Technische Universität Ilmenau

Department of Electrical Engineering and Information Technology

Bachelor Thesis

Evaluation of Open Source Hardware for Rapid Prototyping of Advanced Ultrasonic Testing Methods

---

Submitted by:	Tim Treichel
Field of study:	Computer Science And Systems Engineering
Area of specialization:	Telekommunikationstechnik

Institute for Information Technology

Electronic Measurements and Signal Processing

Professor:	Univ.-Prof. Dr.-Ing. Del Galdo, Giovanni
Advisor:	Dr.-Ing. Römer, Florian

Acknowledgments

First and foremost, I would like to thank my thesis advisors Univ.-Prof. Giovanni Del Galdo and Dr.-Ing. Florian Fömer. Their assistance, guidance and understanding were crucial during the creation of this thesis. I would also like to thank them for providing me with all the necessary facilities and hardware.

Furthermore, I have to thank the creator of the un0rick project, Luc Jonveaux, for creating the basis of my thesis and for taking the time to answer many questions that I had about the un0rick board.

I am also grateful to my fellow students for helping me to settle in at the laboratory and for giving me useful literature recommendations.

Most importantly I would like to thank my family. I am especially thankful to my Mother, Father, Brother and my Grandmothers, which offered me incredible support without which, this thesis probably never would have been completed.

Abstract

The Un0rick project is an open source hardware board for ultrasonic testing. In comparison to common proprietary hardware it is a very low cost solution, which enables complete access to the raw acquisition data and customization options down to the firmware level. The aim of this thesis is to evaluate the hardware in terms of the capability for the use of advanced ultrasonic testing methods, such as coded signals. Furthermore, it also aims to function as additional documentation for the Un0rick project, since the projects documentation is scattered in multiple places and sometimes inconsistent or non existing.

In the first part of the thesis some necessary fundamentals of ultrasonic testing are covered. After this, the working principles of the Un0rick board are explained, with focus on signal generation, measurement and processing. This includes a view into the mechanisms of the default firmware. In the context of this thesis, custom software was created. The own implemented software includes a python library, which enables communication with the Un0rick board, as well as a web-server, which can display live updating acquisitions in a web-browser. In the last part of this thesis custom firmware and software features, which implements binary coded signals, were developed.

Finally, this thesis concludes that the Un0rick hardware is a well suited platform for low cost development (or for education purposes) of ultrasonic testing systems. Although it takes some effort to implement features for custom applications, it is possible to do so, due to the fully open documentation of the project. Advanced ultrasonic testing methods such as coded signals are possible by implementing features on the firmware and software side. Implementing other methods, e.g. ultrasonic imaging, may also need additional hardware.

Contents

List of Tables

List of Figures

1 Introduction

Ultrasonic waves appear frequently in nature. Whales and dolphins use ultrasound for underwater object detection and communication. Similar to whales and dolphins, bats are able to navigate on land and in the air by emitting ultrasonic waves. Other animals, for example moths, cats and dogs are also able to hear parts of the ultrasonic spectrum. Ultrasound also occurs in nature without the influence of living creatures, for example at waterfalls or when rain falls on lakes.

Even though humans can not hear ultrasonic waves, they have many applications in modern medicine, technology, science, food industry, navigation, oceanography, seismology, etc. A widely known application are fetal images, but there are other areas of medicine where ultrasonic imaging is used. The useful ability of making diagnostics on the inside of bodies can also be applied to objects. Therefore, ultrasound is widely used in nondestructive testing. In science ultrasonic waves are used to understand properties of solids and liquids. Navigating ships on and below the water, while detecting icebergs, rocks and other ships, was one of the first applications that lead to the development of ultrasound technology and it is still used to this day[Che02, DPV97].

2 Ultrasonic Testing

2.1 Fundamentals of Ultrasonic Testing

2.1.1 Ultrasound

In physics, sound describes the propagation of mechanical vibrations in matter. Sound can traverse solids, liquids and gas, however it can not exist in empty space because the particles of the materials themselves are vibrating. Sound is not a transport of mater. The particles are vibrating around their neutral positions and transfer their vibration to nearby particles. The sound wave moves with a constant velocity through the medium. The velocity of the propagating wave is specific to the material it traverses and it is called the *velocity of sound* c. The states of the vibrating particles are reappearing in a constant distance throughout the sound wave. This distance is called *wavelength* λ. The number of wave oscillations in one second is called *frequency* f. Those three properties, c, λ and f are in relation to one another (Equation (2.1)) [DPV97, KK90].

$$c = \frac{\lambda}{f} \tag{2.1}$$

Humans can hear sound in a varying ranges of frequencies from 20 Hz to 20 kHz. This range is slightly different between individuals and diminishes with older age. The frequencies of sound waves characterize the pitch we hear. Sound below 20 Hz is called *infrasound (infrasonic)*, while sound waves above 20 kHz are called *ultrasound (ultrasonic)* and sound waves with even higher frequencies than 1 GHz are called *hypersound (hypersonic)*.

2.1.2 Creation of Ultrasonic Waves

Sound waves in air are very common. They are created by human speech, animal noises, etc. To create specific acoustic signals, for example in speakers, thin vibrating membranes out of paper or plastic are used. The vibrations are caused by electromagnetic or electrostatic forces. If build small enough, they can create sound waves with

frequencies up to 1 MHz. However, this method can not create ultrasonic waves in liquids or solids, because the density of liquid or solid materials is higher, therefore only sound waves with low frequencies can be created. To create ultrasonic waves in solid or liquid materials usually a vibrating solid object is used. The vibration is created by the utilization of specific physical effects[DPV97].

Commonly used is the piezoelectric effect, which appears in some materials. Objects with piezoelectric properties can change their physical shape when an electric potential is applied to them. The opposite is also true, i.e. piezoelectric materials create an electric potential when their shape is altered, for example by mechanical pressure (for further details refer to Section 2.1.3). This effect makes it possible to cause vibrations in the material by applying specific voltage signals to it.

Another way to create ultrasonic waves in metals or highly conductive materials is by way of electrodynamic methods. In those methods a high frequency current is induced inside the object. At the same time a static magnetic field causes the Lorentz force, which is applied to the conductive particles inside the metal, rearranging them perpendicular to the flow of current. This causes an ordered, high frequency oscillation of particles, which propagates inside the material. Transducers that operate on those principles are called *Electromagnetic Acoustic Transducers - EMATs*. They can create ultrasonic waves inside materials without touching them, which for example makes them suitable to operate on high temperature metals[DPV97, KK90].

Another way to create ultrasound in metals, without touching them is by utilizing lasers. If a light impulse, caused by a laser, hits the surface of a metal, some of the energy is converted into heat. This heats up the metal in a small surface area, which causes small volume expansion of this area. If the light impulses are short enough, it is possible to create high frequency vibrations inside the metal, which will propagate parallel to the surface area[DPV97, KK90].

2.1.3 Piezoelectric Effect

As mentioned before, the piezoelectric effect is a physical phenomenon that is widely used to create and receive ultrasonic wave signals. It was discovered, that some materials create electric charges on their surface when they are exposed to external mechanical pressure, which causes deformation. This is called the direct piezoelectric effect and materials, which possess those properties are called piezoelectric materials. This phenomenon is also reversible, which means that piezoelectric objects change their form when an electrical potential is applied to them. This is called the inverse piezoelectric

effect. The second effect can be used to create ultrasonic waves, by making the piezo oscillate. The first effect can be used to measure ultrasonic waves. Therefore, a piezoelectric material can be used as an ultrasonic transceiver and receiver.[DPV97, KK90]

2.1.3.1 Cause of the Piezoelectric Effect

The cause of the piezoelectric effect lies in the ionic crystalline structure. Piezoelectric materials have an asymmetry in their *elementary cell*, which creates electrical potentials when the molecules are shifted. The *elementary cell (or unit cell)* of a crystal is a small parallelepiped structure of particles, which builds up the whole crystal. Depending in which direction pressure is applied or released to the material and therefore to its elementary unit, different (in direction and strength) electrical potentials are or are not formed [DPV97, KK90, San93].

As a simplified example, the piezoelectric effect is visualized for quartz, which is one of the first piezoelectric materials used for technical applications, is shown in Figure 2.1. In Figure 2.1 an elementary cell of a quartz crystal is placed between two electrodes, which measure the electrical potential at the surface. The chemical formula of quartz is SiO_2. It consist of Si^{4+} and O^{2-} ions. The elementary cell is build from 3 silicon ions and 6 oxygen ions as shown in Figure 2.1 (for simplification two O^{2-} ions are illustrated as one unit).

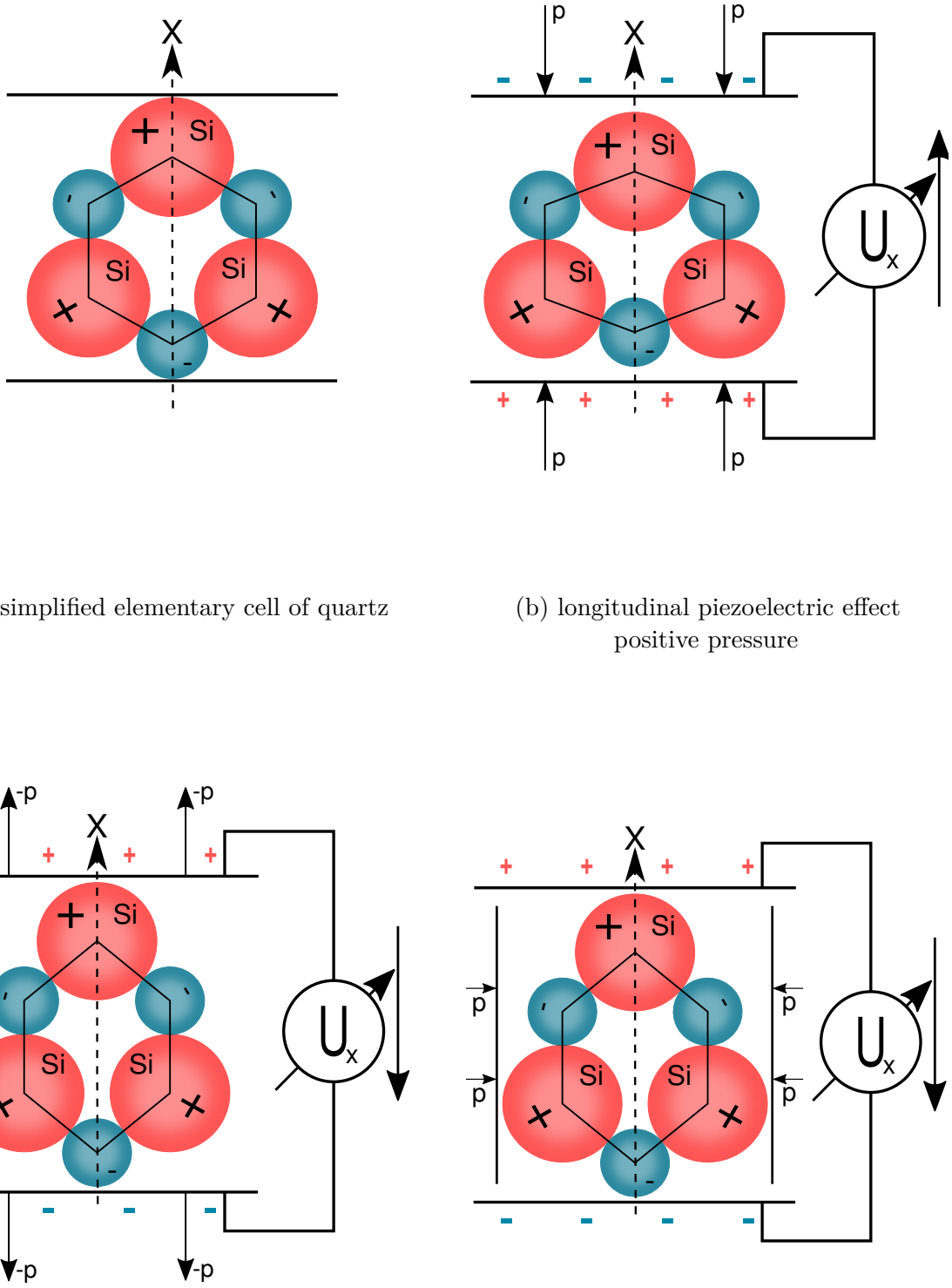
In Figure 2.1 (a) no pressure is applied to the crystal. Its elementary cell is in a balanced formation and no electric potentials occur at the crystals surface.

In Figure 2.1 (b) positive pressure is applied to the crystal along the x-axis. This causes slight shifting of the ions so that the positive Si^{4+} ions are closer to the bottom electrode and the negative O^{2-} ions are closer to the top electrode. This results in a difference of electrical potential between the two electrodes.

In Figure 2.1 (c) negative pressure is applied to the crystal along the x-axis, which has the opposite effect, which results in an inverted potential difference.

In Figure 2.1 (d) the pressure is applied along the y-axis, which causes similar results as the negative pressure along the x-axis shown in Figure 2.1 (c).

If the pressure is applied along the same axis as the resulting electrical field (as in Figures 2.1 (b) and 2.1 (c)) it is called longitudinal piezoelectric effect, whereas it is called transversal piezoelectric effect when the causing pressure and the resulting electrical field are perpendicular to each other (as in Figure 2.1 (d)).



(a) simplified elementary cell of quartz

(b) longitudinal piezoelectric effect positive pressure

(c) longitudinal piezoelectric effect negative pressure

(d) transversal piezoelectric effect positive pressure

Figure 2.1: The cause of the piezoelectric effect in quartz (X-cut) (from [DPV97])

2.1.3.2 Piezoelectric Constants

The correlations between electrical and mechanical processes that occur in piezoelectric materials are described through piezoelectric constants. These constants are unique for each material and the correlations are directional.

The following Equations (2.2) to (2.4) are simplified and apply to a piezoelectric plate. The original equations describe the relations between the directional mechanical quantities *Stress* T_μ and *Strain* S_μ and the directional electrical quantities *electric field* E_k and *electric displacement* D_k. All the lower indices represent the direction. In concrete cases those indices are usually numbers from 1 to 6, with 1, 2, 3 representing the x, y, z axis and 4, 5, 6 representing shearing in the yz, zx, xy plain.

For the direct piezoelectric effect the *piezoelectric deformation constant* h_{ij} describes the relation between the change of thickness Δx_j, which was caused by external mechanical influences and the corresponding open circuit potential. That is

$$h_{ij} = \frac{U_i}{\Delta x_j}\,. \tag{2.2}$$

Furthermore, there is also the *piezoelectric pressure constant* g_{ij}, which describes the relation between the external pressure that causes the deformation, the corresponding voltage and the thickness d. The *piezoelectric pressure constant* is given by

$$g_{ij} = \frac{U_i}{d \cdot p_j}\,. \tag{2.3}$$

For the indirect piezoelectric effect the *piezoelectric modulus* d_{ij} describes the relationship between an electrical potential, which is applied to the material and the corresponding change in thickness Δx_j, as per

$$d_{ij} = \frac{\Delta x_j}{U_i}\,. \tag{2.4}$$

The efficiency between the conversion of applied mechanical energy and correlating electrical energy and vise versa is described through the *piezoelectric coupling coefficient* k. That is

$$k = \sqrt{\frac{Mechanical\,energy\,stored}{Electrical\,energy\,applied}} = \sqrt{\frac{Electrical\,energy\,stored}{Mechanical\,energy\,applied}}\,. \tag{2.5}$$

This section is a compilation of references[KK90, JPW96, DPV97, TEKP10]

2.1.3.3 Piezoelectric Plate As Oscillator

A piezoelectric plate can be forced to oscillate by applying an alternating voltage to it. The plate will expand and contract in the rhythm of the applied voltage frequency. Similar to other oscillating objects the piezoelectric plate has its own *natural frequency* f_0, which is related to its *thickness* d by

$$f_0 = \frac{c}{2 \cdot d} \,. \tag{2.6}$$

If an alternating voltage with the natural frequency is applied to a piezoelectric plate, the plate will oscillate in this frequency, however the oscillation will dampen over time, if the voltage is no longer applied. The damping occurs because of internal friction and energy conversion in the form of ultrasonic waves. The second cause, the creation of ultrasonic waves, is more significant in causing attenuation than the internal friction, while also being the desired effect. The attenuation of the oscillation is measured in the *damping coefficient* b, which describes the amplitude decrement from one oscillation to the next, as illustrated in Figure 2.2

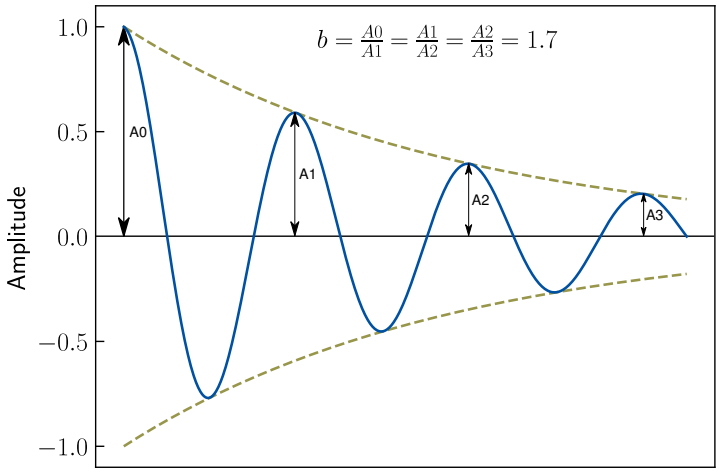


Figure 2.2: Damping of an oscillating piezoelectric plate (from [KK90])

The plate can also be exited from other frequencies than its natural frequency. If that is the case, the plate will oscillate at the exciting frequency, after an initial stabilization process. The amplitude of the oscillation depends on the applied frequency. The largest possible amplitude occurs at the *resonance frequency* f_r, which is usually slightly different from the *natural frequency* of the plate. The largest thickness change, i.e. the amplitude at *resonance frequency* f_r, and its ratio to the amplitude at a static

thickness change (see Equation (2.4)) is described in the *Quality* Q by

$$Q = \frac{\Delta x_{f_r}}{\Delta x_{stat}} \,. \tag{2.7}$$

The *Quality* Q is also related to the *damping coefficient* b (Equation (2.8)) and the *Bandwidth* B as per

$$Q = \frac{\pi}{\ln b} \,, \tag{2.8}$$

$$Q = \frac{f_r}{B} \,. \tag{2.9}$$

Thereby B represents the frequency range around the *resonance frequency*, where the amplitude does not drop below the maximum amplitude by a factor of $\frac{1}{\sqrt{2}}$, i.e. $-3\,\mathrm{dB}$. However, the relation between *Quality* Q and *Bandwidth* B, as described in Equation (2.9), is only applicable if the damping is low or moderate.

In Figure 2.3 you can see the *change of thickness* Δx of a piezoelectric plate over the *frequency* f of the applied voltage and the corresponding oscillation. The lower the frequency, the closer is Δx to the *static thickness change*, which occurs at $0\,Hz$, i.e. under the influence of a constant voltage, and is set to 1 in Figure 2.3. As above mentioned, the maximum *change of thickness* occurs at *resonance frequency* f_r, which is equal to $1\,MHz$ in Figure 2.3.

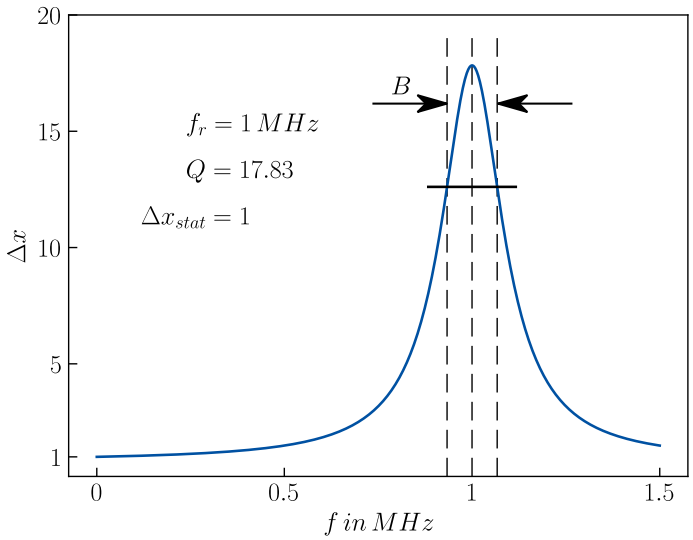


Figure 2.3: Quality of an oscillating piezoelectric plate (from [KK90])

Each material has its own *acoustic impedance* Z, which defines the velocity in which

sound can traverse this material. When a piezoelectric plate with an *acoustic impedance* Z_0 is oscillating between two materials, with according *acoustic impedances* Z_1 and Z_2, as illustrated in Figure 2.4, the *damping coefficient* b can be calculated using Equations (2.10) and (2.11).

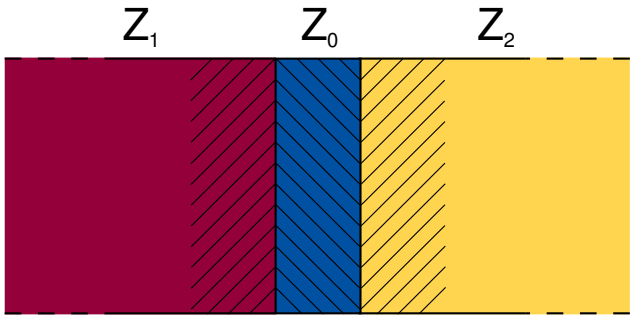


Figure 2.4: Piezoelectric Plate with Z_0 between two materials with Z_1 and Z_2 (from [KK90])

If Z_1 and Z_2 are both greater or both less than Z_0 i.e. if both contiguous materials are sonically softer or sonically harder then the material of the oscillating plate, the *damping coefficient* can be calculated according to

$$b = \frac{(Z_0 + Z_1)(Z_0 + Z_2)}{(Z_0 - Z_1)(Z_0 - Z_2)}\,. \tag{2.10}$$

However, if one of Z_1 or Z_2 is higher then Z_0, while the other impedance is lower then Z_0, the *damping coefficient* must be calculated according to

$$b = \frac{(Z_0 + Z_1)^2(Z_0 + Z_2)^2}{(Z_0 - Z_1)^2(Z_0 - Z_2)^2}\,. \tag{2.11}$$

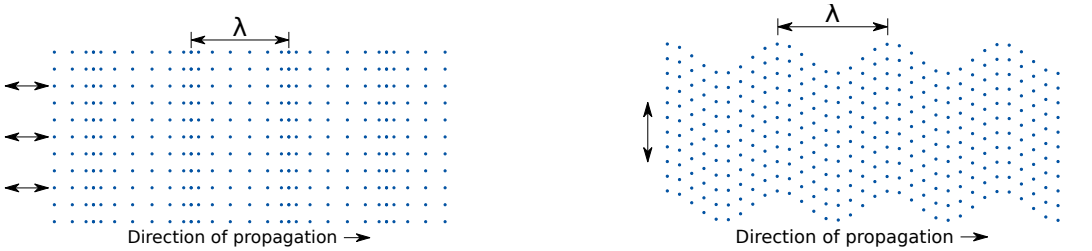
The second described scenario is experiencing significantly stronger damping where the *natural frequency* of the piezoelectric plate is only half of the frequency described in Equation (2.6), which is expressed by

$$f_0 = \frac{1}{2} \cdot \frac{c}{2 \cdot d} = \frac{c}{4 \cdot d}\,. \tag{2.12}$$

In summary damping naturally occurs when ultrasonic waves are created from oscillation, because the waves result as a conversion of energy. While the attenuation decreases the amplitude of the ultrasonic signal, it also enables the transmission of short pulses, which are commonly used in ultrasonic testing. [KK90]

2.1.4 Ultrasonic Waves in Materials

Ultrasonic waves can be differentiated in the various ways they propagate through the material. The two probably most commonly used distinctions are the differentiation in *longitudinal waves* and *transversal waves*, both of which are illustrated in Figure 2.5. The former term describes waves with a direction of propagation which is parallel to the direction of the oscillation, as shown in Figure 2.5 (a). *Longitudinal waves* can propagate in solids, liquids and gases. In contrast to this, *transversal waves* are waves, which have a direction of propagation that is perpendicular to the direction of the oscillation. This behavior is illustrated in Figure 2.5 (b). Furthermore, *transversal waves* can only propagate in solid materials. The speed of *transversal waves* is generally lower compared to the speed of *longitudinal waves*. [DPV97, KK90]



(a) Longitudinal waves (from [KK90]) (b) Transversal waves (from [KK90])

Figure 2.5: Distinction of waves

The speed of sound waves is depending on the material the wave traverses. An important material depended parameter that gives information about the velocity of propagating sound waves is the *acoustic impedance* Z. The *acoustic impedance* is calculated from the *velocity of sound* c in the material and the *density* ρ of the material. It is given by

$$Z = \rho \cdot c\,. \tag{2.13}$$

If an ultrasonic wave hits perpendicular at an interface between two different materials with *acoustic impedances* Z_1 and Z_2, a part of the wave is reflected, while the remaining part gets through. This is illustrated in Figure 2.6.
The ratio between the reflected part and the transmitted part can be expressed by the *reflection coefficient* R and the *transmission coefficient* T, which are given by

$$R = \frac{Z_2 - Z_1}{Z_1 + Z_2} \tag{2.14}$$

and

$$T = \frac{2 \cdot Z_2}{Z_1 + Z_2}\,. \tag{2.15}$$

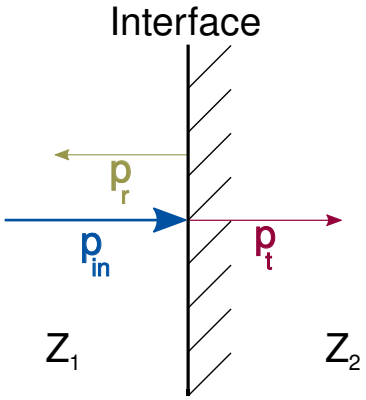


Figure 2.6: Reflection of ultrasonic waves at an interface between two materials (from [DPV97])

An incoming wave with the *sound pressure* p_{in} will split into a reflected wave with *sound pressure* p_r and a transmitted wave with *sound pressure* p_t, which can be calculated according to

$$p_r = p_{in} \cdot R \qquad (2.16) \qquad \text{and} \qquad p_t = p_{in} \cdot T\,. \qquad (2.17)$$

When Z_1 is much bigger then Z_2 then most of the wave is reflected. A common scenario, where this is the case, is when an ultrasonic wave crosses from steal to water. In contrary, the closer Z_1 and Z_2 are to one another, the less reflection occurs. The phase of the reflected wave is inverted when $R < 0$.[DPV97]

2.2 Ultrasonic Testing Methods

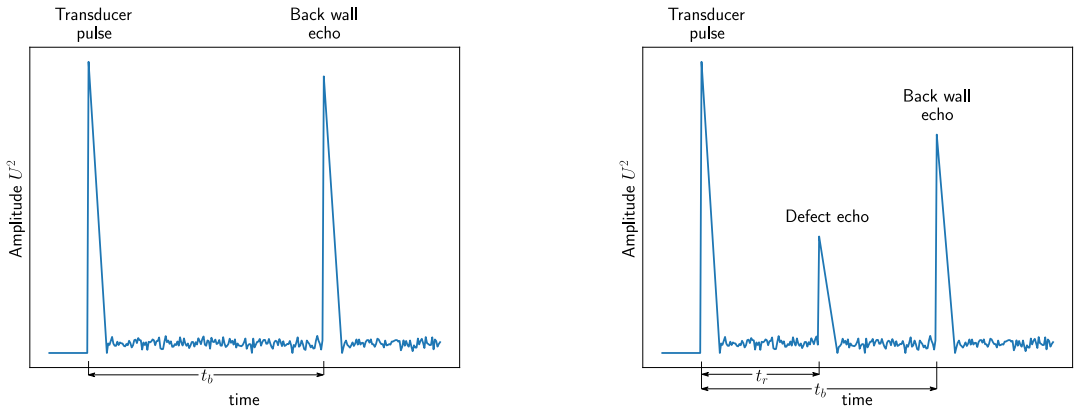
2.2.1 Pulse-echo

As the name suggests the pulse-echo method describes a procedure during which an ultrasonic pulse is send through the material and the reflections (echoes) are measured. The ultrasonic waves are propagating through the material with an *acoustic impedance* Z_0 and are reflected when the waves cross into a section with a different *acoustic impedance* Z_r.

