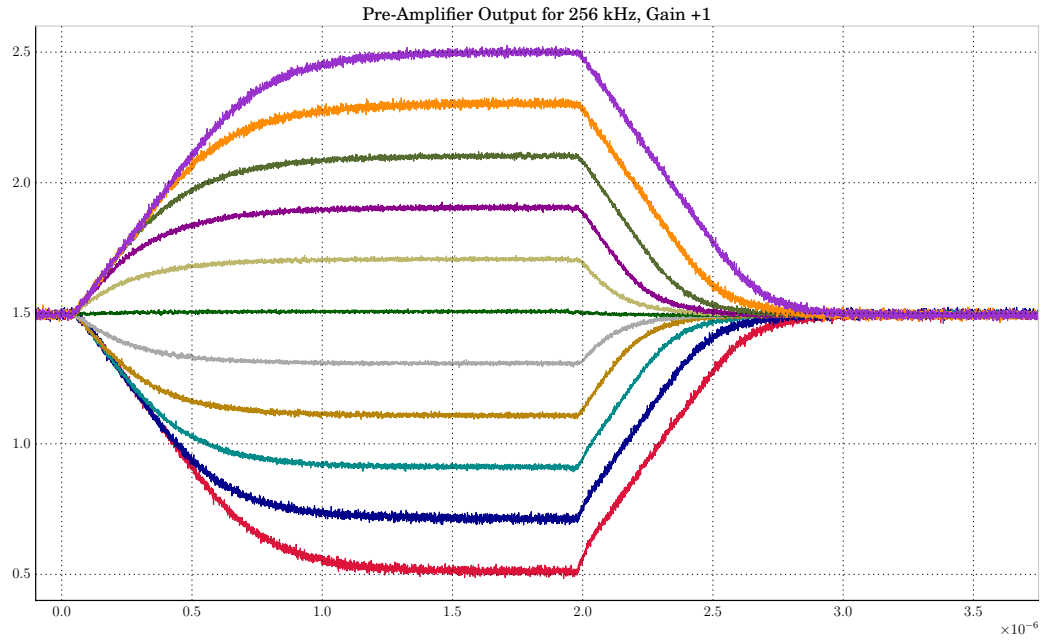


Figure 4.4: Output voltage of the pre-amplifier for 11 different DC input voltage levels.

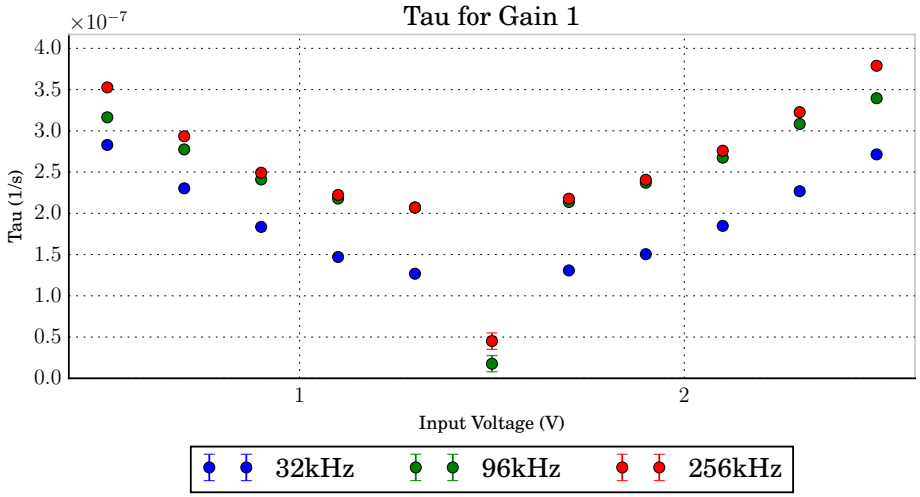


Figure 4.5: Fitted parameter tau (RC time constant) in function of DC input voltage.

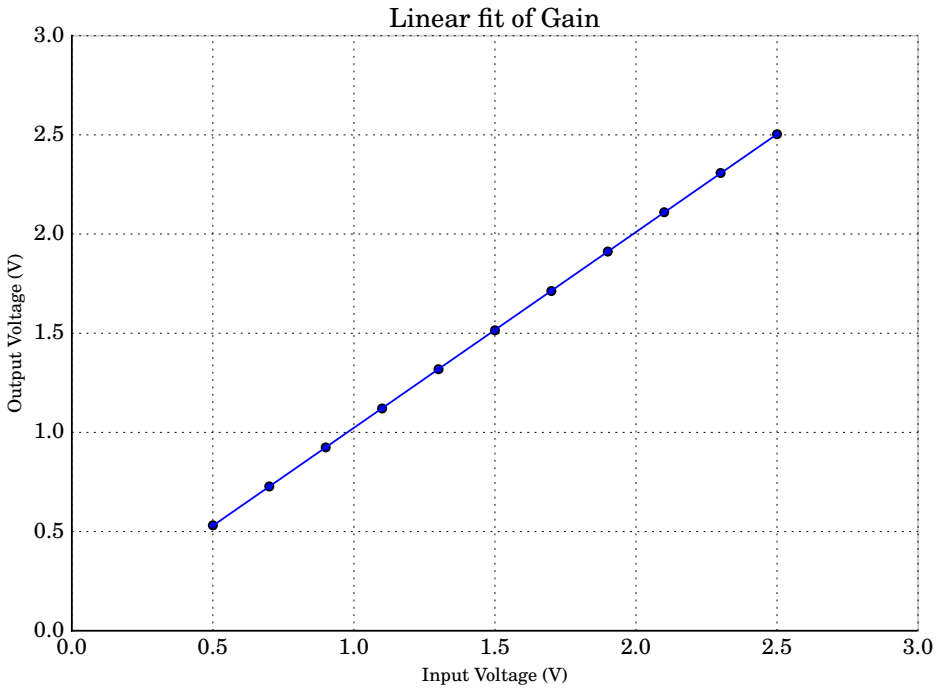


Figure 4.6: Linear fit of DC input vs output voltage. The resulting slope is directly the effective gain.

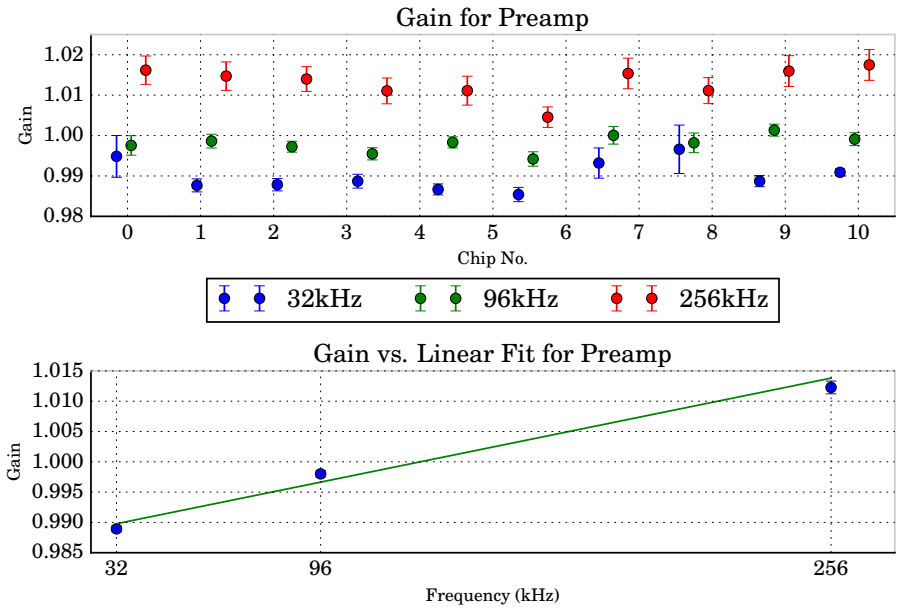


Figure 4.7: Effective gains of pre-amplifier at different sampling frequencies. The configured gain is 1.

