Master Quantum Engineering

Year 1 Internship Report

February - June 2024

Software & Hardware Migration for Analog & Digital I/O in Rydberg-Assisted Quantum Engineering of Light Experiment

Ningshun Chen

Supervised by

Alexei Ourjoumtsev & Sébastien Garcia

Introduction

Currently in Master 1 Quantum Engineering at École Normale Supérieure — Université Paris Sciences et Lettres, I carried out my second semester internship in the Quantum Photonics team of Collège de France. The team is directed by Alexei Ourjoumtsev, and is composed of CNRS researcher Sébastien Garcia, PhD students Valentin Magro and Antoine Covolo, and two other interns Mathieu Girard and Yi Li.

The central project of the research team is the "quantum engineering of light in an ensemble of Rydberg atoms in a cavity." The experimental apparatus of this endeavour aims to create coherent interactions between optical photons and a cold cloud of Rubidium located inside the cavity. This setup allows the team to control a superposition state between a Rydberg excitation and a photon in a coherent fashion [7].

The goal of this internship was to facilitate and contribute to the software migration process from LabVIEW, a proprietary language used in the experiment for programming applications for digital and analog inputs/outputs (I/O), to open-source languages such as Python or C. A successful migration away from LabVIEW that eliminates its usage in totality has the potential to increase code development efficiency, make digital and analog I/O processes more adaptable, and drastically reduce the cost of experimental operations.

Table of Contents

Chapter 1

The Experiment

This chapter presents a short theoretical background of the experiment, with an introduction to Rydberg atoms and their properties, as well as an overview of the experimental setup. It concludes with a demonstration of my research internship's central problem, which revolved around migrating software programs and hardware devices used in the experiment for digital and analog I/O purposes.

1.1 General Presentation

1.1.1 Interactions Between Photons & Rydberg Atoms

Photons don't interact with each other in free space, which posses a problem if we want to create logical gates with qubits using optical light. In order to have strong and coherent interactions between photons, it's possible to use an intermediary material thanks to non-linearity of light-matter interaction. In effect, if one atom absorbs a photon, it will not be able to absorb ~~another~~ one without a re-emission. A second photon arriving just after a photon absorption will experience a much higher potential compared to the first photon. Intuitively, we can have a single atom or molecule in free space and shoot photons at it, which unfortunately yields a low probability of absorption and random directions of spontaneous emission. This implies that the information about the emitted photon is lost, which is undesirable if we want to create a quantum logic gate having the property of reversibility.

To circumvent such undesired effects, the Quantum Photonics team in which I did my internship chose to use a cloud assembly of N Rubidium 87 atoms coupled to Rydberg states that is confined within a cavity of medium-finesse. A cavity confinement gives control over the direction of photon re-emission, and the Rydberg state introduces non-linearity that prevents the absorption of a second photon. A very fine cavity increases the probability of absorption, but introduces problems ~~such as loss in the cavity, mirror quality, and beam size~~. Thus a cloud assembly of N atoms for which the coupling is multiplied by $\sqrt{N}$ was preferred as it permitted a larger cavity with medium finesse, making the experiment more swiftly repeatable.

Rydberg atoms are atoms with excited quantum states close to ionization, i.e. with ~~principle~~ quantum number $n \gg 1$. In this experiment, ^{87}Rb atoms are excited towards spherical states $nS_{1/2}$ with $n \sim 100$. A probe beam transitions the $5^2S_{1/2}$ state to the $5^2P_{1/2}$ state, and then a control beam ~~that~~ transitions the 5^2P state to the Rydberg $nS_{1/2}$ state.

The low overlap between the ground state and excited state wavefunctions gives Rydberg states a very long lifetime and very narrow line-width. A Rydberg excitation in the atom cloud leads to strong Van der Waals interactions with other atoms, contributing to high polarizabilities, such that no other Rydberg excitation can occur within a "~~blocking~~" sphere of radius R_b around the Rydberg atom. This Rydberg "~~blocking~~" phenomenon is illustrated in Figure 1.1. The experimental

parameters used in the lab results in $R_b \simeq 15\ \mu$m to 20 μm, while the density of the atomic cloud has a Gaussian standard deviation of 5 μm. This signifies that only one Rydberg excitation can exist within the atom cloud under our experimental conditions.

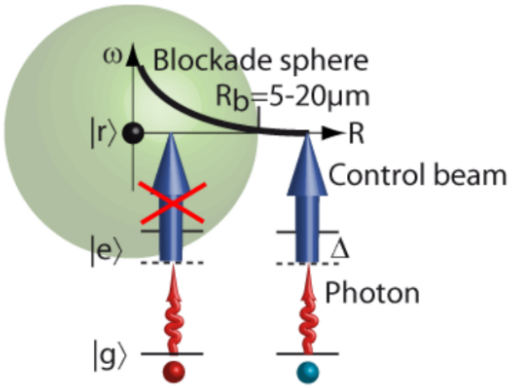
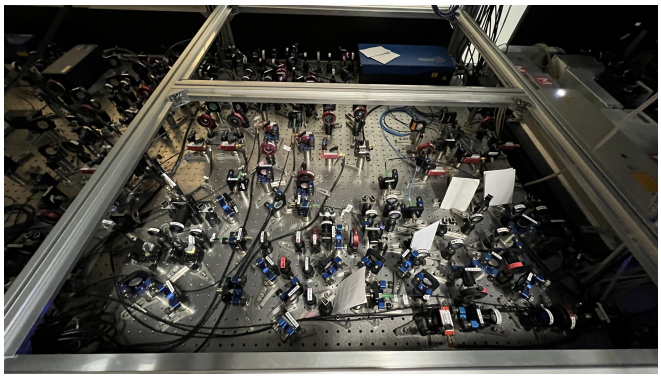


Figure 1.1: Schematic diagram of the Rydberg blocking effect. The Van der Waals dipole interaction of an existing Rydberg excitation prevents a second photon from being resonant within a ~~blocking~~ radius R_b around the first excitation. If the atom cloud is smaller than R_b, then only one Rydberg excitation can exist within the entire cloud.

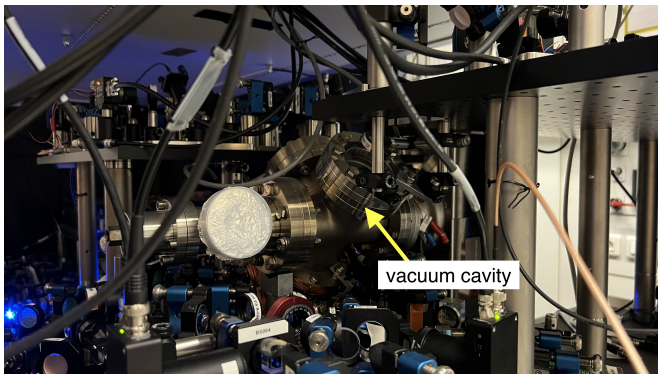
1.1.2 Experimental Setup

The entire experimental setup is very involved and sizable, and is divided into three parts. One part is used to control the different lasers, a second part prepares the different laser beams, and a third part houses the vacuum chamber in which the Rubidium atoms are prepared. A section of the first two parts is shown in Figure 1.2a and the vacuum ~~cavity~~ is shown in Figure 1.2b.

