

Integrating FPGA-Based Signal Generation Hardware into Experiment Control System

Student Paper**Author(s):**

Bugnon, Cédric; Lazzaroni, Lorenzo

Publication date:

2024

Permanent link:

<https://doi.org/10.3929/ethz-b-000668511>

Rights / license:

[In Copyright - Non-Commercial Use Permitted](#)



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Integrating FPGA-Based Signal Generation Hardware into Experiment Control System

Group Project

Student Cédric Bugnon
Student Lorenzo Lazzaroni
cbugnon@student.ethz.ch
llazzaroni@student.ethz.ch

Quantum Optics Laboratory (QOL)
ETH Zürich

Supervisors:

Jacob Fricke (QOL)
Jeffrey Mohan (QOL)
Bahadır Dönmez (Trapped Ion Quantum Information)
Peter Clements (Engineering Unit in Quantum Center)
Dr. Kadir Akin (Engineering Unit in Quantum Center)
Prof. Dr. Tilman Esslinger (QOL)

April 8, 2024

Acknowledgements

First, we would like to express our gratitude to Prof. Dr. Tilman Esslinger for giving us the opportunity to participate in a group project at the Quantum Optics Laboratory. We gained many important skills and knowledge during our project which will be very useful for our academic career.

We would also like to express our gratitude to all our supervisors: Jacob Fricke, Jeffrey Mohan, Bahadır Dönmez, Peter Clements, and Dr. Kadir Akin. They were instrumental in our project, always there to address our questions and provide invaluable guidance and support. Their expertise, patience, and unwavering dedication significantly contributed to the success of our endeavor.

Additionally, we extend our appreciation to Alexander Frank, whose eagerness to assist with hardware-related issues and soldering was invaluable throughout our project.

Abstract

Achieving high fidelity for the measurement and control of quantum experiments imposes strict requirements on the precision and stability of surrounding electronics. Controlling electronics from a central device is more challenging when they are distributed distantly in a laboratory and require analog signals where effects like ground loops and radiative cross-talk can limit their performance. The project aims to solve the noise issues in long coaxial cables with a flexible and scalable system. The scalability is achieved by showcasing that multiple arbitrary waveform generators (AWG) can be synchronized and connected to one central device. The main idea is to transmit and receive signals via low voltage differential signal (LVDS) carried by the shielded and twisted pairs in the Ethernet cable, and then generate and sample the signals locally near the electronics that require analog inputs or provide outputs. The interface between the single ended signals and differential signals is very important as the signal integrity depends strongly on this transition. Since it is very vulnerable it is therefore important to protect it against any kind of damage that could occur from the outside. This project aims to showcase and improve the durability and to increase the signal quality of the differential signal interface.

In conjunction with the hardware advancements outlined in the previous section, this project addresses the critical software infrastructure necessary for the integration and control of distributed electronics in quantum experiments. Central to the software component is the development of a robust framework written in C++, designed to orchestrate the control and measurement tasks of quantum experiments across distributed electronics. The framework encompasses a modular architecture that accommodates the integration of various devices represented as classes within the software. A significant focus of the project is the incorporation of FPGA functionality into the existing software framework. This involves the creation of a dedicated class for the FPGA, facilitating the communication with other device classes. Moreover, the software component comprises an interface that provides intuitive access to FPGA programming functionalities and allows users to program and control the FPGA seamlessly, enhancing the overall usability and accessibility of the system.

Contents

Acknowledgements	i
Abstract	ii
1 Introduction	1
1.1 System Setup	1
1.2 Improving the Hardware Interface	2
1.3 Integrating the FPGA in the Control Software	3
2 Hardware and System Performance Evaluation	4
2.1 Investigating Factors Behind LVDS Chip Failures	4
2.1.1 Inspection of the LVDS Output Pins	4
2.1.2 Measurement Results of the Output Channels and Discussion	5
2.1.3 ESD Protection Diodes	7
2.1.4 ESD Gun Tests	7
2.1.5 Discussion of Results	11
2.1.6 Improvements in PCB Design	11
2.2 Enclosure Design	14
2.2.1 Design of the Enclosure	14
2.3 Scalability of the Low-Noise Eight-Channel AWG	16
2.3.1 Procedure to Connect two FPGAs	16
2.3.2 Generation and Results of the synchronized SPI Signals	17
3 High-Level Control Software	19
3.1 Understanding the Main Structure of the Software Used in the Lab	19
3.1.1 <i>ExperimentControl</i>	19
3.1.2 ExpWiz	21
3.2 Understanding the Low-Level Software Implemented in Python	23
3.2.1 Configuration Flags	23
3.2.2 Metadata	24
3.2.3 Waveform Samples	24
3.3 Integrating the DeviceQODAC in the Control Software	24

CONTENTS

3.3.1	Translating the Low-Level Software Implemented in Python	25
3.3.2	Quick Guide to the Setup	26
3.3.3	Steps Towards a Working Environment	27
3.3.4	Playing the Desired Waveforms with the FPGA, Programmed on <i>ExperimentControl</i>	33
4	Conclusion	34
5	Possible Improvements	35
5.1	High-Level Control Software	35
5.2	Hardware and System Performance Evaluation	35

Introduction

1.1 System Setup

The system designed by Shijia Chen [1] contains a control computer, a Zybo Z7-20 Field Programmable Gate Array (FPGA), a custom-designed Printed Circuit Board (PCB), and a custom-designed Digital-to-Analog Converter (DAC) board. This system is built upon the previous system by Bahadır Dönmez [2]. Figure 1.1 shows the Low-Noise Eight-Channel Arbitrary waveform generator (AWG) created by Shijia.

The goal of the system is to generate arbitrary low-noise waveforms and to send these signals to receiving electronics that require analog inputs or provide outputs. The waveforms will be sent via low voltage differential signals in an Ethernet cable to a DAC chip which converts the digital signals to analog signals.

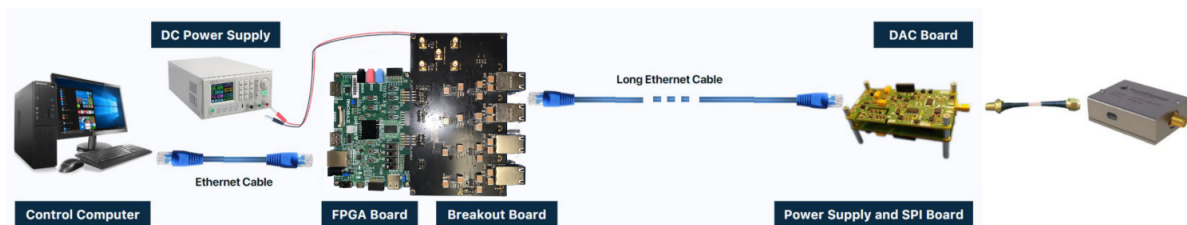


Figure 1.1: System setup, taken from [1]

The system is able to generate arbitrary low-noise waveforms and supports a minimum of eight channels. The procedure to create these waveforms is given below:

1. The control computer runs a Python script which sends metadata and the digital waveform to the FPGA via a Transmission Control Protocol (TCP) connection through an Ethernet connection cable.
2. The FPGA uses the received digital signals to create Single-ended serial peripheral interface (SPI) signals which it sends then to the breakout board.
3. The breakout board converts the SPI signals to Low-voltage differential signaling (LVDS) pairs and sends them to the DAC board through an Ethernet connection cable.
4. The SPI board transforms these LVDS pairs back to single-ended SPI signals.
5. The DAC board then converts the SPI signals to analog signals. The dac board can then be connected via Bayonet Neill–Concelman (BNC) connectors to the receiver hardware.

1. INTRODUCTION

1.2 Improving the Hardware Interface

The hardware interface between the single ended signals and differential signals is crucial to the signal quality and in the end to the viability of the whole system. As the current system lacks any kind of protection against Electrostatic discharge (ESD) events [1], certain LVDS drivers have been damaged during use. Therefore, the system needs protection measures against these events to protect it in future use and to have a long living and robust device in an experimental environment. This will be achieved by introducing ESD protection diodes. Additionally, the whole systems needs to be scalable which means that the user should be able to have multiple AWGs connected to one control computer. This step is not yet established, as currently only one system can be connected and controlled from a PC.

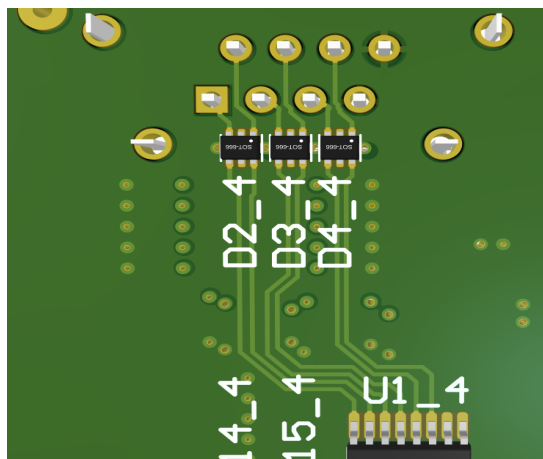


Figure 1.2: TVS Diodes (D2_4, D3_4, D4-4) at output of LVDS chips

To achieve the proposed goals the following tasks had to be completed:

1. The source responsible for damaging the LVDS chips needs to be identified.
2. The durability of the protection devices needs to be tested to guarantee that in the future the LVDS chips cannot be damaged anymore.
3. Changes to the PCB:
 - (a) The ESD diodes need to be strategically placed on the PCB to enhance their utility.
 - (b) As the system currently has no labels and silkscreens for all components, they need to be added for an improved overview and a quicker assembly of the PCB.
4. For additional protection, a custom-designed enclosure needs to be crafted to shield the PCB and the FPGA.
5. The capability to connect two FPGAs to one PC and to synchronize them using an external trigger signal needs to be demonstrated.

The Figure 1.2 shows a section of the previously designed PCB [1], showcasing the newly added ESD protection diodes.

1. INTRODUCTION

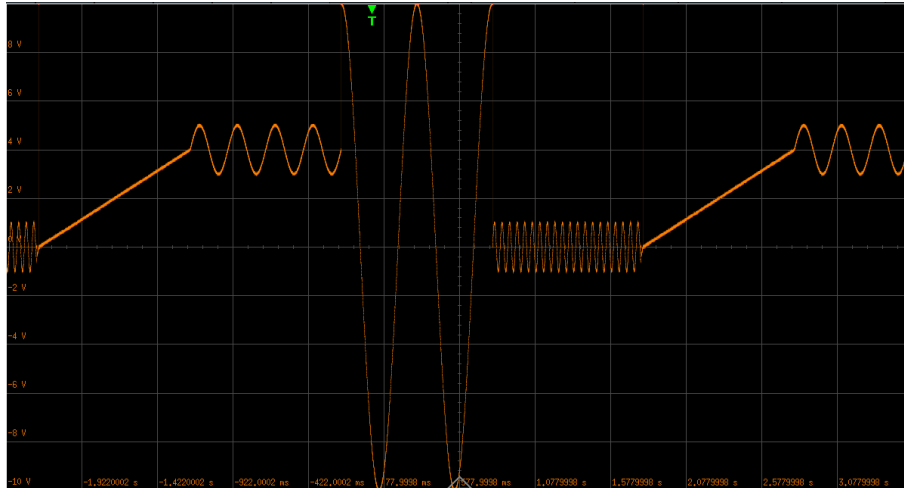


Figure 1.3: waveform played by the FPGA (oscilloscope screenshot)

1.3 Integrating the FPGA in the Control Software

The quantum optics laboratory's control software is used to program a large number of devices which can be used to control quantum optics experiments. It already comprised the settings to collect the data to be sent to the FPGA, however, it lacked the piece of code to successfully send all this information to the FPGA. The main tasks to accomplish this goal were:

1. Understanding the main structure of the quantum control software used in the laboratory;
2. Understanding the low-level software implemented in Python for communicating with the FPGA;
3. Converting the low-level software implemented in Python to C++ and integrating it into the control software;
4. Generating quantum control waveforms in the PC, transferring them to the RAM of the FPGA, and sending the control commands to the FPGA.

Figure 1.3 shows an example of the waveforms the FPGA is able to play, once all the data are transmitted correctly to it. While this was previously accomplished through a Python script, the goal was to integrate it inside of the quantum optics control software.

Hardware and System Performance Evaluation

2.1 Investigating Factors Behind LVDS Chip Failures

The objective of this section is to identify and address the problem of LVDS chip failures observed in the PCB. Currently, as per Shijia's breakout board [1], among the expected eight functional channels, channel six and seven exhibit signal corruption issues. Additionally, attempts to send waveform signals from the control computer result in the absence of correct wave function generation.

2.1.1 Inspection of the LVDS Output Pins

During our investigation into the output connectors of the LVDS chips, we employed a "Keysight U1252B" Multimeter to measure the resistance of the output pins to ground. The pins represent the positive and negative logic signals of SPI, in particular the differential pairs of the Clock Signal (CLK), the Chip Select (CS), and the Main Out, Sub In (MOSI) signal path. Among these, the MOSI line is particularly crucial as it carries data from the main device to the sub-device.

In Table 2.1, we conducted measurements on an unoperated LVDS chip to establish a reference measurement of a chip expected to be entirely intact. Under normal conditions a LVDS chip that is inoperative should always exhibit a Open Loop (O.L) behaviour across all output channels, except for the ground connection.