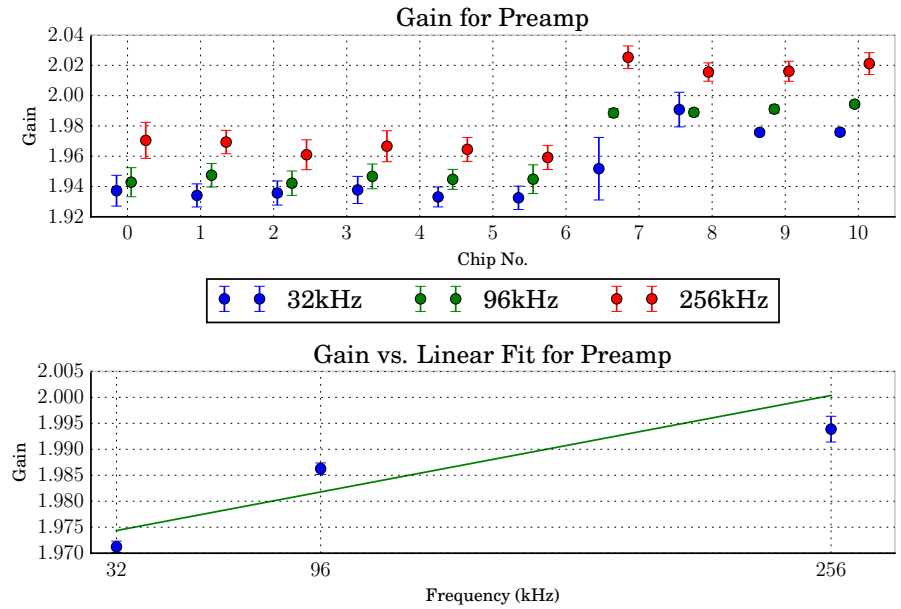


Figure 4.8: Effective gains of pre-amplifier at different sampling frequencies. The configured gain is 2.

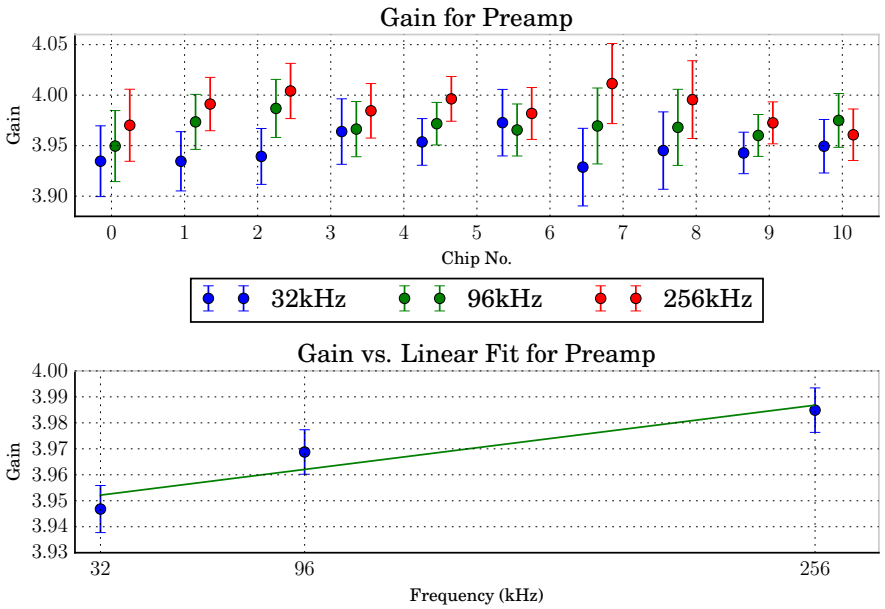


Figure 4.9: Effective gains of pre-amplifier at different sampling frequencies. The configured gain is 4.

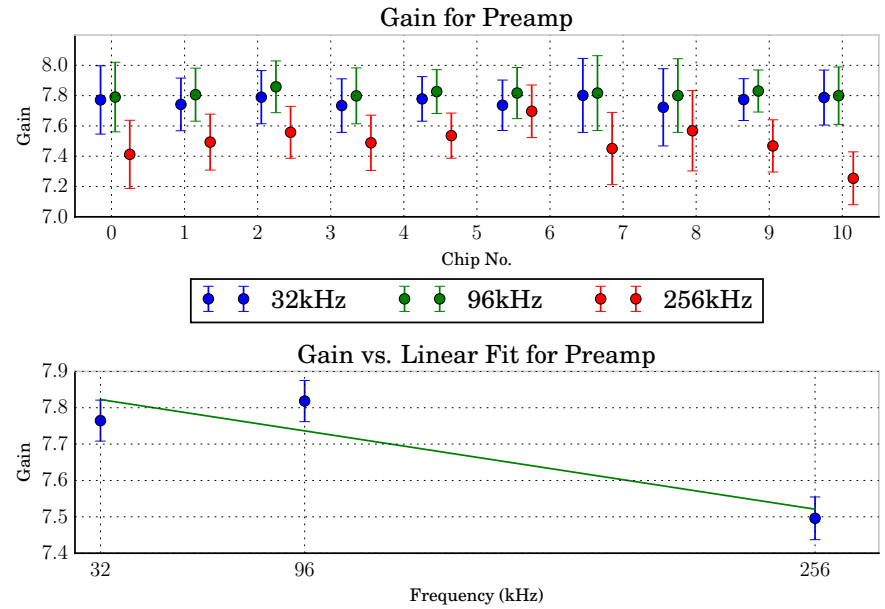


Figure 4.10: Effective gains of pre-amplifier at different sampling frequencies. The configured gain is 8.

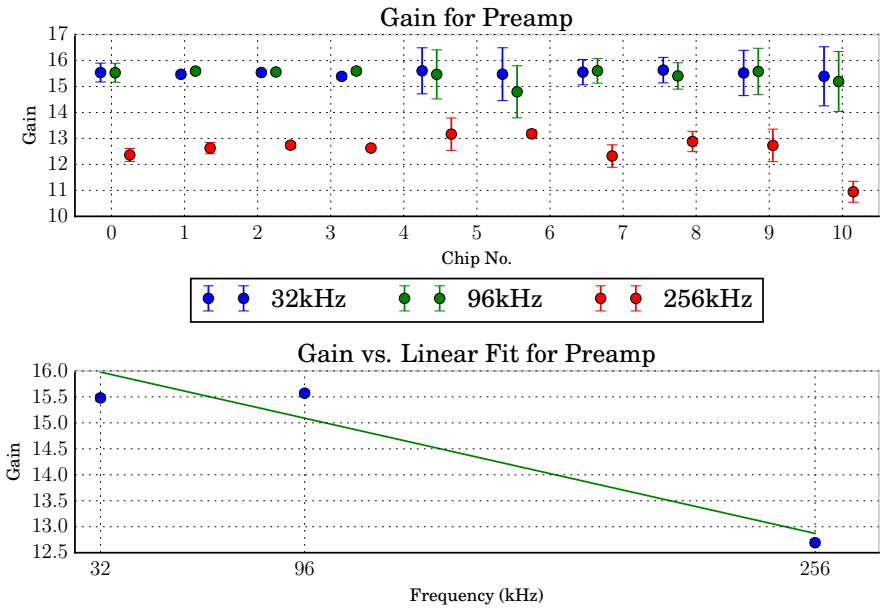


Figure 4.11: Effective gains of pre-amplifier at different sampling frequencies. The configured gain is 16.