The experimental setup already somewhat reveals that the start-to-end process of the experiment – from laser preparation, atomic cloud preparation, to Rydberg excitation via controlled laser behavior, etc. – involves many calculated steps. Therefore, the experiment requires various digital and analog I/O modules and numerous respective I/O channels that send and receive instruction, signals, and data samples for experimental monitoring and processing. As a consequence, a smart and efficient data acquisition (DAQ) mechanism becomes crucial.



(a) Various physical devices used to control laser beams.



(b) Vacuum ~~cavity~~ in which the Rb atom cloud is confined and prepared.

Figure 1.2: Experimental setup.

1.2 Hardware & Software Requirements

A key necessity of the experiment is the ability to control the time scale between the needs of the atomic and optical preparations. The atomic manipulation requires control from microseconds to a few seconds, while the optical ~~part lasts about 100 nanoseconds~~. Consequently, the hardware and software platforms must be able to accommodate such particular demands.

1.2.1 Hardware

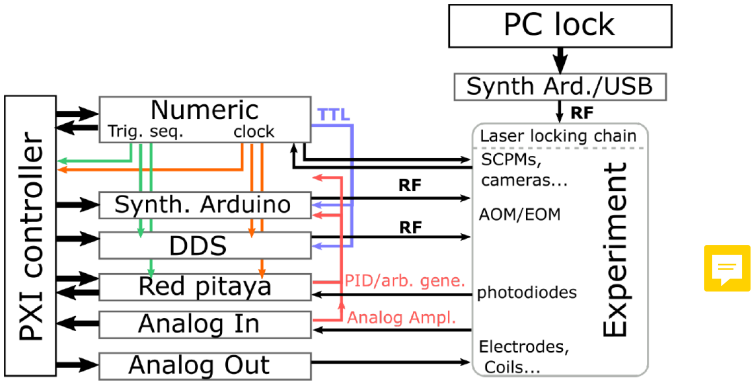


Figure 1.3: Control of the experiment.

A schematic of the experimental control system is shown in Figure 1.3. It consists of a PXI (PCI eXtensions for Instrumentation) system that sends instructions to different cards; various instruments synchronized by a trigger signal generated by the numeric card; Arduino channels that produce RF (radio frequency) signals that control laser beams; digital channels that control RF channels, cameras, etc; analog outputs that control RF signals, electrode voltages, etc; analog inputs that record photodiode signals from 100 kHz to 250 MHz; and more.

A photograph of an examlpe PXI system is shown in Figure 1.4, which is a National Instrument (NI) PXIe-1078 chassis with the three main modules of interest built-in: NI PXI-6713 Analog Output, NI 5761 Analog Input, and NI 6581 Digital I/O.

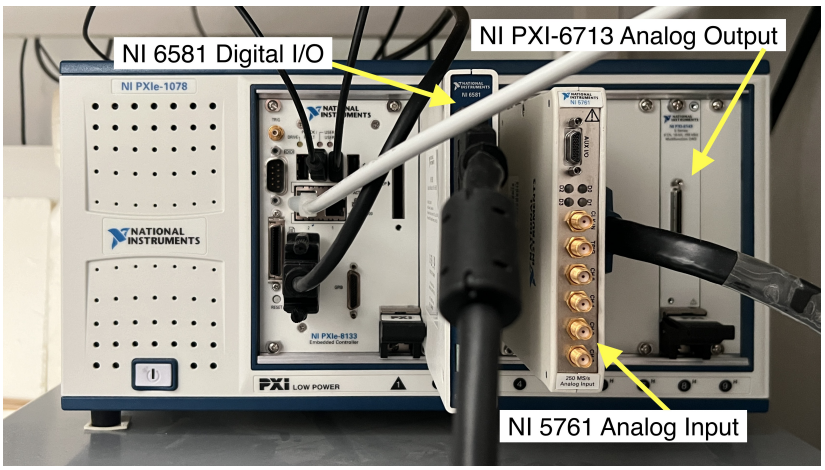


Figure 1.4: An NI PXIe-1078 chassis equipped with the three modules used in the experiment for digital and analog inputs/outputs.

1.2.2 Software

The main control of the platform is executed through a home-made LabVIEW program (VITO) on the PXI system. The program's interface is separated into four main panels: *Select Channels*, *Config Sequence*, *Config Channels*, and *Run Sequence*. A screenshot of the *Config Channels* panel of the VITO program interface is shown in Figure 1.5.

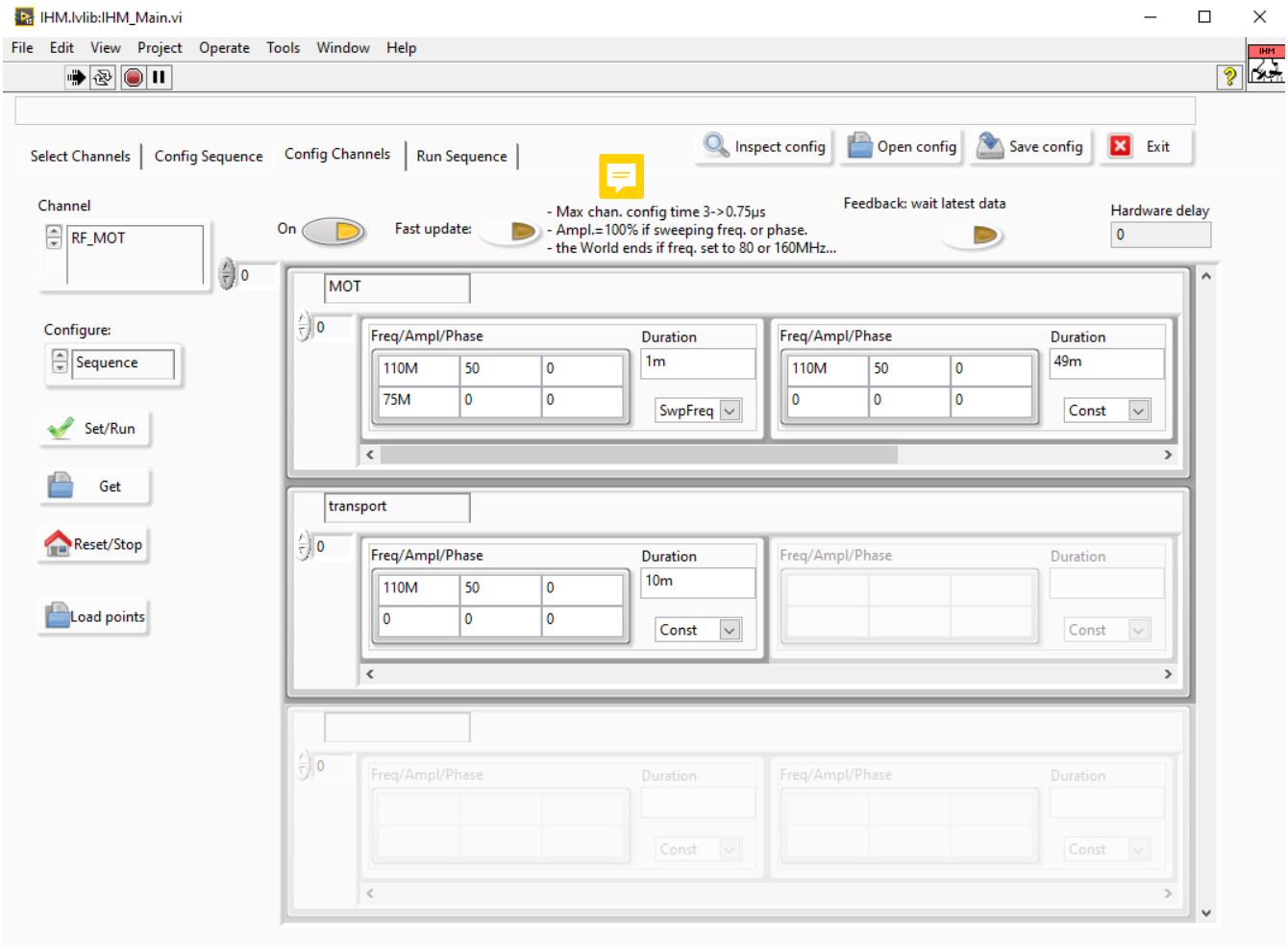


Figure 1.5: VITO, Config Channels.