PINS	red[Ω]	black[Ω]
SCLK_LVDS_N	O.L	O.L
SCLK_LVDS_P	O.L	O.L
MOSI_LVDS_P	O.L	O.L
MOSI_LVDS_N	O.L	O.L
CS_LVDS_N	O.L	O.L
CS_LVDS_P	O.L	O.L
24V_NI	O.L	O.L
PGND	0.23	0.21

Table 2.1: Reference measurements on an unoperated LVDS chip

2. HARDWARE AND SYSTEM PERFORMANCE EVALUATION

2.1.2 Measurement Results of the Output Channels and Discussion

The tables provided below present the measured resistances in Ohms (Ω) to ground. In instances where there is no connection possible, denoted as O.L, it signifies an open circuit condition. In the table, I signified with the colours "red" and "black" the polarity of the Multimeter during the measurement process. Specifically, "red" denotes that the positive polarity of the Multimeter was connected to the output pin while the negative polarity was grounded, and vice versa. In Table 2.5 we see the measurements of the broken LVDS chips that produced corrupted signals in red.

We can observe that out of the two faulty LVDS chips, the resistance to ground for pins `MOSI_LVDS_P` and `MOSI_LVDS_N` is significantly reduced, almost to the level of a short circuit. This finding aligns with the corrupted output waveforms observed on the oscilloscope, indicating that a damaged MOSI path disrupts signal transmission. This fact is supported by the claim that supposedly functional chips also exhibit pins with high resistance to ground. For instance, in Table 2.2, the resistance of pin `CS_LVDS_P` in Channel 1 to ground is notably low compared to the reference chip, yet Channel 1 still produces a correct signal. This must be because this working channel still has an open circuit connection to ground in the MOSI signal path. Therefore, we can deduce that only chips with a damaged MOSI signal path fail to generate a correct output signal.

2. HARDWARE AND SYSTEM PERFORMANCE EVALUATION

PINS	red[Ω]	black[Ω]	PINS	red[Ω]	black[Ω]
SCLK_LVDS_N	O.L	O.L	SCLK_LVDS_N	O.L	O.L
SCLK_LVDS_P	O.L	O.L	SCLK_LVDS_P	O.L	O.L
MOSI_LVDS_P	O.L	O.L	MOSI_LVDS_P	O.L	O.L
MOSI_LVDS_N	O.L	O.L	MOSI_LVDS_N	O.L	O.L
CS_LVDS_N	501.17	O.L	CS_LVDS_N	501	O.L
CS_LVDS_P	10.79	10.72	CS_LVDS_P	10.72	10.73
24V_NI	O.L	O.L	24V_NI	O.L	O.L
PGND	2.11	1.87	PGND	2.14	2.82

Table 2.2: Measurements of Channel 1 left and Channel 2 right

PINS	red[Ω]	black[Ω]	PINS	red[Ω]	black[Ω]
SCLK_LVDS_N	O.L	O.L	SCLK_LVDS_N	O.L	O.L
SCLK_LVDS_P	O.L	O.L	SCLK_LVDS_P	O.L	O.L
MOSI_LVDS_P	O.L	O.L	MOSI_LVDS_P	O.L	O.L
MOSI_LVDS_N	O.L	O.L	MOSI_LVDS_N	O.L	O.L
CS_LVDS_N	502.92	O.L	CS_LVDS_N	O.L	O.L
CS_LVDS_P	150.63	150.35	CS_LVDS_P	O.L	O.L
24V_NI	O.L	O.L	24V_NI	O.L	O.L
PGND	2.23	2.25	PGND	2.13	2.21

Table 2.3: Measurements of Channel 3 left and Channel 4 right

PINS	red[Ω]	black[Ω]	PINS	red[Ω]	black[Ω]
SCLK_LVDS_N	O.L	O.L	SCLK_LVDS_N	O.L	O.L
SCLK_LVDS_P	O.L	O.L	SCLK_LVDS_P	O.L	O.L
MOSI_LVDS_P	O.L	O.L	MOSI_LVDS_P	O.L	O.L
MOSI_LVDS_N	O.L	O.L	MOSI_LVDS_N	O.L	O.L
CS_LVDS_N	501.2	O.L	CS_LVDS_N	484.3	O.L
CS_LVDS_P	4.31	4.3	CS_LVDS_P	4.67	4.67
24V_NI	O.L	286	24V_NI	O.L	284
PGND	1.56	1.58	PGND	1.73	1.67

Table 2.4: Measurements of Channel 5 left and Channel 8 right

PINS	red[Ω]	black[Ω]	PINS	red[Ω]	black[Ω]
SCLK_LVDS_N	O.L	O.L	SCLK_LVDS_N	O.L	O.L
SCLK_LVDS_P	O.L	O.L	SCLK_LVDS_P	O.L	O.L
MOSI_LVDS_P	20.11	20.11	MOSI_LVDS_P	498	O.L
MOSI_LVDS_N	495	O.L	MOSI_LVDS_N	10.9	10.89
CS_LVDS_N	496	O.L	CS_LVDS_N	O.L	O.L
CS_LVDS_P	6.44	6.4	CS_LVDS_P	O.L	O.L
24V_NI	O.L	284.5	24V_NI	O.L	283
PGND	1.6	1.58	PGND	1.83	1.83

Table 2.5: Measurements of Channel 6 left and Channel 7 right

2. HARDWARE AND SYSTEM PERFORMANCE EVALUATION

2.1.3 ESD Protection Diodes

One of the factors contributing to the breakdown of the LVDS chips are ESD events. ESD events can occur, for example, when a person walks over a carpet or a floor with poorly conductive shoes, charges themselves up with static electricity, and then touches the integrated circuit, resulting in an ESD strike. To determine whether this is the root cause of the chips breaking, we utilized an ESD simulation device capable of generating rapid, high-voltage pulses [3].

A way to protect electronic devices against ESD is with Transient Voltage Suppressor (TVS) Diodes [4]. When the voltage on the transient voltage suppressor reaches a certain threshold, the avalanche effect in the semiconductor causes the p-n junction to be conductive which then creates a low-impedance path to ground and can redirect the high-voltage pulse that the circuit receives. The reason why we use TVS diodes is because their response time is extremely fast in ranges of a few picoseconds. It is crucial for our purposes to employ bidirectional TVS diodes because ESD events can occur not only from the signal line but also from the ground region. Moreover, since our circuit handles differential signals consisting of both positive and negative polarity, the use of bidirectional diodes is recommended [5]. The LVDS chips we utilize come equipped with ESD protection rated at 4kV. To demonstrate their susceptibility to ESD events, we had to surpass this threshold. Furthermore, we aimed to illustrate that by employing TVS diodes with a higher voltage rating, we could safeguard the LVDS chips.

2.1.4 ESD Gun Tests

In our experiments, we utilized three distinct types of TVS diodes, each with varying rated voltages and connection configurations. These diodes were soldered onto the output pins of a freshly assembled PCB equipped with unused LVDS chips. The experiment was conducted by delivering voltage pulses to the output pins of the chips. Figure 2.1 displays the ESD gun used in the experiment, while Figure 2.2 showcases the setup featuring the soldered diodes.



Figure 2.1: ESD Simulation Device used in the experiment [6]

2. HARDWARE AND SYSTEM PERFORMANCE EVALUATION

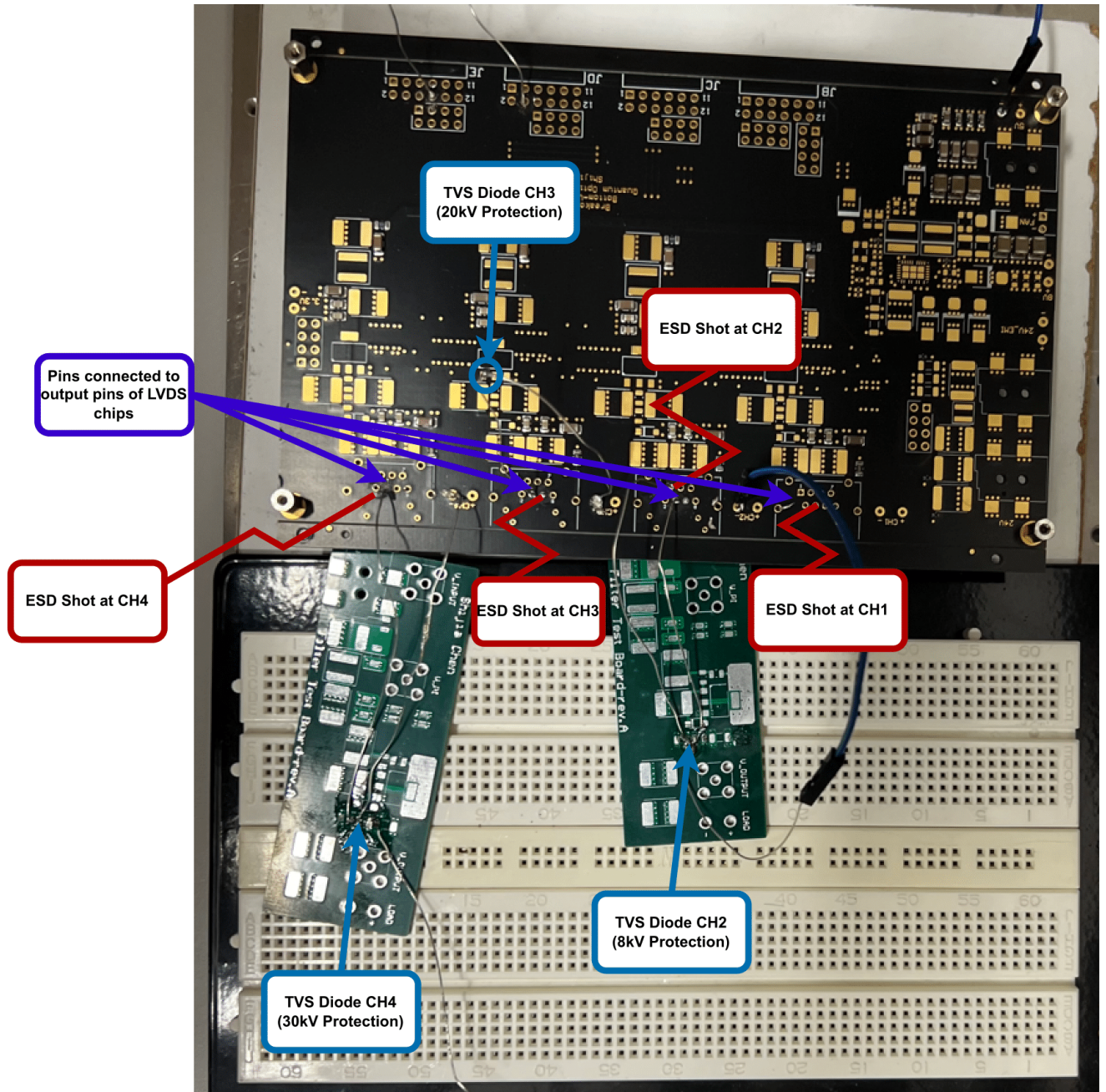


Figure 2.2: Experiment setup: On the top, the black PCB with the to be tested LVDS chips soldered onto it. On the bottom, the ESD protection diodes soldered onto a green spare PCB. The labels indicate where the ESD shots have been applied, the protection level of each channel, and where the diodes were placed.

2. HARDWARE AND SYSTEM PERFORMANCE EVALUATION

We equipped each of the four channels with varying levels of protection. Channel 1 relies solely on the integrated ESD protection of the chip itself (4kV) and has no additional diode. Channel 2 is equipped with a bidirectional TVS diode designed for differential signals up to 8kV (TVS DIODE 5.5VWM 8VC SOT9X3). Channel 3 features a unidirectional TVS diode protecting it up to 20kV (ESD7004, SZESD7004), while Channel 4 is equipped with a unidirectional protection diode offering protection up to 30kV (SOD-882 esd protection). For the unidirectional diodes we can apply the voltage only on the output pins and not at ground. To have a constant reference we also applied the voltage only at the output pins for the bidirectional diodes. But in general one wants to use bidirectional diodes in the final implementation, as it also protects the chips from the ground. But for our tests it was sufficient to only apply the shocks at the output pins as we wanted to showcase the durability of the diodes.

The use of long cables is not ideal for TVS diodes as it can lead to reflections and thus to signal integrity corruption while using it normally. In these tests we had to use long cables as we did not have a place to solder them. But for testing the diodes durability with the ESD gun, it is not a problem to have long cables as we are solely interested if the LVDS chips break and not if the signal quality is good.

Initially, we subjected each channel to a 4kV voltage pulse to test the effectiveness of the integrated protection. If a resistance measures O.L, it corresponds to $100M\Omega$ to ensure visual clarity in the plot. Different pins of each channel were utilized in the plots; however, it can be assumed that all pins behave similarly. Therefore, if one pin is damaged during the tests, we can switch to another pin and continue testing. All of our tests were conducted with the contact discharge mode of the ESD simulation device.