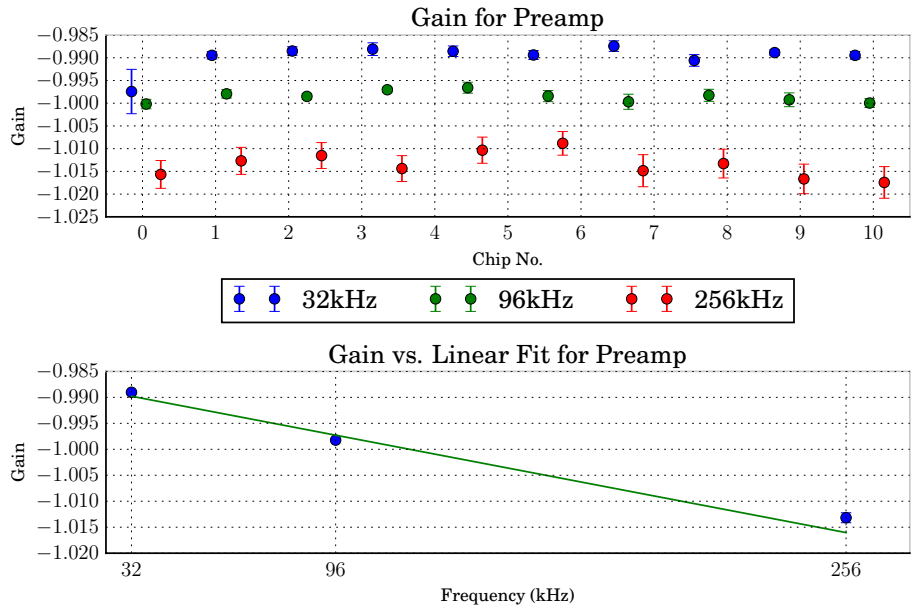


Figure 4.12: Effective gains of pre-amplifier at different sampling frequencies. The configured gain is -1.

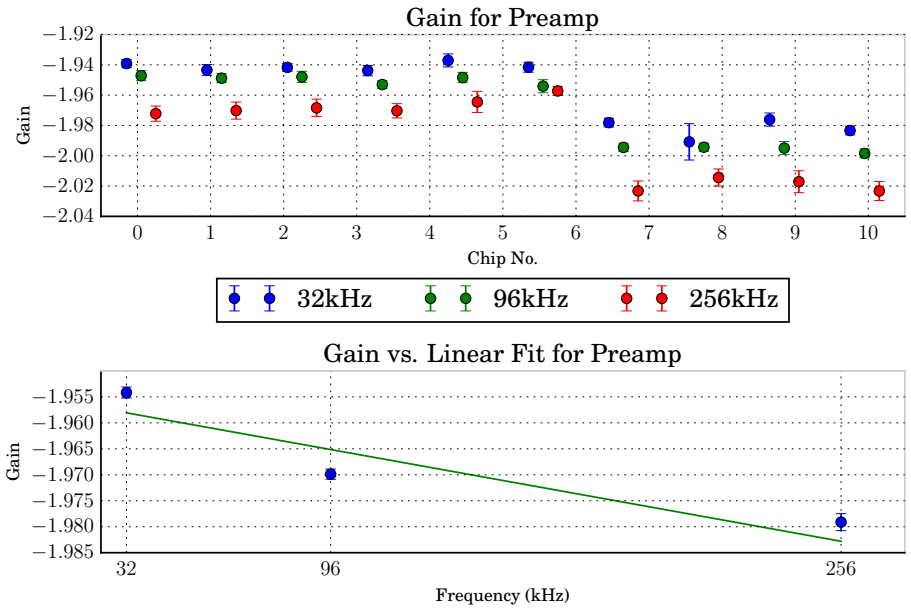


Figure 4.13: Effective gains of pre-amplifier at different sampling frequencies. The configured gain is -2.

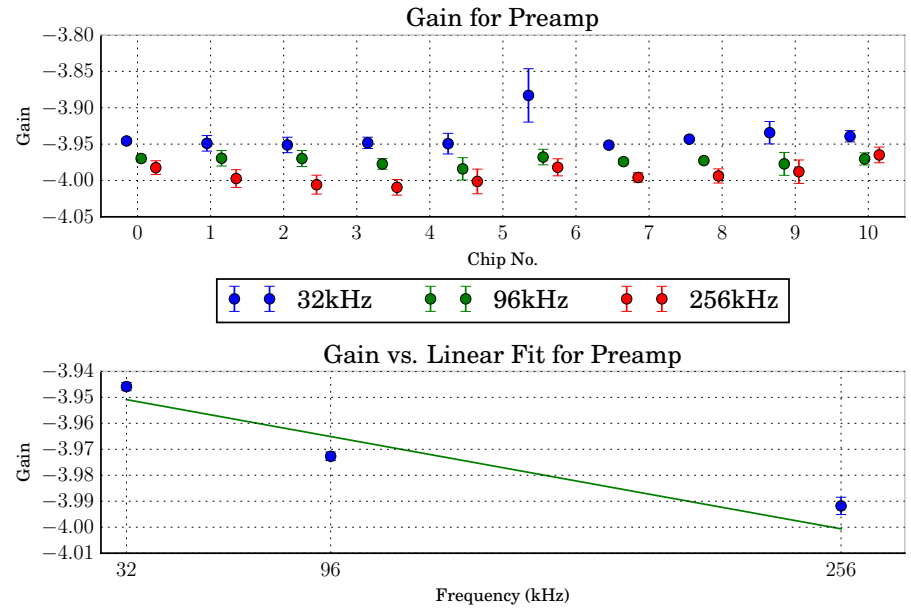


Figure 4.14: Effective gains of pre-amplifier at different sampling frequencies. The configured gain is -4.

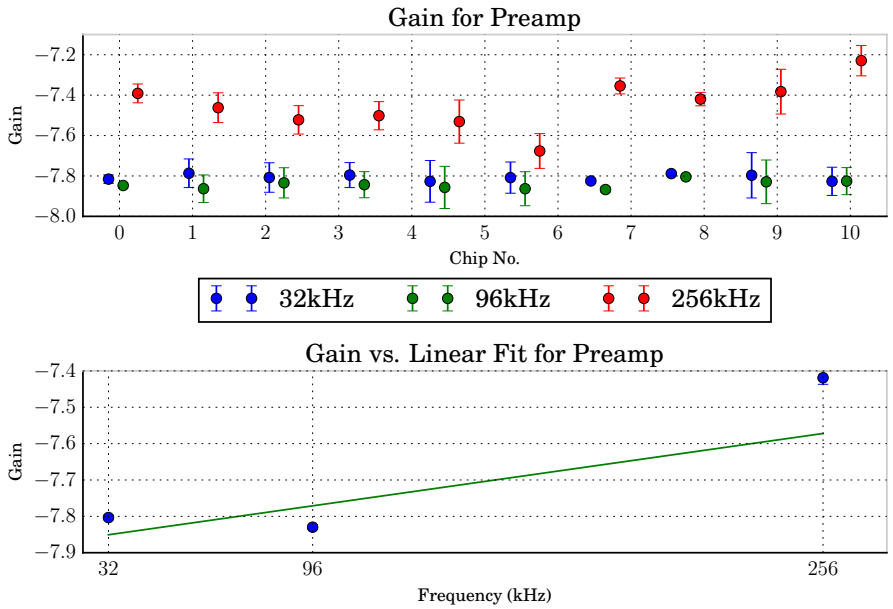


Figure 4.15: Effective gains of pre-amplifier at different sampling frequencies. The configured gain is -8.