1.3 The Costly Problem with LabVIEW

1.3.1 LabVIEW 2022

LabVIEW (Laboratory Virtual Instrument Engineering Workbench) is a system-design platform and development environment for visual programming created and owned by National Instruments. Currently, the experiment uses the 2015 version of LabVIEW for the development of the VITO control system on the PXI. Starting in 2022, NI software moved to a subscription model [2]. This means that in order to update the current version of LabVIEW used in the experiment to the newest, subscription-based version (2022 and beyond), so that the software framework used in the experiment does not become outdated, a costly annual fee becomes inevitable.

From this arose the problem that my internship was centered around. That is, is it feasible to migrate the software development environment from LabVIEW, a proprietary visual programming

language, to open-source languages, such as Python or C? And if so, what are the critical steps of such a migration process?

In the ideal situation where LabVIEW can be successfully replaced, not only will the cost of experimental operations be reduced by avoiding the annual subscription fee, the development process of necessary software programs also has the potential of becoming more efficient, making digital and analog I/O processes more adaptable to changing experimental needs.

Since LabVIEW is used to develop software programs for digital and analog I/O modules, it is then necessary to investigate whether and which open-source programming languages are viable alternatives for each module.

1.3.2 NI PXI-6713 Analog Output

The NI PXI-6713 Analog Output module is a 12-bit, 8-channel, 1 MS/s high-speed "simple" card designed to deliver simultaneous, multichannel updates for control and waveform output applications, such as stimulus-response, power supply control, high-speed, deterministic control, and sensor/signal simulation. It is considered a "simple" module because it does not contain a field-programmable gate array (FPGAs are elaborated in Section 2.1) that must be programmed by a hardware language to execute desired functions.

This module is controlled by the NI-DAQmx instrument driver ~~for data acquisition purposes~~, which allows the configuration and programming of the DAQ system from low-level OS and device control to high-level programming languages. Currently, for the experiment, programs for the NI PXI-6713 Analog Output module are developed in LabVIEW, which integrates the NI-DAQmx driver. Fortunately, NI-DAQmx can also be used with Python, C, and other languages, which is exactly what we need for software migration.

One of my internship objectives was to translate sample NI-DAQmx programs already written in LabVIEW to Python and C. The reason was to familiarize myself with LabVIEW, to examine the compatibility of such language migration, and to conduct runtime tests comparing the speeds of the translated programs between the two open-source languages. The two sample LabVIEW programs chosen were **OnDemand**, which configures the NI PXI-6713 module to output a continuous, constant signal voltage, and **Finite**, which configures the module to output a finite, periodical signal voltage.

OnDemand	Average Runtime (ms)	
	$V_{out} = 5$ V	$V_{out} = 7.5$ V
Python	2.085	2.130
C	1.904	1.800

a. Average runtimes for the LabVIEW OnDemand program translated to Python and C. Each average runtime was calculated over 10 separate tests, with 10,000 runs per test. The constant, on-demand ouput analog voltage was set at 5 V and 7.5 V for different tests.

Finite	Average Runtime (ms)	
	$f = 0.1$ MHz	$f = 1$ MHz
Python	21.450	21.538
C	20.020	20.976

b. Average runtimes for the LabVIEW Finite program translated to Python and C. Each average runtime was calculated over 10 separate tests, with 1,000 runs per test. The sample time was fixed to 10 ms for all runs, while the sampling rate f was set to 0.1 MHz and the recommended maximum of 1 MHz for different tests.

Table 1.1

With the help of APIs (application programming interfaces) and tutorial programs provided by NI [4][3] as references, I successfully translated these two simple NI-DAQmx programs from LabVIEW to Python and C and performed runtime tests. The runtime results are summarized in Table 1.1.

The results show that the sample LabVIEW programs translated to C generally have a slightly faster runtime compared to those translated to Python. Although OnDemand and Finite are quite

simple in complexity, we can generalize this result beyond these two sample programs. This is because the Python NI-DAQmx package was implemented as a complex, highly object-oriented wrapper around the NI-DAQmx C API using the **ctypes** Python library [4].

Consequently, when migrating LabVIEW programs that configure the NI PXI-6713 Analog Output module, both Python and C are viable open-source language alternatives. C provides a faster runtime as it is more primitive than Python, while Python is generally easier to learn, code, and debug.

1.3.3 NI 6581 Digital I/O & NI 5761 Analog Input

The NI 6581 Digital I/O & NI 5761 Analog Input modules are "smart" cards because they provide wider ranges of customizable functions via their respective on-board FPGA (field-programmable gate arrays). Figure 1.6 shows the software and hardware schematics of how such an FPGA works in these two modules.