2. HARDWARE AND SYSTEM PERFORMANCE EVALUATION

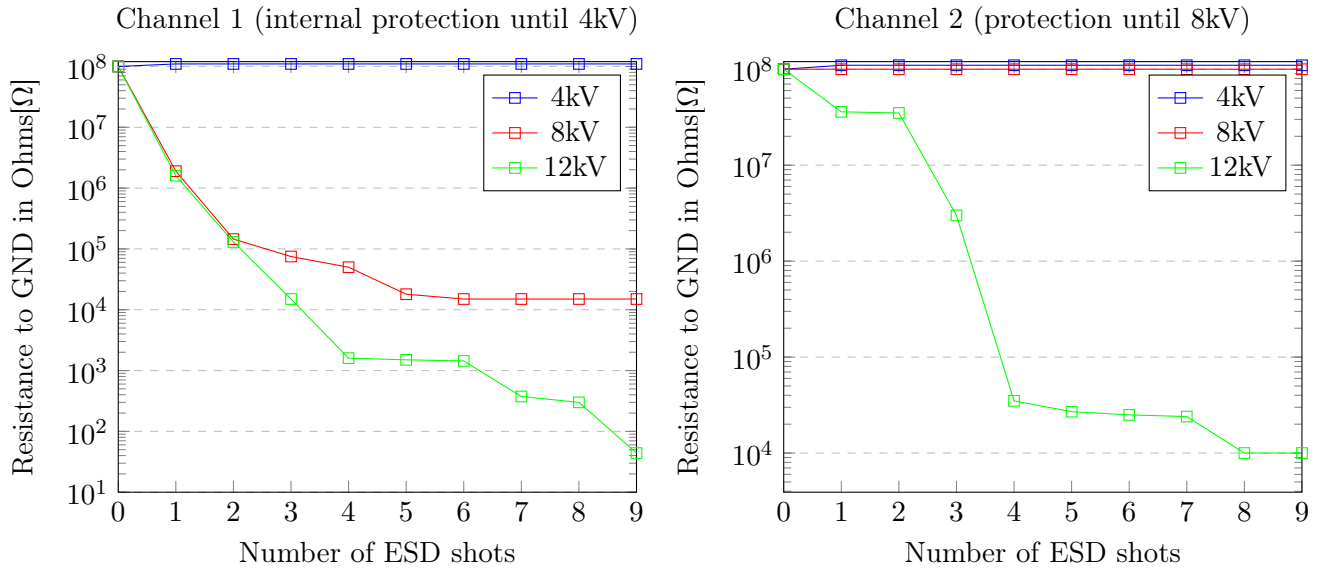


Figure 2.3: ESD Simulation Tests Ch1 and Ch2

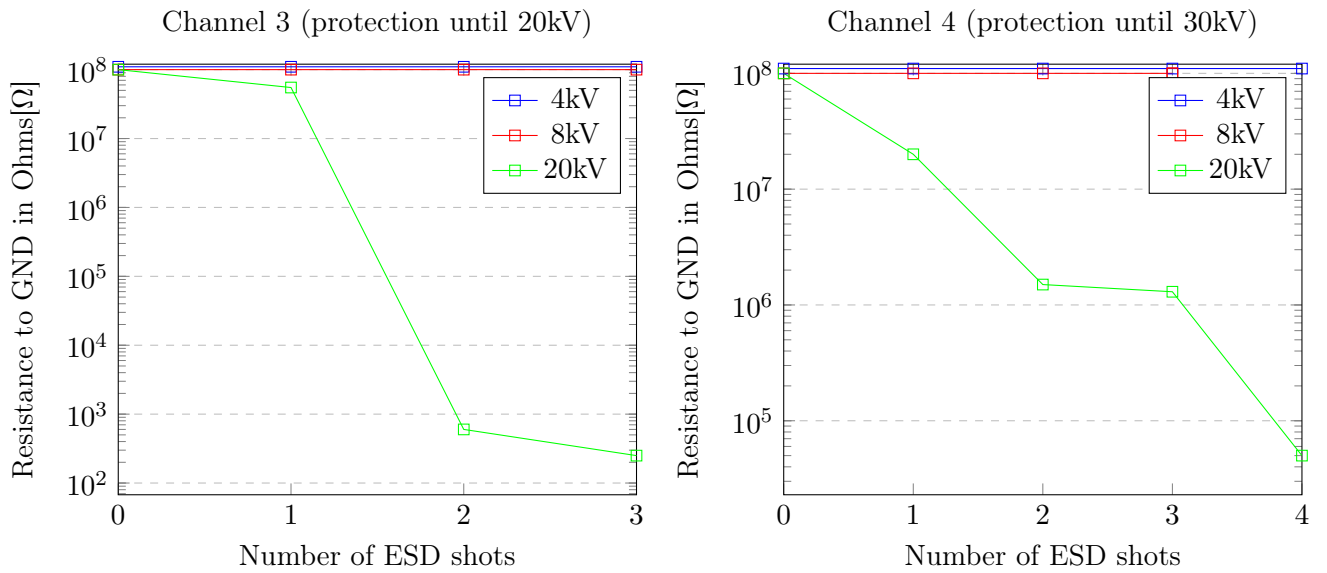


Figure 2.4: ESD Simulation Tests Ch3 and Ch4

2. HARDWARE AND SYSTEM PERFORMANCE EVALUATION

2.1.5 Discussion of Results

As anticipated, the tests conducted with a 4kV voltage burst on all diodes demonstrated that the protection mechanism functions reliably, even under repeated voltage applications. So we can safely assume that the basic protection provided by the manufacturer works fine, thus the ESD shots that the LVDS chips received through contact discharge must have been much higher than that threshold.

Channel 1:

As the applied voltage level increases, we observe a significant decrease in measured resistance to ground in the absence of additional protection, especially evident with 12kV shots. Notably, for 8kV shots, the resistance stabilizes at a fixed value, contrasting with the continued decrease seen with higher voltage shots.

Channel 2:

This channel demonstrates resilience against repeated 8kV voltage bursts; however, at 12kV, the resistance begins to decrease significantly, as expected. Similarly to the behavior we observed in Channel 1.

Channel 3:

While Channel 3 withstands small to moderate voltage bursts, it exhibits signs of failure after just two ESD shots at the 20kV level, despite the diode's rated protection. This issue may stem from a falsely applied voltage shock or a faulty connection at the diode. Naturally, it must be acknowledged that the rated voltage serves as an upper limit. Therefore, even if we apply this voltage precisely, there remains a possibility that the diode may not fully protect against such voltage due to voltage fluctuations in the ESD simulation device. Additionally, under normal circumstances TVS diodes should be connected with a short cable, but in our setup the cables to the diodes were of a few centimeters, as we didn't have space to solder them directly onto the PCB. Also, ordering new PCBs just to test diodes would not have been financially worth as we are trying to destroy the LVDS chips. Therefore, there is a possibility that the 20kV and 30kV shocks still reached to the chips.

Channel 4:

In Channel 4, the impact of 20kV is substantial, despite the TVS diode being rated at 30kV. Similarly to Channel 3, we assume that due to the considerable length of the cable connected to the diode, there was a possibility that the 20kV and 30kV ESD shocks reached the LVDS chip before the TVS diode could redirect the high voltage.

One can see a clear connection between the resistances to ground of the LVDS chips and the applied voltage bursts so we can conclude that the damage was most likely due to a contact discharge by humans. Additionally, the symptoms of the damaged LVDS chips that we observed in section 2.1.2 are very similar to those recreated during our ESD simulation tests. It is therefore important to add an additional protection on top of what is already integrated. That is why we decided to improve the PCB design of Shijia [1] by adding TVS diodes at the outputs of the vulnerable chips.

2.1.6 Improvements in PCB Design

In the PCB design, it's crucial to position the diodes in close proximity to the output connector pins, especially those directly connected to the socket where the Ethernet connector will be plugged in. This placement ensures that in the event of an ESD occurrence, the voltage pulse is intercepted long before it reaches the LVDS chips.

During testing, we observed an issue when attempting to transmit SPI signals to the LVDS chips

2. HARDWARE AND SYSTEM PERFORMANCE EVALUATION

that had previously endured ESD events. Despite their resilience to ESD, we encountered unwanted errors in the oscilloscope's differential signal readings. This problem stemmed from our testing setup, where diodes were connected to lengthy (8-10cm) copper wires for testing. Given the high-frequency nature of the SPI data transmission (operating at 31.25MHz), the extended wiring introduced signal reflections and superposition, leading to signal corruption.

Therefore it is advisable to have the TVS diode placed directly onto the signal line to minimize signal reflections and maintain signal integrity.

We therefore incorporated TVS diodes which are made for differential signals, like the one in Channel 2, where the data lines go through the diode, as can be seen in picture 2.6. We use the diode "USBLC6-2P6" which provides very low capacitance ESD protection up until 8kV contact discharge produced by STMicroelectronics. In figure 2.6 one can see the different connections through the diode. The data lines to be protected are connected to pins 1, 3, 4, 6 of the diode. Pin 2 serves as the Ground (GND) connection, while pin 5 functions as the Common Collector Voltage (VCC) connection.

In figures 2.7 and 2.8 one can see the TVS diodes incorporated into the PCB design. As, mentioned before they are placed very close to the output pins.



Figure 2.5: TVS diode in our setup [7]

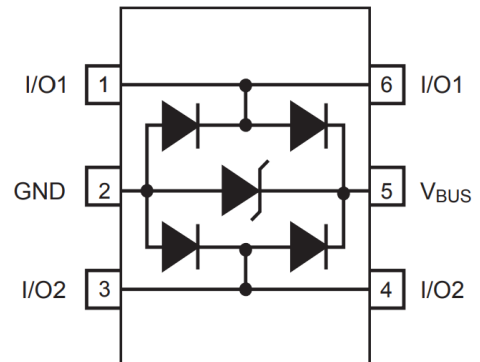


Figure 2.6: Signal line paths through the diode [8]

2. HARDWARE AND SYSTEM PERFORMANCE EVALUATION

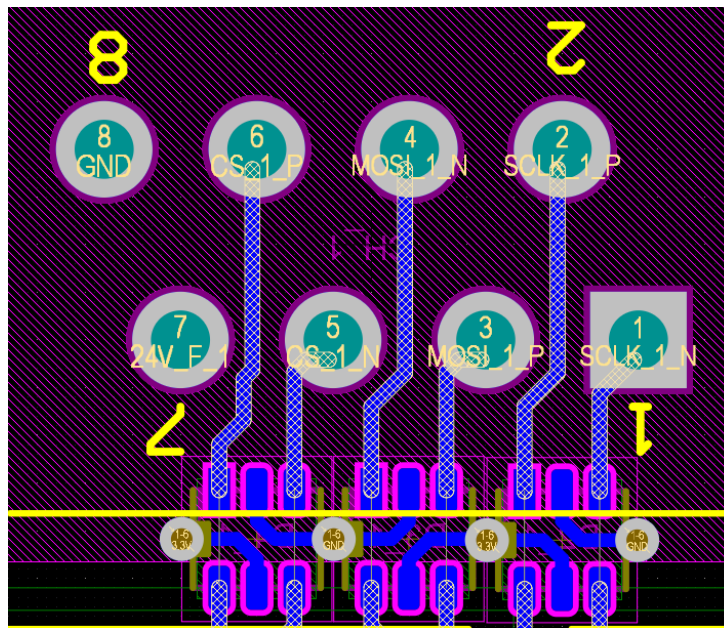


Figure 2.7: Close-up of the LVDS chips close to output pins (PCB Desing)

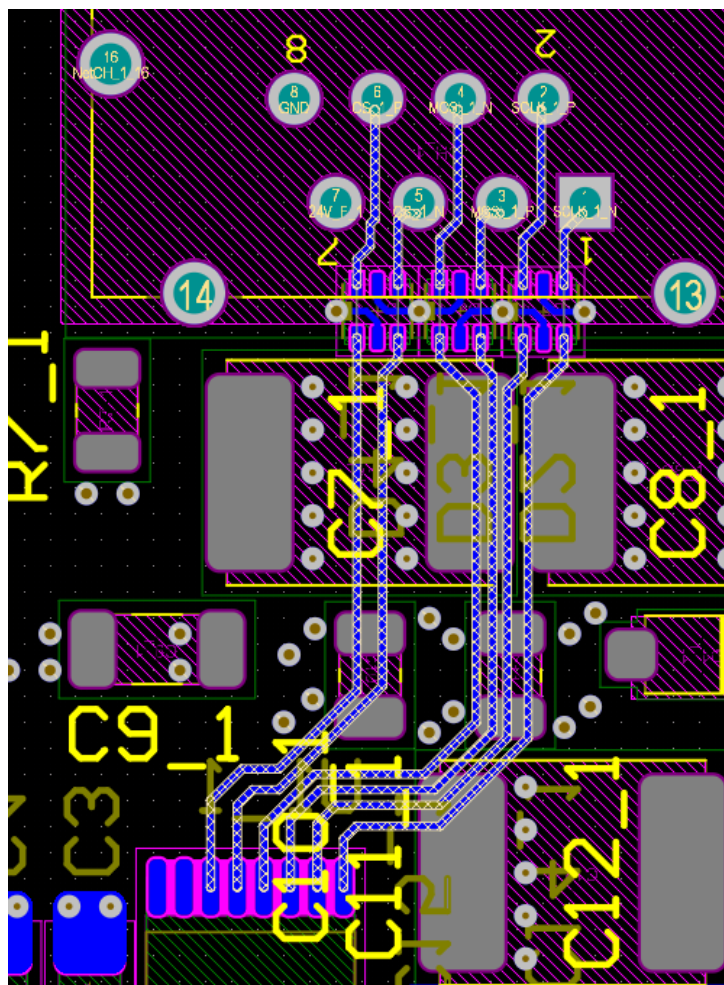


Figure 2.8: Connecting lines going through the diodes (PCB Desing)

2.2 Enclosure Design

In the previous section, we observed the critical significance of ESD protection, particularly in devices susceptible to human touch. To fortify our defense against these potentially damaging events, the subsequent important endeavor involves crafting an enclosure for the entire system consisting of the FPGA and the PCB. This enclosure serves as a fundamental barrier, ensuring basic protection for the device while affording users greater ease in handling, thereby mitigating the need for excessive caution.

Yet another crucial advantage of employing a metal enclosure is its ability to diminish susceptibility to environmental electromagnetic noise. Such interference threatens to signal quality, potentially introducing unwanted errors that undermine our objective of achieving low-noise waveforms. By adopting a metal enclosure, we effectively shield the device from electromagnetic disturbances emanating from other equipment within the laboratory environment, thereby safeguarding the integrity of our signals. In this section, we will show the design steps and the final result of the 3D printed enclosure.