A specific example object in ultrasonic testing would be a metal (e.g. steel) cuboid. When creating ultrasonic waves in the cuboid they will propagate trough the object until they reach the back wall of the cuboid, where a portion of the waves are reflected, while some other portion will leave the object. The reflections will continue to

propagate and reflect through the cuboid until the energy of the wave is completely depleted.

When measuring the ultrasonic waves with an piezoelectric receiver and plotting the measured amplitudes over time the initial pulse and every reflection are visible as peaks. This plot is called an A-scan and two examples are shown in Figure 2.7.



(a) A-scan with no defects

(b) A-scan with a defect

Figure 2.7: Schematic Pulse Echo A-scans (from [KK90])

Assuming the metal cuboid is a flawless whole piece of the same metal, the plot would only show two, with constant time reappearing, peaks, as illustrated in Figure 2.7 (a). The first peak is always the initial pulse or a reflection at the front wall of the cuboid, while the second peak is always a reflection at the back wall of the cuboid. When the object is not flawless, for example a cuboid with defects, cracks, holes, corrosion, etc. multiple different peaks will be visible. Assuming the same flawless cuboid has a defect in its center, the A-scan will still show the initial peak and the back wall echo peak at the same position as the previous plot. However, it will have a third (probably smaller) peak between those two peaks, which indicates the reflection at the crack. The back wall peaks will also be lower in amplitude as in the previous A-scan because the wave loses energy when reflecting at the crack as well as at the back wall. With a sufficient quality of measurements the *depth* d of the back wall and the *depth* of the crack can be calculated from the *time difference* Δt of the peaks. Considering that the sound wave traverses the way two times, because the echos are measured, the *depth* d can be calculated by

$$d = \frac{c \cdot \Delta t}{2} . \tag{2.18}$$

By applying Equation (2.18) to the measured signals in Figure 2.7 (b) it is possible to calculate the back wall depth d_b, i.e. the thickness of the cuboid and the depth of the defect d_r (see Equation (2.19)).

$$d_b = \frac{c \cdot t_b}{2} \qquad d_r = \frac{c \cdot t_r}{2} \tag{2.19}$$

The simplest form of a pulse-echo acquisition can be performed with a single beam normalprobe, as illustrated in Figure 2.8. It can produce ultrasonic waves, which traverse the material perpendicular to the touching surface.

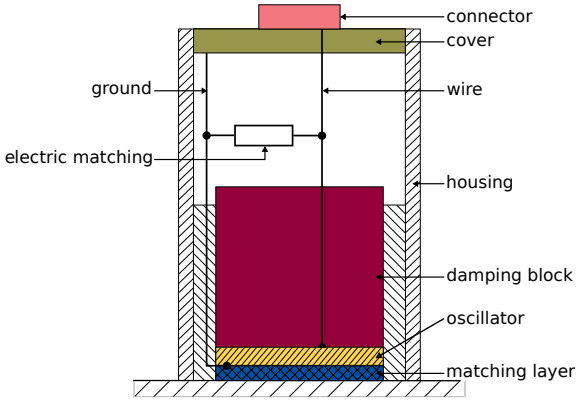


Figure 2.8: Single beam normalprobe for pulse echo acquisitions (from [KK90, Bro12])

The sound waves are created by an oscillating single-crystal plate. The oscillations traverse from the oscillator into the matching layer and from there into the material. The matching layer should match the acoustic impedance of the test object. Therefore, different probes with different matching layers have to be used when measuring different materials, e.g. other probes are used when measuring in water then when measuring in steal. The damping block reduces the production of ultrasonic waves at the back side of the oscillator and it attenuates the oscillation, which results in shorter oscillation cycles. This is necessary when transmitting short pulses. The electronic matching, the wires and the piezoelectric crystal form an electrical circuit, which can be connected to the ultrasonic testing system at the connector. Furthermore, these components are matched to impedance of the ultrasonic testing system. [KK90, Bro12]

2.2.2 Through-Transmission

In the simplest form of the through-transmission method (or shadow method) an ultrasonic transceiver and a secondary receiver are placed at opposite ends of the test object. The receiver measures the waves that passed through and exited the object. If a defect is in the path between the transceiver and receiver, it reflects the incoming ultrasonic waves and creates a shadow. This shadow is then measured by the receiver in the form of lower voltages, due to the lack of ultrasonic waves which reach the receiver. However it is also possible to use the through-transmission method on reflections of the generated waves, e.g. when using angled probes. The use of continuous waves often causes standing waves, which affect the measurements negatively. Therefore, methods like frequency modulation or spread spectrum have to be used in order to counteract the negative effect of standing waves [KK90].

2.2.3 Transmit-Receive

With an transmitter-receiver (TR) probe, as illustrated in Figure 2.9, it is possible to measure the ultrasonic waves, while simultaneously transmitting them, with a single probe. This increases the accuracy of the measurement.

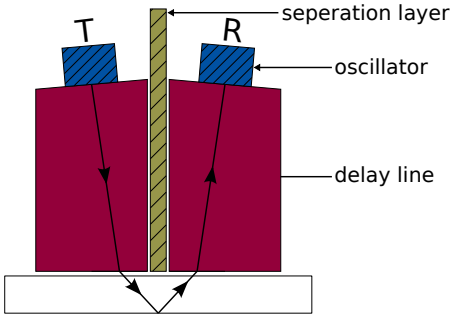


Figure 2.9: Transmitter-receiver (TR) Transducer (from [KK90])

To achieve this, two piezoelectric crystals are used. One of the crystal acts as the transmitter, while the other acts as the receiver at the same time. To ensure, that the receiver measures the reflected ultrasonic waves and not only oscillations of the transmitter, receiver and transmitter are shielded from one another, acoustically as well as electrically. Therefore, a separating layer, which is typically made from cork or foamed plastic, is installed between both piezos. This layer has a high damping effect on ultrasound. Furthermore, the layer contains electrical screening in order to prevent electrical cross coupling.

The transmitter and receiver are both mounted in the same housing, but each one has its own connector, which connects to a separate twin cable. The angle of the ultrasonic waves that are sent by a TR probe is not perfectly straight, so that the reflected wave hits the receiver, which is located slightly offset to the transmitter. This means that the track of the waves forms a triangle with the surface of the object and that the traversed distance is longer than Equation (2.18) describes. This has to be considered and the formula has to be compensated. The main purpose of TR probes is to measure flaws close to the surface of the material. This is possible because, the delay line eliminates the dead zone of typical single beam normalprobes.

2.2.4 Coded Signals

When measuring signals, in general, the signal-to-noise-ratio (SNR) is important. The measurements are not very useful when the distinction between signal (information) and noise can not be made. This also applies to ultrasonic testing. When transmitting ultrasonic signals the SNR can be improved by applying more *sound pressure*, i.e. by applying higher voltages to the transducer. However, there is always a limit to the maximum *sound pressure* that can be applied. This limit may be of physical or other nature, e.g. in the medical use of ultrasound, the maximum peak intensity is limited because of health concerns. However, there is another way to increase the energy which is injected into the system in order to increase the SNR. By increasing the duration of the transmission while using specific techniques the transmitted energy can be accumulated after the signal is received. One such technique is the transmission of barker codes. Barker codes have the property of forming peaks with lower side lobes when being autocorrelated. This property is illustrated in Figure 2.10. Therefore, it is possible, to transmit such barker codes and correlate the received signal with the code sequence itself in order to gain better SNR. [Mis01].

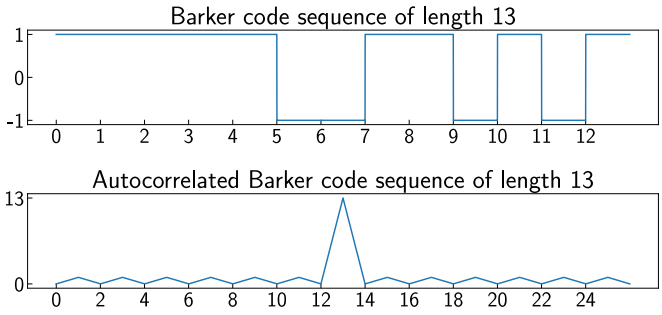


Figure 2.10: Barker Code of length 13 and its autocorrelation function

3 Open Source Ultrasonic Testing Hardware - Un0rick

Ultrasonic testing systems are usually proprietary commercial products, which don't allow access to the hardware, firmware or raw data of the measurements. Furthermore, they are often quite expensive, which makes them unattractive for educational, research and development purposes.

Open source hardware, such as the Un0rick project, is a potential solution to those problems, however there are not many open source hardware projects for ultrasonic applications.

In the specific case of the Un0rick project, all of the electrical schematics, PCB design, firmware, software and documentation are public on github and fall under the *TAPR Open Hardware License*. This makes it an interesting project for educational, research and development purposes [Jon21a].

3.1 Un0rick - Open Source Ultrasonic Testing Board

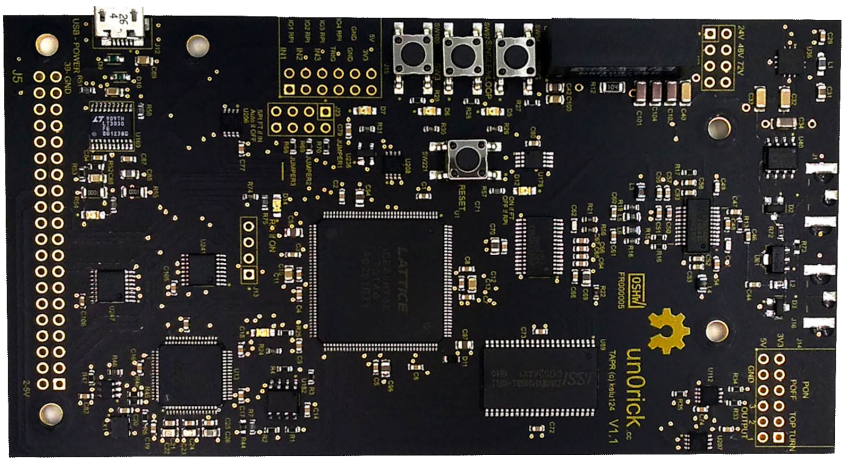


Figure 3.1: Un0rick open source ultrasound hardware board (from [Jon20c])

3.1.1 System and Function Overview

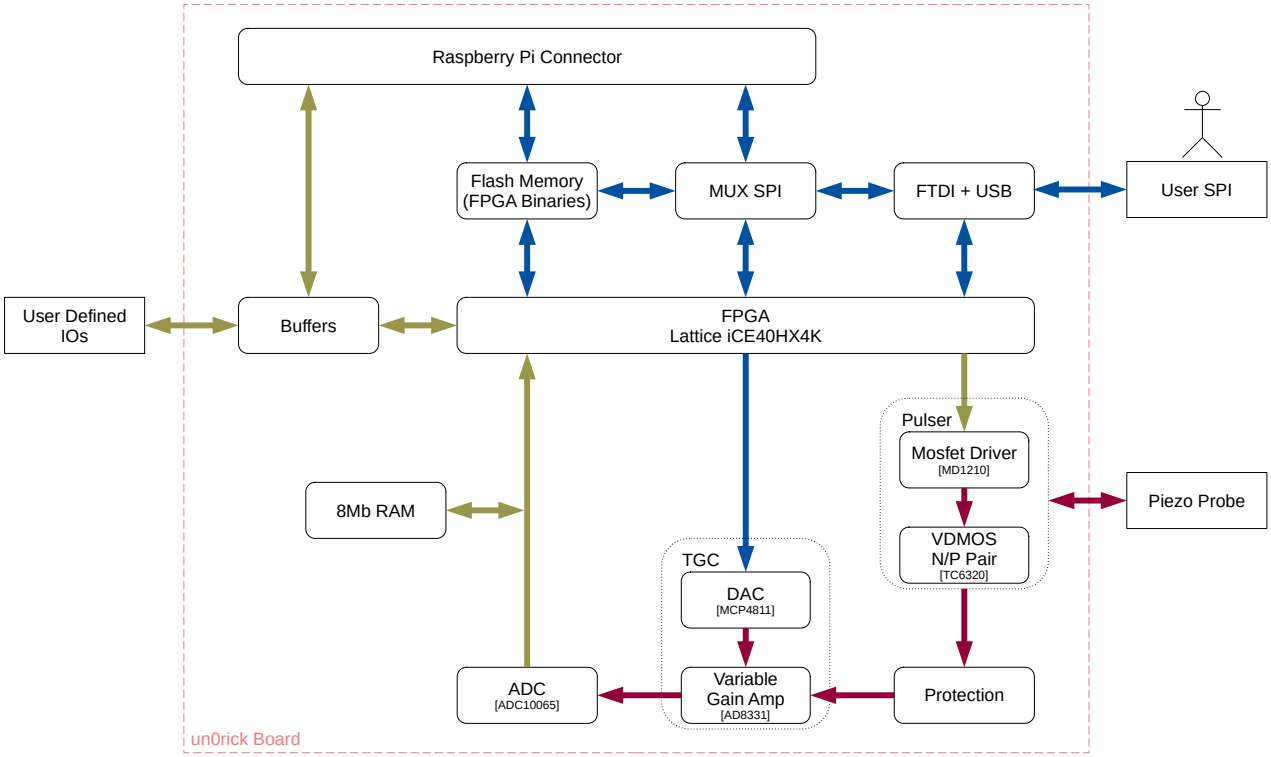


Figure 3.2: Un0rick system block diagram (from [Jon21b])

Figure 3.2 shows a system overview of the Un0rick board. The main interfaces of the board are the user SPI interface, which enables the user to control the board from a Raspberry Pi or a USB cable and the coaxial connector for a piezoelectric transducer probe. Everything on the board is controlled by the FPGA, including the SPI communication [Jon21b].

The following Sections 3.1.2, 3.1.3 and 3.1.3.1 to 3.1.3.3 are attempting to explain the low level process of an ultrasonic acquisition on the Un0rick board in detail. This process is not explicitly documented, but of interest when trying to understand the mechanisms of the board e.g. in order to implement custom testing methods. Therefore the following sections are a compilation of information obtained from the Un0rick schematics as well as the FPGA source code and any other documentation available [Jon20a, Jon20b, Jon20c, Jon20e, Jon21a, Jon19].

3.1.2 Signal Generation

The signal that is applied to the piezoelectric transducer is a single unipolar pulse. The output level can be configured only physically, by placing a jumper bridge (J22). It can be set to $5\,V$, $25\,V$, $50\,V$ and $75\,V$. The pulse shape is controlled by the on board FPGA. It uses two output pins, named Pon and Poff , that control the logic levels of the high voltage pulser circuit, which is shown in Figure B.3. The high voltage pulser mainly consist out of two integrated circuits (ICs) [Jon20a, Jon20e]:

1. The MD1210
   High Speed Dual MOSFET Driver

2. The TC6320
   N-Channel and P-Channel Enhancement-Mode MOSFET Pair

These two ICs are designed to work together as high voltage pulser or as a driver circuit for a piezoelectric transducer. The high voltage pulser circuit of the Un0rick board is very similar to the typical application circuit described in the data sheet of the TC6320 , with the difference being the implemented voltage sources, which causes the unipolar pulse on the Un0rick board, although the design for a bipolar output signal is described in the data sheet.

The TC6320 is a combination of a n-channel power MOSFET (VDMOS) and a p-channel power MOSFET, which both operate in enhancement-mode. This allows the fast switching, similar to a CMOSFET, which implements a gate.

The MD1210 operates as a driver circuit for the TC6320 . It converts the logic level signals that come from the FPGA output pins to the appropriate high and low levels of the TC6320 input pins. As shown in Table 3.1, the MD1210 simultaneously works as an inverter [Dat16b, Dat19].

This allows the FPGA to control the pulse with the Pon and Poff and therefore the pulse shape is dependent on the firmware implementation of the FPGA. In the VHDL firmware version the signal is configured with 3 registers which are shown in Table 3.2. These registers are configurable time values, which control the output levels of the Pon and Poff pin and therefore the shape of the pulse. This behavior is visualized in Figure 3.3.

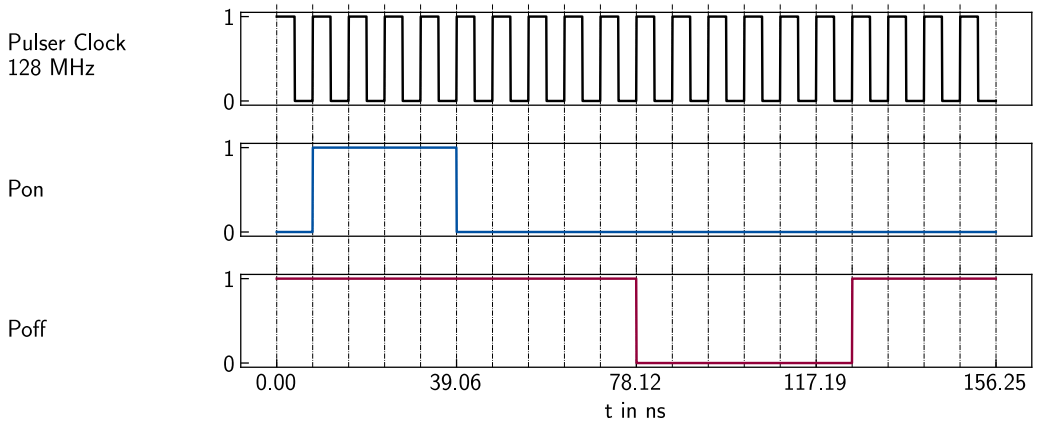


Figure 3.3: Pon and Poff during signal generation
`sEEPon` $= 04_{\mathrm{H}}$, `sEEPonPoff` $= 09_{\mathrm{H}}$, `sEEPoff` $= 10_{\mathrm{H}}$

Logic Inputs			Output	
OE	**INA (from Pon)**	**INB (from Poff)**	**OUTA**	**OUTB**
High	Low	Low	V_H	V_H
High	Low	High	V_H	V_L
High	High	Low	V_L	V_H
High	High	High	V_L	V_L
Low	X	X	V_H	V_L

Table 3.1: MD1210 Logic Inputs to Outputs Table (from [Dat19])

The default value for `Pon` is 0, while the default value for `Poff` is 1. When `Pon` is set to high, while `Poff` is also set to high, the p-channel VDMOS is activated and the voltage of the signal is pulled up, towards the configured voltage. In the opposite case of both `Pon` and `Poff` being set to low, the n-channel VDMOS is activated and the signal is dampened. When both pins are on their default levels, i.e. `Pon` is 0 and "Poff is 1, both MOSFETs are deactivated and the signal is neither pulled up or dampened, which is especially useful while measuring received ultrasonic waves.

A pulse during signal generation is always shaped the same way. First `Pon` is set from 0 to 1 and stays high for `sEEPon` clock cycles or `sEEPon`$\cdot 7.8125\,ns$. After this time `Pon` is set to 0 while `Poff` remains 1 for the time difference of `sEEPonPoff` and `sEEPon` . Following this, `Poff` is set from high to low for the time difference of `sEEPoff` and `sEEPonPoff` . After this both pins return to their default value, being 0 for `Pon` and 1 for `Poff` . Like all the other configuration registers, `sEEPon` , `sEEPonPoff` and `sEEPoff` can be set using SPI before an acquisition [Jon20a, Jon20e, Jon21d, Jon19, Dat16b, Dat19].

Register	Size (Bits)	Default Value in Hex	Unit
`sEEPon`	8	14_{H}	$\frac{1}{128}\,\mu s = 7.8125\,ns$
`sEEPonPoff`	8	$0\mathrm{A}_{\mathrm{H}}$	$\frac{1}{128}\,\mu s = 7.8125\,ns$
`sEEPoff`	16	$\mathrm{C8}_{\mathrm{H}}$	$\frac{1}{128}\,\mu s = 7.8125\,ns$

Table 3.2: FPGA registers for pulse shape configuration (from [Jon19])

3.1.3 Data Acquisition

The signal that is generated by the piezoelectric probe is sampled by an `ADC10065` analog digital converter. However, before the electrical signal gets sampled, it first gets amplified by the time gain compensation (TGC) and filtered by an aliasing filter. The corresponding schematics are shown in Figures B.1, B.2 and B.4 [Jon20a, Dat13].

3.1.3.1 Time Gain Compensation - TGC

The TGC consists of an `AD8331` variable gain amplifier (VGA) and an `MCP4811` digital to analog converter (DAC) as shown in Figures B.1 and B.4. The `MCP4811` is set by the FPGA via SPI. The output voltage of the DAC is used as reference voltage by the `AD8331` . This makes it possible to change the amplification of the measured signal, with increasing time, during the acquisition. In the VHDL firmware version (v1.1) the FPGA has 42 configuration registers in order to configure the TGC before an acquisition. These configuration registers represent the DAC values with a $5\,\mu s$ time spacing [Jon20a, Jon20c, Jon20e, Dat10, Dat16a].

3.1.3.2 Anti-Aliasing Low Pass Filter

The electrical signal that is generated by the piezoelectric receiver is send through a single ended wire to the `AD8331` VGA. The output of the amplifier is a differential signal, which is filtered by a differential anti-aliasing low pass filter, which is shown in Figures 3.4 and B.4. Beside the schematics of the board, no official documentation of this filter is released. However, in the field of measurement and signal processing, this could be important information. Therefore, the anti-aliasing filter was calculated and independently simulated with spice [Jon20a].