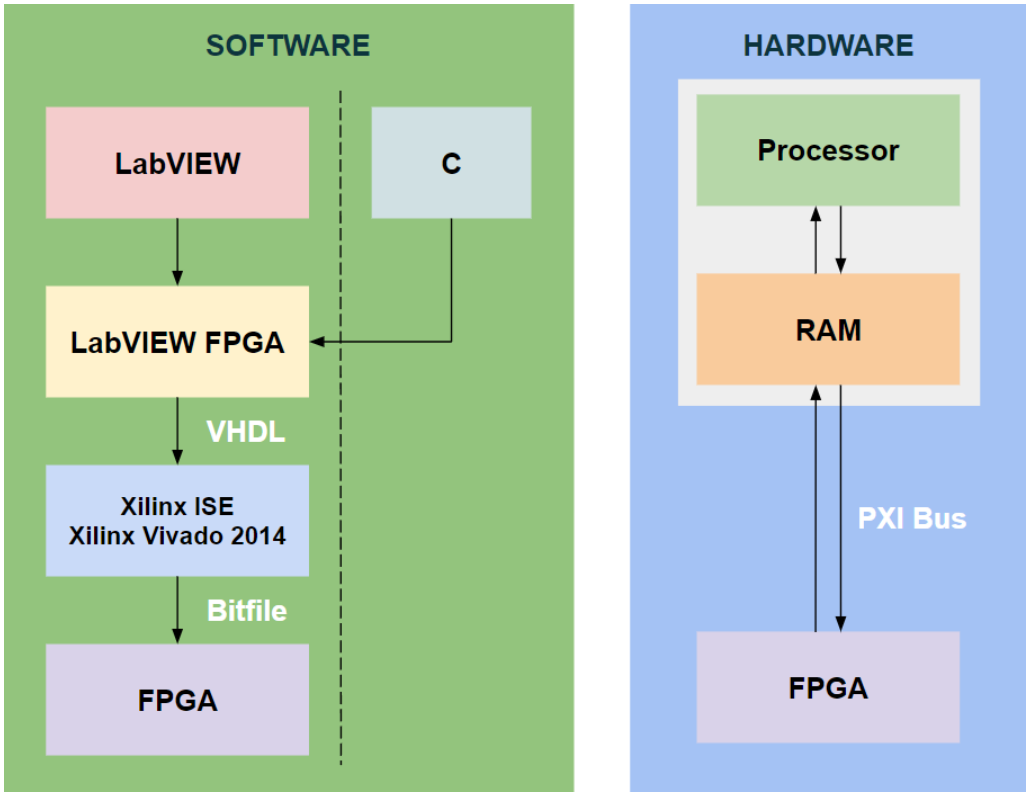


Figure 1.6: Schematics of software and hardware communication processes for NI "smart" modules. On the software side, the FPGA is programmed by a bitfile generated using Xilinx Vivado environment application. The bitfile corresponds to a hardware design program written in a hardware description language (HDL), such as Verilog or VHDL. This hardware design is then communicated via the LabVIEW FPGA software module to LabVIEW or C programs for front-end purposes. On the hardware side, the FPGA communicates with the on-board RAM (random access memory) via the PXI bus, and a processor works with the data stored in the RAM.

For both the NI 6581 Digital I/O and the NI 5761 Analog Input modules, the files that describe how the FPGA communicates with the RAM via the PXI bus was encrypted by NI for proprietary reasons. This introduced a bottleneck in our endeavor as it prevented us from replacing LabVIEW with Python or C like it could easily be done for the NI PXI-6713 Analog Output module. As a result, we must look into alternative FPGA-based devices for digital I/O and analog input that could replace these two "smart" modules and hopefully relieve the experiment from such a proprietary stranglehold.

Chapter 2

Replacing NI FPGA Cards

This chapter contains the bulk of my internship work. It elaborates the exploration of hardware alternatives to the NI 6581 Digital I/O and NI 5761 Analog Input modules that challenged software migration away from LabVIEW, due to proprietary encryption associated with the hardware.

In order to familiarize myself with the basics of FPGA and HDL, I started learning to program the Digilent Cmod A7-35T module that could replace the NI 6581 module for digital I/O purposes. Then I proceeded to learn to operate and to explore the capabilities of the Red Pitaya ~~17~~ module, a more complex single-board computer with the potential of replacing the NI 5761 module for analog input purposes.

2.1 Introduction to FPGA & HDL

Field Programmable Gate Arrays An FPGA is an integrated circuit that comes with configurable logic blocks and other features. FPGAs differ from traditional integrated circuits in that they can be programmed and reprogrammed by users, such that they allow the creation of custom digital circuits by connecting the configurable blocks as needed. Hence they are "field-programmable," indicating that their abilities are adjustable and not hardwired by the manufacturer. Additionally, FPGAs excel at parallel processing and can even outperform central processing units (CPUs) for certain tasks.

Hardware Design Language HDLs (such as VHDL or Verilog) are used to describe the behavior of digital circuits, which can be thought of as programming languages specifically tailored for hardware design. HDL code is compiled into a bit pattern file (or bitfile) that specifies the connections inside the FPGA, defining its behaviours. When the FPGA is booted, the bitfile is loaded into the circuit, making the necessary connections and configuring its functionality.

For the experimental development of hardware programs, the Vivado design softare envirement is used. It includes tools for design entry, synthesis, place and route, and verification and simulation.

2.2 Digilent Cmod A7-35T as Digital I/O

The Digilent Cmod A7-35T is a small, 48-pin board built around a Xilinx Artix 7 FPGA. It also includes a USB-JTAG programming circuit, USB-UART bridge, clock source, Pmod host connector, SRAM, basic I/O devices, and a Quad SPI Flash.

It was ~~already~~ known that ~~the~~ 44 digital FPGA I/O signals ~~of the Cmod A7-35T~~ allow it to functionally replace the NI 6581 Digital I/O module in the experimen[illegible] So for the purpose of

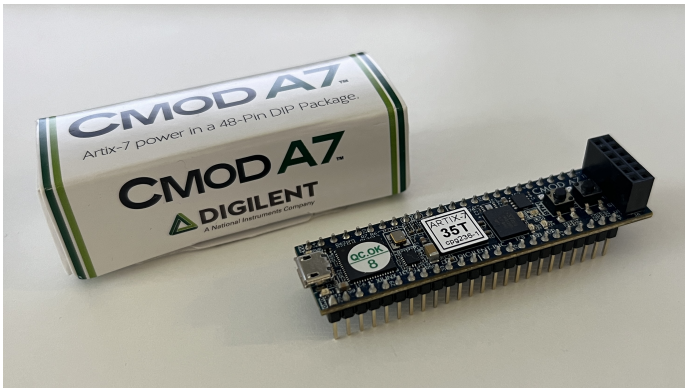
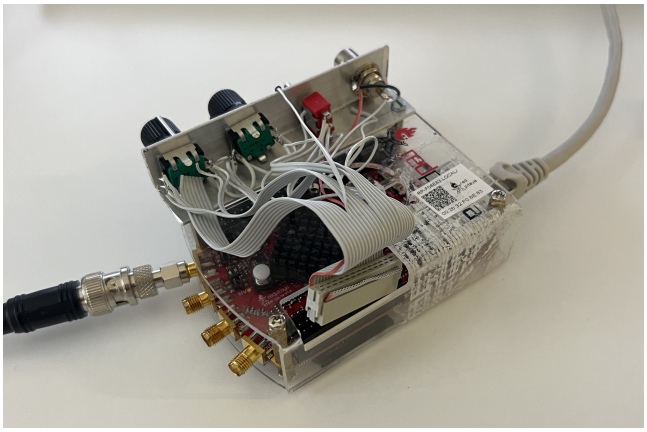


Figure 2.1: The Digilent Cmod A7-35T module.

learning to program FPGAs using HDLs, I walked through the tutorials provided by Digilent as well as created my own programs that sent, received, and processed digital I/O signals to and from the Cmod A7-35T via its UART port.

2.3 Red Pitaya 17 as Analog Input



(a) Red Pitaya STEMlab 125-14 model single-board computer.

(b) Red Pitaya web homepage interface with default applications.

Figure 2.2: The Red Pitaya.

Red Pitaya is a single-board computer first created to be an alternative for various expensive laboratory and control measurements. The experiment uses the STEMlab 125-14 model running on ecosystem version 1.04. The model includes a Xilinx system-on-chip (SoC) with a CPU and an FPGA, two 125 MS/s inputs and two 125 MS/s outputs coupled with 14-bit analog-to-digital (ADC) and digital-to-analog (DAC) converters, ethernet connectivity, and more. The Red Pitaya is well known for its proprietary hardware design and open-source software development framework. By default, the Red Pitaya can function as an oscilloscope, an LCR meter, a spectrum analyzer, logic analyzer, and so on [6]. For the experiment, we wish to exploit Red Pitaya's 125 MS/s inputs coupled to the ADC, which could replace the NI 5761 Analog Input module for data acquisition purposes.

To get started with STEMlab 125-14, I followed the introductory tutorials provided by Red Pitaya. The tutorials covered the basics of hardware and software programming associated with the Red Pitaya, which differs from that of the Digilent Cmod A7, as the Red Pitaya is much more complex. Following this, I moved onto exploring whether the Red Pitaya could satisfy key experimental requirements, which include sufficient memory storage for data acquisition, input noise limits, and discontinuous data acquisition.