2.2.1 Design of the Enclosure

An important feature of the enclosure is to prevent users from touching any surface of the device that isn't necessary. Therefore, we've ensured that the enclosure fits tightly, making it close to impossible to manipulate any components from the outside. The enclosure features four stilts for securely fastening the FPGA which are drilled in through the enclosure from the bottom side. So it is advisable to use plastic screws in order to prevent any ESDs going through the enclosure to the FPGA. It's advisable to refrain from frequently removing the FPGA and breakout board from the enclosure due to the tight fit, which could potentially damage certain components of both the FPGA and PCB during removal. Furthermore, we've incorporated holes for connectors, allowing the power and SMA connectors of the PCB to be securely screwed in place, thereby minimizing contact with interior components. This can be seen in the 3D model 2.11. We've installed ventilation holes on both the left and right sides of the enclosure to ensure good airflow for the components, preventing overheating. We have also designed a lid which can be screwed on the enclosure. The print took around 23 hours to complete and it has been printed with PLA filament material. PLA can start to lose its form and become pliable at temperatures above 60°C so it is not recommended to go beyond that threshold.

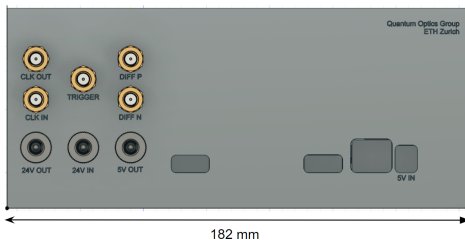


Figure 2.9: Back View

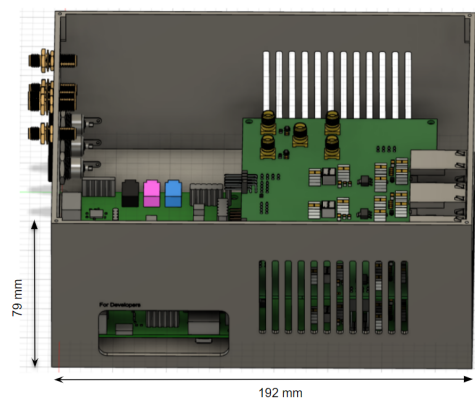


Figure 2.10: Top-down View

Figure 2.11: 3D model of enclosure design

2. HARDWARE AND SYSTEM PERFORMANCE EVALUATION

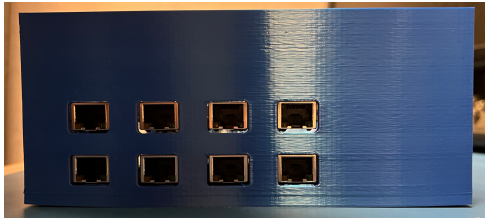


Figure 2.12: Front View



Figure 2.13: Back View



Figure 2.14: Back View

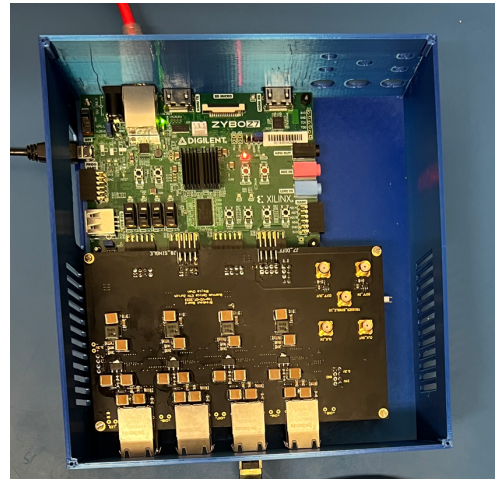


Figure 2.15: Top View

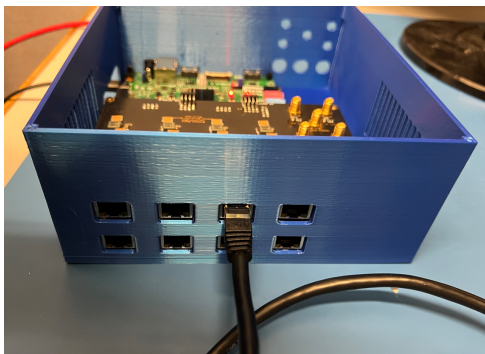


Figure 2.16: Top-down View



Figure 2.17: Top-down View

3D-printed enclosure with the FPGA and PCB incorporated in it with and without the lid.

2.3 Scalability of the Low-Noise Eight-Channel AWG

A bottleneck hindering scalability is the current system's reliance on only one FPGA connected to the control PC. Enhancing the system to accommodate multiple FPGAs connected simultaneously, generating identical signals, would be advantageous. Fortunately, Shijia's system [1] offers the capability to employ an external input trigger, enabling synchronization for concurrent signal transmission. Currently, the Python control software is only made to accommodate only one FPGA but by having two python scripts open at the same time and making small changes to the code, it is possible to connect the PC with two FPGAs at the same time.

2.3.1 Procedure to Connect two FPGAs

Following the part of the Python Script where the configuration flag is defined which determines how the waveforms are going to be played either in single or continuous play mode and additionally if one would like to use the external or debug trigger mode. In this code example the debug trigger mode is activated. The FPGA will wait until it receives the trigger signal, the DACs are already configured.

```

1 # IP address of the FPGA board running the TCP/IP server
2 server_ip = "192.168.1.XX"
3 # Port number the server is listening to
4 server_port = 5001
5 # Use internal trigger (0) external trigger (1) debug trigger (2)
6 trigger_mode = 2
7
8
9 # Use internal clock (0) external clock (1)
10 clock_mode = 0
11 # Use channel 1 for testing on single channel
12 channel_states = int('0b11111111',2) #00000100 for channel 3,
13 # channel 1 must always be enabled, since common clock
14 # Combine all individual flags to obtain 32-bit config flag
15 # Shift left and add
16 config_flag = (debug_messages
17 + (command << 1)
18 + (play_mode << 4)
19 + (trigger_mode << 5)
20 + (clock_mode << 7)
21 + (channel_states << 8))

```

To connect two FPGAs to one PC each one needs it's own Internet Protocol (IP) address. So the last number of

```

1 server_ip = "192.168.1.XX"

```

needs to be changed so it's different to all the other FPGAs had before. XX is therefore arbitrary but needs to be different. Additionally, in the file `tcp_perf_server.h` the macro `TCP_SERVER_IP_ADDRESS` needs to be changed such that it matches the IP address in the Python Code. Here is the discussed C code snippet:

```

1 /* Server port to listen on */
2 #define TCP_SERVER_PORT 5001
3 /* Server IP information */
4 #define TCP_SERVER_IP_ADDRESS "192.168.1.XX"
5 #define TCP_SERVER_IP_MASK "255.255.255.0"
6 #define TCP_SERVER_GW_ADDRESS "192.168.1.1"

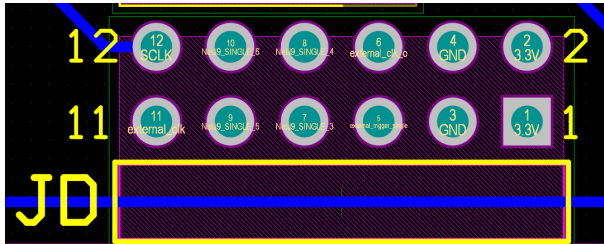
```

2. HARDWARE AND SYSTEM PERFORMANCE EVALUATION

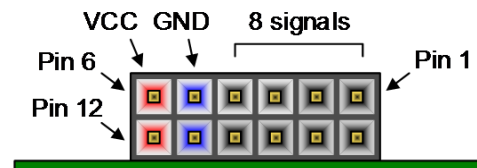
After defining or changing the macros the FPGA needs to be reprogrammed or it will not be possible to set up the TCP connection. After the Python Script has been executed, the console should read:

All DACs are configured, playing initial value and waiting for trigger.

If that is the case, a trigger can be applied at both FPGAs to start sending the SPI signals at the same time. To use `trigger_mode = 2`, the pin 5 (`external_trigger_single`) needs to be used of the JD input pins, this corresponds to the pin 10 on the FPGA.



((a)) input pins of the PCB



((b)) PMOS output pins of FPGA

2.3.2 Generation and Results of the synchronized SPI Signals

To generate the actual trigger signal one can use a wave generator and connect it to the two FPGAs. The following measurements were achieved while using pulse mode with a frequency of 1kHz, a high voltage of 3.3V, low voltage of 0.0V and a duty cycle of 50%.

On the following page, the Measurements presented in Figure 2.19 reveal a yellow signal representing the trigger pulse. The orange and green signals depict the SPI signals originating from the FPGAs. One can observe that there exists a latency of approximately 400ns between the transmission of the trigger pulse and the initiation of the first SPI signals. Moreover, the SPI signals from both FPGAs are synchronized to a few nanoseconds depending on the length of trigger pulse cable, thereby achieving our intended objective.

2. HARDWARE AND SYSTEM PERFORMANCE EVALUATION

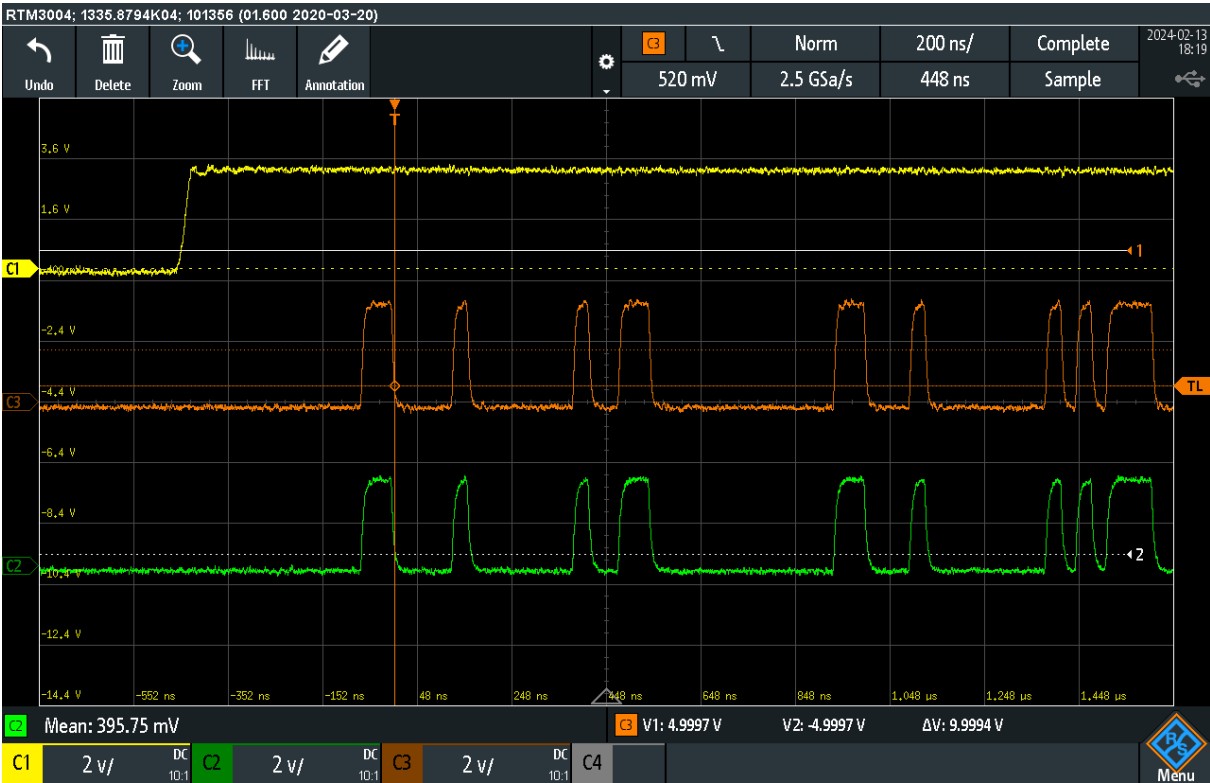


Figure 2.19: Oscilloscope measurement from two FPGAs (Simultaneous SPI signal generation initiated by a common trigger pulse)

High-Level Control Software

3.1 Understanding the Main Structure of the Software Used in the Lab

3.1.1 *ExperimentControl*

The starting point of the software used in the lab is the application `ExperimentControl`. It is from this application that the setup of the experiment can be chosen. Thanks to this application we can control the devices connected to the computer, program the timings of the events and generate the waveforms. After everything has been set up and the program is run, the application generates an XML file which is read by the software and the devices are initialized.

To understand how to set up the devices, the events and the waveforms, I suggest reading one of the `ctr` files saved in the control computer (which can directly be opened on `ExperimentControl`), as one can understand where to click when setting up the devices.

Figure 3.1 shows how `ExperimentControl` shows itself. The first step is to add a device. To do that, click on `Hardware>I/O Devices`. At this point, choose the QODAC device. Change the number in the field `Port` to 5001. After having done that, click `ok` and `close`. Once the device has been initialized, set up all the channels, from Channel 0 to Channel 7. To do that, click on `Hardware>Channels Setup...` and then click on the single channels. After having clicked on one channel, just click on `enable` and `show`.

After having done that, Figure 3.2 shows how the central field should look like. The next step is creating events, i.e. time windows in which the device will produce the desired output. It is important to note that a series of events, with the same starting and ending times, corresponds to every channel; these events can contain different formulas, enabling different channels to have different outputs in the same time window. To create an event, click on `Event`; assign a name to the timing edge, say `one`, and give it a formula. If the event is the first one, write 0 in the formula. This will create the first timing edge. As every event consists of two timing edges, to create at least an event create a second timing edge; to do that, click again on `Event`, assign a name to the event, and give it a formula. In this case the formula must be `previousEnabled + t`, and write the desired duration of the first event instead of `t`.