In order to analyze the filter, the differential filter (Figure 3.4) was transformed into a single ended filter, which is shown in Figure 3.5. This transformation can be applied by removing the horizontal components of one end (in this case $R_{1'}$, $L_{1'}$, $C_{1'}$ and $R_{2'}$ were removed), while keeping the horizontal components of the other end

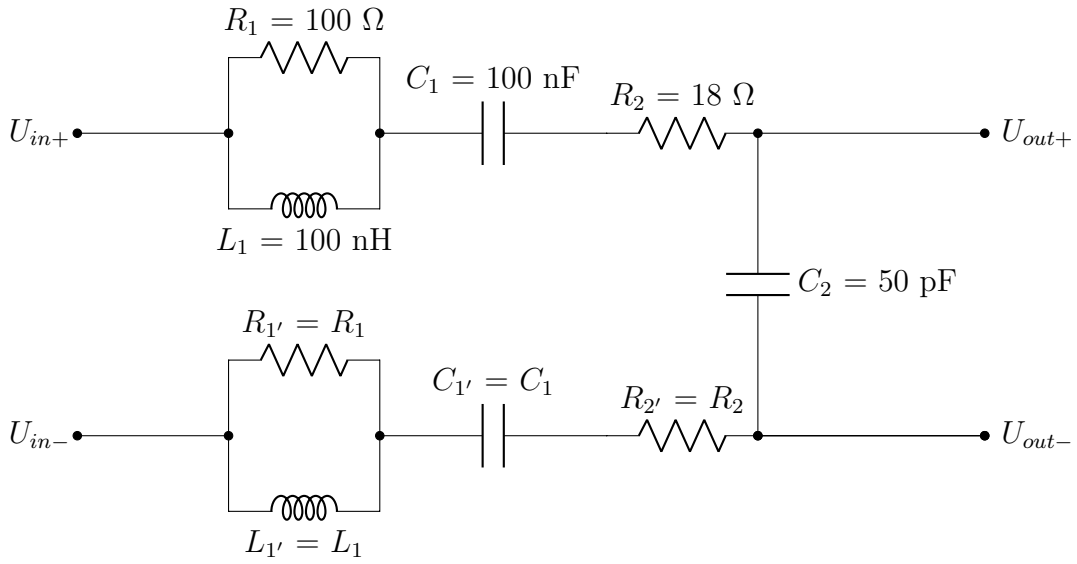


Figure 3.4: Differential anti-aliasing low pass filter

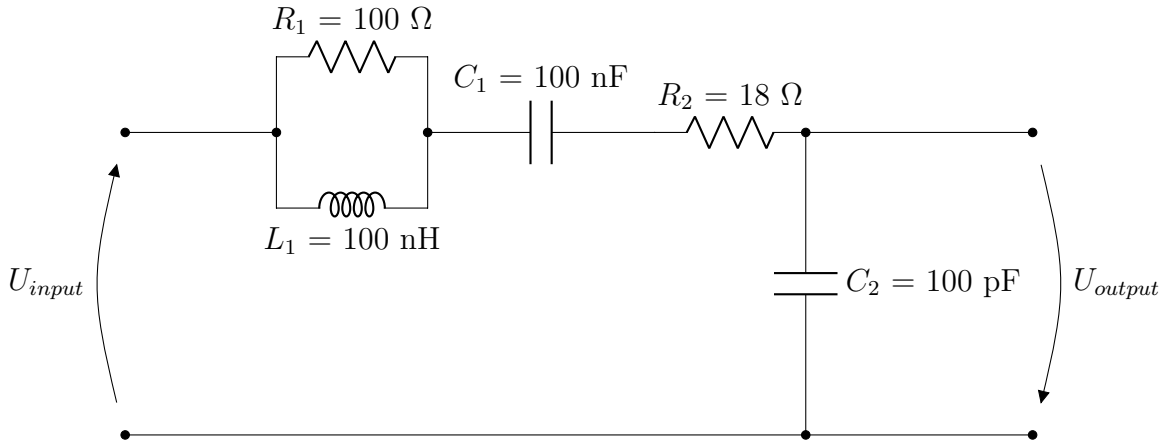


Figure 3.5: Single ended anti-aliasing low pass filter after transformation

unchanged. The capacity of the vertical shunt capacitor in the differential circuit is half of the capacitor in the single ended circuit, therefore the capacity of C_2 has to doubled during the transformation. [Cha10]

The corresponding single ended filter has the same properties, i.e. the same transfer function, as the differential filter. The transfer function $H(s)$ was calculated on the basis of the simplified single ended circuit following Equations (3.1) to (3.7). As a result, other typical filter characteristics can be described by Equations (3.9), (3.10), (3.12), (3.14) and (3.16).

$$H(s) = \frac{U_{output}}{U_{input}} \tag{3.1}$$

$$H(s) = \frac{Z_3}{Z_1 + Z_2 + Z_3} \text{ with } Z_1 = \frac{sL_1R_1}{sL_1 + R_1}\ ,\ Z_2 = R_2 + \frac{1}{sC_1}\ ,\ Z_3 = \frac{1}{sC_2} \tag{3.2}$$

$$H(s) = \frac{\frac{1}{sC_2}}{[\frac{sL_1R_1}{sL_1+R_1}] + [R_2 + \frac{1}{sC_1}] + [\frac{1}{sC_2}]} \tag{3.3}$$

$$H(s) = \frac{1}{[\frac{s^2C_2L_1R_1}{sL_1+R_1}] + [sC_2R_2] + [\frac{C_2}{C_1} + 1]} \tag{3.4}$$

$$H(s) = \frac{sL_1 + R_1}{[s^2C_2L_1R_1] + [s^2C_2L_1R_2 + sC_2R_1R_2] + [s(\frac{C_2}{C_1} + 1)L_1 + (\frac{C_2}{C_1} + 1)R_1]} \tag{3.5}$$

$$H(s) = \frac{sL_1 + R_1}{s^2C_2L_1(R_1 + R_2) + s(C_2R_1R_2 + (\frac{C_2}{C_1} + 1)L_1) + (\frac{C_2}{C_1} + 1)R_1} \tag{3.6}$$

Transfer Funtion:

$$H(s) = \frac{sy_1 + y_0}{s^2x_2 + sx_1 + x_0} \quad \text{with} \quad \begin{aligned} y_1 &= L_1 \\ y_0 &= R_1 \\ x_2 &= C_2L_1(R_1 + R_2) \\ x_1 &= C_2R_1R_2 + (\frac{C_2}{C_1} + 1)L_1 \\ x_0 &= (\frac{C_2}{C_1} + 1)R_1 \end{aligned} \tag{3.7}$$

Frequency Response:

$$H(j\omega) = \frac{j\omega y_1 + y_0}{-\omega^2x_2 + j\omega x_1 + x_0} \tag{3.8}$$

$$H(j\omega) = \frac{y_0 + j\omega y_1}{(x_0 - \omega^2x_2) + j\omega x_1} \tag{3.9}$$

Amplitude:

$$|H(j\omega)| = \frac{\sqrt{y_0^2 + (\omega y_1)^2}}{\sqrt{(x_0 - \omega^2x_2)^2 + (\omega x_1)^2}} \tag{3.10}$$

Phase:

$$\phi(\omega) = arg\{H(j\omega)\} \tag{3.11}$$

$$\phi(\omega) = arctan(\frac{\omega y_1}{y_0}) - arctan(\frac{\omega x_1}{x_0 - \omega^2 x_2}) \tag{3.12}$$

Group Delay:

$$\tau_g(\omega) = -\frac{d\phi(\omega)}{d\omega} \tag{3.13}$$

$$\tau_g(\omega) = \frac{x_1(\omega^2 x_2 + x_0)}{\omega^4 x_2^2 + \omega^2 x_1^2 - 2\omega^2 x_0 x_2 + x_0^2} - \frac{y_0 y_1}{\omega^2 y_1^2 + y_0^2} \tag{3.14}$$

Phase Delay:

$$\tau_\phi(\omega) = -\frac{\phi(\omega)}{\omega} \tag{3.15}$$

$$\tau_\phi(\omega) = \frac{arctan(\frac{\omega x_1}{x_0 - \omega^2 x_2}) - arctan(\frac{\omega y_1}{y_0})}{\omega} \tag{3.16}$$

To verify the accuracy of the calculated transfer function (Equation (3.7)) as well as the transformation from the differential filter to a single ended filter, both circuits (Figures 3.4 and 3.5) were simulated in spice (*ngspice*) and the corresponding bode plots are visualized in Figure 3.6. The similarity of the plots verifies the correctness of the transformation and the calculations.

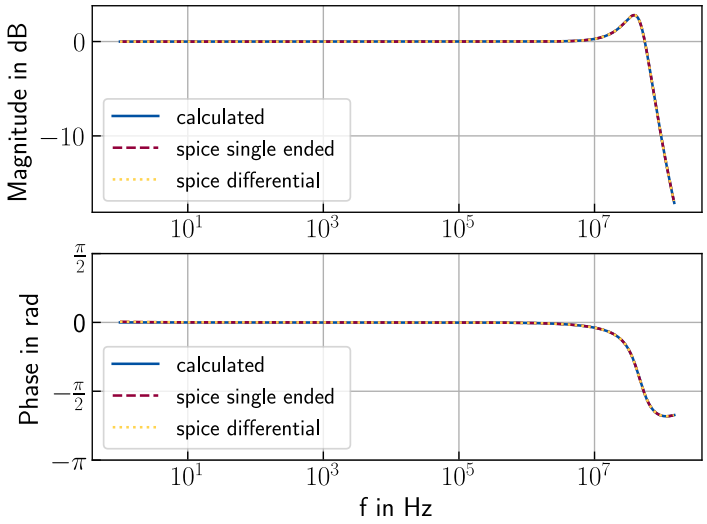


Figure 3.6: Bode plot of calculated transfer function and simulated (spice) circuits

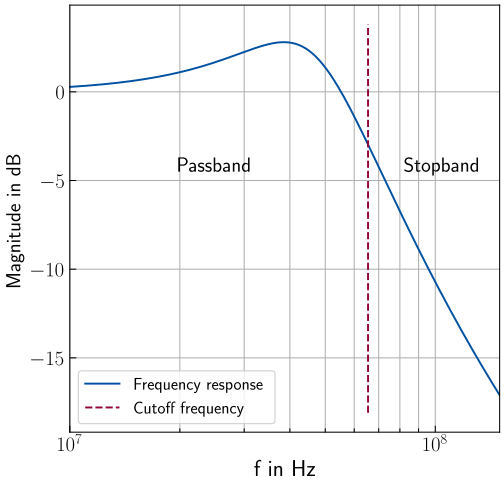
When observing the frequency response of the bode plot (Figure 3.6) the low pass characteristic is of this filter is obvious. The gain is constant throughout the lower frequencies until it peaks around $40\,MHz$ whereafter it drops. The cutoff frequency at $-3\,dB$ is at around $65\,MHz$, which is shown in detail in Figure 3.7 (a). This is unexpectedly high, since the sampling rate of the ADC is $64\,MHz$, which usually is combined, with a cutoff frequency half as big.

The cutoff frequency is higher then the *Nyquist frequency* of the ADC, therefore, in theory, aliasing can occur. In reality it probably is not very likely for the aliasing to occur in the form of ultrasonic waves, as long as common transducers are used, because the frequency range between $32\,MHz$ and $64\,MHz$ (in which the aliasing can occur), is not part of the signal bandwidth and therefore should mostly contain low level noise. However measurements with ultra-high-frequency ultrasonic transducers could experience aliasing with the currently implemented filter. Furthermore, aliasing can always occur on the electrical signal trays, especially since the Un0rick board has high frequency analog signals as well as high frequency digital signals (FPGA) on the same board.

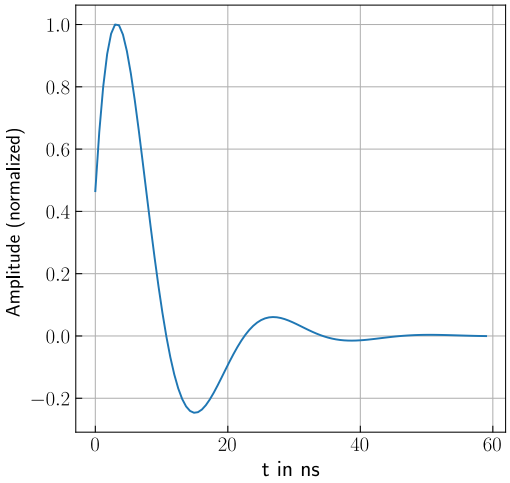
In the verilog firmware version of the FPGA implementation it is possible to enable the double rate mode. In this mode two acquisitions are done one after the other and the corresponding samples are interleaved in post processing, so that the resulting signal has an artificial sample rate of $128\,MHz$. This can improve aliasing when sampling with $64\,MHz$, however it does not improve aliasing when sampling with lower frequencies.

It is also notable that this filter is not a standard filter like i.e. a Butterworth or Chebyshev filter. It is rather based on the low pass filter circuits shown in the data sheet of the `AD8331` .

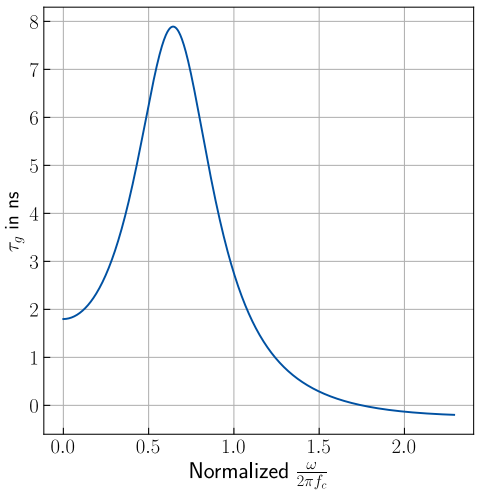
The bode plot (Figure 3.6) shows the magnitude (Equation (3.10) in dB) and phase (Equation (3.12)) of the frequency response (Equation (3.9)), while Figure 3.7 (a) shows the magnitude around the cutoff frequency. Group and phase delay (Equations (3.14) and (3.16)) are plotted in Figures 3.7 (c) and 3.7 (d), respectively. The plots for the magnitude, phase and impulse response (Figure 3.7 (b)) were calculated by the use of the python *SciPy* signal processing library [Jon20a, Jon20d, Jon20e, Jon21f, Dat16a].



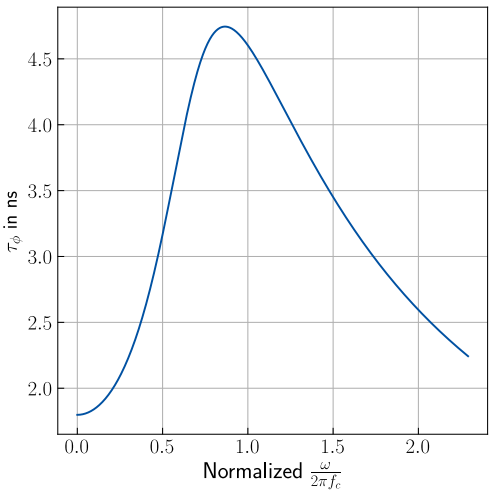
(a) Frequency response with cutoff frequency



(b) Impulse response



(c) Group delay



(d) Phase delay

Figure 3.7: More characteristics of the Un0rick anti-aliasing low pass filter

3.1.3.3 Acquisition Settings

Besides the parameters mentioned in Section 3.1.2 there are other settings, which have to be configured before an acquisition. Those settings are listed in Table 3.3.

The `sEEDelayACQ` register sets the start delay of the ADC sampling in clock cycles. If this value is set to 0 the ADC is starting the sampling process at the same time as the pulse generation starts. This however might be impractical, especially when using a pulse-echo transducer, because the first samples will contain the oscillations and noise which is generated by the pulsing transducer. By setting this value to a higher value, the starting time of the sampling process can be set after the initial impulse entered the material and the pulse-echo transducer is only receiving the reflected ultrasonic waves.

The `sEEACQ` register configures after how many clock cycles a single acquisition should end. Its important to note that the beginning of an acquisition is always the start of the pulse generation and not the time specified in `sEEDelayACQ`. Concluding this, the number of clock cycles in which the ADC is writing samples during a single acquisition is `sEEACQ` − `sEEDelayACQ`.

It is also possible to trigger multiple acquisitions after one another. In order to do this, the register `sEESingleCont` has to be set to 1. If `sEESingleCont` is set to 0 instead, only one acquisition will be triggered. If `sEESingleCont` is set it is necessary to specify the number of repeated acquisitions in the `sEETrigCounter` register. The time spacing between one acquisition to the following triggering is configurable in the `sEEPeriod` register [Jon20a, Jon20e, Jon21d, Jon19].

Register	Size (Bits)	Default Value in Hex	Unit
`sEEDelayACQ`	16	$02BC_H$	$\frac{1}{128}\,\mu s = 7.8125\,ns$
`sEEACQ`	16	$32C8_H$	$\frac{1}{128}\,\mu s = 7.8125\,ns$
`sEEPeriod`	24	$186A0_H$	$\frac{1}{128}\,\mu s = 7.8125\,ns$
`sEETrigCounter`	8	$0A_H$	number of repetitions
`sEESingleCont`	1	00_H	Boolean

Table 3.3: FPGA registers for acquisition settings

3.1.4 SPI Interface

The FPGA firmware implements configuration registers, which can be set prior to an acquisition. It also implements a SPI Interface which allows to write values to these

registers. This creates compatibility with many different devices, like microcontrollers, (single-board) computers or basically any other device that implements SPI.

The data that is measured during an acquisition is stored in the external on board SRAM. Similar to the configuration registers, the data can also be accessed by using the SPI interface, however in this case it is a reading operation for the single samples.

The interface is implemented internally by a simple finite-state machine, which is illustrated as a graph in Figure 3.8.

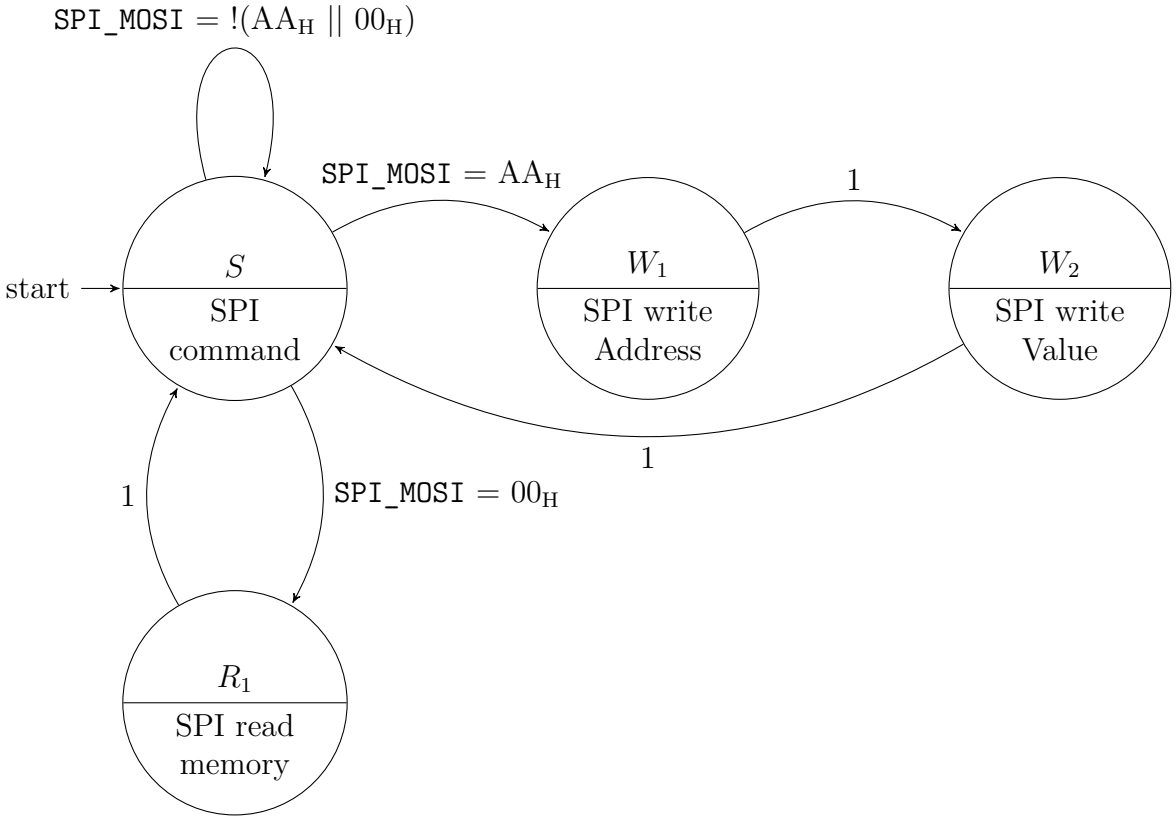


Figure 3.8: Finite-state machine that implements the SPI-Interface on the FPGA in the Un0rick firmware (v1.1)

To write a configuration register first AA_H is send to the FPGA. This indicates that a writing operation is to be performed (Figure 3.8, state S). After this the FPGA expects the address of the register. Every configuration register is assigned to an 8 bit address. This address is multiplexed so that the destination of the write operation can be determined. Configuration parameters which exceed the size of 8 bit have multiple addresses, e.g. a 16 bit parameter will have two addresses, one for the high byte and one for the low byte (Figure 3.8, state W_1). After the address is transmitted, the next value that is sent to the FPGA will be written to the register with the specified address (Figure 3.8, state W_2).

To read the measured data 00_H has to be send to the FPGA. This requests one byte of measured data (Figure 3.8, state S). The FPGA will read one byte of data at the current location of the memory pointer and transmit it (Figure 3.8, state R_1). After this, the memory pointer is incremented automatically, so that the next byte can be read, by transmitting 00_H again. A sample is a 10 bit ADC Value, which is encoded into a 16 bit value. Therefore, in order to read N samples the reading operation has to be carried out $2 \cdot N$ times, i.e. one has to transmit 00_H and store the received byte $2 \cdot N$ times [Jon20e, Jon21d, Jon19]..

3.1.5 Setup and Measurements

3.1.5.1 Obtaining Firmware

There are different versions of compiled binaries available in the `bins` directory of the official github repository. One can simply download those files and flash them onto the board. It is also possible to compile them from their source files, which is particularly of interest, when one wants to create a custom firmware. Both steps are described further in Section 3.1.5.2.

3.1.5.2 Toolchain

To flash the binaries to the Un0rick board, the jumper `J23` has to be set to `SPI FT if IN`. Then the board can be connect to the PC with a Micro-USB cable. The firmware can than be installed to the board by using a designated software. The vendor of the FPGA (lattice), has its own proprietary software called *Lattice Diamond Programmer* for this. Alternatively, the *Yosis Open SYnthesis Suite* offers an open source solution, the *iceprog* tool, for this purpose [Jon21a, Jon21d].

Tools for flashing the board

- Lattice Diamond Programmer
- iceprog (Yosis)

For firmware setups, *iceprog* was used on a linux VM as well as a native linux machine to flash the device.

To compile the firmware binaries from the source code, the processes for VHDL and Verilog versions have to be distinguished. The official *iCEcube2 Design Software*

from *lattice* can be used to create and compile Verilog and VHDL projects. However, specifically for Verilog source code the open source *Project IceStorm* toolchain can be used for compilation.

Tools for creating binaries from source

- Lattice iCEcube2 Design Software
- Project IceStorm (Yosis + next)

For custom firmware compilation *iCEcube2* was used for VHDL based source code and the *Project IceStorm* toolchain was used for Verilog based source code [Jon20d, Jon21d, Jon21e].

3.1.5.3 Software

In order to control the Un0rick board, as well as configure the acquisition settings and actually perform the acquisitions, SPI communication (as described in Section 3.1.4) is needed. While this can be achieved basically by any device that has an SPI interface, i.e. a PC, microcontroller, etc., the official github repository of the Un0rick board contains two open source python libraries for simple usage with a Raspberry Pi or a PC, depending on the firmware.

pyUn0 Library

The *pyUn0* library is a python library made for the v1.1 firmware version. It utilizes the SPI Interface at the Raspberry Pi 22 Header connector. It is designed to be run on a Raspberry Pi 3 or 4, which is connected to the board with a ribbon cable.

un0usb Library

The *un0usb* library is a python library made for the USB firmware version. It utilizes the FTDI SPI Interface at the USB connector. This makes it possible to connect the board directly to a PC, which is especially convenient when working with different firmware versions, because the binaries are also flashed via the USB connection.

Those python libraries make it convenient to configure the board, do an acquisition and read the measured data from the board. The measured data can then be processed

or visualized in python or other software.