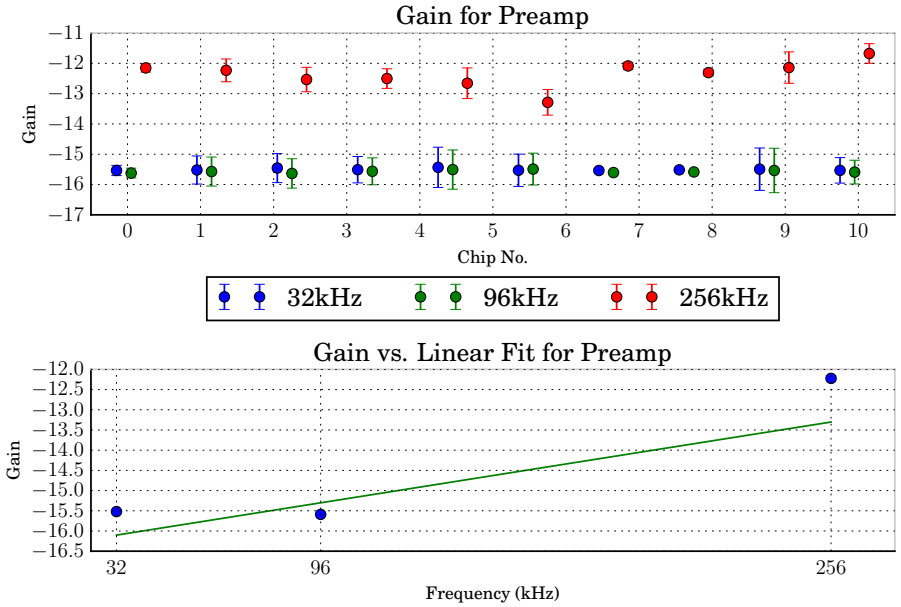


Figure 4.16: Effective gains of pre-amplifier at different sampling frequencies. The configured gain is -16.

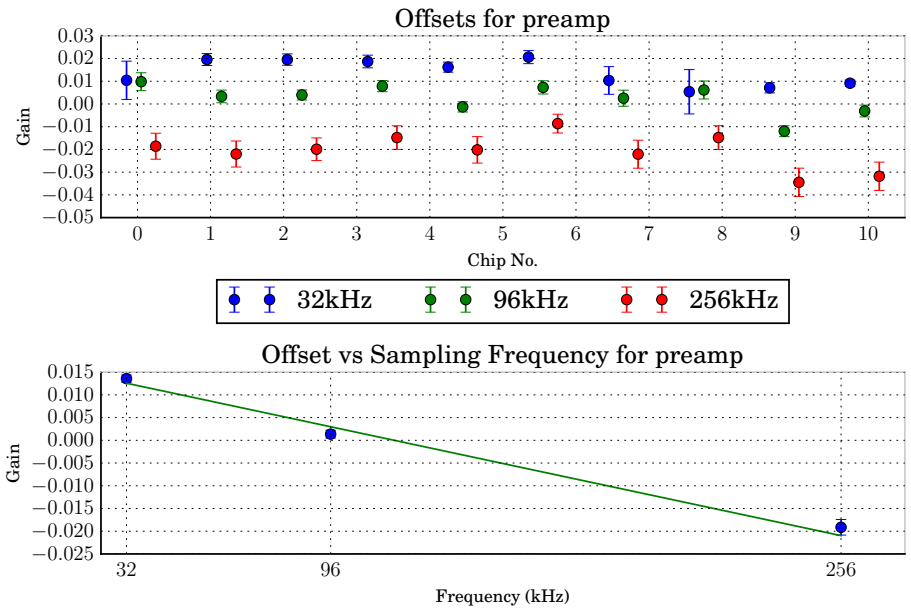


Figure 4.17: Effective offsets of preamp at different sampling frequencies. The configured gain is 1.

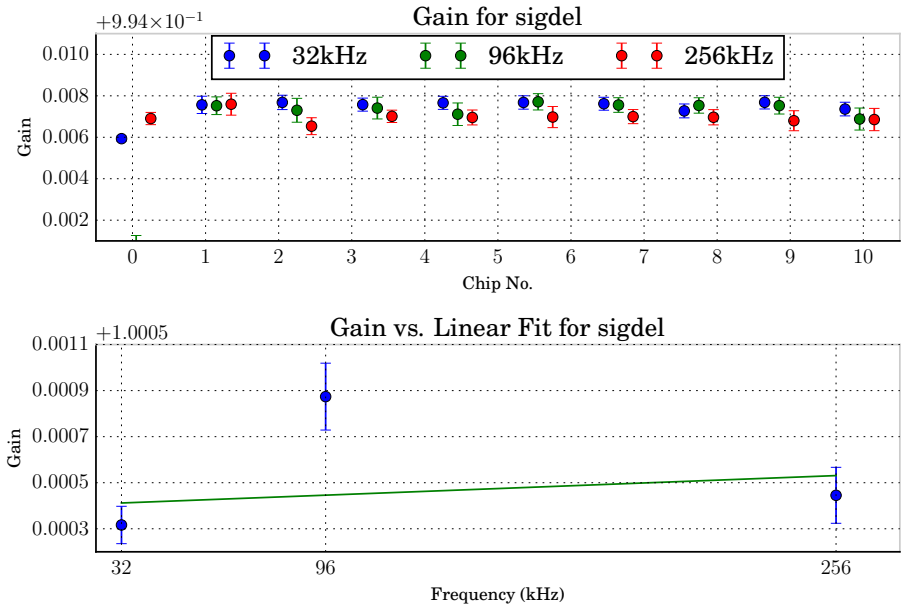


Figure 4.18: Effective gain/linearity of ΔΣ modulator at different sampling frequencies.

4.2 Sigma-Delta Converter: DC Measurements

The ΔΣ modulator was measured directly, isolated from the preamp, by applying a number of constant DC voltages to the `TEST OUT` pin and recording the resulting bit streams using the GPIO Raspberry Pi.

Following the same principles as with the pre-amplifier, for every chip and for every sampling frequency, 11 different input voltages were applied and the output bit stream was recorded. All of the data was then evaluated using a third order sinc-decimation filter to reconstruct a digital version of the measured voltage.

By fitting the linear function $y = mx + q$ to the 11 input/output voltage data points, we expect to obtain a line who's incline is equal to 1 and offset equal to 0. In figure 4.18 each incline of each chip is plotted for each sampling frequency fs. Similarly, the offset of each chip is plotted in figure 4.19.

The results show that there is no significant deviation from the expected values.

During processing of the bit stream data, some anomalies were noticed. Random spikes would occur here and there. See figure 4.20.

The only correlations that could be made were that the spikes occurred more often at higher sampling rates than they do at lower sampling rates. After analysing the time axis to make sure the data points were equidistantly spaced, random deviations were detected. This could possibly be a result of using the rather inaccurate system call `gettimeofday()` for generating time stamps on the Raspberry Pi.

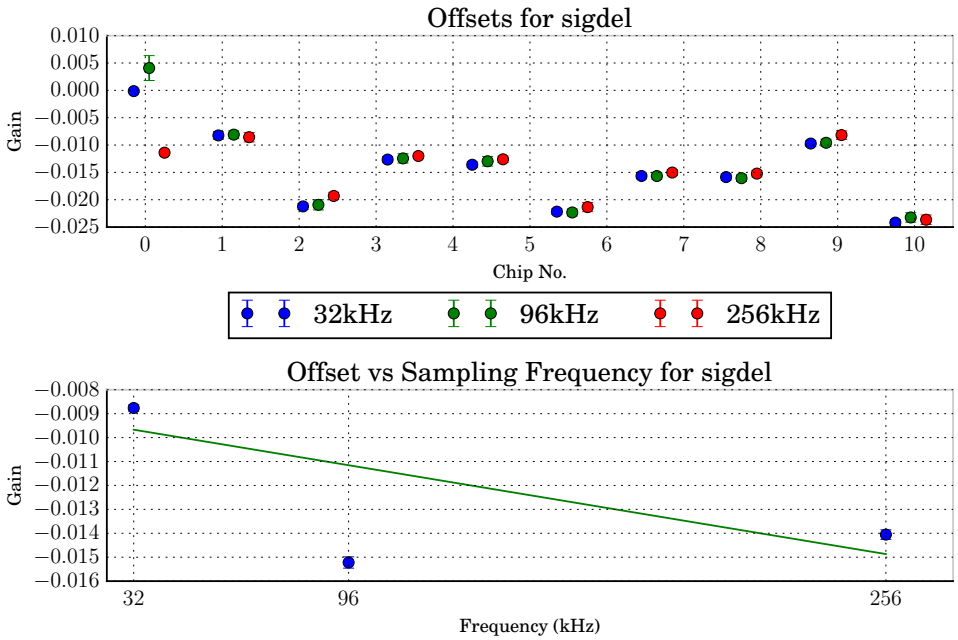


Figure 4.19: Effective offset of ΔΣ modulator at different sampling frequencies.