2.3.1 Input Data Memory Storage & Extension

The NI 5761 Analog Input module currently acquires data for the experiment in the following manner:

$$4 \text{ channels} \cdot 250 \text{ MHz} \cdot 2 \text{ bytes/Sample} \cdot 10 \ \mu\text{s/loop} \cdot 100 \text{ loops} = 2 \text{ MB} \ . \tag{2.1}$$

This is to say that for the module, having 4 analog input channels, where each channel samples data at a rate of 250 MHz, with each data sample having a size of 2 bytes, collects data over a period of 10 μs. This data acquisition process is repeated 100 times per experiment, all over the course of 100 ms. As a result, an amount of 2 MB of data must be stored at a time to be read and processed later.

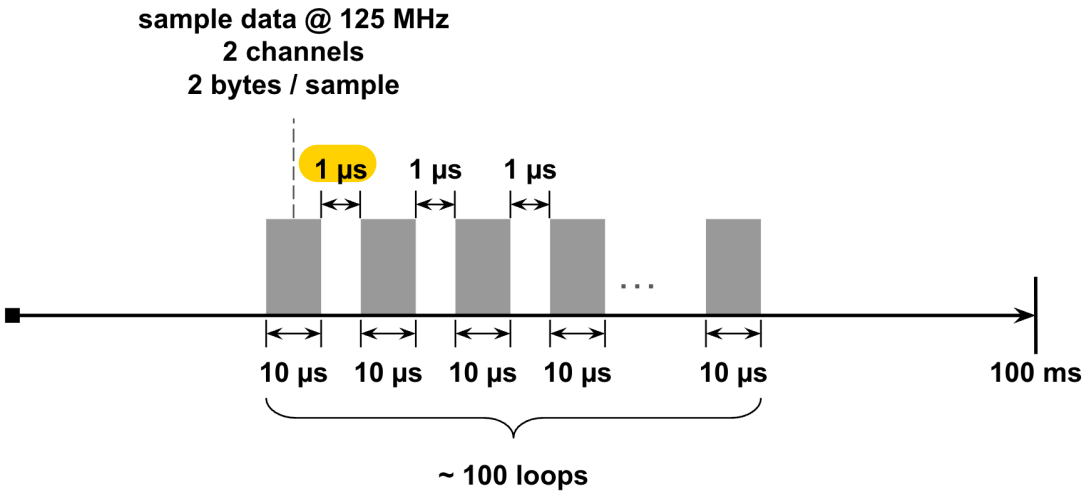


Figure 2.3: Example schematic of how data acquisition would take place for an experimental run using the Red Pitaya, which samples data at a maximum rate of 125 million samples per second.

Applying this acquisition process to the Red Pitaya, where there are 2 input channels and the maximum sampling rate is 125 MHz, we have

$$2 \text{ channels} \cdot 125 \text{ MHz} \cdot 2 \text{ bytes/Sample} \cdot 10 \ \mu\text{s/loop} \cdot 100 \text{ loops} = 0.5 \text{ MB} \ . \tag{2.2}$$

This result implies that we need to be able to configure the Red Pitaya such that it can acquire and store at least 0.5 MB of input data at a given time into the on-board RAM, which has a size of 512 MB.

In order to achieve such required memory capacity, there's a very likely need to manually extend the allocated memory space that stores the converted input signal data on the Red Pitaya. So I began analyzing the Lock-in PID with Oscilloscope application, which proved rather impractical, followed by another unsuccessful attempt with the built-in Streaming service application. Finally, the Deep Memory Acquisition (DMA) feature available only in ecosystem 2.03 or newer seemed to be the most promising input acquisition method in our endeavor to use the Red Pitaya as an alternative to the NI 5761 module.

2.3.1.1 Lock-in+PID

The Lock-in+PID application is currently used in the experiment to maintain the intensity of the laser beam that probes the Rydberg atoms in the cavity. The application was first created by Marcelo Alejandro Luda in the context of a Physics PhD career at the Universidad de Buenos Aires [1]. It includes an oscilloscope application (adapted from the default one that comes with the Red Pitaya) and a lock-in amplifier. Figure 2.4 presents a screenshot of the application interface, showing the

two input signal data collected by the oscilloscope, as well as various other configuration panels. The application allows the user to capture and plot a desired length of signal data that can be easily saved to different file types for further studies. Therefore, we hoped to adapt the oscilloscope of the Lock-in+PID to satisfy the experimental requirements predicted in Equation (2.2).

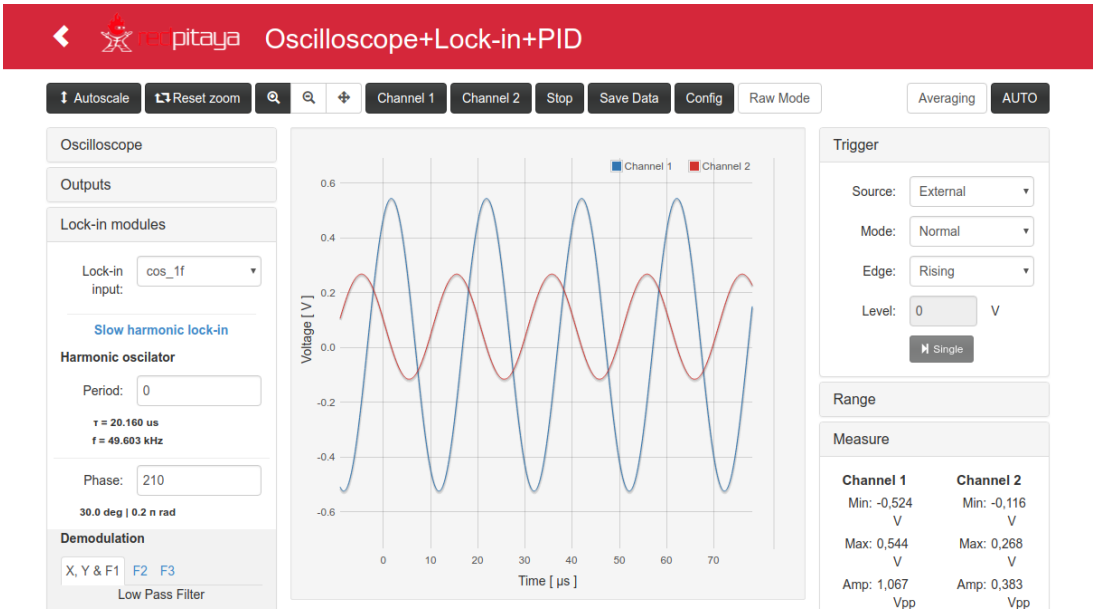


Figure 2.4: Screenshot of the Lock-in+PID application interface.

Hardware Modification By default, the oscilloscope application samples data at the maximum rate of 125 MHz, and occupies 1 IP (intellectual property) core, starting from system address `0x40100000` and ending at `0x401FFFFF` [6]. This corresponds to 16^5 bytes = 1,048,576 bytes $\sim$ 1 MB of memory space. So the objective was then to understand how the oscilloscope of the Lock-in+PID application captures, stores, and reads input data to and from the RAM, so that we can accordingly modify the memory allocation and storage for our experimental needs.