At this point, the central field should look like Figure 3.3. In this case we have one event of the duration of 1 s, delimited by two timing edges. There is actually a second event, which does not play any role for the experiment, but it is important for the software, as every sequence must end with an event without duration delimited by the last timing edge. After that, one can keep on creating new events following the same procedure; note that there will always be one more

3. HIGH-LEVEL CONTROL SOFTWARE

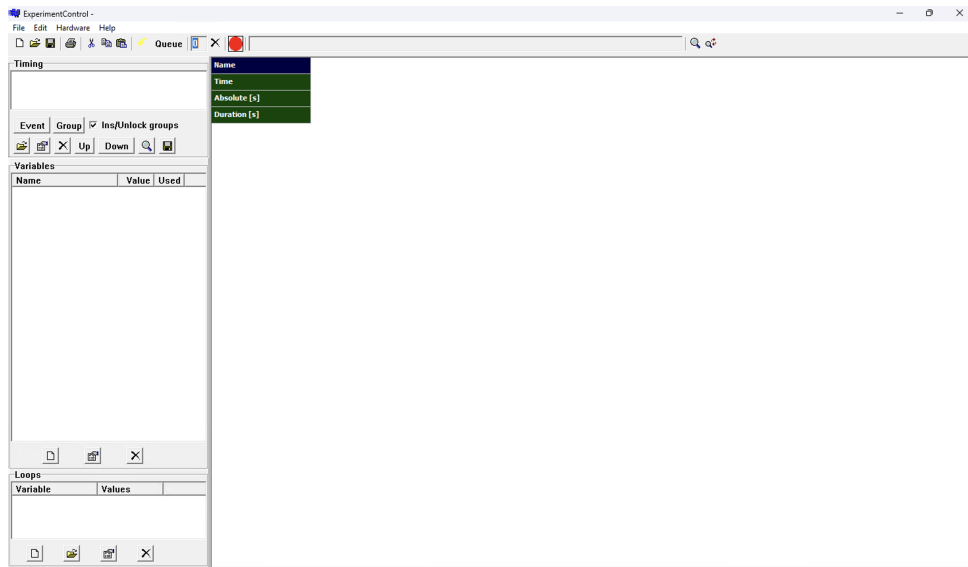


Figure 3.1: *ExperimentControl* layout.

Name
Time
Absolute [s]
Duration [s]
QODAC_0 [V]
QODAC_1 [V]
QODAC_2 [V]
QODAC_3 [V]
QODAC_4 [V]
QODAC_5 [V]
QODAC_6 [V]
QODAC_7 [V]

Figure 3.2: Channels layout.

Name	one	two
Time	0.0	previousEnabled + 1
Absolute [s]	0.0000000	1.0000000
Duration [s]	1.000000E+0	
QODAC_0 [V]	0.0	0.0
QODAC_1 [V]	0.0	0.0
QODAC_2 [V]	0.0	0.0
QODAC_3 [V]	0.0	0.0
QODAC_4 [V]	0.0	0.0
QODAC_5 [V]	0.0	0.0
QODAC_6 [V]	0.0	0.0
QODAC_7 [V]	0.0	0.0

Figure 3.3: How the central field looks like after having added one event lasting 1 second.

3. HIGH-LEVEL CONTROL SOFTWARE

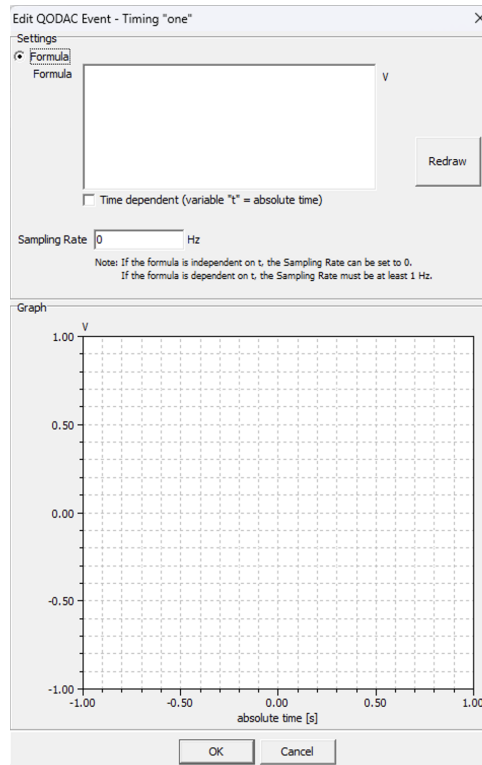


Figure 3.4: Window which opens to set up the output of a channel in a certain event.

event at the end, which has the same meaning as the second event mentioned before.

The next step is to choose the waveform. To do that, double click on the field in Figure 3.3 corresponding to a channel and to an event; this will set the output of the channel in the time window corresponding to that event. The window in Figure 3.4 opens. To choose the output of the channel in that event, write in the field `Formula`. Let us assume we want a sinusoidal wave with a frequency of 1 Hz and amplitude of 5 V. In this case, we would have to write $5 \cdot \sin(2 \cdot \pi \cdot t)$ and tick the button `Time dependent` (otherwise, the software would read the formula as time independent and output only one sample for the whole event). A crucial setting is the `Sampling Rate`. It represents how many samples the FPGA will save for this event, and set the duration of each sample accordingly. Let us assume we write 100 in this field. In this case the computer will upload to the FPGA 100 samples per second, so 100 samples for this event (which lasts one second); each sample will last for $1/\text{Sampling Rate}$ seconds, in this case 10 microseconds. Of course, the higher the Sampling Rate, the higher the precision of the output of the FPGA, but more on this later. Figure 3.7 shows how the window will look like after we click on the button `Redraw`. To save everything, we have to click on the button `ok`. This procedure can be followed to set every event for every channel up, and obtain the desired waveforms.

3.1.2 ExpWiz

Let us now move on to the code. All the code is contained in the folder `ExpWiz`, which is located in `Development`. The file of interest for our purposes is `DeviceQODAC.h`. It contains the declaration of the class `DeviceQODAC`, which represents the device we set up in the previous section. This class is a subclass of the classes `IODevice` and `TCPIPDevice`, and inherits all of their attributes. The most important member variables of `DeviceQODAC` are `buffer` and `buffer_idx`.

3. HIGH-LEVEL CONTROL SOFTWARE

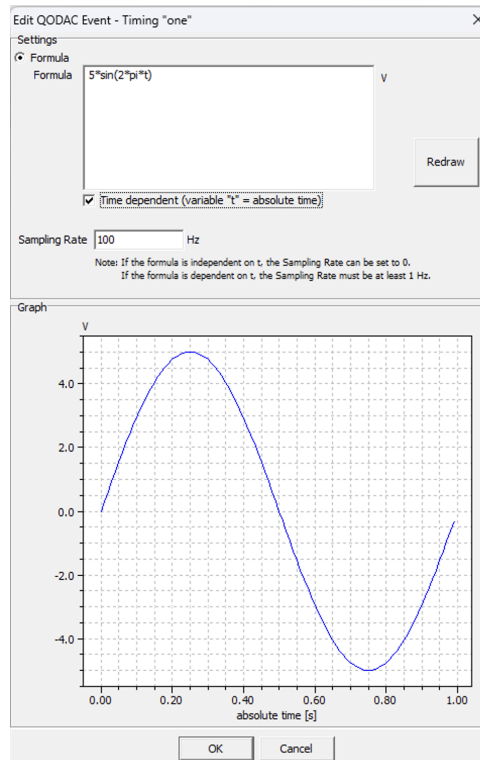


Figure 3.5: How the window looks like after clicking on Redraw.

The first one is an array which contains all the integers which will be sent to the FPGA, and the second one is the number of bytes contained in the first one. The two important member functions of `DeviceQODAC` are `GenerateEvents`, which is where the Python code was translated and integrated inside of the control software, and `sendCommands`, which sends the buffer contained inside of `DeviceQODAC` to the FPGA.

`DeviceQODAC` is a subclass of `IODevice`. Looking into this class we find the member variables `triggerExternal` and `clockExternal`, which come from what was set in `ExperimentControl`. Another important member variable is `channels`, which is an instance of the class `IOChannelList`. `channels` represents a list of the channels which was set before in `ExperimentControl`.

Let us now delve into the class `IOChannel`. It contains the member variables `show` and `enabled`, which again were set in `ExperimentControl`. Another important member variable is `events`, which is an instance of the class `IOEventList`. `events` represents a list of all the events which were set before in `ExperimentControl`.

Let us examine the class `IOEvent`. It contains the corresponding channel, and the other important argument is `timing`. It is an instance of the class `TimingEvent` and it can call its member function `getValue()`, and this will give us the value of `t` at the start of the event. Another important value is `eventSamplingRate` contained in the `QODACEventFormula` (which is a subclass of `IOEvent`). `QODACEventFormula` is also a subclass of the class `EventWaveformFormula`, which contains the formula `f`, which in turn can be evaluated by its member function `evaluate()` to obtain the value of the function (written in ExpWiz) corresponding to that event. Note that the class `formula` contains an instance of the class `FormulaEnvironment`, to whom the notion of time can be given by calling the member function `addVariable`.

Now that the classes, which are interesting for our purposes, have been analyzed, let us move on to the overall working mechanism of the software. The important files are `Generator.cpp`,

3. HIGH-LEVEL CONTROL SOFTWARE

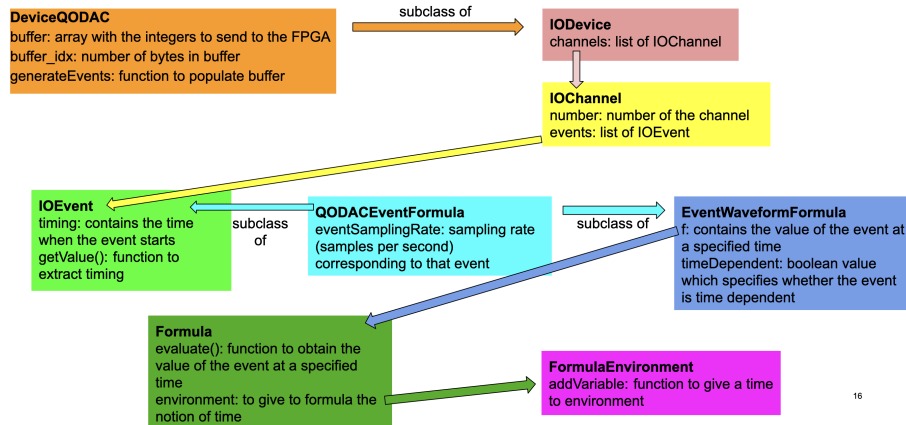


Figure 3.6: Visual representation of the classes of interest in ExpWiz

SDIMain.cpp and Player.cpp. These files working together initialize all the devices which were set up in ExperimentControl, read the information contained in the XML file generated by ExperimentControl and pass all the information to the instances of the various classes representing the devices.

3.2 Understanding the Low-Level Software Implemented in Python

3.2.1 Configuration Flags

The first thing which the low-level software implemented in Python does is to set up some of the settings to be sent to the FPGA. They are collected in a 32-bit configuration flag, called `config_flag`. The parameters are the following:

1. `debug_messages`: it occupies one bit. If it is 1, the debug messages are sent to the PC, otherwise they aren't;
2. `command`: it occupies three bits; if it is 1, which is the value of interest, the data are uploaded to the DDR and the waveforms are played by the FPGA.
3. `play_mode`: it occupies one bit. If it is 0, the FPGA will be in single mode and the waveforms will be played only once, otherwise the waveforms will be played continuously;
4. `trigger_mode`: it occupies two bits. If it is 0 the FPGA will use an internal trigger, if it is 1 the FPGA will use an external trigger, if is 2 the FPGA will use a debug trigger;
5. `clock_mode`: it occupies one bit. If it is 0, the FPGA will use an internal clock, otherwise it will use an external clock;
6. `channel_states`: it occupies 8 bits. If it is set to 11111111 (in the decimal system 255), which is our value of interest, all channels are enabled. Every 1 corresponds to an enabled channel.

All these configuration flags are combined together in one 32-bit doubleword configuration flag called `config_flag`. This is done by summing `debug_messages`, `command` shifted by one bit,

3. HIGH-LEVEL CONTROL SOFTWARE

`play_mode` shifted by 4 bits, `trigger_mode` shifted by 5 bits, `clock_mode` shifted by 7 bits and `channel_states` shifted by 8 bits.

3.2.2 Metadata

After having set the configuration flag, the Python script builds an array containing the metadata to be sent to the FPGA, called `metadata`. The metadata are organized as follows:

1. linearity error compensation: one 32-bit integer per channel; for +10V/-10V voltage references, this value must be 12;
2. initial/safe voltages for analog outputs: one 32-bit integer per channel; for +10V/-10V references, DAC word for 0V is 524288;
3. list of number of events: one 32-bit integer per channel, representing how many events are expected for every channel (1 for one event, 2 for two events etc.);
4. list of sampling periods: one 32-bit integer per event, representing the time period for which one sample will be played. Note: it is one 32-bit integer per event; if, for example, Channel 1 has two events, the metadata must be extended by two 32-bit integers for Channel 1, one with the sampling period of the first event and one with the sampling period of the second event. The value of `sampling_period` has the unit of microseconds;
5. list of number of samples for each event: one 32-bit integer per event, representing how many samples will be stored in the DDR for that specific event. Note: again, it is one value per event. This value is directly dependent on the sampling period and the duration of the event.