Custom Software

In the frame of this bachelor thesis a custom python library was created. The first reason for its creation was to understand how the Un0rick board and the SPI interface works. The second reason was to improve the readability of the existing pyUn0 library which was the basis of this new custom library.

This new python library was then used in all measurements which where done during this thesis, while using the VHDL firmware version v1.1. The library covers board configuration, making acquisitions, post-processing and data visualization. It was used in python scripts and Jupyter notebooks.

Furthermore, a simple python HTTP server was implemented during this thesis. This server was based on the custom python library and serves a client interface written in HTML, CSS and JavaScript. The server implements the SPI communication with the Un0rick board and exposes it with a REST API to the client. This was used to create a live updating A-scan display, with sliders and buttons for more user friendly board configuration then plain code. An image of the client interface is shown in Figure 3.9.

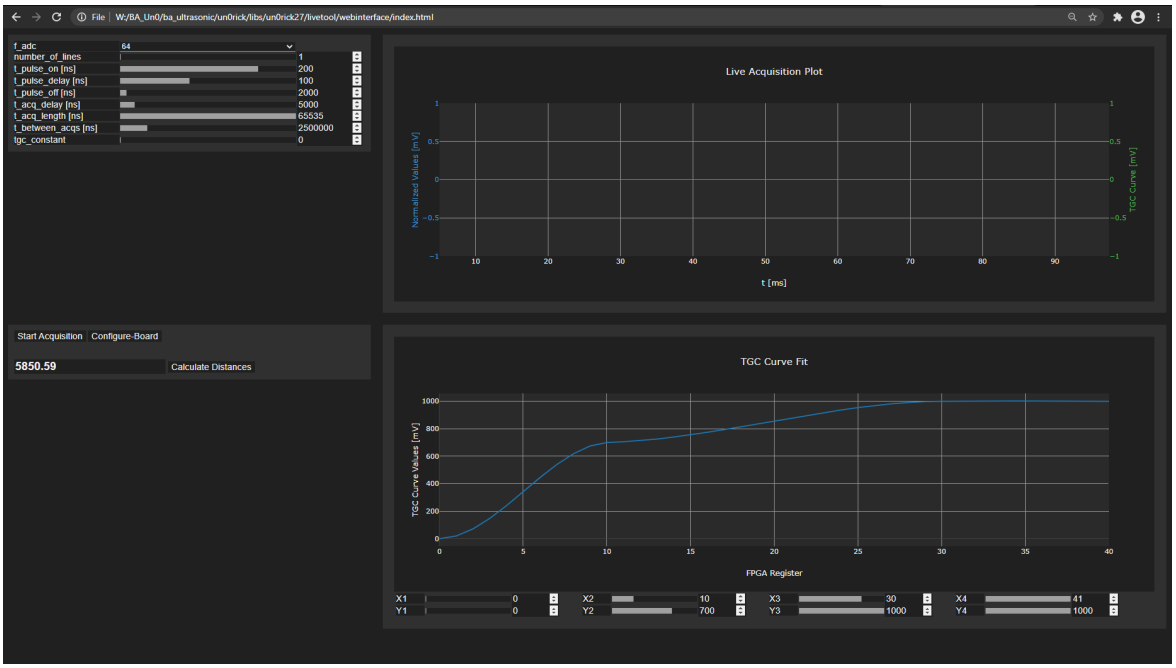


Figure 3.9: Custom web interface - GUI configuration with live updating A-scans

This server can be run on the Raspberry Pi while the web interface can be used

by a client in the network. The server was only implemented for the VHDL firmware version v1.1. When running this tool a single line acquisition is made and the measured data is shown as an A-scan on the website. This is done continuously in a loop with about 1 FPS. While this frame rate is quite low it still enables the possibility to move the ultrasonic probe on the test object and see the changes in the measurements immediately.

The web interface was not designed to show real time plots with a high frame rate, but rather to enable comfortable configuration of the board settings with a compact GUI. Therefore, it is assumable that higher frame rates are possible when according performance optimizations are made. Especially server side optimizations, when triggering the acquisitions and reading the data of the acquisition via SPI should have a significant impact on performance.

This tool was also made to show the versatility of the Un0rick board and its SPI interface. Instead of a website it is also imaginable to create APIs to other software, i.e. *matlab*, *simulink*, etc.

3.1.5.4 Post-Processing - Baseline Correction

While making different measurements, with different acquisition settings, all of the A-scans had a non constant baseline. The baseline would start at a value greater then 0 and converge against 0 over time, similar to the form of an exponential function like e^{-x}. Usually A-scans have a constant baseline at 0 (x-axis) and the falling baselines of the measurements are probably caused by an impedance mismatch between the ultrasonic probe and the board. While an impedance matching itself was not performed yet, it was possible to correct the baselines simply by applying a digital high pass filter in post processing, because the baseline shift itself is in the low frequency range.

```
import scipy.signal as signal

def apply_baseline_correction(y):
        MHz = 10**6
        sos = signal.butter(10, 2*MHz, fs=64*MHz, btype='highpass',
                            analog=False, output='sos')
        return signal.sosfilt(sos,y)
```

This high pass filter was implemented in python with the help of the *SciPy signal processing* library. For the filter a Butterworth filter of order 10 with a cutoff frequency

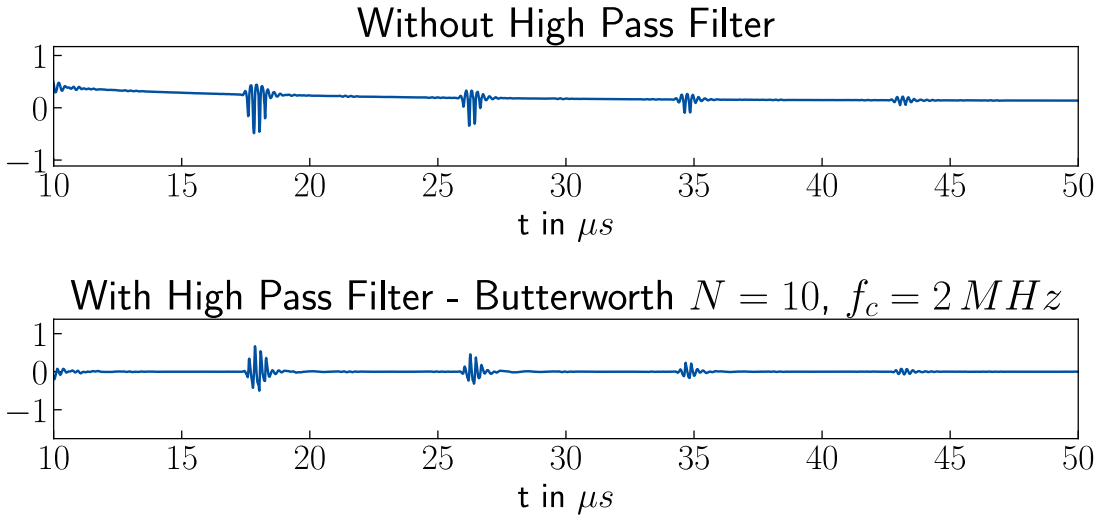


Figure 3.10: Baseline Correction in Post-processing

at $2\,MHz$ was used because it blocks unnecessary lower frequencies while being flat in the passband. This showed good results as Figure 3.10 illustrates.

3.1.6 Test Measurements

In order to test if the board and software was working correctly test measurements where made during this thesis. Those measurements also provide insights about the accuracy of the board and can maybe serve as reference of future measurements.

3.1.6.1 Specimen and Setup

For the measurements the Un0rick board was connected to a Raspberry Pi 3 with a ribbon cable at the dedicated connectors. As ultrasonic transducer and receiver the `Krautkrämer MB4S-N 53465 (4 MHz 10 ⌀)` straight beam probe was used and the highest available voltage source of the Un0rick board ($72\,V$) was selected on `J22`. The probe was connected with a coaxial cable, which was soldered directly onto the connector pads (with no SMA connector). The Raspberry Pi was connected to a PC with an Ethernet cable, which enabled control of the Raspberry Pi via ssh. The positioning of the ultrasonic transducer was supervised with the use of the web interface described in Section 3.1.5.3. The measurements themselves were performed with the use of a python Jupyter notebook. Water was used as ultrasonic couplant between the transducer and the specimen. The complete setup is illustrated in Figure 3.11.

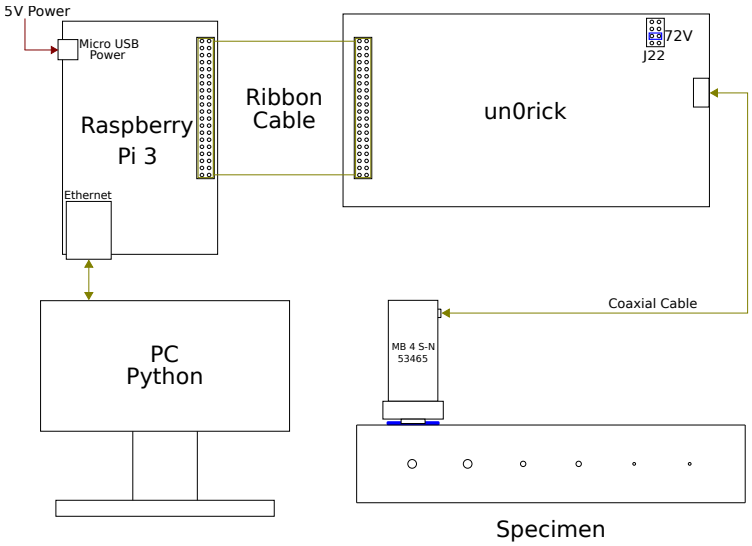


Figure 3.11: Setup for measurements

The specimen used has six drill holes with different depths and diameters, as described in Figure 3.13. The specimen was measured in two different orientations, from the top and from the front, which changes the surface of the measured defect from aa orthogonal plane to a rounded surface. In each orientation, the first measurement position has no drill hole. This enables measuring of the backwall echo without any defects. After this, each drill hole is measured, beginning from the utmost hole with biggest diameter ($4.7\,mm$), to the utmost hole with the smallest diameter ($1.5\,mm$). The two different orientations with their corresponding seven positions are shown in Figure 3.12.

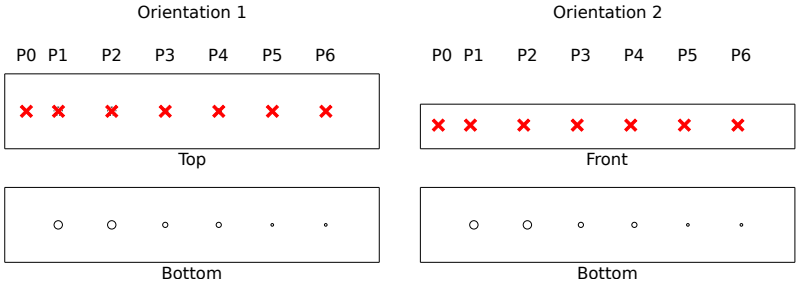


Figure 3.12: Transducer positioning during the measurements

In each position three individual single take pulse echo measurements were taken. Each acquisition had the same settings, which were similar to the default settings of

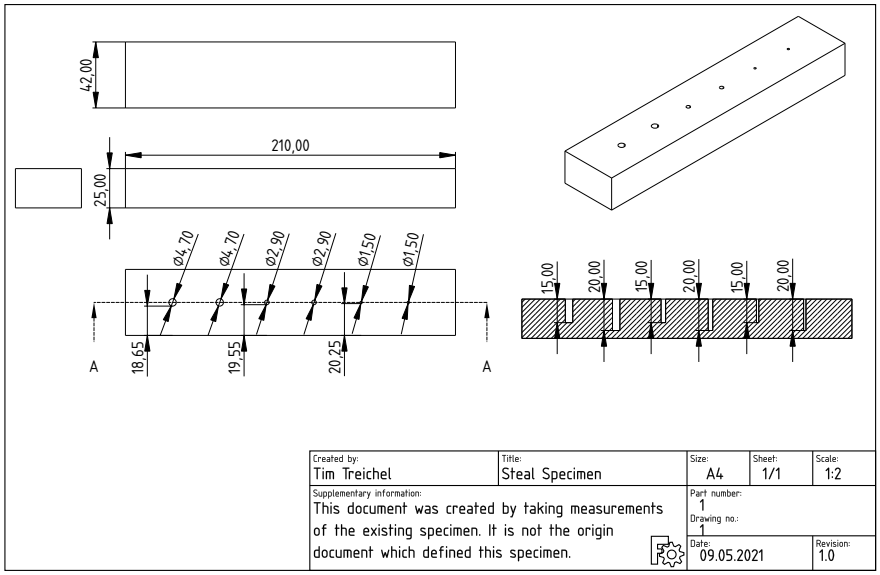


Figure 3.13: Specimen used during the measurements

the Un0rick python library for single take acquisitions. The registers of the FPGA were configured according to following python code:

```
from Un0rick27 import board as ub
from Un0rick27 import acquisition
from Un0rick27 import measurement

len_tgc = len(ub.FPGARegister.TGC_CURVE_REGS) # number of TGC curve registers
acq_data = acquisition.AcquisitionData()
acq_data.settings.number_of_lines = 1       # single line acquisition
acq_data.settings.f_adc = 64                # sample frequency 64 MHz
acq_data.settings.t_pulse_on = 200          # sEEPon       <- 200  ns
acq_data.settings.t_pulse_delay = 100       # SEEPonPoff   <- 100  ns
acq_data.settings.t_pulse_off = 2000        # SEEPoff      <- 2000 ns
acq_data.settings.t_acq_delay = 5000        # sEEDelayACQ  <- 5000 ns
acq_data.settings.t_acq_length = 190000     # sEEACQ       <- 190  us
acq_data.settings.t_between_acqs = 2500000 # sEEPeriod    <- 2.5  ms

# TGC Registers are set linear with an increase of 10
acq_data.settings.tgc_curve = [10*x for x in range(len_tgc)]
```

3.1.6.2 Evaluation of Measurements

All measurements are included in the Appendix A. Overall all measurements contain the expected echos. However, there are definitely noticeable differences in the signal to noise ratio between different positions. All measurements at position 0 (independent of the orientation) have a good signal to noise ratio and the echos are very good distinguishable from the rest of the signal. An example of this can be seen in Figure 3.14.

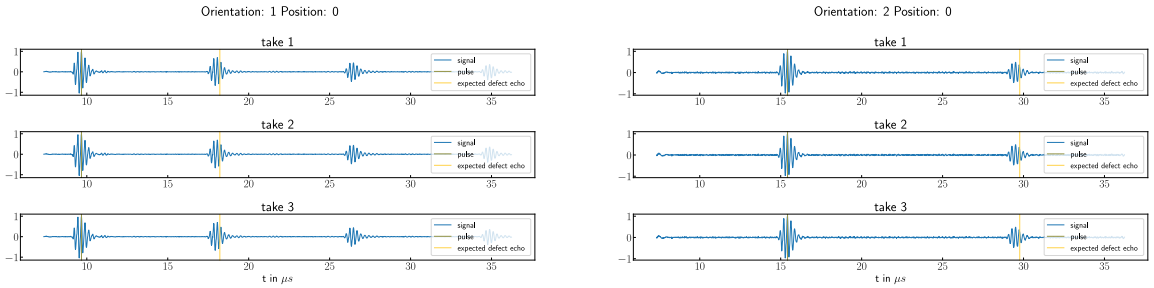


Figure 3.14: Well distinguishable echos, when measuring the backwall without defects

However, when measuring at other positions (with defects) some echos are very distinguishable while others are very hard to notice. This issue is visible in Figure 3.15.

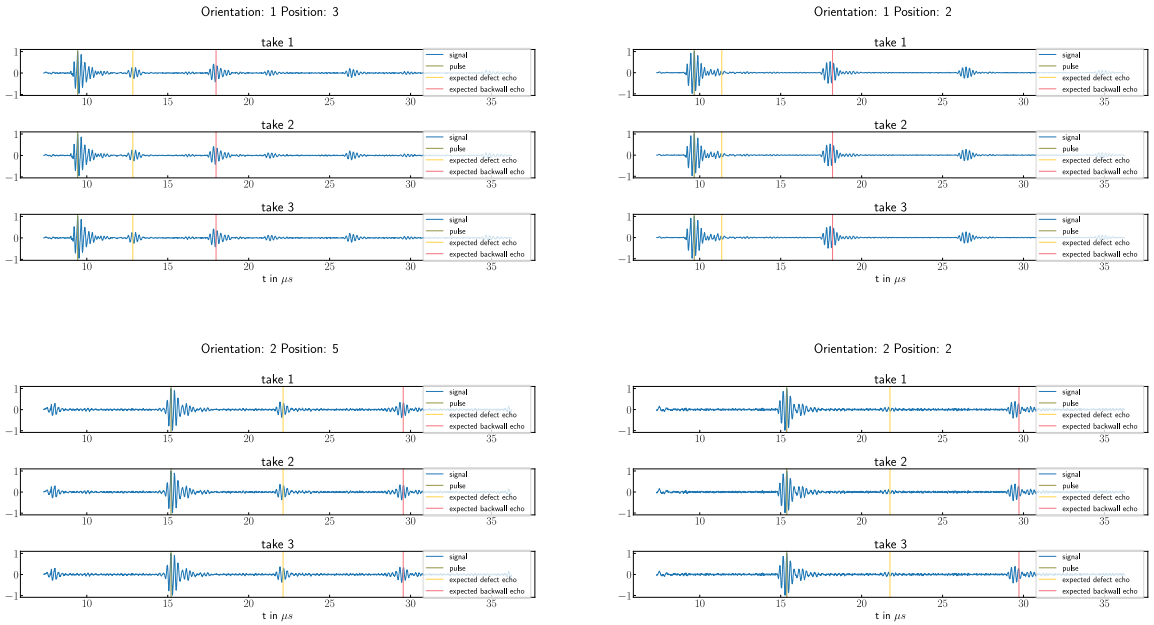


Figure 3.15: Clearly distinguishable echos on the left side, difficult distinguishable echos on right side

The measurements shown on the rights side contain their respective defect echos at their expected positions, however the amplitude of those echos is so low, that it would

be very hard to notice them. This issue would be especially concerning in applications where no other acquisitions for comparison exist, or the position of the defects are unknown.

Furthermore, it is interesting to notice, that all acquisitions which were taken in the second orientation of the specimen, show higher noise levels then acquisitions taken in first orientation as shown in Figure 3.16. This is likely due to all the different reflections that occur at the rounded surface of the drill holes in orientation two.

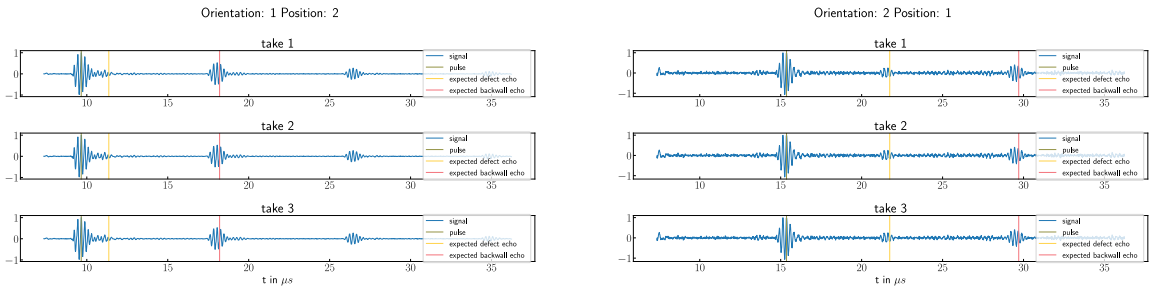


Figure 3.16: Comparison of noise levels between different orientations

The only acquisition with unsuspected results is the measurement at orientation 1, position 1 which is shown in Figure 3.17.

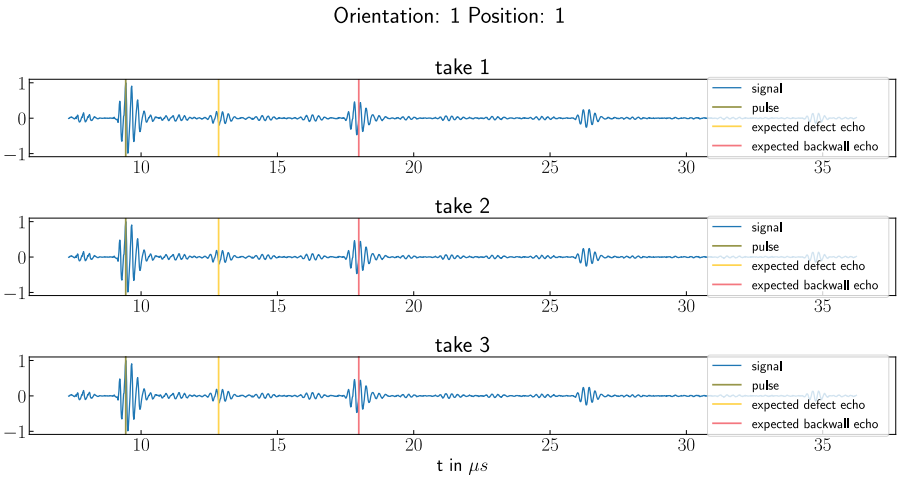


Figure 3.17: Unexplainable defect echos

They seem to indicate multiple defect echos, although only one defect should be visible. The cause for those results are unknown, however it is possible that the root lies in the specimen. Concluding, most measurements show expected results, however in some cases the defect echos where difficult to identify due to bad signal to noise ratio. The signal-to-noise ratio (SNR) could probably be improved by performing an

electrical impedance matching between the transducer and the board or by using a different transducer entirely [Rat19].

4 Experimental Firmware Implementation - Coded Signals

The Un0rick board, due to its design and the open source documentation, is very flexible when it comes to software. Furthermore, not only the software is open source but also the firmware is open, which enables the possibilities for more low level changes.

In order to implement advanced ultrasonic testing methods, such as for example ultrasound imaging or coded signals, changes to the firmware can be applied.

In the frame of this bachelor thesis the possibility to transmit coded signals in the form of Barker codes, was implemented in the firmware of the Un0rick board.

4.1 Approach 1 - Custom Pulse Generation with Multiple Windows

The pulse generation of the official firmware is described in detail in Section 3.1.2. This implementation is very simple and does only allow to configure the timings of the `Pon` and `Poff` pins. However, the sequence of values they have is fixed. This configuration only allows to fire one pulse during each acquisition. The only way to send multiple pulses is with a multi-line acquisition, which is a repetition of a single acquisition in a configured period.

In order to have more control over the pulse generation a custom firmware, based on the source code of the official firmware was developed. The idea of the new pulse generation is to have multiple windows of time in which the `Pon` and `Poff` pins, as well as the window length (time) can be configured individually. In order to achieve the same signal generation as in the official firmware 4 timing windows t_1, t_2, t_3, t_4 are needed, where the sequence of pin states P_{t_i} =(`Pon`, `Poff`) is $P_{t_1} = (1, 1)$, $P_{t_2} = (0, 1)$, $P_{t_3} = (0, 0)$ and $P_{t_4} = (0, 1)$.

If one implements 8 such timing windows with their representing pin states one could send two pulses right after each other and even more pulses when implementing more windows. In order to decode all possible combinations of `Pon` and `Poff` 2 bits are

necessary. The state combination $P_{t_i} = (1, 0)$ should never occur because it shortens the high voltage source to ground.

The timing windows were implemented as 8 bit counter registers. The pin state combinations were implemented as 2 bit values, which are grouped in 8 bit registers. Therefore, all the pin state combinations for the first 4 timing windows would be stored in the first 8 bit register, where the highest two bits represent the pin state combination for the first timing window and the lowest two bits represent the pin state combination for the fourth timing window.

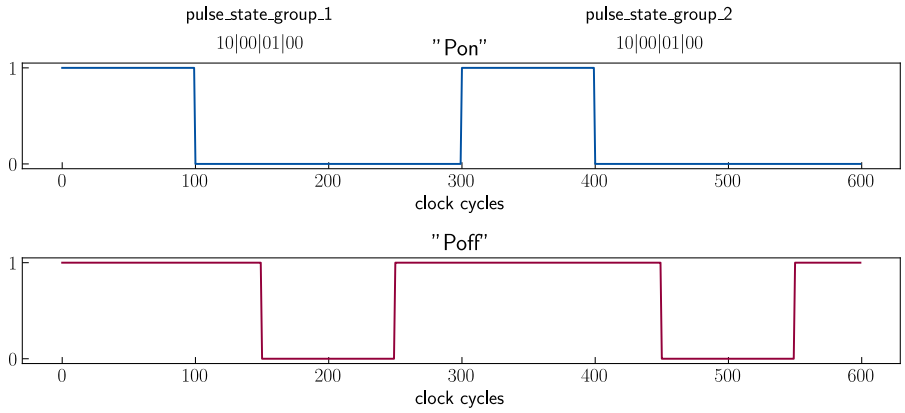
The registers for the timing windows were called `pulse_count_X` where `X` is the number of the timing window and the pin state combination for each `pulse_count_X` was called `pulse_state_X`. The assignment of a `pulse_state_X` to the real pin values was implemented as shown in Table 4.1.