To begin, I explored the Verilog code that configures the oscilloscope hardware, which is comprised of the following main parts:

- **Trigger:** configures the trigger for the ADC
- **ADC input data filtering:** the input signal data captured and converted by the ADC are filtered based on configuration coefficients
- **Input data decimation:** the rate of data captured is reduced while maintaining the essential information
- **Buffer RAM:** writing to and reading from the buffer RAM where the data captured by the ADC are temporarily stored
- **AXI (Advanced eXtensible Interface) connection:** protocol that enables efficient communication between different components on the Red Pitaya
- **System bus connection:** assigns values for various system address variables and stores input data read from the buffer RAM to the designated addresses on the allocated IP core

The Verilog code of the oscilloscope reveals that $2^{14} = 16,384$ samples of ADC data per input channel are stored into the appropriate IP core memory space. By default, each sample takes up only 14 bits, but is stored in a 32-bit (4-byte) chunk of system memory, with the upper 18 bits

being insignificant. This is because the Red Pitaya transfers system memory data in 4-byte chunks. Correspondingly, 65,536 bytes of data are saved to system memory at once by the oscilloscope hardware per input channel, with channel A data starting from system address `0x40110000` and channel B data starting from system address `0x40120000`. Given that 1 MB of memory space is reserved for the oscilloscope application, we are then able to exploit the unused space starting from system address `0x40130000` for our purposes.

To extend the amount of data saved to system memory at once, I modified the Verilog code in the **system bus connection** section by extending the system address into the unused memory addresses and storing the 14 significant bits of channel B sample into the unused bits of the 4-byte chunk for each channel A sample. This allowed an 8-fold increase in captured data storage into the system memory per channel, corresponding to 131,072 samples or roughly 0.25 MB per channel, meeting our predicted experimental needs described by Equation (2.2) exactly.

Software Modification Subsequently, it was necessary to explore and modify accordingly the software code written in C that reads and processes the ADC input data stored in the system RAM for the front-end oscilloscope web application. This included updating the extended memory addresses and sample length for data reading, and extracting appropriately each sample for channel A and B now stored side-by-side in 4-byte chunks.

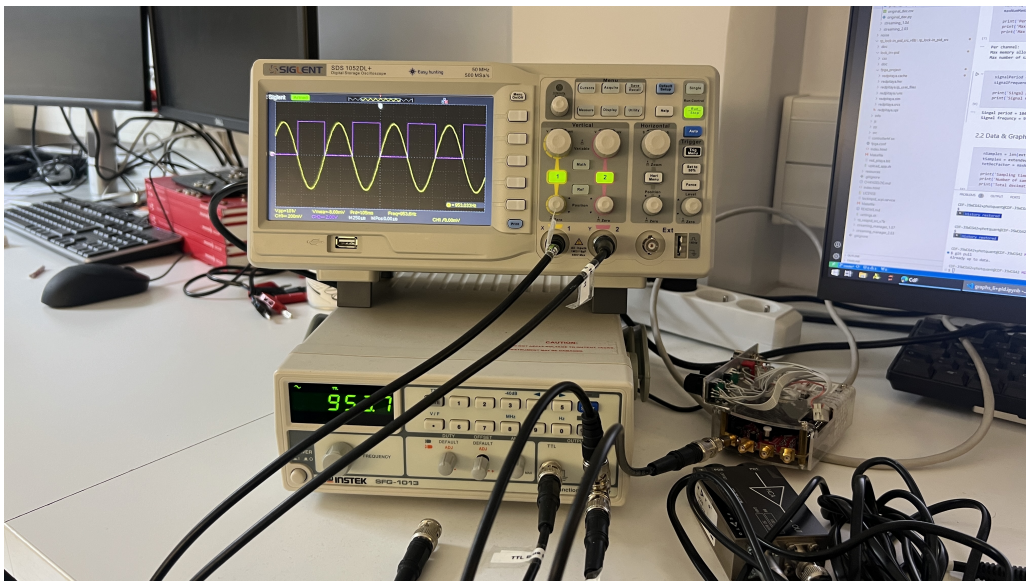


Figure 2.5: Input signal generator (bottom) for conducting tests for the Red Pitaya and external oscilloscope (top) to visualize and confirm the configured input signals.

Tests A simple method to test whether the memory was successfully extended is to monitor the memory addresses that were previously unused consecutively and observe the value stored. For example, executing in the Red Pitaya kernal `monitor 0x40130000` multiple times by hand, and reading the value stored at this memory address that's not used by default. If the memory was not extended correctly, the output values will only fluctuate slightly due to noise. If the memory was extended correctly, the output values will fluctuate by a large amount because varying sample points of the input signal will be stored at that address. This method is by no means comprehensive but is sufficient as a quick check.

Additionally, we want to configure the input signal at a frequency such that sampling exactly 1 period of the signal will fill up the entire memory space dedicated to a single channel, i.e.

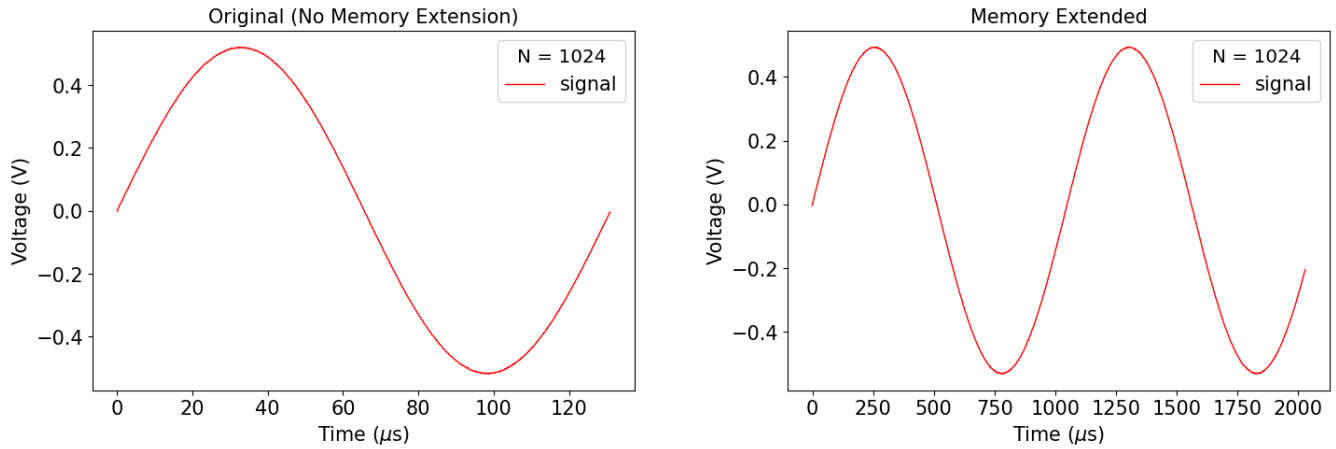
$$f_{\text{full}} = \frac{\text{sampling rate}}{\text{max number of samples storable in memory at once}} \,. \tag{2.3}$$

Using the original oscilloscope application as an example, which stores up to 16,384 samples at once, at a sampling rate of 125 MHz, this corresponds to a $f_{\text{full}} = 7629$ Hz and a period of $T_{\text{full}} = 131$ μs. And for the oscilloscope application with 8-fold extended memory space, $f_{\text{full}} = 954$ Hz and $T_{\text{full}} = 1049$ μs. This is so that capturing exactly 1 period of the signal data using the oscilloscope of the Lock-in+PID, we should expect corresponding numbers of data sample points in the saved signal data files.