3.2.3 Waveform Samples

After having set the metadata, the Python script finally collects the waveform samples. They are collected in an array with the size equal to the sum of all the numbers of samples per event. First they are computed as values in Volts, going from -10V to 10V, then they are passed to the function `voltages_to_dac_words`, which extracts the value the FPGA expects for that voltage (0 for -10, 1048576 for 10, all the values in between are scaled linearly).

Once the samples are put together in the array `waveform_samples`, a new array called `data` is put together, consisting of `config_flag` concatenated with `metadata` and `waveform_samples`. Ultimately `data` is sent to the FPGA.

3.3 Integrating the DeviceQODAC in the Control Software

The main work I have done throughout this project is translating the Python script which sets up the values in the DDR of the FPGA in C++ and integrating it into the control software used in the lab. After having done that, most of the work has consisted of debugging. In this section I am going to explain how I have translated the Python script (with the substantial help of Jeffrey Mohan); after that I am going to provide a quick guide to the procedure that must be followed to make the FPGA play the desired waveforms, and finally, I am going to show the steps I have done towards having a working environment.

3. HIGH-LEVEL CONTROL SOFTWARE

3.3.1 Translating the Low-Level Software Implemented in Python

I will now go over the code written to integrate the FPGA in the control software. The function I have written is `DeviceQODAC::generateEvents`. In the beginning, the function prepares the metadata like it was done in the python script: the vectors `lin_error_comp`, `initial_value` and `num_events` are initialized. Also, `sampling_periods` and `num_samples` are initialized as vectors of vectors; a two-dimensional space is needed because there sampling periods and numbers of samples for every event in every channel. Finally, the vector containing the samples is initialized (`samples`).

After that, the function contains a loop which iterates over every channel. The iteration is done through the aforementioned `channel` list. For every iteration over the channels the values in `lin_error_comp` and `num_events` of the corresponding channel are updated by accessing the data in the channel class. Next, two vectors called `sampling_periods_ch` and `num_samples_ch` are initialized. They will contain the sampling periods and the numbers of samples for every event in the corresponding channel and will be pushed into the matrices `sampling_periods` and `num_samples`.

After that, the function contains a loop which iterates over every event in the corresponding channel. This is done by iterating over the aforementioned `events` list present in every channel class. The variables `timeCurrentEvent` and `timeNextEvent` (which are obtained through the member function `getTimingEvent()`) will give the time span corresponding to that event; note that `timeNextEvent` is read from the next event in the list, while the first event is initialized as NULL and the first `timeCurrentEvent` is initialized as zero, and the first event initialized as NULL is skipped as soon as the first `timeNextEvent` is read; at the end of the iteration corresponding to that event, `timeCurrentTime` takes over the value of `timeNextEvent`. Next, the loop computes the duration corresponding to a single sample in that event as $(\text{timeNextEvent} - \text{timeCurrentEvent}) / \text{eventSamplingRate}$. This value multiplied by `1e6` is pushed in the vector `sampling_periods_ch` (remember that the unit of a sampling period is microseconds).

After that, there is a last loop which iterates over the time, which always increases by the time span corresponding to one sample; this loop evaluates the function which was written in `ExpWiz` and pushes that value into the vector `samples`. This evaluation is done through the function `evaluate()`, which in turn is passed to the function `applyFunction`, the equivalent of `voltage_to_dac_words` in the python script. The loop keeps track of the number of samples added to `samples`, and updates the vector `num_samples_ch`.

To sum up, this loop updates the information which will go into the metadata and extracts the samples out of the information contained in the classes, storing them in the vector `samples`. It also handles two edge cases: the first one is when an event is time-dependent, and in this case, we save only one sample for the whole event; the second one is the last event. In this case, what changes with respect to the other events is that, instead of taking the time of the next event as upper bound of its time span, we take the time `maxTime`, which is one of the arguments given to the function.

We have now arrived at the end of the loop which iterates over the channels. Once we have collected all the information for what we wrote on `ExperimentControl`, we can finally set up the configuration flag and the metadata as it was done in the Python script. First of all, we initialize the variables `debug_messages`, `command`, `play_mode`, `triggerExternal`, `clockExternal` and `channel_states` and combine them together in the 32 bytes integer `config_flag` exactly as it was done in the Python script. Subsequently, we collect the final data to be sent to the FPGA. This is done with the array `buffer`, which is a member variable of the class `DeviceQODAC`; first of all its size is calculated, then the necessary space is allocated, and finally it is populated with

3. HIGH-LEVEL CONTROL SOFTWARE



Figure 3.7: Visual representation of the function `generateEvents`

the values in `config_flag`, `lin_comp_errorr`, `intial_value`, `num_events`, `sampling_periods`, `num_samples` and `samples`. Every time a value is pushed into the array `buffer`, the integer `buffer_idx` (which is another member variable of the class `DeviceQODAC`) is increased by one; finally, it is multiplied by four, because it represents the number of bytes contained in the buffer.

3.3.2 Quick Guide to the Setup

I will now provide a quick step guide to the setup, and explain how to make the FPGA play the desired waveforms with the QO software. The FPGA must be connected to the power supply, and also must be connected to the computer via the USB cable (to program it from Vitis) and the Ethernet cable (to send the data).

1. step 1: launch `ExperimentControl`; I have already explained the functioning of the application in 3.1.1.
2. step 2: open RX, and click on play (the green button). This will start the code, which now will expect a file from `ExperimentControl` to set the classes in the software up. This will make the application `ExperimentRunner` start. RX is the application where we make the code start from, but we can modify it on VSCode without any problems; when opening RX, we will see a notice that the code has been changed, and we just need to accept the chngement.
3. step 3: program the FPGA; launch Vitis, open the correct environment and program the FPGA. Programming it means making it wait for data, and the FPGA is now in that state.
4. step 4: hit the yellow button on `ExperimentControl` as in Figure 3.8; this will give the file coming from the whole setup on `ExperimentControl` to the control software. Now, if you look at `ExperimentRunner` you will see continuous messages being printed; the software sends data to the FPGA over Ethernet continuously, precisely it sends the data, waits for a period of time corresponding to the total duration of the events and sends the data again (the status of the FPGA does not matter). If everything worked fine, the FPGA should

3. HIGH-LEVEL CONTROL SOFTWARE

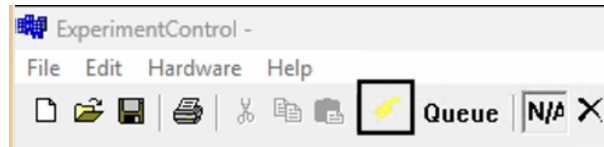


Figure 3.8: Yellow button to hit to make pass the setup to the code

receive the first packet of data, go to the status of play (meaning it continuously plays the waveforms), and not accept data coming from the PC anymore. This is the reason why you need to make sure you closed ExperimentRunner after the FPGA is playing the waveforms; if you don't close it, the computer will keep sending the data to the FPGA (which will not accept it), but if you reprogram the FPGA to make it wait for data (step 3), it will accept the data coming from the computer and possibly corresponding to what you programmed before.

5. step 5: close ExperimentRunner (reason explained in step 4).

Finally, if you want to run the Python code, open the anaconda terminal; look for the file `TestProgram.py` in ExpWiz and copy its path in the anaconda terminal. Run it by typing the command `python TestProgram.py` on anaconda. This will launch the program and the computer will directly send data to the FPGA, therefore you need to make sure the FPGA is expecting the data (step 3).

3.3.3 Steps Towards a Working Environment

In this section I will explain the debugging steps and the process which led to a a working environment.

The first step was translating the Python code in C++ and integrating it into the control software. Of course the code was not perfect in the beginning, and I had to debug it. I have done that by setting small experiments up, corresponding to the cases I wanted to test, then printing the vector `buffer` to another file, finally inspecting this printed vector and checking whether it was as expected (also by comparison with the Python script). After this step, I was almost sure that the C++ code had no flaws and was ready to test it directly on the FPGA.

The first time I tried to run the C++ code on the FPGA something very strange happened: when reading the messages the FPGA was sending to the computer, I saw that the FPGA was continuously opening and closing communication sessions with the computer. What happened, for example, could be the following: the FPGA accepts a first connection, writes to the memory the configuration flag and the metadata, then the connection is lost, a new connection is accepted, the FPGA writes to the memory the data corresponding to Channel 1, then the connection is lost, a new one is accepted and so on. This behavior was very different from the behavior observed with the Python script, but nevertheless I wanted to see if the FPGA was able to play something.

To do that I connected the FPGA to the oscilloscope and observed the played waveforms. After some attempts, I noticed that the FPGA played only the data corresponding to the beginning of Channel 1; then, no matter what should come afterwards, the FPGA played that same data over and over again (always corresponding to the beginning of Channel 1). Since I was sure the C++ code was generating the correct buffer, because I had "manually" inspected before, I came to the conclusion that something was wrong with the communication between computer

3. HIGH-LEVEL CONTROL SOFTWARE

and FPGA, also considering what was happening with the messages coming from the FPGA (continuous closing and opening of the session).

To verify my conclusion, I inspected the data saved in the memory of the FPGA. To do that, I installed Vivado and Vitis on the QO computer, and ran some sessions in the debug mode. I would program the FPGA from Vitis in debug mode, launch the C++ code from the computer, place a breakpoint where the FPGA would start to play the waveforms and inspect the data saved in the memory of the FPGA. After having done that, I could read the memory of the FPGA: I saw that the beginning of the first channel was written correctly, then, for some unknown reason, at some point in the middle of the memory reserved for Channel 1, instead of having the correct data, there were again the configuration flag, the metadata and the data corresponding to the beginning of Channel 1. This repetition of data occurred other times in later memory addresses. This was totally unexpected, because the configuration flag and the metadata are supposed to be saved somewhere else in the memory of the FPGA. However, this confirmed my hypothesis that something was wrong with the communication between computer and FPGA, and not with the C++ code.

At this point, we thought that the reason why the data repeated in the FPGA was the following: the computer was sending all the data over and over again, and the FPGA would continuously accept it, even without having finished to write everything to the memory. This meant that the FPGA would start to write to the memory, and before having finished, accept new data from the computer and write the newly accepted data instead of the old ones.

At this point, we saw two different debugging strategies in front of us. The first one would be to change the C++ code in the control software to slow down the process of sending: I just changed the function which wrote the bytes to the FPGA to make it sleep for a certain amount of time after having sent all the data, but this approach did not work: after the end of the data corresponding to the first packet, the data I saw in the FPGA were data corresponding to another packet, not the second one but a completely different one; I did not investigate further because it was clear to me that this was not a viable solution. The second strategy we adopted was to modify the C code in the FPGA. When the FPGA accepted the first chunk of data, it shouldn't be open to accepting new data, but close any new connection. This solution did not work, because the FPGA, as I noticed later, was already programmed to not accept new data before all the data had been sent.

However, inspecting the code inside of the FPGA was still beneficial because we noticed a variable, called `payload`, which was a pointer to the data received in one packet from the computer. Its size, defined by the constant `tcp_mss`, was equal to 1400 and represented the number of bytes the FPGA was supposed to receive in one packet of data. We also noticed that the number of bytes written to the memory of the FPGA was exactly 1400; after these 1400 bytes, there was this phenomenon of repeating of the data. We came to the conclusion that the following was happening: the FPGA receives the first packet of data, for some unknown reason something goes wrong with the communication, the computer stops to send the data, the FPGA does not have all the data it expects (this information is contained in the metadata) and still waits for the data, the computer opens a new connection and sends a new packet of data, the FPGA accepts the first chunk of data (which explains why the data are repeated in the memory), and this sequence goes on until the FPGA has received all the data it expected. A posteriori, this interpretation was the correct one. I note that this conclusion was different from the first one, when we thought that the problem was only due to the FPGA not keeping up with the pace of the computer.

At this point we adopted a new strategy: we thought that the problem was connected to

3. HIGH-LEVEL CONTROL SOFTWARE

the size of the packets. In fact, the standard value of `tcp_mss` was 1460, while for some reason in the FPGA it was set to 1400. We argued that the computer was expecting the FPGA to receive the data in packets of 1460 bytes, and since this was not happening, it would close the communication. Therefore, we modified the variable `tcp_mss` to 1460, but this approach did not work, the FPGA would now just receive one packet with the size of 1460 bytes. Plus, the Python script does not work with `tcp_mss` set to 1460, as it does not end the communication and does not successfully transfer the last packet of data. Next, we tried to set it to 1444; in this case the Python script worked, but the control software still did not work. We left `tcp_mss` set to 1444. One could think that a solution would be just to set this variable to the wished size, but this approach would not work because the Ethernet communication protocol does not allow to exchange packets larger than 1500 bytes.

This debugging strategy was nevertheless beneficial, because I could understand where the mistake was likely happening: something was wrong with the communication between the computer and the FPGA. The next strategy we adopted was to further investigate the code inside of the FPGA, and we tried to erase the part of code that raised an exception. This did not work, but at least we understood that the problem was not coming from the FPGA, but rather from the computer. All these strategies might not have worked, but they allowed us to narrow down the problem and understand where it was coming from.

The next strategy we adopted was to download the application Wireshark and to inspect the exchange of information over the Ethernet between the FPGA and the computer, also to better understand what was causing the issue. I compared the messages printed when running the Python script and when running the control software. I will now briefly explain the meaning of the messages I have been able to see.