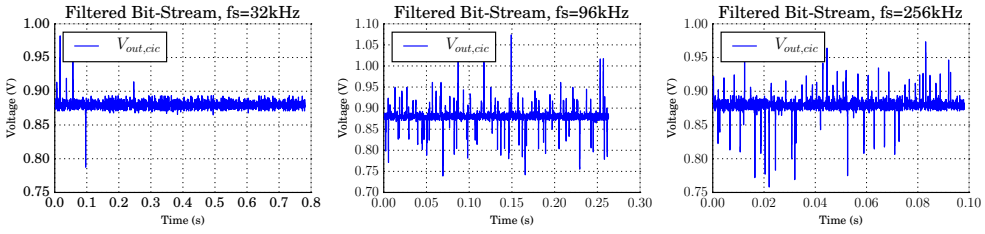


Figure 4.20: Large, random spikes after filtering the bit stream.

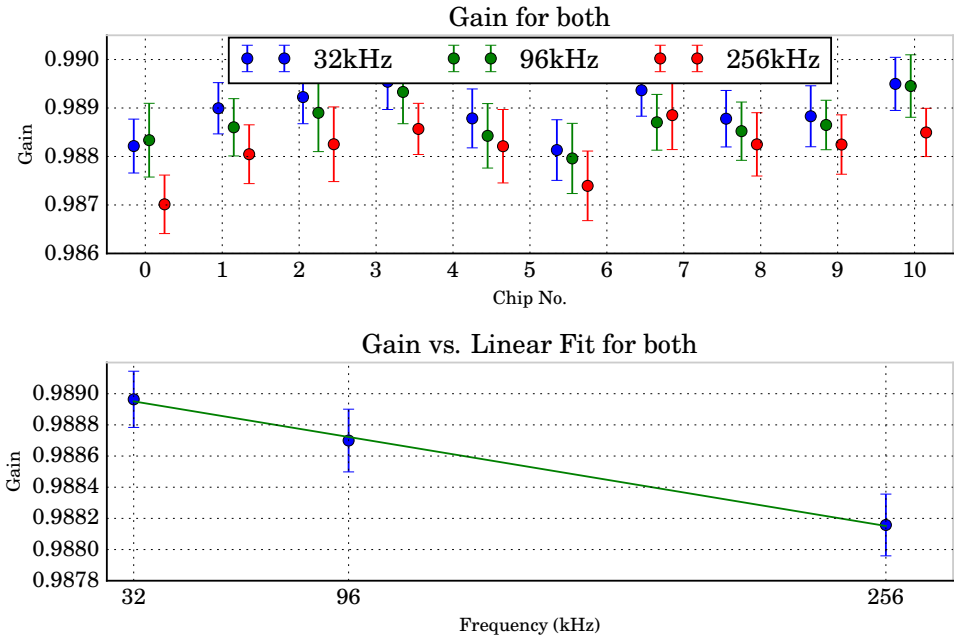


Figure 4.21: Effective gain of the whole system at different sampling frequencies. The configured gain is 1.

The deviations are large enough to suggest the possibility that bits went missing from the bit stream.

4.3 Complete System: DC Measurements

The same measurements were performed on the complete chip: That is, the pre-amplifier and the $\Delta\Sigma$ modulator were measured together as one.

As expected, the rather imperfect results from the pre-amplifier combined with the perfect results from the $\Delta\Sigma$ modulator result in a semi-imperfect system. Figures 4.21 and 4.22 show the various gains and offsets of the complete system.

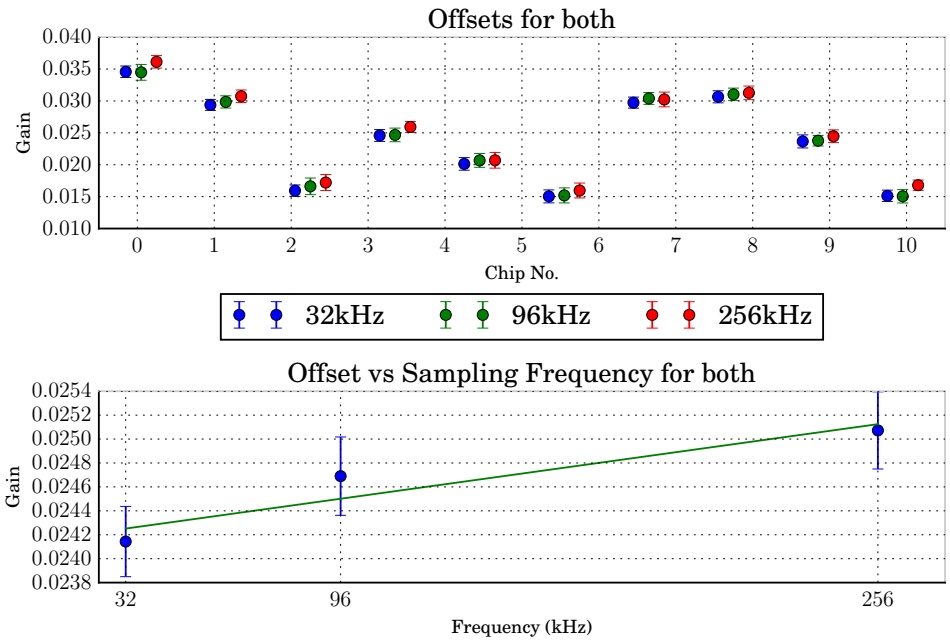


Figure 4.22: Effective offset of the whole system at different sampling frequencies. The configured gain is 1.

5

Conclusions and Outlook

This project's main objectives were to create and document a comprehensive test bench and the corresponding data processing tools, and to measure and assess the performance and characteristics of the $\Delta\Sigma$ modulator. Originally, the plan was to to measure new chip designs which build upon the first iteration. However, as we found out during the project, the new chips would not arrive in time to perform these measurements. Therefore, we fell back to the original chip design in our measurements.

Developing the test bench and the data processing tools turned out to be a noticeably more complex undertaking than originally anticipated, and progress was not as swift as we had hoped. Additionally, some time was lost due to an incorrect component having been soldered onto the test board, requiring time for finding and fixing the error. Lastly, due to our lack of familiarity with the subject, we occasionally got hung up on the wrong details, investing time without generating much usable output.

This resulted in some delays, which means not all objectives have been achieved at this point. Specifically, mapping the measurement results to the internal architecture of the chip via simulations still needs to be done. However, the test bench and data processing tools have been developed to a satisfying and very usable state, allowing for quick measurement of a multitude of chips and efficient analysis of the resulting data.

As has already been established in the previous work [2], the poor characteristics of the pre-amplifier are the main cause for diminished performance of the device. In this work, measurements have shown the pre-amplifier having a rather large offset and a gain that varies strongly depending on the operating frequency. Deviations in effective gain can be traced back to slow rise and fall times of the switched capacitor OTA; When the operating frequency is close to the limit (around 256 kHz), the target output voltage of the pre-amplifier can no longer be reached within one clock cycle and the converted value deviates strongly as a result.