Pulse State (Dec)	Pulse State (Bin)	`Pon`	`Pon`	Description
0	00	0	1	nothing
1	01	0	0	pull voltage down
2	10	1	1	pull voltage up
3	11	0	1	nothing

Table 4.1: Assignment of pulse state values to actual pin states

A grouping of 4 `pulse_state_X` values was called `pulse_state_group_Y`, where `Y` $= 1$ includes all the pulse states for `X` $= \{1, 2, 3, 4\}$. A simple example for 8 timing windows is visualized in Figure 4.1. It shows two identical pulses, one after the other.

This implementation, while it seems to be a very flexible option, has a fatal flaw. In order to transmit a Barker code sequence, multiple pulses and delays have to be transmitted. Assuming that 4 timing windows are needed to shape 1 pulse and 1 timing window is needed to transmit 1 delay, the total amount of windows needed increases rapidly with the length of the Barker code sequence. For example, in order to transmit a Barker code of length 5, $17 = 4 \cdot 4 + 1$ timing windows and 5 corresponding pulse state groups are needed. This means that the number of used FPGA cells increases with the length of the Barker code. This causes lower clock speeds and eventually the clock speeds are so low that the firmware can not perform its task. When trying to compile the source code with 17 timing windows and 5 pulse state groups the timing analysis reported that the clock frequency of the pulser PLL is below $80\,MHz$, which is way lower than the expected $128\,MHz$. Therefore, this implementation is not feasible and the second approach, which is described in detail in Section 4.2, was used instead.



pulse_count_X	1	2	3	4	5	6	7	8
value (clock cycles)	100	50	100	50	100	50	100	50

Figure 4.1: Custom signal generation for 8 timing windows

4.2 Approach 2 - Pulse Sequence with Final State Machine

Although the first approach failed, which is described in Section 4.1, it still offered insights to the problem. Combining those insights with some restrains on flexibility, the requirements and features of the new implementation can be derived:

1. It is crucial to minimize the cells used by the FPGA in order for the firmware to run properly.
2. Barker code sequences have different lengths and consist of pulses and delays.
3. The length of the code sequence, as well as the code sequence itself has to be configurable.
4. A single pulse has the same duration as a single delay. A single pulse transmits a 1 and a single delay transmits a 0 of the Barker code.
5. Pulses have to be configurable in shape and duration while delay only has to be configurable in duration.
6. Pulses and delays stay the same during the code sequence.

This led to the design of this new implementation. A configurable bit vector is representing the Barker code sequence in combination with a configurable length register. A final state machine iterates over the bits of the bit vector until it reaches

the configured length, which signals the end of the code sequence. When the value of the currently select bit is 1, a pulse is transmitted and when the value is 0, a delay is transmitted. Similar to the official firmware a pulse is a fixed sequence of 4 pin states P_{t_i} =(`Pon`, `Poff`) and the duration of each pin state can be configured with a designated register. The sequence of pin states is fixed and the same as in the official firmware with $P_{t_1} = (1, 1)$, $P_{t_2} = (0, 1)$, $P_{t_3} = (0, 0)$ and $P_{t_4} = (0, 1)$. Similar to this the delay has also a designated register, which configures its duration. The pulse state for the delay is fixed with $P_z = (0, 1)$. During the transmission of the code the pulse and delay configuration does not change. The Barker code sequence always starts at the most significant bit of the bit vector. The iteration over the bits of the bit vector is achieved by left shifting the vector by 1 bit after each iteration. The final state machine is visualized in Figure 4.2.

4.2.1 Implementing the FSM into the Verilog Source Code

This custom firmware implementation is based on the Verilog source code [Jon20d]. Since this new FSM handles the whole signal generation parts of the official source code have to be removed or changed:

`verilog/src/rtl/acq.v`

Removed the registers which are no longer used:

```
27  //input  wire [PULSER_LEN_W-1:0]   pulser_on_len,
28  //input  wire [PULSER_LEN_W-1:0]   pulser_off_len,
29  //input  wire [PULSER_LEN_W-1:0]   pulser_init_len,
30  //input  wire [PULSER_LEN_W-1:0]   pulser_inter_len,
31  //input  wire                      pulser_drmode,
```

Changed pulser_init_len_drmode since double rate mode is not used:

```
139 //assign pulser_init_len_drmode = (pulser_drmode && line_even) ?
 ↪  pulser_init_len_even : pulser_init_len_odd;
140 assign pulser_init_len_drmode = pulser_init_len_odd;
```

Removed everything from line 141 to 201 (Aquisition FSM remains)

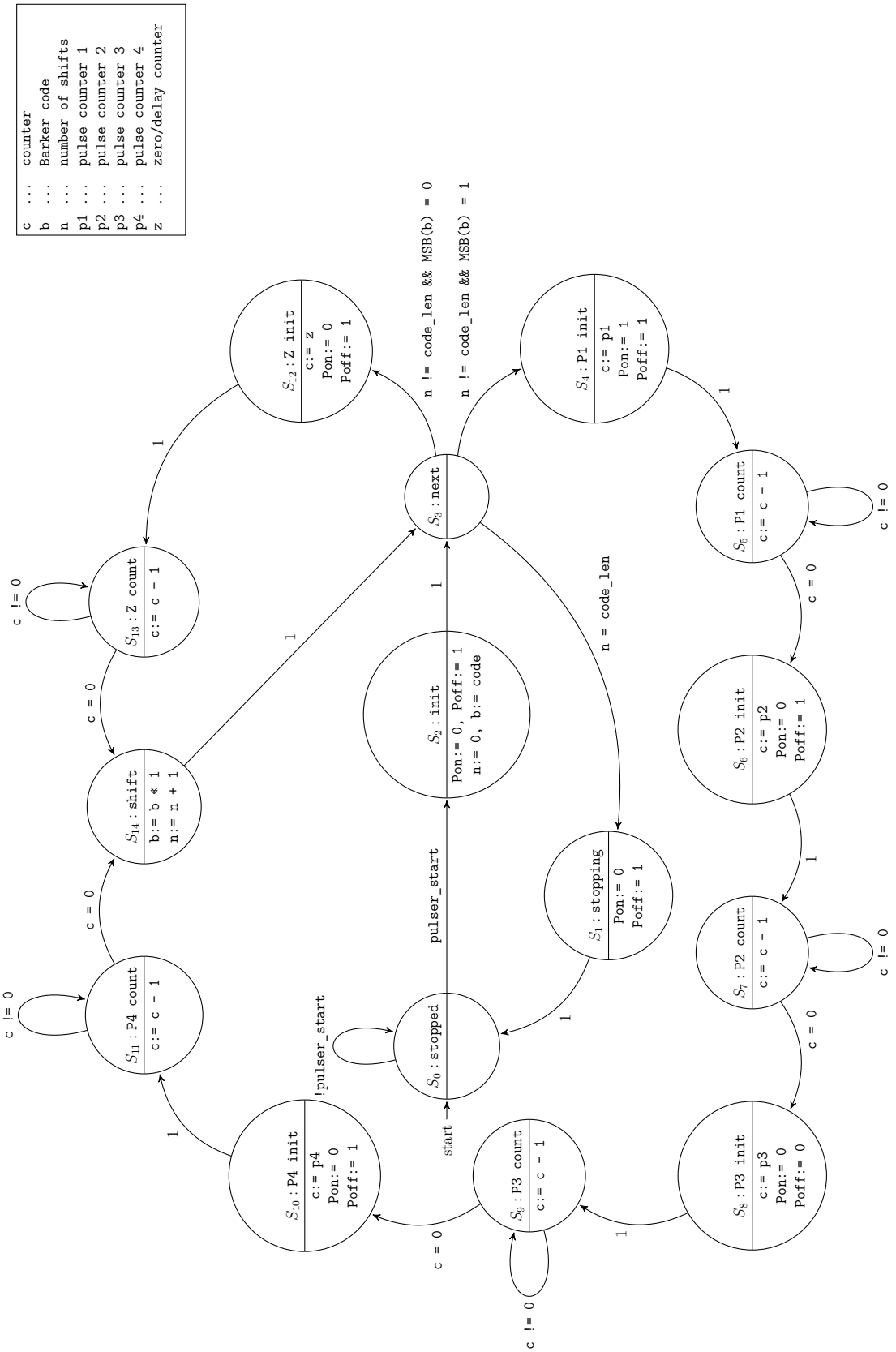


Figure 4.2: Finite-state machine that implements the pulser for the coded signal implementation

After removing the old pulse generation the new pulser FSM can be implemented:

verilog/src/rtl/acq.v

```
30  //input  wire [PULSER_LEN_W-1:0]   pulser_inter_len,
31  //input  wire                      pulser_drmode,
```

New control status registers (inputs of module acq):

```
32  input wire [7:0]  pulse1,    // pulse state 1 duration
33  input wire [7:0]  pulse2,    // pulse state 2 duration
34  input wire [7:0]  pulse3,    // pulse state 3 duration
35  input wire [7:0]  pulse4,    // pulse state 4 duration
36  input wire [7:0]  zero,      // delay duration
37  input wire [7:0]  code_len,  // Barker code
38  input wire [15:0] code_wire, // Barker code sequence
39  // DAC
...
142 assign pulser_init_len_drmode = pulser_init_len_odd;
```

Defining the FSM states:

```
144 localparam PULSER_FSM_STOPPED      = 0;
145 localparam PULSER_FSM_STOPPING     = 1;
146 localparam PULSER_FSM_INIT         = 2;
147 localparam PULSER_FSM_NEXT         = 3;
148 localparam PULSER_FSM_PULSE1_INIT  = 4;
149 localparam PULSER_FSM_PULSE1_COUNT = 5;
150 localparam PULSER_FSM_PULSE2_INIT  = 6;
151 localparam PULSER_FSM_PULSE2_COUNT = 7;
152 localparam PULSER_FSM_PULSE3_INIT  = 8;
153 localparam PULSER_FSM_PULSE3_COUNT = 9;
154 localparam PULSER_FSM_PULSE4_INIT  = 10;
155 localparam PULSER_FSM_PULSE4_COUNT = 11;
156 localparam PULSER_FSM_ZERO_INIT    = 12;
157 localparam PULSER_FSM_ZERO_COUNT   = 13;
158 localparam PULSER_FSM_SHIFT_CODE   = 14;
```

```
Defining FSM registers:
160 reg[4:0]  pulser_fsm_state;
161 reg[4:0]  pulser_fsm_state_next;
162 reg       pulser_fsm_last_state;
163 reg[15:0] code;
164 reg       did_shift;
165 reg[7:0]  num_shifts;

FSM state update:
167 always @(posedge pulser_clk or posedge rst) begin
168     if(rst)
169         pulser_fsm_state <= PULSER_FSM_STOPPED;
170     else
171         pulser_fsm_state <= pulser_fsm_state_next;
172 end

FSM state behavior:
174 always @(posedge pulser_clk) begin
175     pulser_fsm_state_next = pulser_fsm_state;
176
177     case (pulser_fsm_state)
178         PULSER_FSM_STOPPED: begin
179             if(pulser_start)
180                 pulser_fsm_state_next <= PULSER_FSM_INIT;
181             else
182                 pulser_fsm_state_next <= PULSER_FSM_STOPPED;
183         end
184
185         PULSER_FSM_INIT: begin
186             code <= code_wire;
187             pulser_cnt  <= 8'h00;
188             did_shift <= 1'b0;
189             num_shifts <= 8'h00;
190
191             pulser_fsm_state_next <= PULSER_FSM_NEXT;
192             pulser_fsm_last_state <= 1'b0;
```

```
193             pulser_busy  <= 1'b1;
194             pulser_on  <= 1'b0;
195             pulser_off <= 1'b1;
196         end
197
198         PULSER_FSM_NEXT: begin
199             if(num_shifts == code_len)
200                 pulser_fsm_state_next <= PULSER_FSM_STOPPING;
201             else if(code[15])
202                 pulser_fsm_state_next <= PULSER_FSM_PULSE1_INIT;
203             else
204                 pulser_fsm_state_next <= PULSER_FSM_ZERO_INIT;
205
206             did_shift <= 1'b0;
207         end
208
209         PULSER_FSM_STOPPING: begin
210             pulser_busy <= 1'b0;
211             pulser_on  <= 1'b0;
212             pulser_off <= 1'b1;
213             pulser_fsm_state_next <= PULSER_FSM_STOPPED;
214         end
215
216         // PULSE 1
217         PULSER_FSM_PULSE1_INIT: begin
218             pulser_cnt    <= pulse1;
219             pulser_on  <= 1'b1;
220             pulser_off <= 1'b1;
221             pulser_fsm_state_next <= PULSER_FSM_PULSE1_COUNT;
222         end
223
224         PULSER_FSM_PULSE1_COUNT: begin
225             if(!pulser_cnt)
226                 pulser_fsm_state_next <= PULSER_FSM_PULSE2_INIT;
227             else begin
228                 pulser_fsm_state_next <= PULSER_FSM_PULSE1_COUNT;
```

```
229                 pulser_cnt <= pulser_cnt - 1;
230             end
231         end
232
233         // PULSE 2
234         PULSER_FSM_PULSE2_INIT: begin
235             pulser_cnt    <= pulse2;
236             pulser_on  <= 1'b0;
237             pulser_off <= 1'b1;
238             pulser_fsm_state_next <= PULSER_FSM_PULSE2_COUNT;
239         end
240
241         PULSER_FSM_PULSE2_COUNT: begin
242             if(!pulser_cnt)
243                 pulser_fsm_state_next <= PULSER_FSM_PULSE3_INIT;
244             else begin
245                 pulser_fsm_state_next <= PULSER_FSM_PULSE2_COUNT;
246                 pulser_cnt <= pulser_cnt - 1;
247             end
248         end
249
250         // PULSE 3
251         PULSER_FSM_PULSE3_INIT: begin
252             pulser_cnt    <= pulse3;
253             pulser_on  <= 1'b0;
254             pulser_off <= 1'b0;
255             pulser_fsm_state_next <= PULSER_FSM_PULSE3_COUNT;
256         end
257
258         PULSER_FSM_PULSE3_COUNT: begin
259             if(!pulser_cnt)
260                 pulser_fsm_state_next <= PULSER_FSM_PULSE4_INIT;
261             else begin
262                 pulser_fsm_state_next <= PULSER_FSM_PULSE3_COUNT;
263                 pulser_cnt <= pulser_cnt - 1;
264             end
```

```
265         end
266
267         // PULSE 4
268         PULSER_FSM_PULSE4_INIT: begin
269             pulser_cnt    <= pulse4;
270             pulser_on  <= 1'b0;
271             pulser_off <= 1'b1;
272             pulser_fsm_state_next <= PULSER_FSM_PULSE4_COUNT;
273         end
274
275         PULSER_FSM_PULSE4_COUNT: begin
276             if(!pulser_cnt)
277                 pulser_fsm_state_next <= PULSER_FSM_SHIFT_CODE;
278             else begin
279                 pulser_fsm_state_next <= PULSER_FSM_PULSE4_COUNT;
280                 pulser_cnt <= pulser_cnt - 1;
281             end
282         end
283
284         // ZERO
285         PULSER_FSM_ZERO_INIT: begin
286             pulser_cnt <= zero;
287             pulser_on  <= 1'b0;
288             pulser_off <= 1'b1;
289             pulser_fsm_state_next <= PULSER_FSM_ZERO_COUNT;
290         end
291
292         PULSER_FSM_ZERO_COUNT: begin
293             if(!pulser_cnt)
294                 pulser_fsm_state_next <= PULSER_FSM_SHIFT_CODE;
295             else begin
296                 pulser_fsm_state_next <= PULSER_FSM_ZERO_COUNT;
297                 pulser_cnt <= pulser_cnt - 1;
298             end
299         end
300
```

```
301         PULSER_FSM_SHIFT_CODE: begin
302             pulser_fsm_state_next <= PULSER_FSM_NEXT;
303             if(!did_shift) begin
304                 code <= code << 1;
305                 did_shift <= 1'b1;
306                 num_shifts <= num_shifts + 1;
307             end
308         end
309     endcase
310 end
```

In order to implement the SPI interface for the new control status registers, the `verilog/util/csr_map/csr_map.yaml` file has to be updated and a new `verilog/src/rtl/csr_decoder.vh` file has to be generated by using the `util/csr_map/gen_csr.py` script. Furthermore, the input and output ports in `verilog/src/rtl/top.v` and `verilog/src/rtl/csr.v` have to be updated. Lastly the read and write operations can be implemented in `verilog/src/rtl/csr.v` by following the pattern of other control status registers.

4.2.2 Post Synthesis Simulation

In Figure 4.3 the results of post synthesis simulation of a barker code of length 2 is visualized. All control status registers which control a pulse or delay duration were set to 02_{H}. However, the actual number of clock cycles is higher than 2 because the FSM needs additional cycles to switch to the next state. Furthermore, the number of clock cycles of `pulse4` and `zero` increases because of the bit shifting. The actual timings are shown in Table 4.2. The post synthesis simulation was performed by compiling the source code with *Yosis*, simulating it with *Icarus Verilog* and viewing the results with *GTKWave*.

Register	Hex Value	Real Number of Clock Cycles	Time in ns
`pulse1`	02_{H}	$6 = 4 + 2$	$46.875 = 6 \cdot 7.8125$ ns
`pulse2`	02_{H}	$6 = 4 + 2$	$46.875 = 6 \cdot 7.8125$ ns
`pulse3`	02_{H}	$6 = 4 + 2$	$46.875 = 6 \cdot 7.8125$ ns
`pulse4`	02_{H}	$10 = 8 + 2$	$78.125 = 10 \cdot 7.8125$ ns
`zero`	02_{H}	$10 = 8 + 2$	$78.125 = 10 \cdot 7.8125$ ns

Table 4.2: Correlation between register values and real duration values

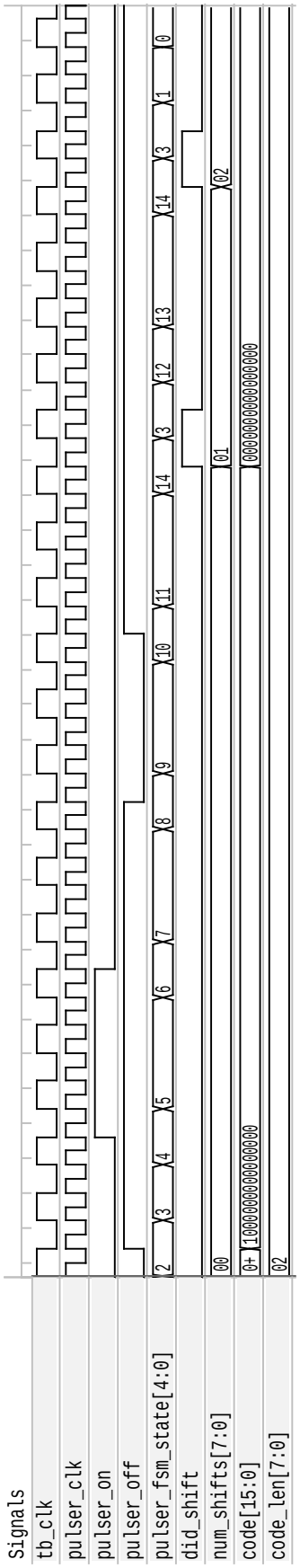


Figure 4.3: Post synthesis simulation of coded signals pulser FSM
`pulse1` = 02_{H}, `pulse2` = 02_{H}, `pulse3` = 02_{H}, `pulse4` = 02_{H}, `zero` = 02_{H},
`code_wire` = 1000 0000 0000 0000_{H}, `code_len` = 02_{H}

4.2.3 Coded Signal Measurements

The acquisitions where performed under similar conditions to those described in Section 3.1.6.1. As ultrasonic transducer and receiver the `Krautkrämer MB4S-N 53465 (4 MHz 10 ⌀)` straight beam probe was used and the highest available voltage source of the Un0rick board ($72\,V$) was selected on `J22`. However, this time the Un0rick board was directly controlled from a pc. This is possible because the firmware is based on the Verilog version, which enables SPI communication via USB. Figure 4.4 illustrates the experiment setup. The acquisitions were only performed in orientation 2 and position 0. The reason for this is, that this is the position with the largest depth. The transducer is only a simple pulse echo probe. If the wall depth is too short, the echos will arrive at the probe while it is still transmitting the signal, which will corrupt the received signal. In order to measure with the longest possible code lengths the deepest wall depth was needed. In order to measure such close distances with long code lengths, a TR probe is needed. The Un0rick board supports TR probes, but in order to use them the resistor `R72` has to be desoldered and the resistor `R73` has to be soldered to the board instead [Jon21c].

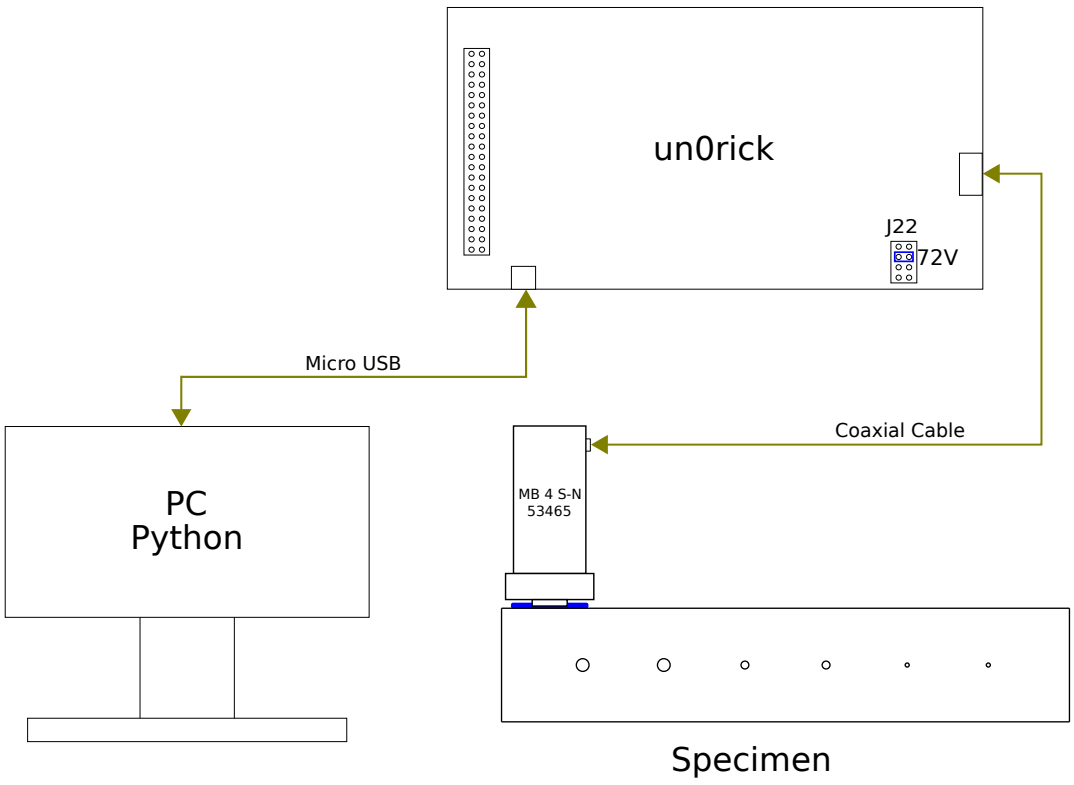


Figure 4.4: Coded signals acquisition setup

Following settings where used during the acquisition:

```
import un0usb as USB

fpga = USB.FpgaControl('ftdi://ftdi:2232:/', spi_freq=8E6)
fpga.reload() # reload configuration
fpga.reset() # reset fpga

code_len = 7 # or 5
fpga.csr.pulse1 = 10
fpga.csr.pulse2 = 5
fpga.csr.pulse3 = 200
fpga.csr.pulse4 = 2
fpga.csr.zero   = 190

if code_len == 5:
        codelen  = 0x5
        codewire = 0b1110_1000_0000_0000

if code_len == 7:
        codelen  = 0x07
        codewire = 0b1110_0100_0000_0000

offset = 1
fpga.csr.codelen  = codelen + offset      # delay start of acq
fpga.csr.codewire = (codewire >> offset)  # delay start of acq

data = custom_acquisition(fpga, gain=None)
make_plot(data, code_len)
fpga.disconnect()
```

Those settings where chosen manually so that the duration of a single pulse and a single delay are very similar. This was achieved by visually looking at a plot and manually adjusting the `zero` register value. It is also notable that the `code_len` and the `code_wire` register were both offset by 1 in order to delay the start of the transmission.