Figure 3.9 shows the exchange of messages when running the Python script. Let us analyze the entries we can see in a row: the first one is referred to how many messages have been exchanged since Wireshark had started to read the communication; the second one is referred to the time which has passed since Wireshark had started to read the communication; the third and the fourth ones are respectively the address of the sender and the address of the receiver, 192.168.1.100 for the computer and 192.168.1.10 for the FPGA; next, we see the entry "TCP" and another number (not of interest for our purposes); the seventh one signals the ports that are involved in the communication (50882 in Figure 3.9 for the computer, 5001 for the FPGA); next, we see numbers corresponding to the labels `Seq`, `Ack`, `Win`, `Len`, `MSS`, `WS` and `SACK_PERM`. Of these last entries the only interesting one is `Len`, which represents the number bytes which are being sent in the corresponding exchange. All the other entries are not interesting for our purposes.

Let us now analyze the exchange of messages we can observe when running the Python script. The first two messages (the gray ones) are the initialization of the communication: the first one is sent from the computer, which starts the communication, and the second one is the FPGA's acknowledgement. Then we see a series of blue messages. The first one comes from the computer; it shows that the computer is starting to send the data (it is still a "protocol message", as its length is zero). After that, we see messages always coming together in couples: the computer sends a packet of data and the FPGA sends it back to the computer. Both packets have the size of 1444 in the first exchange, which is the value we previously gave to the variable `tcp_mss`; this also shows that the size of the payload does not represent a problem for the communication, as the control software can adapt the size of the packets it is sending. This exchange of messages is continued until all the data have been sent.

Let us now analyze the exchange of messages we can observe when running the control

3. HIGH-LEVEL CONTROL SOFTWARE

24	40.091045	192.168.1.100	192.168.1.10	TCP	66	50882 → 5001	[SYN]	Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
25	40.091319	192.168.1.10	192.168.1.100	TCP	60	5001 → 50882	[SYN, ACK]	Seq=0 Ack=1 Win=2048 Len=0 MSS=1444
26	40.091348	192.168.1.100	192.168.1.10	TCP	54	50882 → 5001	[ACK]	Seq=1 Ack=1 Win=64240 Len=0
27	40.091652	192.168.1.100	192.168.1.10	TCP	1498	50882 → 5001	[ACK]	Seq=1 Ack=1 Win=64240 Len=1444
28	40.214246	192.168.1.10	192.168.1.100	TCP	1498	5001 → 50882	[ACK]	Seq=1 Ack=1445 Win=2048 Len=1444
29	40.214271	192.168.1.100	192.168.1.10	TCP	1498	50882 → 5001	[ACK]	Seq=1445 Ack=1445 Win=64980 Len=1444
30	40.214996	192.168.1.10	192.168.1.100	TCP	81	5001 → 50882	[ACK]	Seq=1445 Ack=2889 Win=2048 Len=27
31	40.215019	192.168.1.100	192.168.1.10	TCP	1498	50882 → 5001	[ACK]	Seq=2889 Ack=1472 Win=64953 Len=1444
32	40.221542	192.168.1.10	192.168.1.100	TCP	140	5001 → 50882	[ACK]	Seq=1472 Ack=4333 Win=2048 Len=86
33	40.221563	192.168.1.100	192.168.1.10	TCP	1498	50882 → 5001	[ACK]	Seq=4333 Ack=1558 Win=64867 Len=1444
34	40.221969	192.168.1.10	192.168.1.100	TCP	60	5001 → 50882	[ACK]	Seq=1558 Ack=5777 Win=2048 Len=0
35	40.221978	192.168.1.100	192.168.1.10	TCP	1498	50882 → 5001	[ACK]	Seq=5777 Ack=1558 Win=64867 Len=1444
36	40.222356	192.168.1.10	192.168.1.100	TCP	60	5001 → 50882	[ACK]	Seq=1558 Ack=7221 Win=2048 Len=0
37	40.222363	192.168.1.100	192.168.1.10	TCP	1498	50882 → 5001	[ACK]	Seq=7221 Ack=1558 Win=64867 Len=1444
38	40.229097	192.168.1.10	192.168.1.100	TCP	140	5001 → 50882	[ACK]	Seq=1558 Ack=8665 Win=2048 Len=86
39	40.229107	192.168.1.100	192.168.1.10	TCP	1498	50882 → 5001	[ACK]	Seq=8665 Ack=1644 Win=64781 Len=1444
40	40.229486	192.168.1.10	192.168.1.100	TCP	60	5001 → 50882	[ACK]	Seq=1644 Ack=10109 Win=2048 Len=0
41	40.229493	192.168.1.100	192.168.1.10	TCP	1498	50882 → 5001	[ACK]	Seq=10109 Ack=1644 Win=64781 Len=1444
42	40.229856	192.168.1.10	192.168.1.100	TCP	60	5001 → 50882	[ACK]	Seq=1644 Ack=11553 Win=2048 Len=0
43	40.229863	192.168.1.100	192.168.1.10	TCP	1498	50882 → 5001	[ACK]	Seq=11553 Ack=1644 Win=64781 Len=1444
44	40.237008	192.168.1.10	192.168.1.100	TCP	140	5001 → 50882	[ACK]	Seq=1644 Ack=12997 Win=2048 Len=86
45	40.237043	192.168.1.100	192.168.1.10	TCP	1498	50882 → 5001	[ACK]	Seq=12997 Ack=1730 Win=64695 Len=1444
46	40.237457	192.168.1.10	192.168.1.100	TCP	60	5001 → 50882	[ACK]	Seq=1730 Ack=14441 Win=2048 Len=0
47	40.237470	192.168.1.100	192.168.1.10	TCP	1498	50882 → 5001	[ACK]	Seq=14441 Ack=1730 Win=64695 Len=1444
48	40.237846	192.168.1.10	192.168.1.100	TCP	60	5001 → 50882	[ACK]	Seq=1730 Ack=15885 Win=2048 Len=0
49	40.237855	192.168.1.100	192.168.1.10	TCP	1498	50882 → 5001	[ACK]	Seq=15885 Ack=1730 Win=64695 Len=1444

Figure 3.9: Communication over the Ethernet when running the Python script

24	37.623541	192.168.1.100	192.168.1.10	TCP	66	51252 → 5001	[SYN]	Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
25	37.623718	192.168.1.10	192.168.1.100	TCP	60	5001 → 51252	[SYN, ACK]	Seq=0 Ack=1 Win=2048 Len=0 MSS=1444
26	37.623752	192.168.1.100	192.168.1.10	TCP	54	51252 → 5001	[ACK]	Seq=1 Ack=1 Win=64240 Len=0
27	37.623786	192.168.1.100	192.168.1.10	TCP	1498	51252 → 5001	[ACK]	Seq=1 Ack=1 Win=64240 Len=1444
28	37.649591	192.168.1.100	192.168.1.10	TCP	66	51253 → 5001	[SYN]	Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
29	37.746354	192.168.1.10	192.168.1.100	TCP	1498	5001 → 51252	[ACK]	Seq=1 Ack=1445 Win=2048 Len=1444
30	37.746388	192.168.1.100	192.168.1.10	TCP	54	51252 → 5001	[RST, ACK]	Seq=1445 Ack=1445 Win=0 Len=0
31	37.746401	192.168.1.10	192.168.1.100	TCP	60	5001 → 51253	[SYN, ACK]	Seq=0 Ack=1 Win=2048 Len=0 MSS=1444
32	37.746436	192.168.1.100	192.168.1.10	TCP	54	51253 → 5001	[ACK]	Seq=1 Ack=1 Win=64240 Len=0
33	37.746535	192.168.1.100	192.168.1.10	TCP	1498	51253 → 5001	[ACK]	Seq=1 Ack=1 Win=64240 Len=1444
34	37.771148	192.168.1.10	192.168.1.100	TCP	194	5001 → 51253	[ACK]	Seq=1 Ack=1 Win=2048 Len=140
35	37.771206	192.168.1.100	192.168.1.10	TCP	54	51253 → 5001	[RST, ACK]	Seq=1445 Ack=141 Win=0 Len=0
36	37.771222	192.168.1.10	192.168.1.100	TCP	60	5001 → 51253	[ACK]	Seq=141 Ack=1445 Win=2048 Len=0
37	37.774506	192.168.1.100	192.168.1.10	TCP	66	51254 → 5001	[SYN]	Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
38	37.782178	192.168.1.10	192.168.1.100	TCP	60	5001 → 51254	[SYN, ACK]	Seq=0 Ack=1 Win=2048 Len=0 MSS=1444
39	37.782238	192.168.1.100	192.168.1.10	TCP	54	51254 → 5001	[ACK]	Seq=1 Ack=1 Win=64240 Len=0
40	37.782340	192.168.1.100	192.168.1.10	TCP	1498	51254 → 5001	[ACK]	Seq=1 Ack=1 Win=64240 Len=1444
41	37.794531	192.168.1.10	192.168.1.100	TCP	194	5001 → 51254	[ACK]	Seq=1 Ack=1 Win=2048 Len=140
42	37.794576	192.168.1.100	192.168.1.10	TCP	54	51254 → 5001	[RST, ACK]	Seq=1445 Ack=141 Win=0 Len=0
43	37.805996	192.168.1.100	192.168.1.10	TCP	66	51255 → 5001	[SYN]	Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
44	37.814164	192.168.1.10	192.168.1.100	TCP	60	5001 → 51255	[SYN, ACK]	Seq=0 Ack=1 Win=2048 Len=0 MSS=1444
45	37.814205	192.168.1.100	192.168.1.10	TCP	54	51255 → 5001	[ACK]	Seq=1 Ack=1 Win=64240 Len=0
46	37.814273	192.168.1.100	192.168.1.10	TCP	1498	51255 → 5001	[ACK]	Seq=1 Ack=1 Win=64240 Len=1444
47	37.826666	192.168.1.10	192.168.1.100	TCP	194	5001 → 51255	[ACK]	Seq=1 Ack=1 Win=2048 Len=140
48	37.826666	192.168.1.10	192.168.1.100	TCP	60	5001 → 51255	[ACK]	Seq=141 Ack=1445 Win=2048 Len=0
49	37.826703	192.168.1.100	192.168.1.10	TCP	54	51255 → 5001	[RST, ACK]	Seq=1445 Ack=141 Win=0 Len=0

Figure 3.10: Communication over the Ethernet when running the control software (events of 1 ms)

3. HIGH-LEVEL CONTROL SOFTWARE

49	34.529278	192.168.1.100	192.168.1.10	TCP	66	63685 → 5001	[SYN, Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
50	34.529568	192.168.1.10	192.168.1.100	TCP	60	5001 → 63685	[SYN, ACK] Seq=0 Ack=1 Win=2048 Len=0 MSS=1444
51	34.529708	192.168.1.100	192.168.1.10	TCP	54	63685 → 5001	[ACK] Seq=1 Ack=1 Win=64240 Len=0
52	34.529863	192.168.1.100	192.168.1.10	TCP	1498	63685 → 5001	[ACK] Seq=1 Ack=1 Win=64240 Len=1444
53	34.536460	192.168.1.10	192.168.1.100	TCP	194	5001 → 63685	[ACK] Seq=1 Ack=1 Win=2048 Len=140
54	34.536509	192.168.1.100	192.168.1.10	TCP	54	63685 → 5001	[RST, ACK] Seq=1445 Ack=141 Win=0 Len=0

Figure 3.11: Communication over the Ethernet when running the control software (events of 30 s)

software shown in Figure 3.10. Already at a first glimpse we can see a noticeable difference and imagine that something is going wrong. The first four messages are exactly the same that we see when running the Python script: the computer initializes the communication, the FPGA acknowledges it, the computer send the first "data start" message and the first packet of data. At the fifth message we already see a difference: the computer has opened a new connection (we can see this not only from the color of the message, but also from the number representing the port of the computer, which has now changed from 51252 to 51253). Only the sixth message is the FPGA recognizing that the first packet has arrived . Note that this message is sent to the previous port. This triggers the seventh message (the red one), which is sent by the computer (again by the port 51252) and shows that the computer has now closed the connection because of an error. Then we see the FPGA accepting the new connection from port 51253 and the computer sending a new packet of data (again, the data corresponding to the beginning of the array `buffer`). Now the FPGA does not acknowledge the data coming from the computer: notice that the length of the message is 140, not 1444, meaning that it must be some other kind of message. After that, the computer issues an error and closes the connection from the port 51253. After that, we see this behavior going on.

We could draw some important conclusions from the observation of the exchange of messages. First of all, the error is issued by the computer. Secondly, the computer keeps opening new sessions where it sends the data again from the beginning, and the FPGA accepts it, but it interprets it as new packets (not as the first packet repeated). At this point, I noticed an interesting fact: the computer opens new sessions independently from the messages received from the FPGA, and independently from the status of the communication (whether or not a mistake has occurred). In fact, the fifth message is sent even though no mistake has happened yet. I noticed that the rate at which the computer tries to send new data corresponds to the total duration of the events; later this has been explained to me, but at the time I was unaware of that and I only made a guess. We now took our first hypothesis back into consideration: it was possible that the FPGA was not keeping up to the pace of the computer, which was opening new sessions before the preceding ones were over, and this was causing a intertwining between messages sent to an by the various ports of the computer, which in turn raised the error messages from the computer.

With this new idea, and together with the awareness that the rate at which the computer was opening new sessions was related to the duration of the events, what came to mind was just to program longer events on `ExperimentControl`. Before I was sending data corresponding to a duration of one millisecond, so I tried to send data corresponding the to a duration of 30 seconds, so that the FPGA would have all the time to write to its memory before a new connection was opened. This approach proved to be wrong, but it revealed some information which later proved to be essential to fix the problem.