There are multiple possible reasons for the slow rise and fall times. The transconductance of the OTA might be too low. Increasing it would lead to better performance, but at the cost of higher power consumption. Decreasing the values of the switched capacitors can lead to faster rise and fall times, but at the cost of higher noise and more susceptibility to parasitic capacitances. Considering the fact that the switched capacitors are in the 100 fF-range, and taking into account that the output of the pre-amplifier is directly connected to a lead on the chip (which is typically in the pF-range), the lead may very well have a strong negative influence on the pre-amplifier's performance. This hasn't been tested, but may be worth considering.

Without the aid of simulations, we can't draw more concrete conclusions at this point. On the bright side, though, the $\Delta\Sigma$ modulator itself appears to perform quite well.

Going into P6, we currently see the following tasks as critical:

- Measuring and evaluting the new chip designs, as originally intended for this project. Thanks to the test bench and data processing scripts developed during this project, this should be achievable fairly quickly.
- Performing simulations in order to gain a deeper understanding of how the circuit actually works, and gaining insight into *why* the measurement results look as they do.
- Based on this, isolate critical areas of the design and improve upon them.
- Evaluate whether or not the Raspberry Pi is actually sufficient for the task for which it is being used, and if not, what measures to take. This could mean either optimizing the software or, in the extreme case, replacing the hardware with something of higher performance.

6

References

[1] T. Burgherr, "Sensor Chip," July 2015.

[2] R. Gloor and P. Walther, "Re-Design eines integrierten Verstärkers für einen AD-Wandler-IC," January 2016.

[3] M. Baier and K. Niffenegger, "Sensor Chip mit Sub-SAR ADC," January 2016.

[4] (2002, March) Agilent 33120A 15 MHz Function / Arbitrary Waveform Generator – User's Guide. [Online]. Available: http://www.keysight.com/upload/cmc_upload/All/6C0633120A_USERSGUIDE_ENGLISH.pdf

[5] (2015, July) Keysight Truevolt Series Digitale Multimeter – Bedienungs- und Servicehandbuch. [Online]. Available: http://www.keysight.com/main/redirector.jspx?action=ref&cname=EDITORIAL&ckey=2345839&lc=ger&cc=CH&nfr=-536902435.1119130.00

[6] (2007, February) WAVE R UNNER 6000A SERIES OSCILLOSCOPES Operator's Manual. [Online]. Available: http://cdn.teledynelecroy.com/files/manuals/wr6a-om-e_rev_g.pdf

[7] (2005, July) WAVE R UNNER 6000A SERIES DSO Getting Started Manual. [Online]. Available: http://cdn.teledynelecroy.com/files/manuals/wr6a-gs-rev_b.pdf

[8] (2002, January) WaveRunner Remote Control Manual. [Online]. Available: http://cdn.teledynelecroy.com/files/manuals/wr2_rcm_revb.pdf

[9] R. P. org. (2016) NOOBS. [Online]. Available: https://www.raspberrypi.org/downloads/noobs/

[10] J. Hammons. (2016, October) Bash on Ubuntu on Windows. [Online]. Available: https://msdn.microsoft.com/en-us/commandline/wsl/about

[11] Wikipedia. (2016) iproute2. [Online]. Available: https://en.wikipedia.org/wiki/Iproute2

[12] ——. (2016) ifconfig. [Online]. Available: https://en.wikipedia.org/wiki/Ifconfig

[13] (2017, January) SSH keys. [Online]. Available: https://wiki.archlinux.org/index.php/SSH_keys#Generating_an_SSH_key_pair

[14] Hanspeter Schmid and Alex Huber, "Measuring a Small Number of Samples and the 3-Sigma Fallacy," *IEEE SOLID-STATE CIRCUITS MAGAZINE*, pp. 52–58, June 2014.

Appendices

A Scripts

This section contains the various scripts which were used for remote-controlling equipment and for processing data.

A.1 Waveform Generator Remote Control

```
#!/usr/bin/env python3

import serial
import struct
import sys
import getopt

# ------------------------------------------------------------------------ #
# DESCRIPTION                                                              #
# ------------------------------------------------------------------------ #
# Controls two  33120A arbitrary  function generators via  RS232 connection
# and USB  adapter. One of the generators  is used to output  a square wave
# (to be used as the clock) between  0V and 3V, the other generator is used
# to output a DC signal.

# ------------------------------------------------------------------------ #
# USAGE                                                                    #
# ------------------------------------------------------------------------ #
# ./33120A.py --clock=<frequency>
# ./33120A.py -c <frequency>
# Set the CLK generator's frequency. <frequency> is a value in Hertz.
# Example: Set square wave frequency to 96 kHz
# ./33120A.py -c 96e3

# ./33120A.py --voltage=<voltage>
# ./33120A.py -v <voltage>
# Set the other generator to output a DC voltage of <voltage>. <voltage> is
```

```
# a value in Volts.
# Example: Set a DC voltage of 0.9V:
# ./33120A.py -v 0.9

# ------------------------------------------------------------------------ #
# SETTINGS                                                                 #
# ------------------------------------------------------------------------ #
CLK_DEVICE = '/dev/ttyUSB1'
DC_DEVICE  = '/dev/ttyUSB0'

# ------------------------------------------------------------------------ #
# IMPLEMENTATION                                                           #
# ------------------------------------------------------------------------ #
class FunctionGenerator(object):
    def __init__(self, device):
        self.__gen = serial.Serial(
            port=device,
            baudrate=9600,
            timeout=1,
            parity=serial.PARITY_NONE,
            stopbits=serial.STOPBITS_TWO,
            bytesize=serial.EIGHTBITS
        )
        self.__gen.write(b'OUTP:LOAD INF\n')

    def set_dc(self, voltage):
        self.__gen.write(
            bytes('APPL:DC DEF, DEF, {} V\n'.format(voltage), 'ascii'))

    def set_clk(self, freq):
        self.__gen.write(
            bytes('APPL:SQU {} HZ, 3 VPP, 1.5 V\n'.format(freq), 'ascii'))

    def set_sin(self, frequency, vpp, offset):
        self.__gen.write(
            bytes('APPL:SIN {} HZ, {} VPP, {} V\n'.format(
                frequency, vpp, offset), 'ascii'))


def main(argv):
    try:
        opts,args=getopt.getopt(argv,"hc:v:",["help","clock=","voltage="])
    except getopt.GetoptError:
        print('33120A.py -c <clock frequency> -v <input voltage>')
        sys.exit(2)
    for opt, arg in opts:
        if opt in ("-h", "--help"):
            print('33120A.py -c <clock frequency> -v <input voltage>')
            sys.exit(0)
        elif opt in ("-c", "--clock"):
```

```
            fgClk = FunctionGenerator(CLK_DEVICE)
            fgClk.set_clk(arg)
        elif opt in ("-v", "--voltage"):
            fgVin = FunctionGenerator(DC_DEVICE)
            fgVin.set_dc(float(arg))

if __name__ == '__main__':
    main(sys.argv[1:])
```

A.2 Initialize WaveRunner Oscilloscope

```
#!/usr/bin/env python3

import vxi11

# ------------------------------------------------------------------------ #
# DESCRIPTION                                                              #
# ------------------------------------------------------------------------ #
# Initializes the oscilloscope to relatively sane settings. These will
# usually still not be optimal for our measurements.

# ------------------------------------------------------------------------ #
# USAGE                                                                    #
# ------------------------------------------------------------------------ #
# ./initWaveRunner.py

# ------------------------------------------------------------------------ #
# SETTINGS                                                                 #
# ------------------------------------------------------------------------ #
INSTR_IP='169.254.14.189'

# ------------------------------------------------------------------------ #
# IMPLEMENTATION                                                           #
# ------------------------------------------------------------------------ #
instr=vxi11.Instrument(INSTR_IP)
print(instr.ask('*IDN?'))
instr.write('*RST')
instr.write('TRIG_MODE NORM')
instr.write('TRIG_SELECT EDGE')
instr.write('C1:TRIG_SLOPE POS')
instr.write('C1:TRIG_LEVEL 1V')
instr.write('TIME_DIV 5US')
instr.write('C1:TRACE OFF')
instr.write('C2:TRACE OFF')
instr.write('C3:TRACE ON')
instr.write('C1:COUPLING D1M')
instr.write('C2:COUPLING D1M')
instr.write('C3:COUPLING D1M')
instr.write('C4:COUPLING D1M')
```