Figure 2.5 shows the setup of such a memory test with the Red Pitaya, where a sinusoidal analog signal is created by a signal generator. The signal is connected to an external oscilloscope for monitoring and to the Red Pitaya's channel A input.

Results With the `monitor` command in Red Pitaya's kernal, it was quickly confirmed that ~~that~~ signal values are stored into the previously unused memory addresses starting from `0x40130000` and ending at `0x4018FFFF`.

For the original oscilloscope (without memory space extension), displaying and saving the data for one signal period via the Lock-in+PID application web interface for an input signal at f_{full} was easy and successful. As the application provided configurations that allowed the capturing of exactly one signal period starting at time 0 and ending at T_{full}, as predicted. This is shown in Figure 2.6a, where we observe a single waveform of the input signal, and served as a baseline test. However, capturing exactly one signal data with memory extension proved difficult. The Lock-in+PID web application became buggy and could not as swiftly capture a single waveform. The best signal data I was able to save is shown in Figure 2.6b, where roughly two periods of the wave-form was capture. More critically, it was discovered at the frond-end code for the application further decimates the oscilloscope signal data for display resolution purposes, as all saved signal data files only contained 1024 sample points. As a consequence, we determined that this attempt to modify the Lock-in+PID application is not a practical method for analog input data acquisition purposes. Therefore, alternative methods were explored.



(a) (b)

Figure 2.6: Data exported from the Lock-in+PID oscilloscope application for (a) original and (b) extended allotted system memory space for signal data storage.

2.3.1.2 Data Streaming

Another method to handle data acquisition and memory storage is through the Streaming application, which enables input data to be streamed from the Red Pitaya to a file saved on the system's SD card or on a remote computer via the ethernet protocol. Figure 2.7 shows a screenshot of the application's web interface, where various configurations can be set, including sampling frequency, input channel

selection, input resolution and attenuation, calibration, etc. The data can be saved as the following types: *.wav*, *.tdms*, or *.bin*.

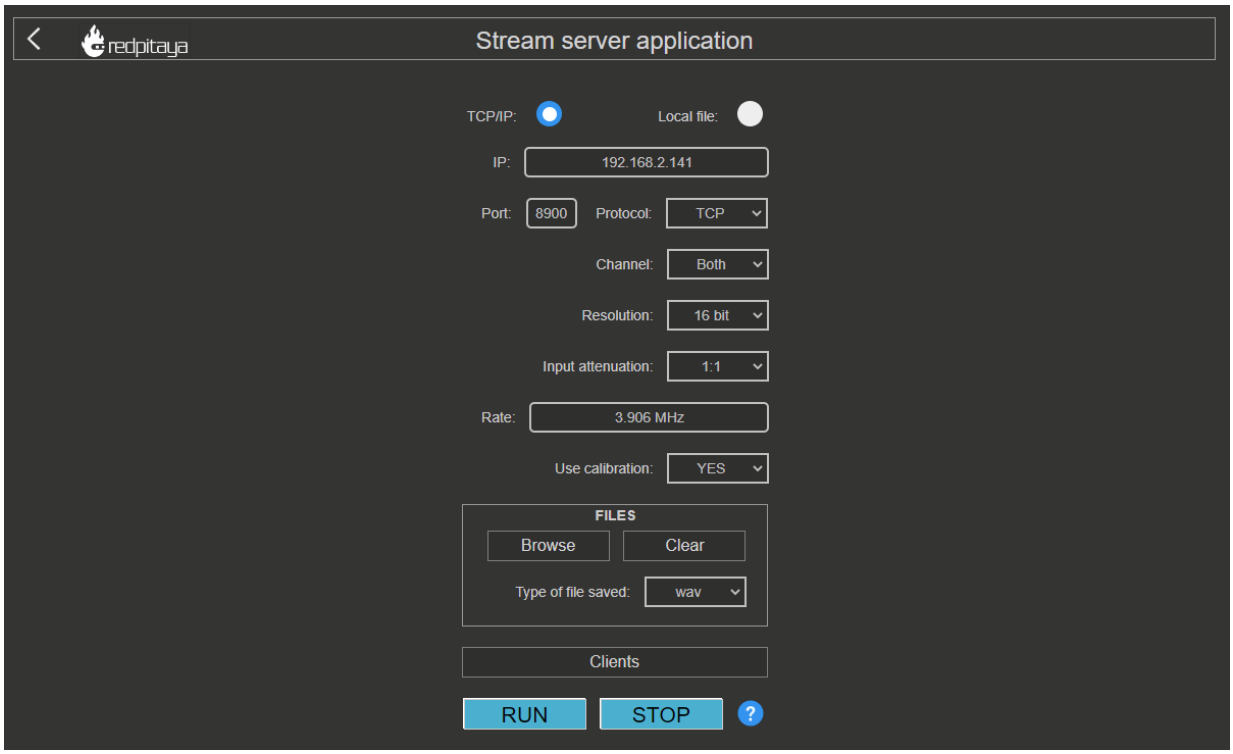


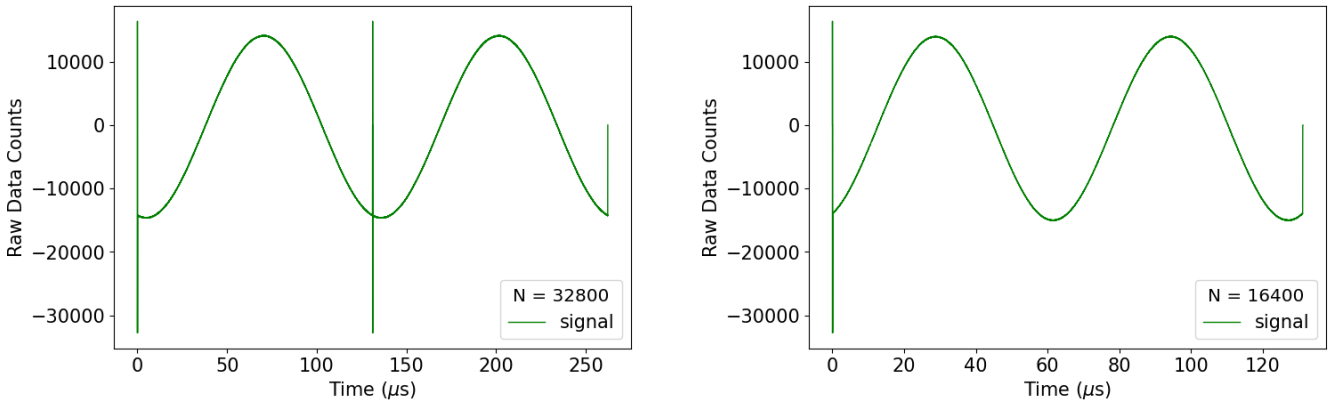
Figure 2.7: Web interface of the Streaming application that comes with the Red Pitaya.

At first glance, the features that come with the Streaming application make it a good candidate for acquiring analog input data. So then it was necessary to discover its mechanism with respect to RAM storage, in order to modify the memory space to acquire sufficient amount of data at once to meet the experimental requirement represented in Equation (2.2).