Let us analyze the exchange of messages we can observe when running the control software shown in Figure 3.11. We see the usual messages of initialization of a connection; next, the computer starts to send the data but again the FPGA does not acknowledge it, and instead sends another kind of message with the length of 140. Note that no new connection has started

3. HIGH-LEVEL CONTROL SOFTWARE

in between, meaning that the intertwining of communication between different ports is not the problem triggering the error in this case. After this non-expected message, the computer raises an error and stops the session. This leaves us in the following situation: the computer has stopped sending the data and has sent only one packet, while the FPGA has not received all the data it was expecting. It is not shown in the picture because in between there were many other messages not related to the communication over the Ethernet, and it would be misleading, but exactly 30 seconds after this exchange a new exchange from another port of the computer would happen, with the same exact results (computer raising an error after the feedback message of the FPGA). The FPGA still can't play the waveforms, because not all the data it was expecting have been written to its memory yet. This exchange would happen over and over again until the FPGA has collected an amount of data corresponding to the amount of data it was expecting; at this point the FPGA would start playing the waveforms and not accept data anymore from the computer. Of course, all the data received by the FPGA would correspond to the first packet of data sent every time by the computer, i.e. the configuration flag, the metadata and the beginning of the Channel 1 samples.

The next step I took was to analyze over Wireshark the exchange of messages between the control software and a working device. What I could observe were messages very similar to the ones observed when running the Python code. This made me realize that there was nothing wrong with how the Ethernet communication was handled in the control software, so it excluded another possible source of the mistake.

What led us to the solution of the problem was a comparison between the exchange when running the Python script and when running the control software. A posteriori, I recognized two reasons why the computer raises errors. The first error in Figure 3.10 was due to the intertwining of connections between different ports: the FPGA was recognizing the data corresponding to a previous communication, while the computer had already started a new one. This is fixed by making the events last longer. The second error in Figure 3.10 and the errors in Figure 3.11 were due to the computer receiving a feedback message it is not expecting. We did not come to that realization by then; we thought it was only a timing issue, because the FPGA was taking too long to send the feedback message.

To solve the second kind of error I explained, we deactivated the debug messages sent by the FPGA, thinking that they were increasing the time the FPGA needed to respond. Now, I realize that this was only a lucky guess; actually, what was probably happening is that the message with length 140 was a debug message sent by the FPGA, not expected by the computer, which in turn would raise an error.

With this change in the settings of the FPGA, we finally did not see the error messages from the computer anymore. However, not all the data were arriving yet, and the phenomenon of repeating of packets was still present. But now, with the communication between computer and FPGA finally working, the situation was much better, and the solution was just half an hour away: I noticed that the amount of bytes received by the FPGA before repeating was not fixed to one packet (1400 bytes) anymore, but it was always around one fourth of the total expected data. The reason was that `buffer_idx`, member variable of the class `DeviceQODAC`, was not meant to be number of integers the computer was supposed to send, but the number of bytes; therefore, since `buffer_idx` was not multiplied by 4, the computer would always send an amount of data corresponding to one fourth of the real amount. The fix was just multiplying `buffer_idx` by 4.

After all these steps, we had finally integrated the FPGA in the control software successfully.

3. HIGH-LEVEL CONTROL SOFTWARE

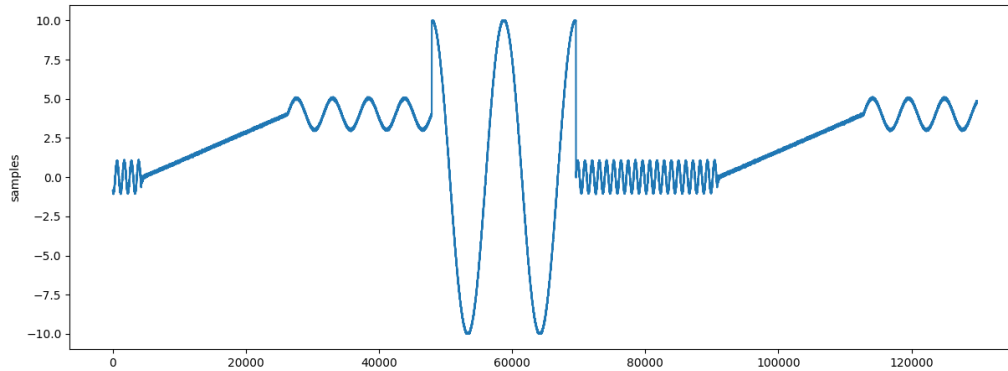


Figure 3.12: various concatenated waveforms (plotted from the .CSV file)

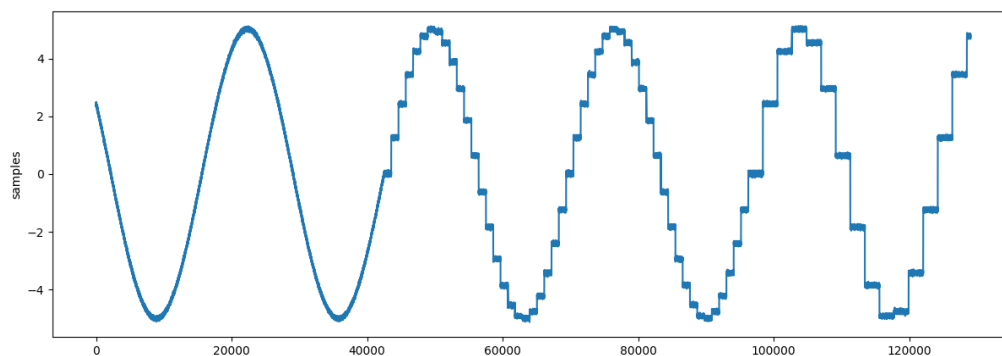


Figure 3.13: sine wave (plotted from the .CSV file)

3.3.4 Playing the Desired Waveforms with the FPGA, Programmed on *ExperimentControl*

In this final section I will show some of the waveforms I was able to play with the FPGA after having debugged the control software and the C code running on the FPGA.

The first waveform is just a concatenation of different functions, played with a total of 4 events. The first one is a ramp lasting for one second and going from 0 to 4 volts; the second one is a sine wave lasting for one second, with a frequency of 4 Hz, amplitude of 1 V and offset of 4 V; the third one is a cosine wave lasting for one second, with frequency of 2 Hz, amplitude of 10 V and offset of 0 V; the last one is another sine wave lasting for one second, with frequency of 20 Hz, amplitude of 1 V and offset of 0 V. Note how the FPGA is able to play discontinuous waveforms.

The second waveform demonstrates how the chosen sampling rate can influence the played waveform. In this case we have a sine wave with a frequency of 2 Hz, which is played with three different sampling rates (meaning how many samples the FPGA produces per second; this quantity is not related to the actual frequency of the signal, but it would be better to choose it accordingly, meaning that longer lasting signals whose derivative is small can have lower sampling rates, while shorter signals with a larger derivative should have a higher sampling rate). The first one has a sampling rate of 500 Hz, the second one has a sampling rate of 50 Hz, the third one has a sampling rate of 20 Hz. Choosing different sampling rates implies different trade-offs: lowering the sampling rate allows to have more space for the samples on the memory of the FPGA, while making the played waveform "rougher".

Conclusion

This thesis introduces several protection enhancements of the low-noise eight-channel arbitrary waveform generator against ESD events. Firstly, the robustness of TVS diodes has undergone thorough testing. These diodes were integrated into the PCB housing the LVDS chips, ensuring direct safeguarding of the chips.

Furthermore, an enclosure has been specifically designed to offer initial protection from human-induced ESD occurrences and to be able to later upgrade to a metal enclosure, aimed at further diminishing electromagnetic interference from the environment.

Additionally, it has been demonstrated that multiple FPGAs can be interconnected to a single control PC and synchronized via a unified trigger input. This synchronization enables simultaneous signal transmission from multiple FPGAs, showcasing the scalability possibilities of the system.

At the end of the project the FPGA has been successfully integrated inside of the quantum optics laboratory's control software. It has been demonstrated that the data sent to the FPGA can be collected with C++ code.

Finally, the Ethernet communication between the computer and the FPGA has been investigated, and all this information has been summarized in this project report.

Possible Improvements

5.1 High-Level Control Software

The FPGA has been successfully integrated into the control software, however there are still two main flaws which could be fixed:

1. **Setting up the configuration flag from ExperimentControl:** at the moment, the configuration flag, which determines the mode which the FPGA will operate in, is hard coded and cannot be set from ExperimentControl. However, it has to be emphasized that the configuration flag already corresponds to the way the FPGA is to be used within quantum optics experiments.
2. **Taking into consideration the case where not all channels are enabled:** currently, the control software always assumes that all channels are enabled. I did not take the first case into account because this case was also not contemplated in the Python script. However, this second problem is minor, and its solution can be implemented very fast.

5.2 Hardware and System Performance Evaluation

Further improvements could be made to refine the system:

1. **Integration of LEDs on the PCB:** Incorporating LEDs to visually indicate the presence of 24V and 5V power connections provides users with immediate feedback on power status, enhancing usability and troubleshooting efficiency.
2. **Testing with TVS Diode:** Even though the TVS diode should have no effect on the SPI signals it is still necessary to conduct tests to ensure that signal quality remains unaffected.
3. **Metal Enclosure Production:** Transitioning to a metal enclosure offers superior protection against environmental electromagnetic noise, thereby enhancing the overall durability and signal integrity of our system.
4. **Temperature Durability of the Enclosure:** Since PLA starts to lose its form and become pliable at temperatures above 60°C, tests about the operating temperature of the FPGA and PCB need to be conducted to ensure the temperature do not go beyond that threshold. And if it does, to utilize the fan connectors of the PCB to redirect the heat.
5. **Utilization of Plastic Screws:** Substituting metal screws with plastic counterparts serves to further isolate the enclosure, minimizing the risk of ESDs.

5. POSSIBLE IMPROVEMENTS

6. **Perfect Fit for Ethernet Connectors** To further minimize the enclosure's dimensions, the Ethernet connectors can be pushed into the wall. This requires the opposite wall to be detachable and screwable to manage the installation.
7. **Python Code Optimization for Multiple FPGAs:** Refining our Python code to allow the use of two FPGAs while only running a single script, which simplifies system management. Afterwards incorporating it into the C++ based control software.

Acronyms

- AWG** arbitrary waveform generator. 1, 2
- BNC** Bayonet Neill–Concelman. 1
- CLK** Clock Signal. 4
- CS** Chip Select. 4
- DAC** Digital-to-Analog Converter. 1, 16
- ESD** Electrostatic discharge. 2, 7–12, 14, 34, 35
- FPGA** Field Programmable Gate Array. 1, 2, 14–17, 34, 35
- GND** Ground. 12
- IP** Internet Protocol. 16
- LVDS** low-voltage differential signaling. 1, 2, 4, 5, 7–9, 11, 13, 34
- MOSI** Main Out, Sub In. 4, 5
- O.L** Open Loop. 4, 5
- PCB** Printed Circuit Board. 1, 2, 4, 7, 8, 11, 12, 14, 15, 17, 34, 35
- SPI** single-ended serial peripheral interface. 1, 4, 11, 12, 17, 35
- TCP** Transmission Control Protocol. 1, 17
- TVS** Transient Voltage Suppressor. 7, 9, 11, 12, 34, 35
- VCC** Common Collector Voltage. 12

Bibliography

- [1] Shijia Chen. “A Low-Noise Eight-Channel Arbitrary Waveform Generator for Quantum Gas Experiments”. en. Master Thesis. Zurich: ETH Zurich, 2023. DOI: [10.3929/ethz-b-000634723](https://doi.org/10.3929/ethz-b-000634723).
- [2] Bahadir Dönmez. “Towards a Distributed Low-Noise Signal Generation System for Quantum Gas Experiments. A Scalable FPGA-Based Design”. en. Master Thesis. Zurich: ETH Zurich, 2022. DOI: [10.3929/ethz-b-000594182](https://doi.org/10.3929/ethz-b-000594182).
- [3] Wikipedia contributors. *ESD simulator* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 2-March-2024]. 2023. URL: https://en.wikipedia.org/w/index.php?title=ESD_simulator&oldid=1146438610.
- [4] Phil Salmony. *ESD Protection Basics: TVS Diodes*. 2023. URL: <https://resources.altium.com/p/esd-protection-basics-tvs-diodes> (visited on 03/03/2024).
- [5] Zachariah Peterson. *PCB Design Guidelines: Using TVS Diode Transient Protection*. 2023. URL: <https://resources.altium.com/p/pcb-design-guidelines-using-tvs-diode-transient-protection> (visited on 03/03/2024).
- [6] Ametek. *esd NX30 ESD generator*. 2024. URL: https://cdn11.bigcommerce.com/s-v339xoy611/images/stencil/1280x1280/products/842/1278/esd_NX30__26704.1628057325.jpg?c=1 (visited on 03/03/2024).
- [7] STMicroelectronics. *SOT-666 TVS Diode*. 2021. URL: <https://www.mouser.ch/images/stmicroelectronics/images/sot-666.jpg> (visited on 03/03/2024).
- [8] STMicroelectronics. *USBLC6-2*. Datasheet. Image taken from page 1, Functional diagram (top view). 2021. URL: <https://www.mouser.ch/datasheet/2/389/cd00050750-1796222.pdf> (visited on 03/03/2024).



Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

INTEGRATING FPGA-BASED SIGNAL GENERATION
HARDWARE INTO EXPERIMENT CONTROL SYSTEM

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):

BUGNON
LAZZARONI

First name(s):

CÉDRIC
LORENZO

With my signature I confirm that



- I have committed none of the forms of plagiarism described in the '[Citation etiquette](#)' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

ZURICH, 24.03.2024

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.