A.3 Configure WaveRunner Oscilloscope

```
#!/usr/bin/env python3

import vxi11
import sys
import getopt

# ------------------------------------------------------------------------ #
# DESCRIPTION                                                              #
# ------------------------------------------------------------------------ #
# Sets voltage/div, time/div and offset for a given oscilloscope trace.
# Needed for generating sane display settings so that we can acquire
# optimal measurement data.

# ------------------------------------------------------------------------ #
# USAGE                                                                    #
# ------------------------------------------------------------------------ #
# Set voltage/div
# ./configWaveRunner.py --vdiv=<value>
# ./configWaveRunner.py -v <value>
# where <value> is a voltage in millivolts
#
# Set time/div
# ./configWaveRunner.py --tdiv=<value>
# ./configWaveRunner.py -t <value>
# where <value> is a time in microseconds
#
# Set offset
# ./configWaveRunner.py --offset=<value>
# ./configWaveRunner.py -o <value>
# where <value> is an offset voltage in millivolts

# ------------------------------------------------------------------------ #
# SETTINGS                                                                 #
# ------------------------------------------------------------------------ #
# The intrument's IP address can be checked on the oscilloscope itself.
# Make sure you have a LAN connection to it.
INSTR_IP='169.254.14.189'
CHANNEL='C3'

# ------------------------------------------------------------------------ #
# IMPLEMENTATION                                                           #
# ------------------------------------------------------------------------ #
class WaveRunner(object):
    def __init__(self, instr_IP, channel):
        self.__gen = vxi11.Instrument(instr_IP)
        #print(self.__gen.ask('*IDN?')) # For diagnostics
        self.__gen.channel = channel
        self.__gen.write(self.__gen.channel + ':TRACE ON')
```

```
    def set_time_div(self,microseconds):
        self.__gen.write(
            self.__gen.channel + ':TIME_DIV ' + microseconds + 'US')

    def set_volt_div(self,millivolts):
        self.__gen.write(
            self.__gen.channel + ':VOLT_DIV ' + millivolts + 'MV')

    def set_offset(self,millivolts):
        self.__gen.write(
            self.__gen.channel + ':OFFSET ' + millivolts + 'MV')

def main(argv):
    instr = WaveRunner(INSTR_IP,CHANNEL)
    try:
        opts,args=getopt.getopt(argv,"t:v:o",["tdiv=","vdiv=","offset="])
    except getopt.GetoptError:
        sys.exit(2)
    for opt, arg in opts:
        if opt in ("-t", "--tdiv"):
            instr.set_time_div(arg)
        elif opt in ("-v", "--vdiv"):
            instr.set_volt_div(arg)
        elif opt in ("-o", "--offset"):
            instr.set_offset(arg)

if __name__ == '__main__':
    main(sys.argv[1:])
```

A.4 Aqcuire Measurement Data Through WaveRunner Oscilloscope

```
#!/usr/bin/env python3

import vxi11
import sys
import getopt

# ------------------------------------------------------------------------ #
# DESCRIPTION                                                              #
# ------------------------------------------------------------------------ #
# Acquires, stores and downloads measurement  data from the oscilloscope to
# the controlling computer. Removes files on oscilloscope after download.

# ------------------------------------------------------------------------ #
# USAGE                                                                    #
# ------------------------------------------------------------------------ #
# ./acquireWaveRunnerData.py \
```

```
#     -channel=<channel> \
#     --remotefile=<remotefile> \
#     --localfile=<localfile>
# Where:
#
# <remotefile>: the filename which is used by the oscilloscope for
# storing a waveform on its HDD. This will usually be of form
# <ch>Trace<number>.txt, for example C1Trace00001.txt for the first
# waveform for the first channel.
#
# NOTE: This cannot be configured remotely, the filename is merely used to
# tell the script which remote file to download from the oscilloscope. If
# the oscilloscope's filename iterator and the script's <remotefile>
# parameter are not in sync, the download will fail. Resetting the
# oscilloscope's counter requires manual intervention on the oscilloscope
# via the "File->Save Waveform" dialog.
#
# <localfile>: The filename to be used for storing the waveform
# on the computer onto which it is downloaded. Example:
# chip01-gain+01-256kHz-1.9V.txt
#
# <channel>: The channel for which a waveform is to be stored.
#
# EXAMPLES:
# ./acquireWaveRunnerData.py \
#     --channel=C2 \
#     --remotefile=C2Trace00000.txt \
#     --localfile=trace1.txt
# or in short form:
# ./acquireWaveRunnerData.py -c C2 -r C2Trace00000.txt -l trace1.txt

# ------------------------------------------------------------------------ #
# SETTINGS                                                                 #
# ------------------------------------------------------------------------ #
instrIP='169.254.14.189'
# CHANNEL='C3'
# Data directory on the oscilloscope.
# NOTE: This must also be configured via the "File->Save Waveform" dialog
# on the oscilloscope itself; merely setting it remotely will not be
# sufficient.
DATA_DIR='D:\\traces'

# ------------------------------------------------------------------------ #
# IMPLEMENTATION                                                           #
# ------------------------------------------------------------------------ #
class waverunner(object):
    def __init__(self, instrIP):
        self.__gen = vxi11.Instrument(instrIP)
        #print(self.__gen.ask('*IDN?'))
```

```
    def store_data(self,channel):
        self.__gen.write('STO ' + channel + ',HDD')

    def transfer_file(self,remotefile,localfile):
        trace_file_path = DATA_DIR + '\\' + remotefile
        data=self.__gen.ask('TRFL? DISK,HDD,FILE,"' + trace_file_path + '"')
        file=open(localfile,'w')
        # NOTE: data is a  Windows text string; newlines are
        # represented by  \n\r.  Its  last line is  a string
        # 'ffffffff' (see LeCroy documentation). This string
        # is  unneeded and  interferes  with processing  the
        # data.   We therefore  cut  off  the string's  last
        # line: the  8 'f'  characters as  well as  the \n\r
        # part.
        file.write(data[:-10])
        file.close()

    def cleanup(self,remotefile):
        trace_file_path = DATA_DIR + '\\' + remotefile
        self.__gen.write('DELF DISK,HDD,FILE,' + trace_file_path)

def main(argv):
    instr = waverunner(instrIP)
    try:
        opts, args = getopt.getopt(
            argv,"c:r:l",["channel=","remotefile=","localfile="])
    except getopt.GetoptError:
        sys.exit(2)

    remotefile = ''
    localfile  = ''
    channel    = ''
    for opt, arg in opts:
        if opt in ("-r", "--remotefile"):
            remotefile=arg
        elif opt in ("-l", "--localfile"):
            localfile=arg
        elif opt in ("-c", "--channel"):
            channel=arg

    instr.store_data(channel)
    instr.transfer_file(remotefile,localfile)
    instr.cleanup(remotefile)


if __name__ == '__main__':
    main(sys.argv[1:])
```