4.2.4 Post Processing

The post processing is very important, because the signal needs to be correlated with the barker code as explained in Section 2.2.4. In the first step the baseline correction is applied to the signal, as described in Section 3.1.5.4. Then the signal is multiplied with a rising linear function, which acts as a digital time gain compensation. After this the envelope of the signal is calculated by using the absolute of its Hilbert transformation:

```
import numpy as np
import scipy.signal as signal

data = apply_baseline_correction(data)

digital_tgc = 0.8*(np.linspace(0,1, len(data)) + 0.5)
data = [data[i] * digital_tgc[i] for i in range(len(data))]

envelope = np.abs(signal.hilbert(data))
```

The envelope then is correlated with the rectangular barker code sequence in order to calculate the decoded signal:

```
def gen_barker_code(rect_width, code_len):
        ones  = [1 for x in range(rect_width)]
        zeros = [0 for x in range(rect_width)]

        code = []

        if code_len == 5:
                code = ones*3 + zeros + ones
        if code_len == 7:
                code = ones*3 + zeros*2 + ones + zeros

        return code

rect_width = 109 # chosen manually
barker_code_rect = gen_barker_code(rect_width, 7)
corr = np.correlate(envelope, barker_code_rect, "same")
```

In Figures 4.5 and 4.6 the results of measuring under the same conditions with a code length of 5 and a code length of 7 are shown. In both cases the peaks are visible in the correlated signal, but in neither case did the signal-to-noise ration (SNR) improve compared to the uncorrelated signal. However, it is clearly visible, that the correlated signal of the code sequence with length 7 has a better SNR than the correlated signal of the code sequence with length 5.

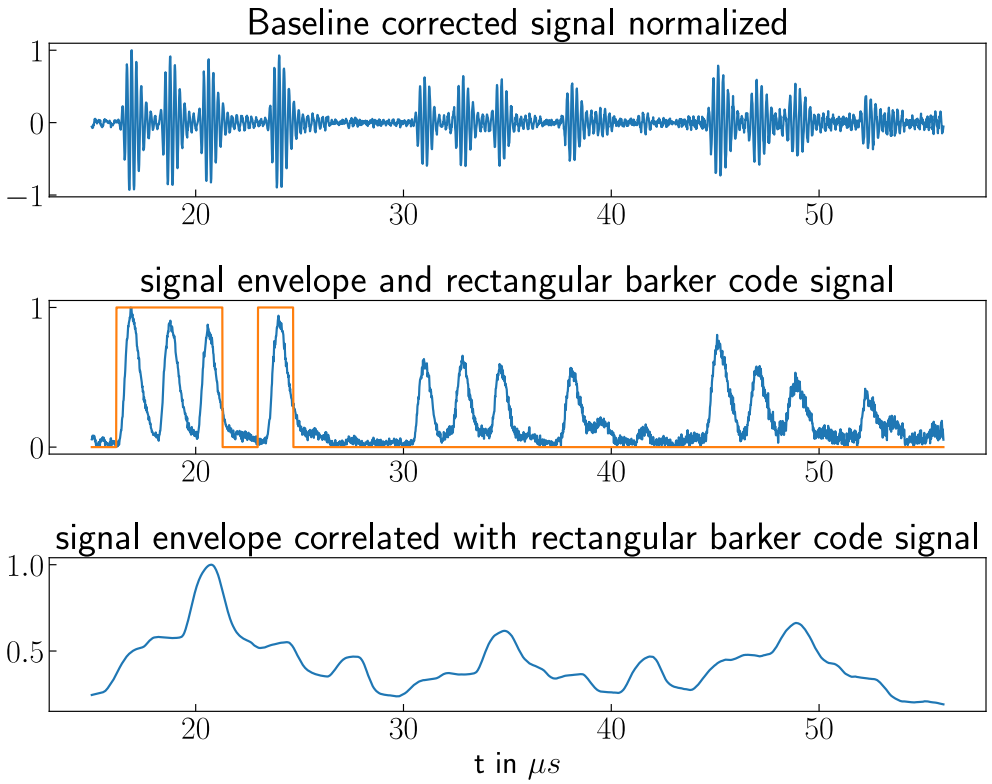


Figure 4.5: Coded signals acquisition with Barker code length 5

Considering this and the theory described in Section 2.2.4, it is assumable that the SNR can be improved when using longer code lengths. In both cases the peaks of the correlated signal are in the center of the code sequence. This is to be expected, because at those the barker code rectangles overlap exactly with the code sequence in the signal, which maximizes the correlation. The positions and time differences of those peaks accurately measure the depth of the specimen.

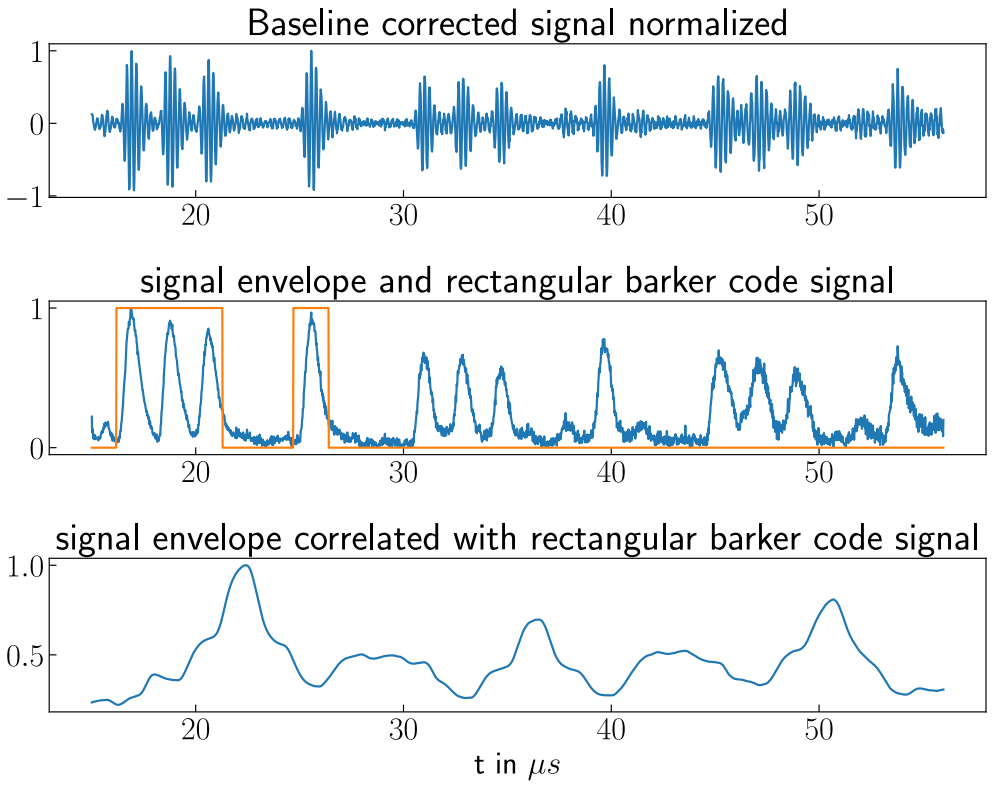


Figure 4.6: Coded signals acquisition with Barker code length 7

4.2.5 Issues

After implementing the SPI API for the new control status registers, the timing analysis tool reports that the required clock frequency of the pulser PLL is caped at around 123 MHz instead of 128 MHz. However, despite of this, the firmware still works on real hardware and acquisitions can be made. Packing the new control status registers into a single SPI read/write group can maybe fix this issue. Furthermore setting the durations of the pulses, the delays as well as the width of the Barker code rectangles manually is not ideal. It should be possible to fix this entirely in software by implementing a calibration routine.

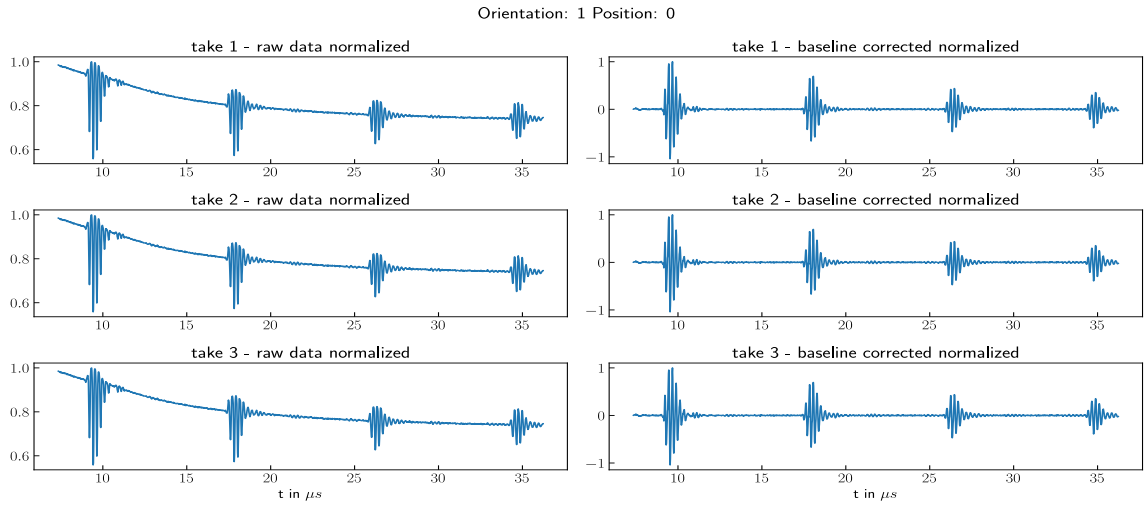
5 Conclusion and Outlook

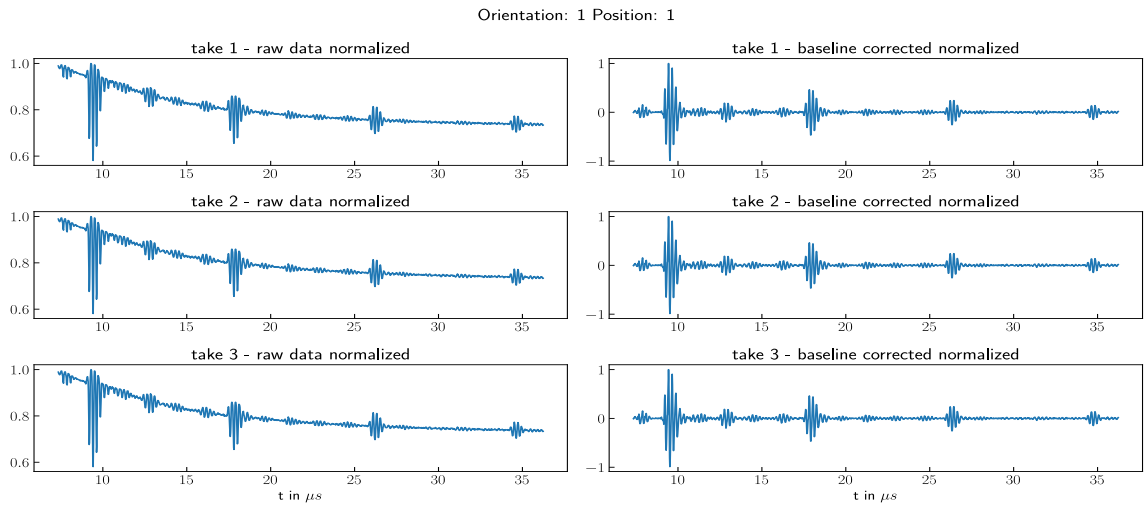
This research covered the fundamentals of ultrasound and ultrasonic testing, as well as different testing methods. Furthermore, this thesis intensely focused on the open source ultrasound project Un0rick. By analyzing the schematics of the hardware, as well as the source code of the firmware and software a low level understanding of the working mechanisms of this ultrasonic testing system was gained. Therefore, this thesis can serve as additional documentation for the Un0rick project, as it covers core mechanisms such as signal generation, anti aliasing, the SPI interface and the development of custom firmware/software. Additionally, measurements were made, in order to verify the accuracy of the board. In the last part of this thesis a custom firmware version was developed, which enables the possibility to send binary coded signals, e.g. Barker codes, in order to improve the SNR of the signal. Although the SNR did not improve, compared to normal transmissions, the implementation worked and can probably be improved in the future. Enabling the use of longer codes, by using an transmitter-receiver probe could potentially increase the SNR. Furthermore, an electrical impedance matching between the board and the transceiver could potentially improve the measurements. Another possibility would be to explore other binary codes then barker codes. Concluding this research, the Un0rick can offer a low cost ultrasonic system, mainly as a development or educational platform. The strongest feature of the Un0rick board is that the entire project (hardware, firmware and software) is open source, which enables great flexibility in its use cases.

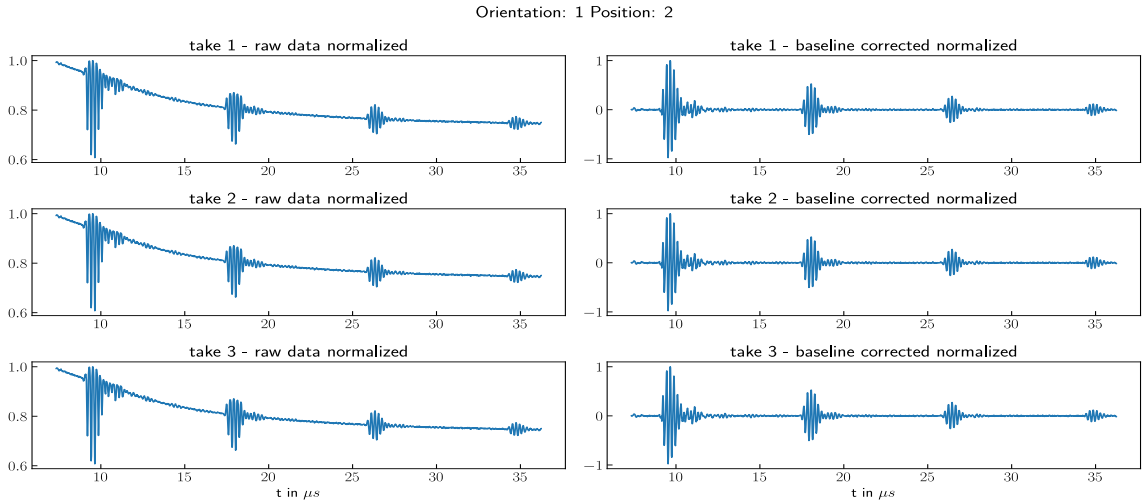
Appendix

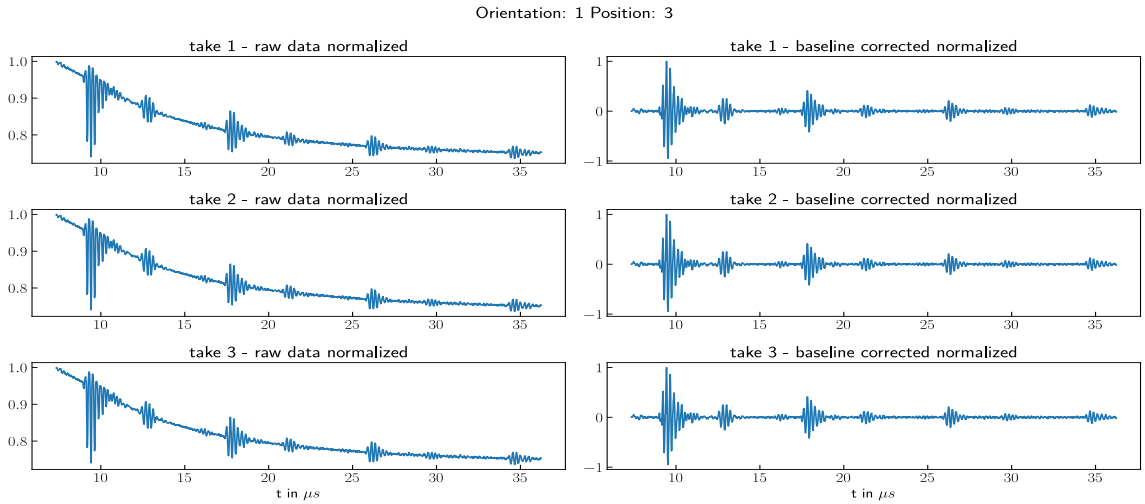
A Measurements

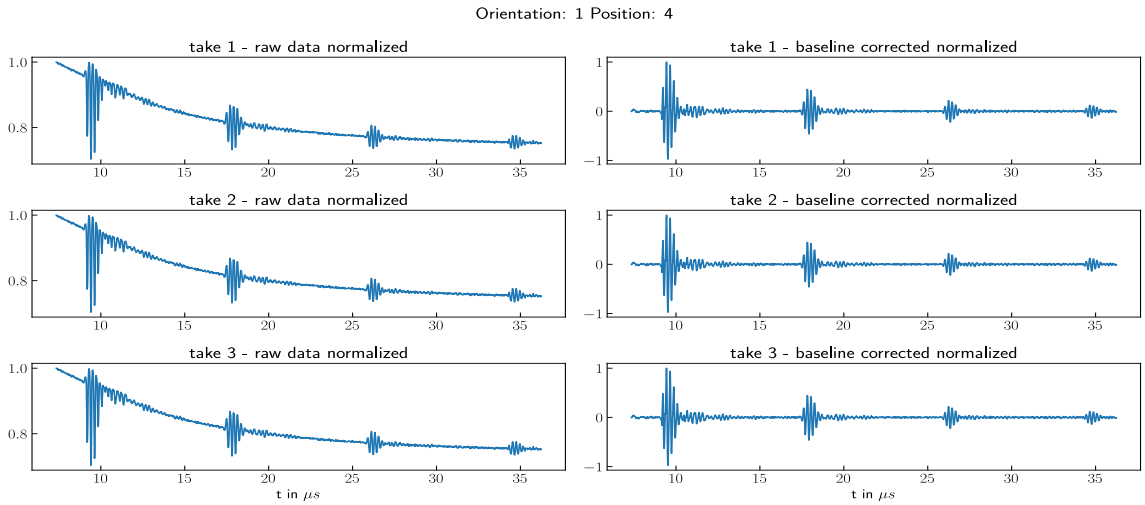
The following plots are the results of the measurements described in Section 3.1.6.2. The first plots show the raw data and the data after applying the baseline correction described in Section 3.1.5.4. The plots after this include marked timestamps where echos are expected.



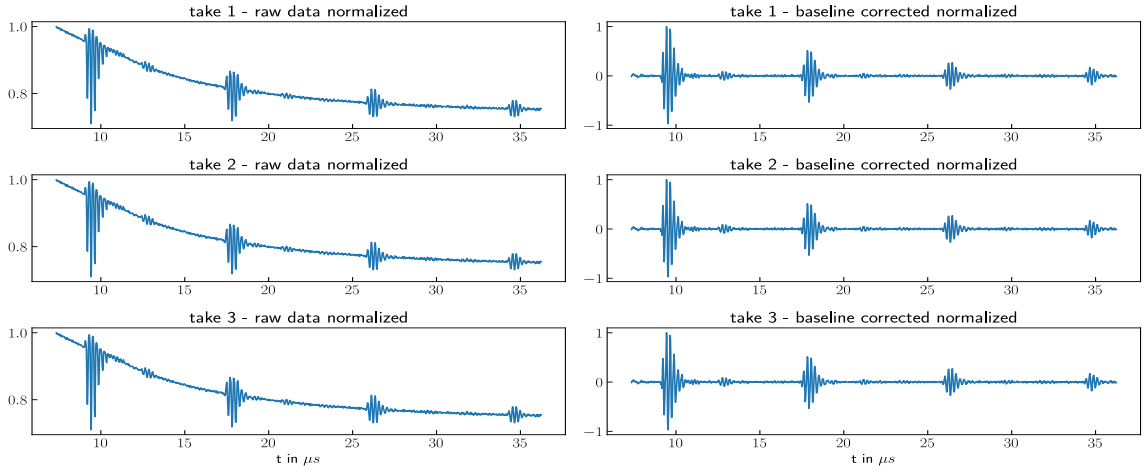



Orientation: 1 Position: 2
take 1 - raw data normalized
take 1 - baseline corrected normalized
take 2 - raw data normalized
take 2 - baseline corrected normalized
take 3 - raw data normalized
take 3 - baseline corrected normalized
t in μs

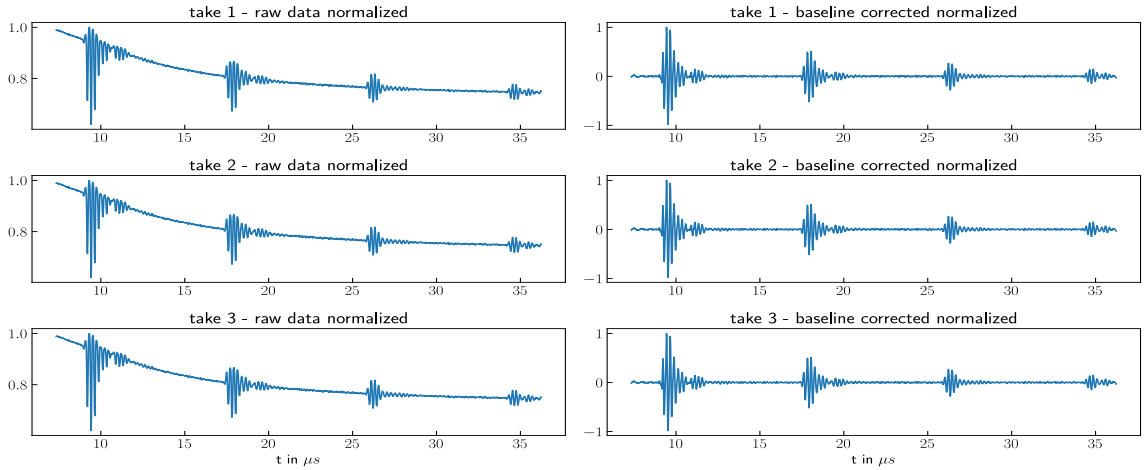

Orientation: 1 Position: 3
take 1 - raw data normalized
take 1 - baseline corrected normalized
take 2 - raw data normalized
take 2 - baseline corrected normalized
take 3 - raw data normalized
take 3 - baseline corrected normalized
t in μs


Orientation: 1 Position: 4
take 1 - raw data normalized
take 1 - baseline corrected normalized
take 2 - raw data normalized
take 2 - baseline corrected normalized
take 3 - raw data normalized
take 3 - baseline corrected normalized
t in μs