A.5 Acquire and Store Bit Stream on Rasbperry Pi

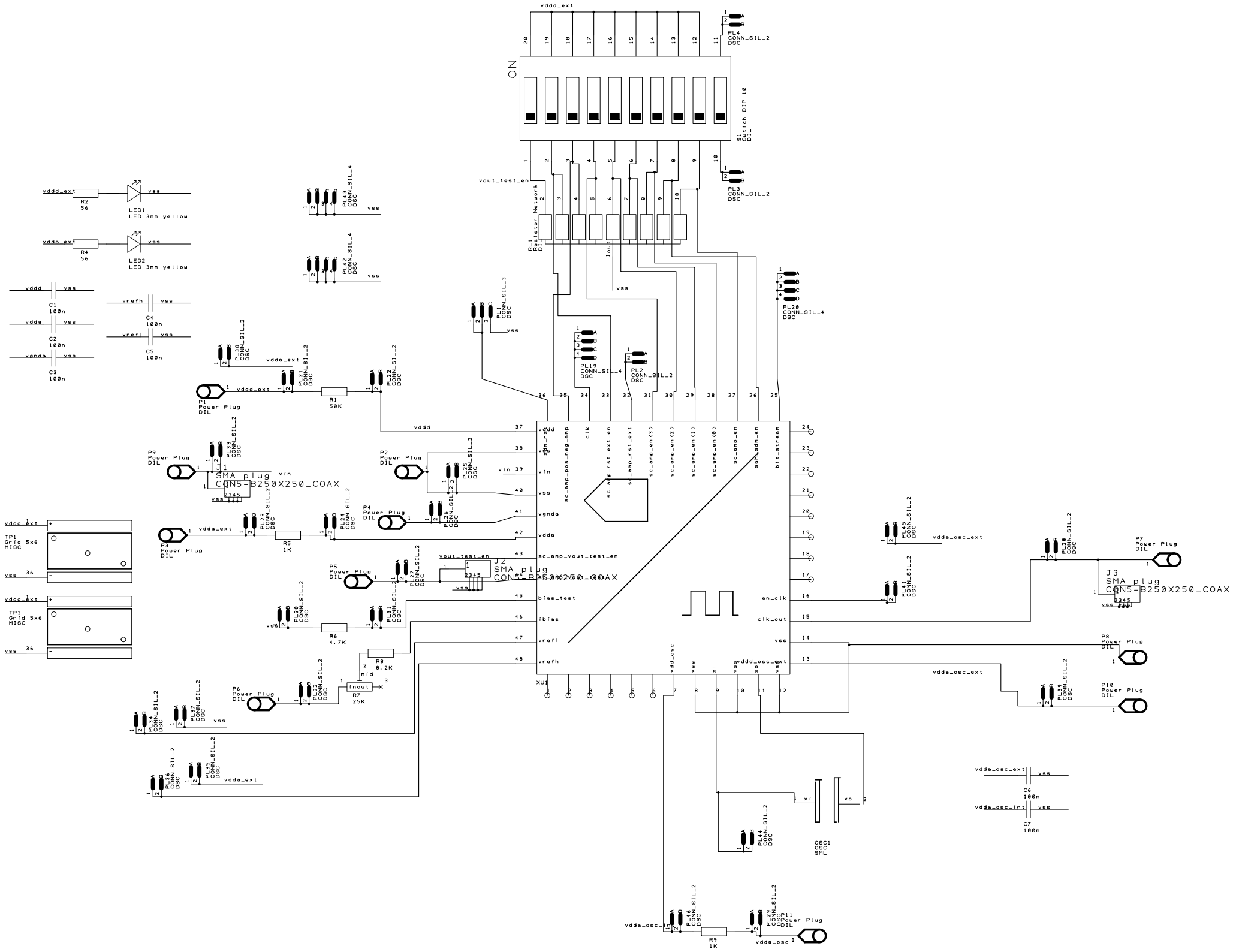
```
#include <iostream>
#include <wiringPi.h>
#include <sys/time.h>
#include <fstream>

enum BufferDataSlots
{
        BUF_DATA,
        BUF_TIME,

        BUF_NUM_SLOTS
};

#define NUM_MEASUREMENTS 25000
#define PIN_CLOCK        17
#define PIN_DATA         18

void initialise_gpio()
{
        wiringPiSetupGpio();

        pinMode(PIN_CLOCK, INPUT);
        pinMode(PIN_DATA, INPUT);
}

void acquire_data(int bitstream_buf[BUF_NUM_SLOTS][NUM_MEASUREMENTS])
{
        timeval end, start;

        gettimeofday(&start, 0);
        for(int i = 0; i != NUM_MEASUREMENTS; ++i)
        {
                // wait for clock to go positive
                while(digitalRead(PIN_CLOCK) == 0) {}

                gettimeofday(&end, 0);
                bitstream_buf[BUF_DATA][i] = digitalRead(PIN_DATA);
                bitstream_buf[BUF_TIME][i]
                        = ((end.tv_sec * 1000000) + end.tv_usec)
                        - ((start.tv_sec * 1000000) + start.tv_usec);

                // wait for clock to go negative
                while(digitalRead(PIN_CLOCK) == 1) {}
        }
}

void save_data(const int bitstream_buf[BUF_NUM_SLOTS][NUM_MEASUREMENTS])
{
```

```
        std::ofstream data_file("bit_stream.txt");
        std::ofstream clock_file("vin_clk.txt");
        for(int i = 0; i != NUM_MEASUREMENTS; ++i)
        {
                long double time_usec = bitstream_buf[BUF_TIME][i] / 1000000.0;
                clock_file <<time_usec<< "     " << "1.2" << "     " <<std::endl;
                if(bitstream_buf[BUF_DATA][i] == 1)
                        data_file << time_usec << "     " << "3.0000" << std::endl;
                else
                        data_file << time_usec << "     " << "1.0e-9" << std::endl;
        }
}

int main()
{
        initialise_gpio();

        /*
         * The IO operations may be too slow, so we store the measurements
         * to a temporary buffer and save the buffer later.
         */
        int bitstream_buf[BUF_NUM_SLOTS][NUM_MEASUREMENTS];
        acquire_data(bitstream_buf);
        save_data(bitstream_buf);

        std::cout << "done" << std::endl;
        return 0;
}
```

B

Test Board Schematic

The following schematic was created by Mr. Burgherr. It is included in this document for convenience so that the reader does not have to go looking for it in the Subversion repository.


vddd_ext
vdda_ext
vss
LED1
LED 3mm yellow
LED2
LED 3mm yellow
R2
56
R4
56
C1
100n
C2
100n
C3
100n
C4
100n
C5
100n
vddd
vdda
vgnda
vrefh
vrefl
ON
S1
Switch DIP 10
RL1
Resistor Network
vout_test_en
iout
PL1 CONN_SIL_2 DSC
PL2 CONN_SIL_2 DSC
PL3 CONN_SIL_2 DSC
PL4 CONN_SIL_2 DSC
PL19 CONN_SIL_4 DSC
PL20 CONN_SIL_4 DSC
PL42 CONN_SIL_4 DSC
PL43 CONN_SIL_4 DSC
PL44 CONN_SIL_3 DSC
P1 Power Plug DIL
P2 Power Plug DIL
P3 Power Plug DIL
P4 Power Plug DIL
P5 Power Plug DIL
P6 Power Plug DIL
P7 Power Plug DIL
P8 Power Plug DIL
P9 Power Plug DIL
P10 Power Plug DIL
P11 Power Plug DIL
R1 50K
R5 1K
R6 4.7K
R7 25K
R8 8.2K
R9 1K
J1 SMA plug CON5-B250X250_COAX
J2 SMA plug CON5-B250X250_COAX
J3 SMA plug CON5-B250X250_COAX
TP1 Grid 5x6 MISC
TP3 Grid 5x6 MISC
XU1
vin
vss
vgnda
vdda
sc_amp_vout_test_en
bias_test
ibias
vrefl
vrefh
sc_amp_pos_neg_amp
clk
sc_amp_rst_ext_en
sc_amp_rst_ext
sc_amp_en<3>
sc_amp_en<2>
sc_amp_en<1>
sc_amp_en<0>
sc_amp_en
sampsdn_en
bit_stream
en_clk
clk_out
vdd_osc
vss
xi
vddd_osc_ext
xo
vdda_osc_ext
C6
100n
C7
100n
OSC1
OSC
SML