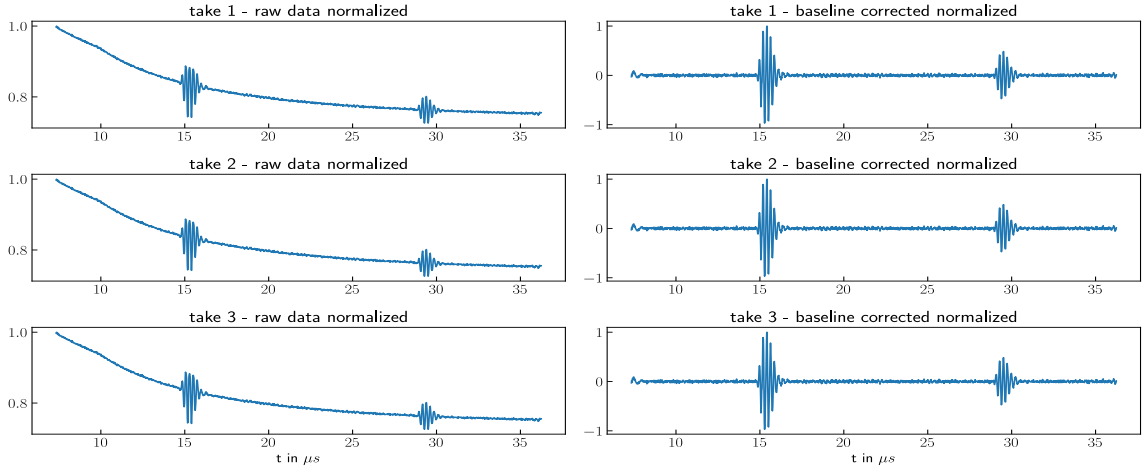
Orientation: 1 Position: 5



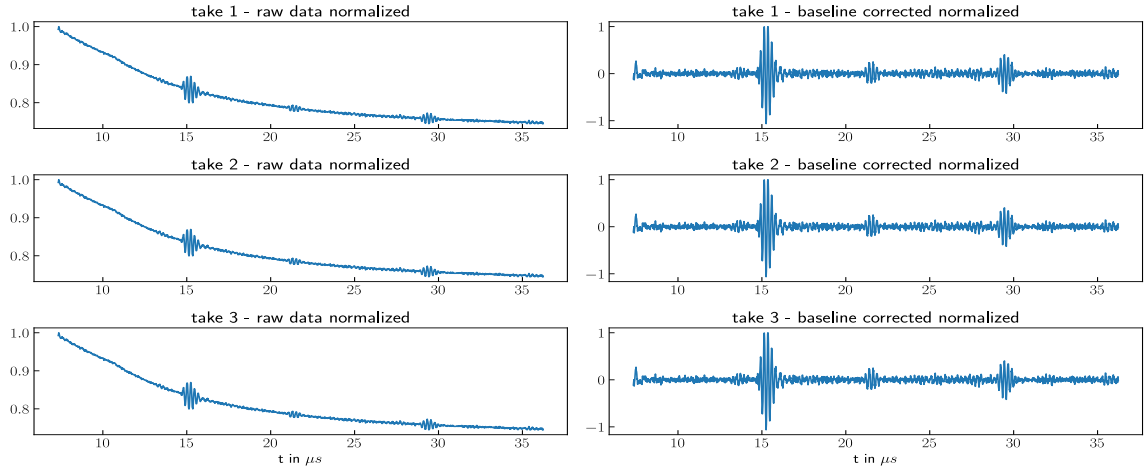
Orientation: 1 Position: 6



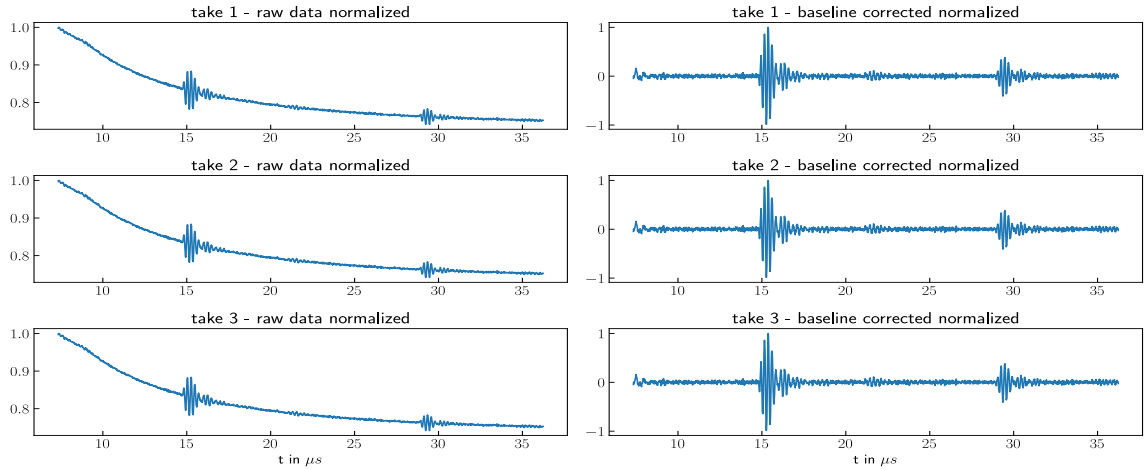
Orientation: 2 Position: 0



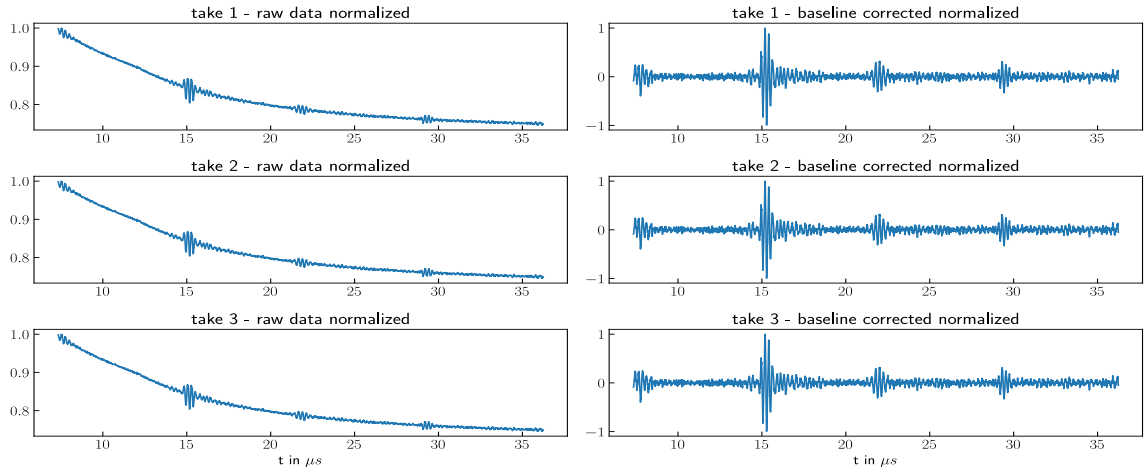
Orientation: 2 Position: 1



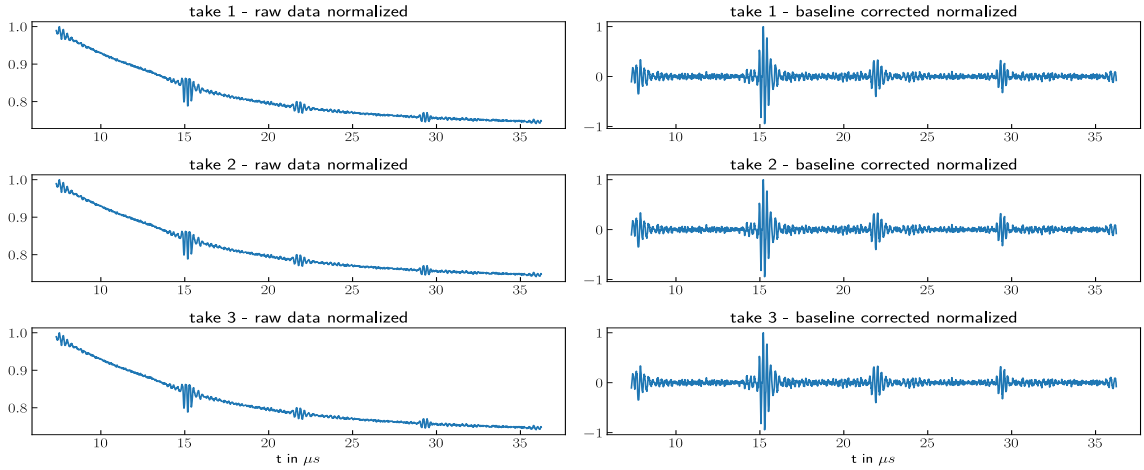
Orientation: 2 Position: 2



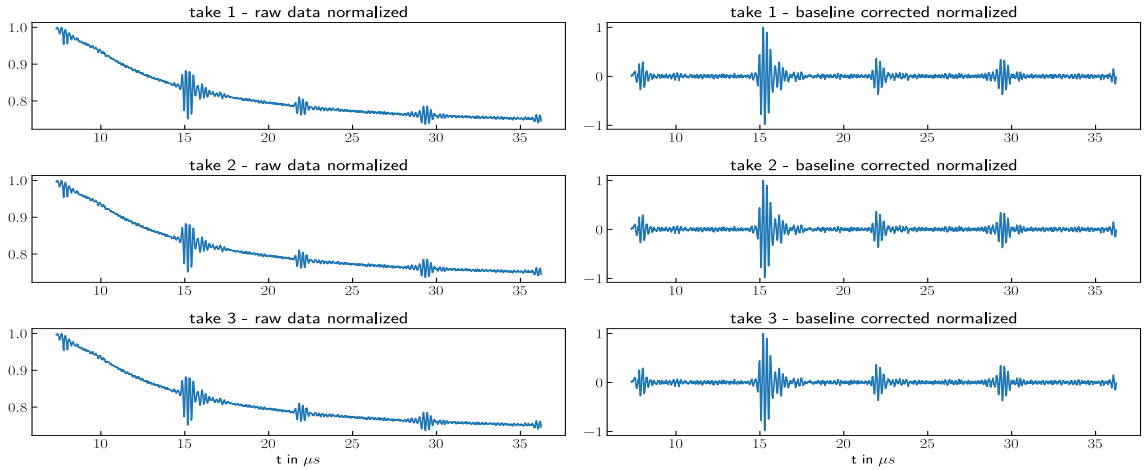
Orientation: 2 Position: 3



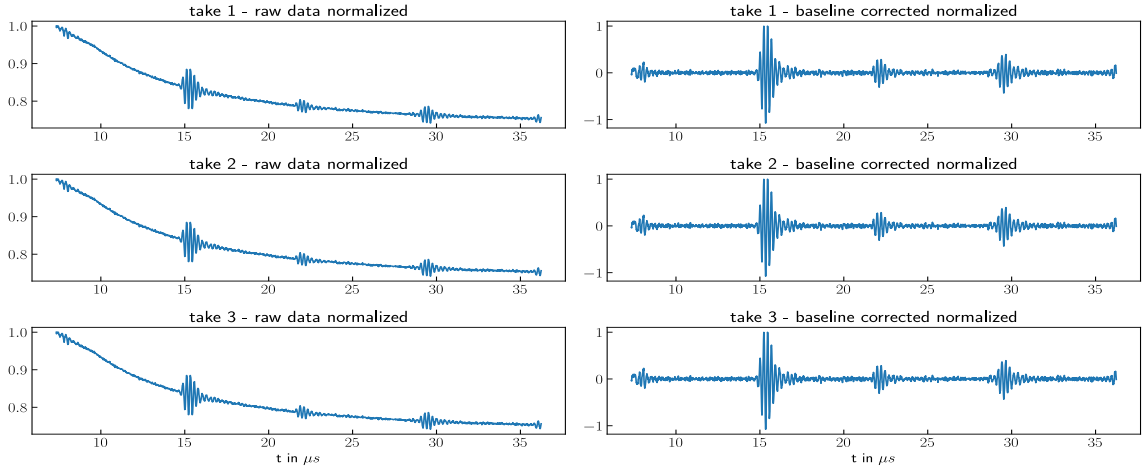
Orientation: 2 Position: 4

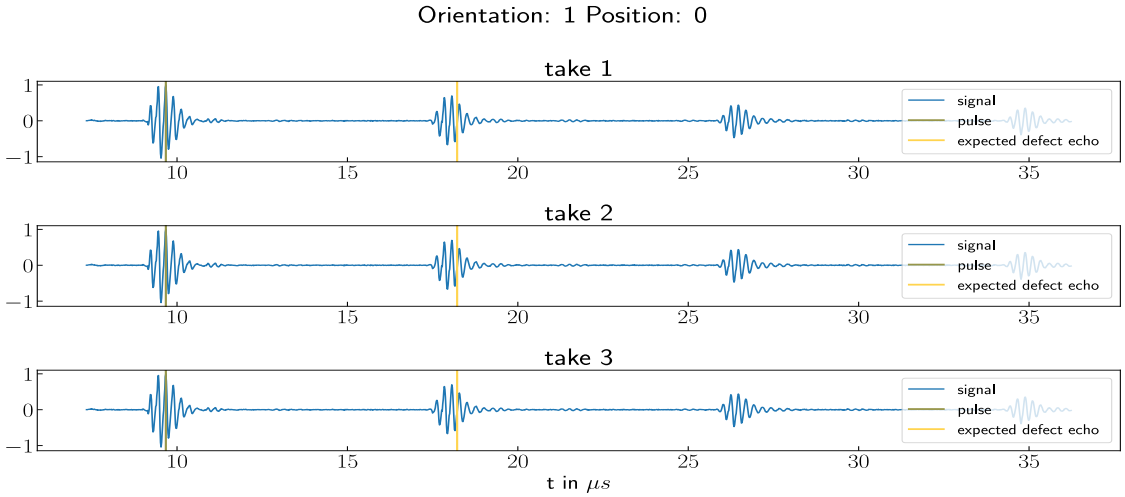


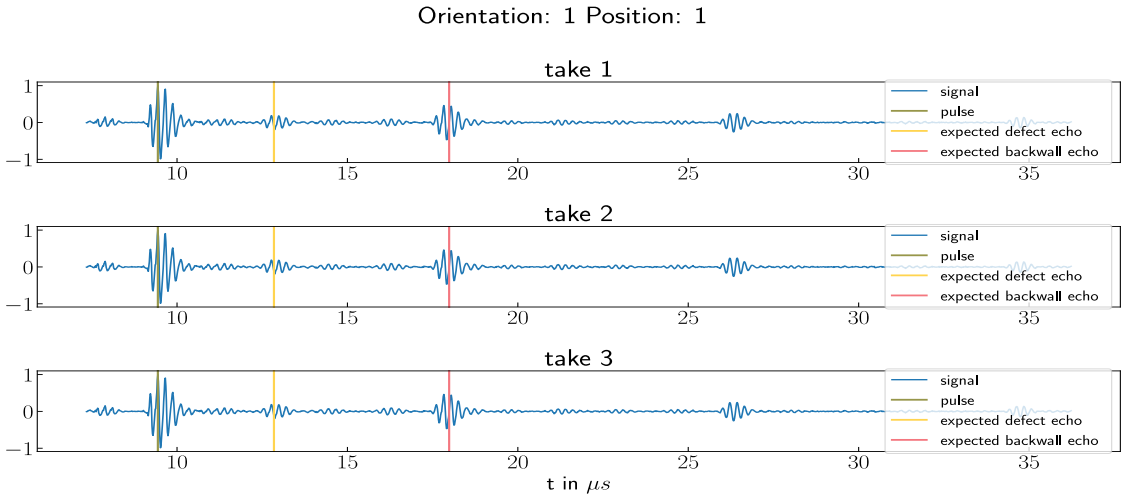
Orientation: 2 Position: 5

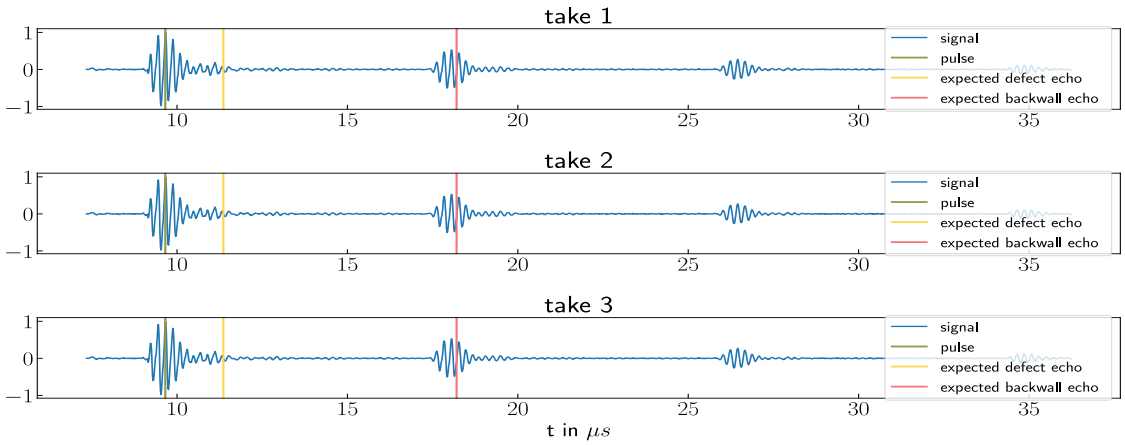


Orientation: 2 Position: 6

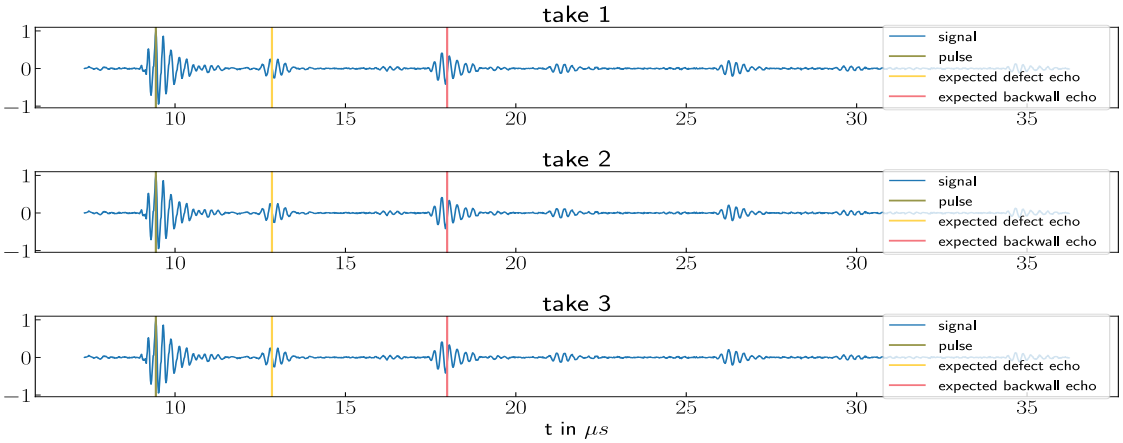



Orientation: 1 Position: 0
take 1
take 2
take 3
signal
pulse
expected defect echo
t in μs

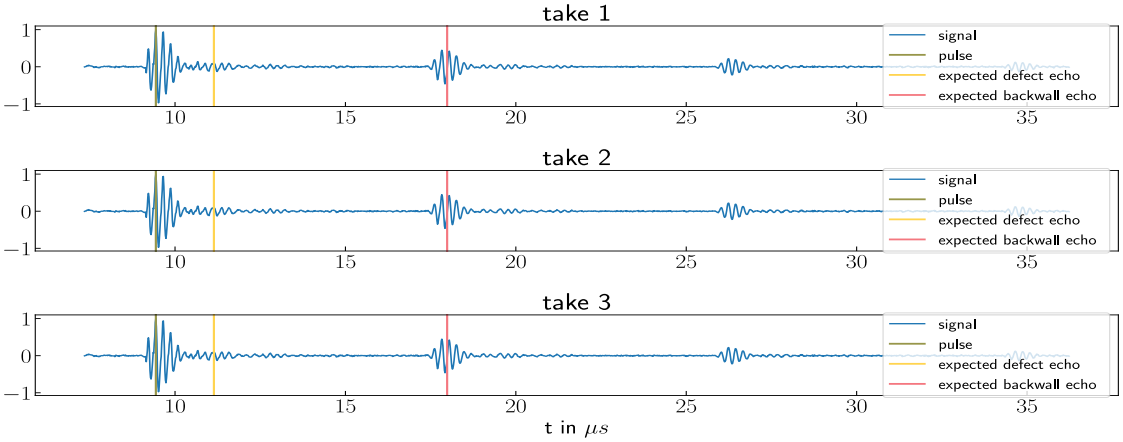

Orientation: 1 Position: 1
take 1
take 2
take 3
signal
pulse
expected defect echo
expected backwall echo
t in μs


Orientation: 1 Position: 2
take 1
take 2
take 3
signal
pulse
expected defect echo
expected backwall echo
t in μs

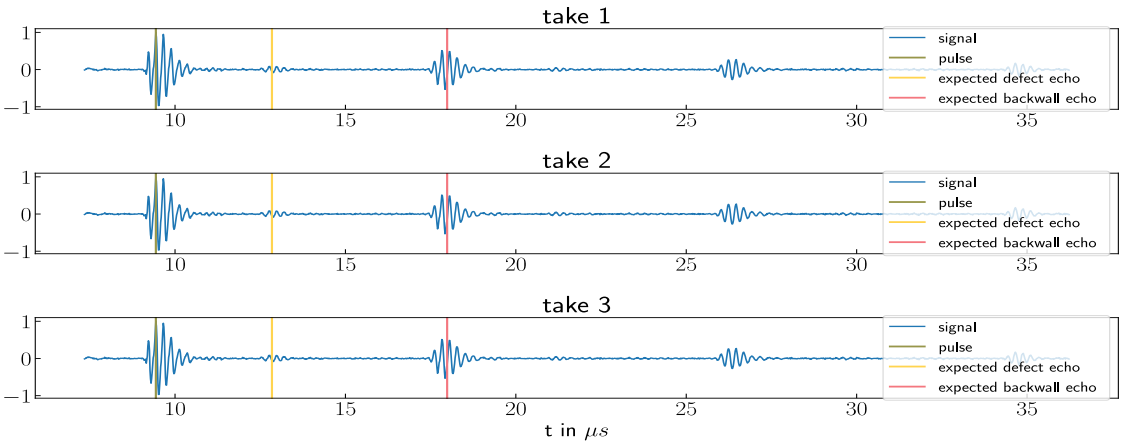
Orientation: 1 Position: 3



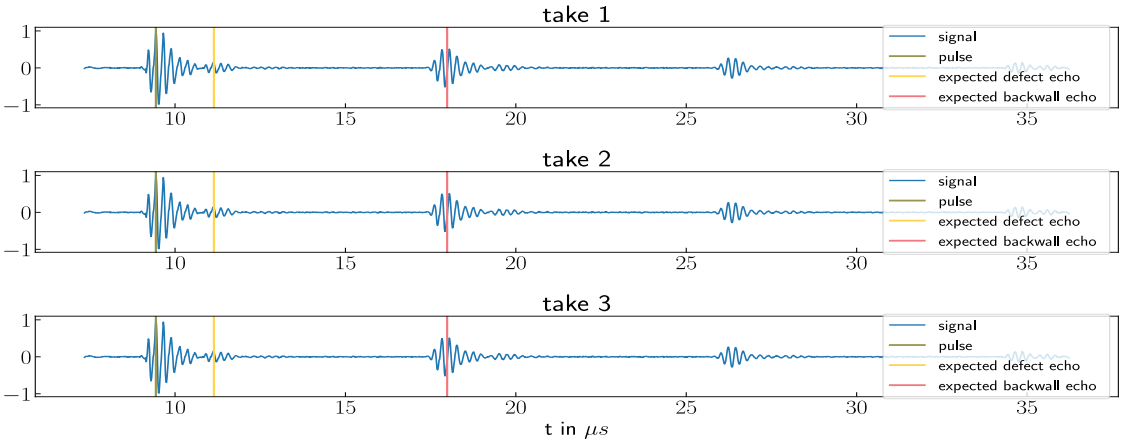
Orientation: 1 Position: 4



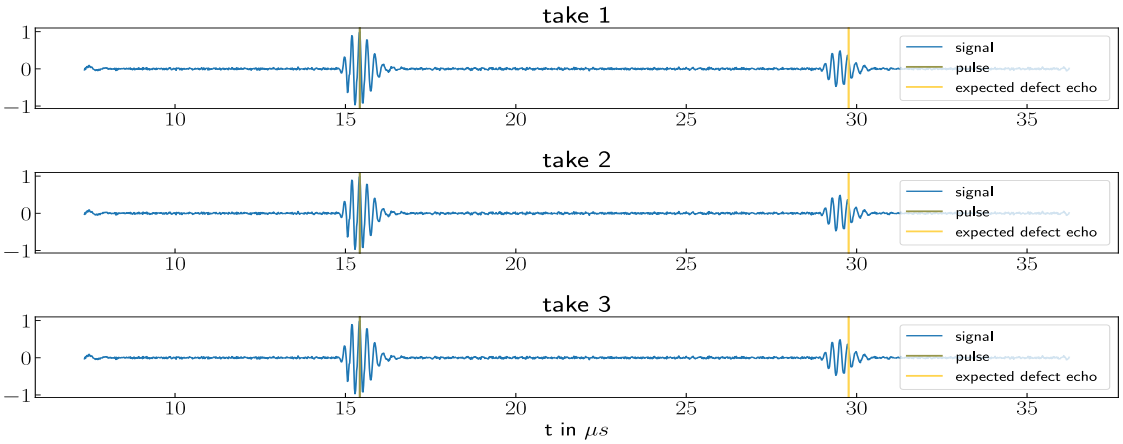
Orientation: 1 Position: 5



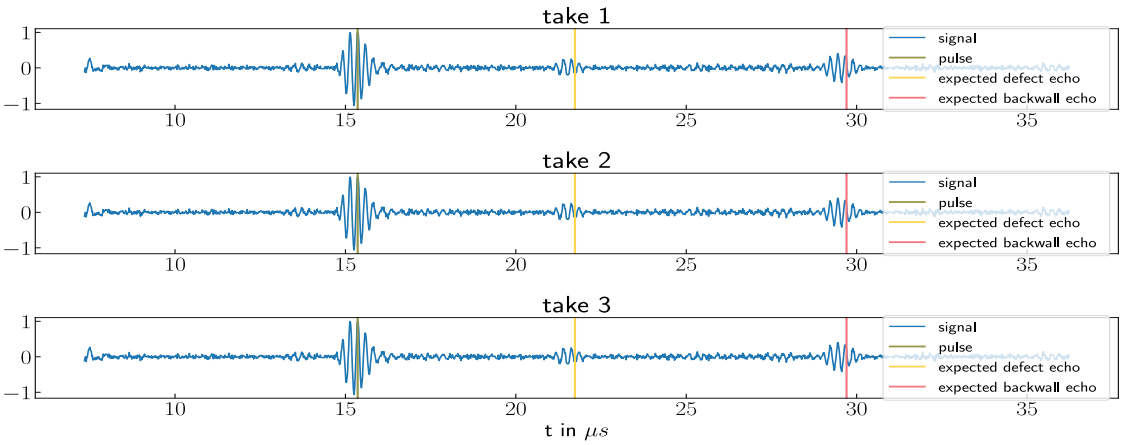
Orientation: 1 Position: 6



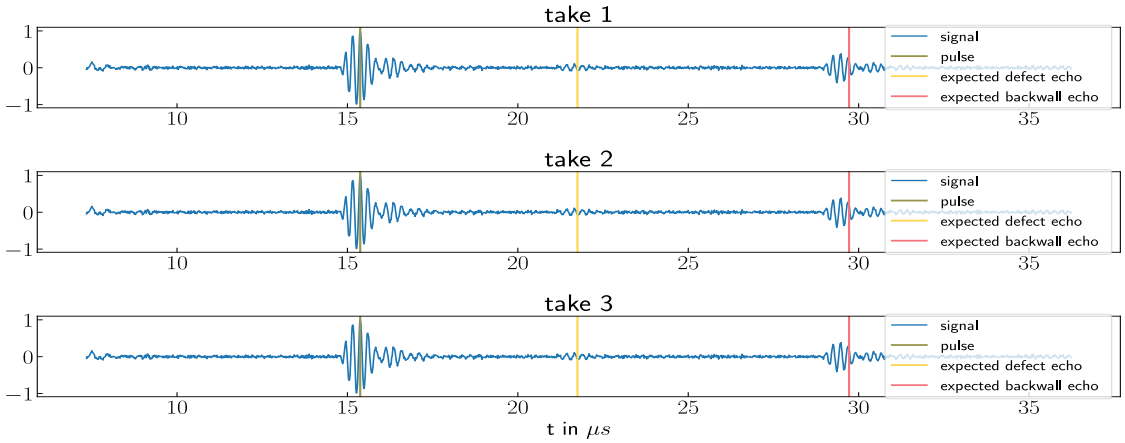
Orientation: 2 Position: 0



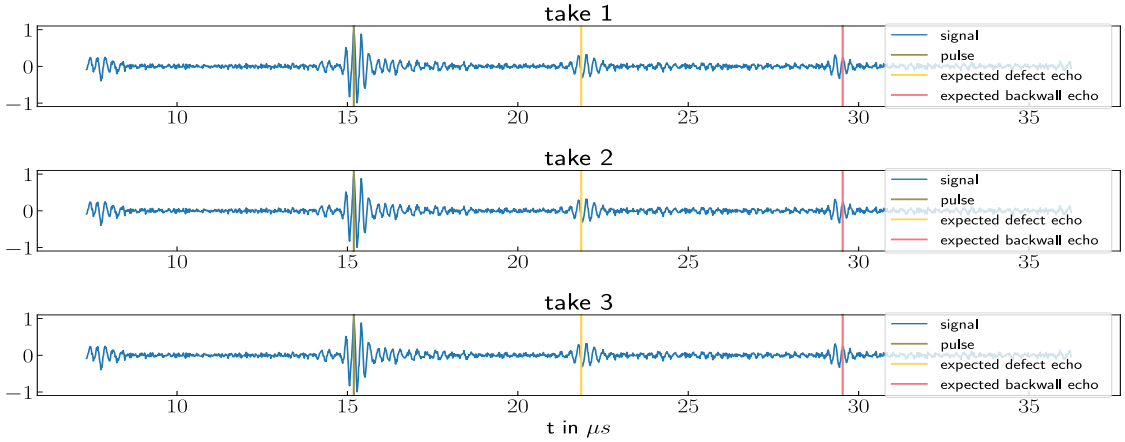
Orientation: 2 Position: 1



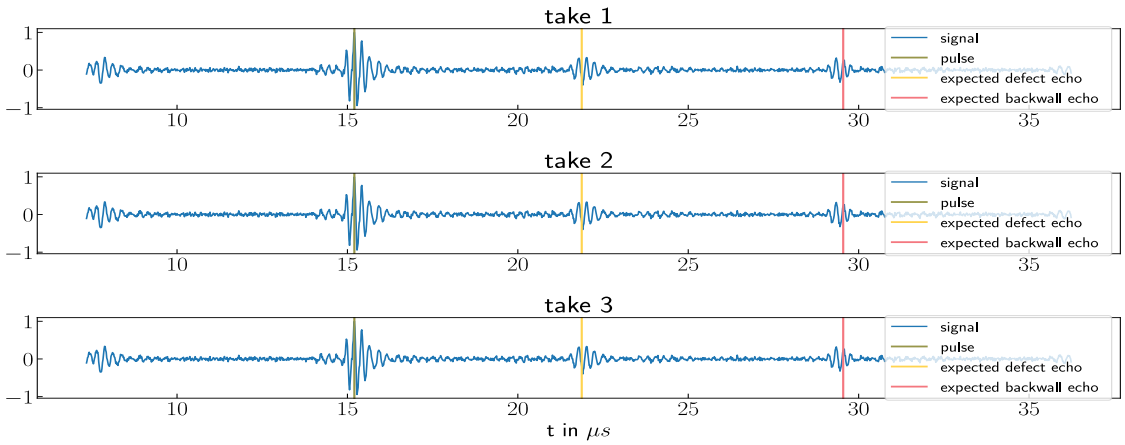
Orientation: 2 Position: 2

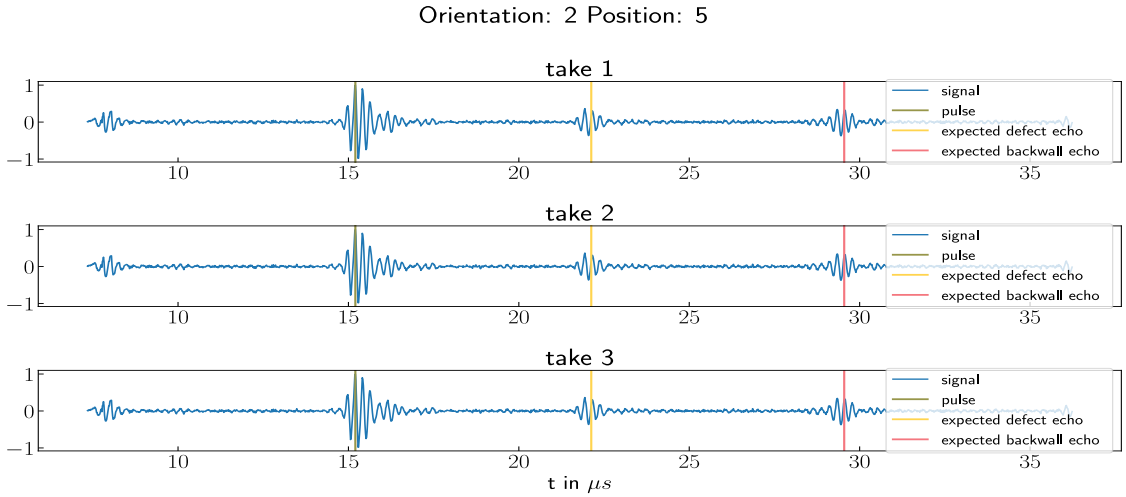


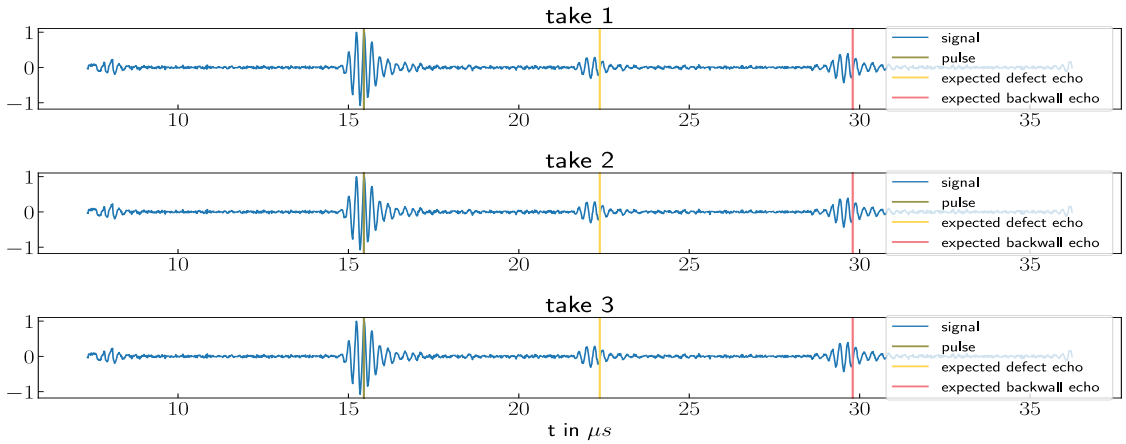
Orientation: 2 Position: 3



Orientation: 2 Position: 4




Orientation: 2 Position: 5
take 1
take 2
take 3
signal
pulse
expected defect echo
expected backwall echo
1
0
−1
10
15
20
25
30
35
t in μs


Orientation: 2 Position: 6
take 1
take 2
take 3
signal
pulse
expected defect echo
expected backwall echo
1
0
−1
10
15
20
25
30
35
t in μs

B Schematics

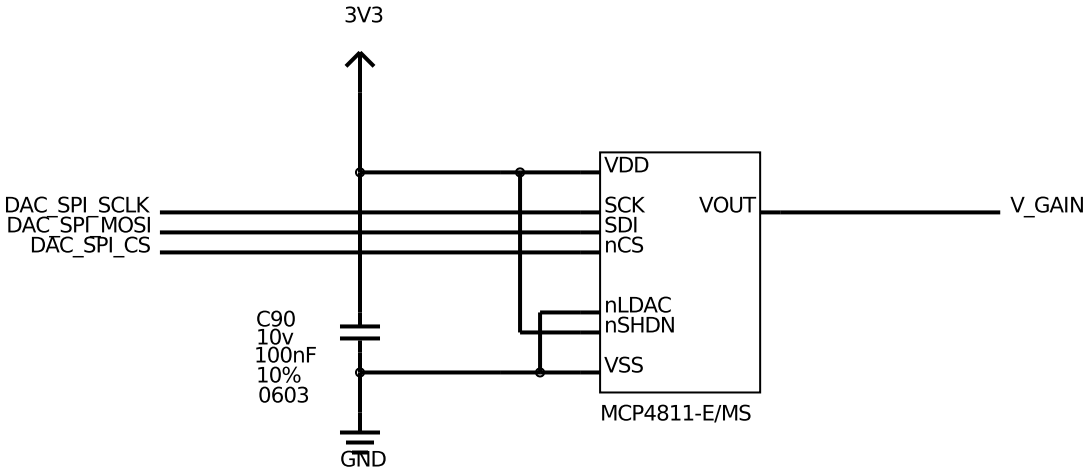


Figure B.1: Un0rick time gain compensation (TGC) DAC - controllable reference voltage for VGA (from [Jon20a])

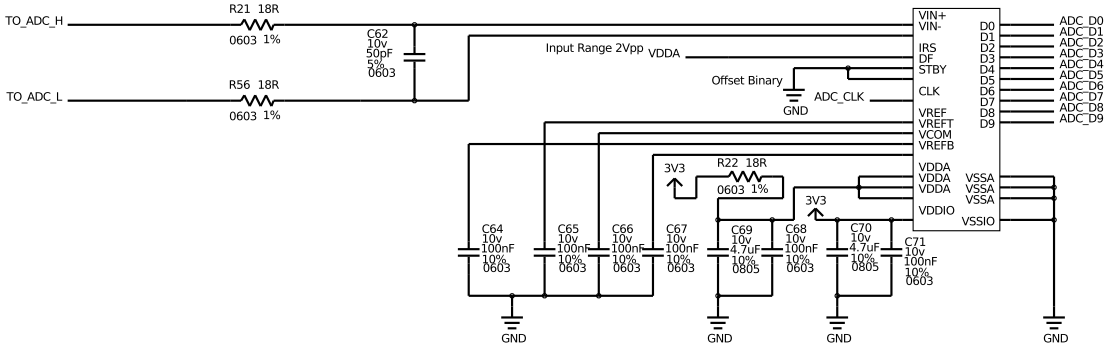


Figure B.2: Un0rick analog to digital converter (ADC) (from [Jon20a])

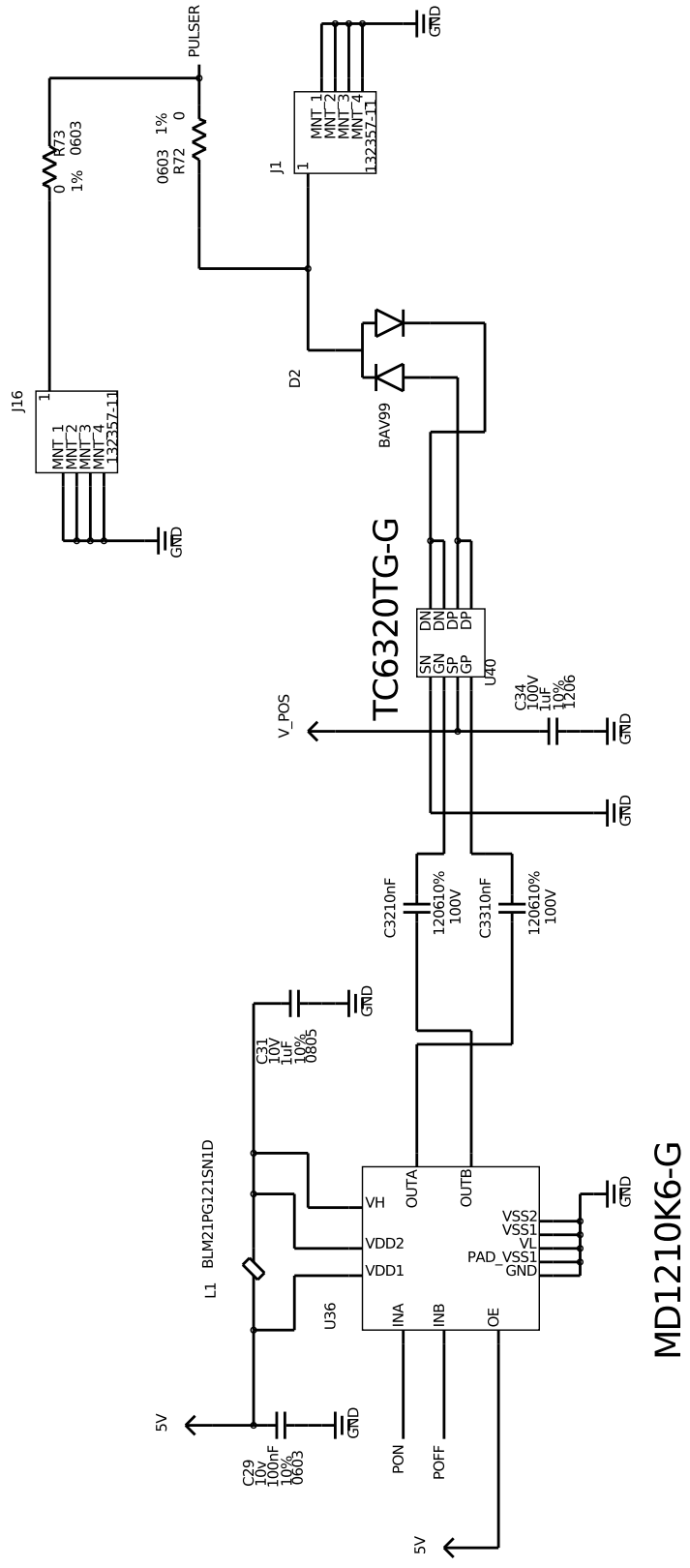


Figure B.3: Un0rick high voltage pulser circuit (from [Jon20a])

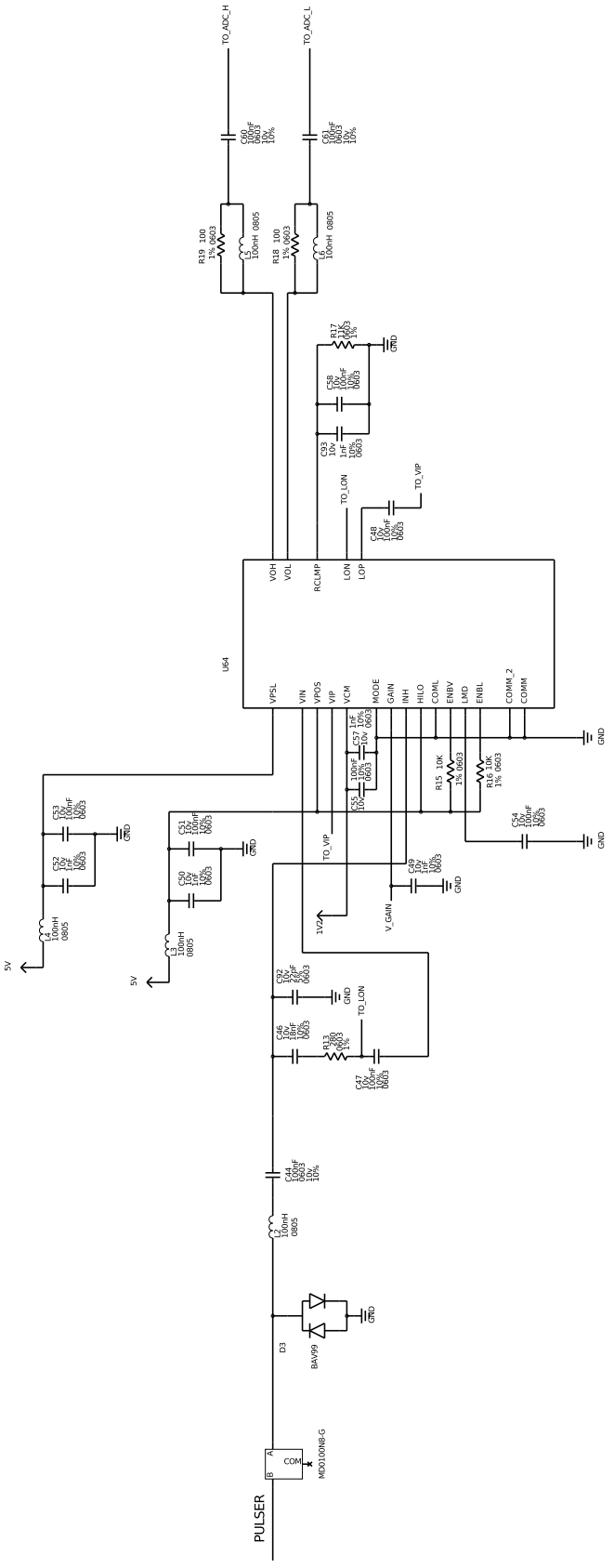


Figure B.4: Un0rick time gain compensation (TGC) variable gain amplifier (from [Jon20a])

Bibliography

[Bro12] Brook, Mark V.: *Ultrasonic inspection technology development and search unit design: examples of practical applications*. John Wiley & Sons, 2012

[Cha10] Chan, Ken: Design of differential filters for high-speed signal chains. In: *Texas Instruments, SLWA053B, rev* (2010)

[Che02] Cheeke, J David N.: *Fundamentals and applications of ultrasonic waves*. 2. CRC press, 2002

[Dat10] Datasheet, MCP4811: *MCP4801/4811/4821 8/10/12-Bit Voltage Output Digital-to-Analog Converter with Internal VREF and SPI Interface*. Microchip Technology Inc., 2010

[Dat13] Datasheet, ADC10065: *ADC10065 10-Bit 65 MSPS 3V A/D Converter*. Texas Instruments, 2013

[Dat16a] Datasheet, AD8331: *Ultralow Noise VGAs with Preamplifier and Programmable RIN*. Analog Devices, 2016

[Dat16b] Datasheet, TC6320: *N-Channel and P-Channel Enhancement-Mode MOSFET Pair*. Microchip Technology Inc., 2016

[Dat19] Datasheet, MD1210: *High-Speed Dual MOSFET Driver*. Microchip Technology Inc., 2019

[DPV97] Deutsch, Volker ; Platte, Michael ; Vogt, Manfred: *Ultraschallprüfung: Grundlagen und industrielle Anwendungen*. 2. Springer-Verlag, 1997

[Jon19] Jonveaux, Luc: un0rick : open-source fpga board for single element ultrasound imaging. (2019), 08. `http://dx.doi.org/10.5281/zenodo.3364559`. – DOI 10.5281/zenodo.3364559

[Jon20a] Jonveaux, Luc: *un0rick/hardware/MATTY-V11.pdf*. `https://github.com/kelu124/un0rick/blob/master/hardware/MATTY-V11.pdf`, 2020. –

commit: 3c23ad7427569d25e06861d360ce2ba78d658636 [online; last visited on 21.07.2021]

[Jon20b] Jonveaux, Luc: *un0rick/pyUn0/pyUn0.py*. `https://github.com/kelu124/un0rick/tree/master/pyUn0/pyUn0.py`, 2020. – commit: b32852932c4c0603bbd98a3b144c3900522c5b81 [online; last visited on 21.07.2021]

[Jon20c] Jonveaux, Luc: *un0rick/un0rick/hardware.md*. `https://github.com/kelu124/un0rick/blob/gh-pages/un0rick/hardware.md`, 2020. – commit: f7ccf9bc9afbe733e39bb636f501efd1a1b15e14 [online; last visited on 21.07.2021]

[Jon20d] Jonveaux, Luc: *un0rick/usb/verilog*. `https://github.com/kelu124/un0rick/tree/master/usb/verilog`, 2020. – commit: 027ab32a98293b8dc1dfefecad237441a34dd425 [online; last visited on 21.07.2021]

[Jon20e] Jonveaux, Luc: *un0rick/vhdl/matty.vhd*. `https://github.com/kelu124/un0rick/blob/master/vhdl/matty.vhd`, 2020. – commit: d99afead2a9076ad31055590ec47ecaa7ecc42b5 [online; last visited on 21.07.2021]

[Jon21a] Jonveaux, Luc: *un0rick*. `https://https://github.com/kelu124/un0rick`, 2021. – commit: ee840bfbc1e3dfda97da96dbd171d617eeebfc3f [online; last visited on 21.07.2021]

[Jon21b] Jonveaux, Luc: *un0rick/un0rick.md*. `https://github.com/kelu124/un0rick/blob/gh-pages/un0rick.md`, 2021. – commit: 97300154d5c2c55c8223c00499bba2c9254966a6 [online; last visited on 21.07.2021]

[Jon21c] Jonveaux, Luc: *un0rick/un0rick/ndt.md*. `https://github.com/kelu124/un0rick/blob/gh-pages/un0rick/ndt.md`, 2021. – commit: 2558c2b31206f4323c5df97fa35b5cd7d8f0cd7d [online; last visited on 21.07.2021]

[Jon21d] Jonveaux, Luc: *un0rick/un0rick/rpi-setup.md*. `https://github.com/kelu124/un0rick/blob/gh-pages/un0rick/rpi-setup.md`, 2021. – com-

mit: 2558c2b31206f4323c5df97fa35b5cd7d8f0cd7d [online; last visited on 21.07.2021]

[Jon21e] Jonveaux, Luc: *un0rick/un0rick/usb-setup.md*. `https://github.com/kelu124/un0rick/blob/gh-pages/un0rick/usb-setup.md`, 2021. – commit: 2558c2b31206f4323c5df97fa35b5cd7d8f0cd7d [online; last visited on 21.07.2021]

[Jon21f] Jonveaux, Luc: *un0rick/usb/python_lib/un0usb-0.2.6.tar.gz*. `https://github.com/kelu124/un0rick/blob/master/usb/python_lib/un0usb-0.2.6.tar.gz`, 2021. – commit: c1bf3674f711674fbac0c3041bc0a6bc87982769 [online; last visited on 21.07.2021]

[JPW96] Jones, DJ ; Prasad, SE ; Wallace, JB: *Piezoelectric materials and their applications part 1. definitions and measurements*. Sensor Technology Limited, 1996

[KK90] Krautkrämer, Josef ; Krautkrämer, Herbert: *Ultrasonic Testing of Materials*. 4. Springer-Verlag, 1990

[Mis01] Misaridis, Athanasios: *Ultrasound imaging using coded signals*, Diss., dec 2001

[Rat19] Rathod, Vivek T.: A review of electric impedance matching techniques for piezoelectric sensors, actuators and transducers. In: *Electronics* 8 (2019), Nr. 2, S. 169

[San93] Sands, Donald E.: *Introduction to crystallography*. Courier Corporation, 1993

[TEKP10] Tichỳ, Jan ; Erhart, Jiri ; Kittinger, Erwin ; Privratska, Jana: *Fundamentals of piezoelectric sensorics: mechanical, dielectric, and thermodynamical properties of piezoelectric materials*. Springer Science & Business Media, 2010

Declaration of Originality

I hereby declare that this thesis is my own original work, which has been composed by me without using aids other than those specified. I have clearly referenced all sources, both published or unpublished, which have been directly or indirectly used in the work. This work has not been, previously or concurrently, submitted to any other examination authority.

Place, Date

Tim Treichel