Tests To familiarize with how the Streaming server works, I started with some sampling tests. First we assume that the application, similar to the oscilloscope, is allotted $16^4 = 65{,}536$ bytes of memory space per channel by default. Sampling at 16 bits per sample, this corresponds to 32,768 samples that can be stored at once. At the maximum sampling rate of 125 MHz, one period of input signal at $f_{\text{full}} = 3815$ Hz will fully occupy this amount of memory space. Using these parameters, input signal data with RAW units were streamed and saved to a *.bin* file to be processed and evaluated.

Results Figure 2.8a shows the streamed signal data plotted with respect to the expected time span. Immediately, we recognize that the actual number of samples taken is greater than the input configuration, as if this number is rounded up to the nearest hundred, which is an unexpected and undesired behavior. Additionally, we see sharp lines at the beginning, middle, and end of the signal, which we hypothesized as indications of memory overflow. These lines imply that the memory space allotted to the Streaming app is actually half the amount predicted, at $16^4/2 = 32{,}768$ bytes or roughly 16,400 samples. Since exactly one period of the signal with $f_{\text{full}} = 3815$ Hz is contained in the "halved" memory space, referring to Equation (2.3), it must mean that the sampling rate was actually not 125 MHz, as entered on the application web interface, but half of that rate. This is another unexpected and undesired behavior of the Streaming application.

Another test was done using 16,384 samples as the maximum sample count that can be stored in the allotted memory space, which at 125 MHz sampling speed (entered on the application web interface), corresponds to $f_{\text{full}} = 7629$ Hz. The signal data plot is shown in Figure 2.8b. The absence



(a) (b)

Figure 2.8: Input signal data streamed from the Streaming application for (a) original and (b) extended allotted system memory space for signal data storage.

of a sharp middle line indicates that memory overflow did not occur, and the fact that 2 signal periods where captured confirms that the actual sample rate was indeed half of what was entered, validating our assumptions and hypothesis from the previous test run.

Conclusion A deeper investigation into the Streaming application's sampling rate setting reveals that only certain number values are accepted by the application. I then explored the front-end Javascript code associated with this input field and found numerous strange bugs. We hoped that upgrading the Red Pitaya ecosystem from the current 1.04 version to the newest 2.03 version will introduce fixes to the Streaming app, for both the sampling count and sampling rate fields. Unfortunately, this was not the case, as the same issues remain in ecosystem 2.03. We determined that it was not worth the effort trying to debug the Streaming application, so we decided to move onto another alternative method for data acquisition for analog inputs.

2.3.1.3 Deep Memory Acquisition (DMA)

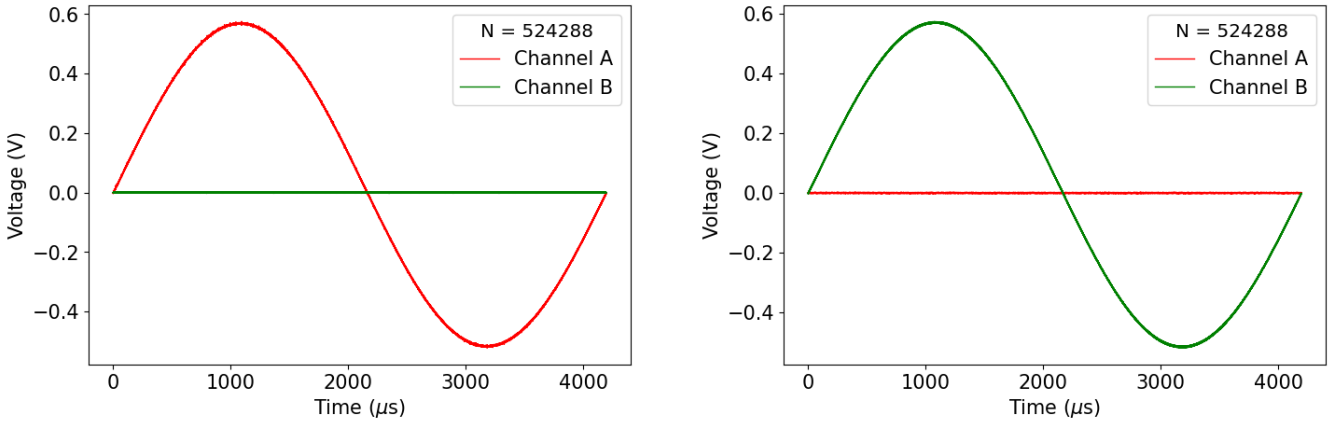
Deep Memory Acquisition is a functionality that allows the Red Pitaya to set a buffer of any size for capturing signal data from the ADC. Only available for ecosystems 2.03 or newer, DMA runs at maximum ADC sampling rate of 125 MHz, writes data directly to the RAM, and relies on the AXI protocol. By default, the maximum size of the buffer (for the two input channels) is set to 2 MB, and data are saved as 32-bits per sample. According to the documentation [6], the maximal recommended DMA region size is 412 MB for STEMlab 125-14. This amount of memory space is beyond the experimental requirement described by Equation (2.2). Therefore, the DMA feature seemed to fit exactly what we needed for analog input acquisition.

To get started with testing the DMA feature, the reserved memory must be reconfigured, followed by a rebuild of the device tree and restart of the Red Pitaya. I doubled the default size of the reserved memory from `0x200000` bytes $\sim$ 2 MB to `0x400000` bytes $\sim$ 4 MB. This buffer region is shared between the two channels, and the exact buffer size for each channel can be configured through programming commands.

Test Acquisitions The Red Pitaya can be controlled remotely through SCPI (Standard Commands for Programmable Instruments) protocol available in Python and MATLAB. It can also be controlled with on-board Python and C API commands. For testing the DMA, I referenced provided examples in the official documentation and used SCPI and API commands in Python. The main

code structure for both acquisitions is similar between the remote and on-board methods; it is as follows:

- Configure signal units, decimation, and trigger source and level
- Get reserved memory region address and size
- Set DMA buffer address and size for each input channel
- Continuously acquire data until buffer is full
- Read data from buffer
- Process and save data to external file



(a) (b)

Figure 2.9: Input signal data acquired through DMA tests where analog signal was only connected to (a) channel A and (b) channel B. The trigger was set at the positive edge of the connected channel signal.

For all tests, the reserved memory was equally distributed among the two input channels, this corresponded to 2 MB of data per channel, or 524,288 samples with each sample sizing 32 bits. Therefore, one period of an input signal at $f_{\text{full}} = 238$ Hz would completely fill up the memory buffer for one channel. Figure 2.9 shows two DMA tests using such configurations, where in each test an input signal was connected only to one channel, with the other channel unconnected. We see that the exact number of samples demanded were acquired and written to the buffer, and that precisely one period of the signal filled up the corresponding channel buffer entirely.

Conclusion The DMA feature appears to be the most promising and powerful candidate method of analog input acquisition using the Red Pitaya that could replace the NI 5761 module in the experiment. Compared to the Streaming application, it does not present issues associated with sampling count, sampling rate, and memory overflow. However, further studies regarding Red Pitaya input noise limits and discontinuous acquisition capabilities must be explored using DMA to definitively confirm if the Red Pitaya can fully meet experimental requirements.

2.3.2 Input Noise

2.3.2.1 Homodyne Detection System

Currently, a critical role of the NI 5761 Analog Input module is the acquisition and processing of data by homodyne detection of the beam coming from the cavity in the presence of Rb atoms. Such a cavity beam is a result of the probe and control pulses sent into the cavity. Homodyne detection

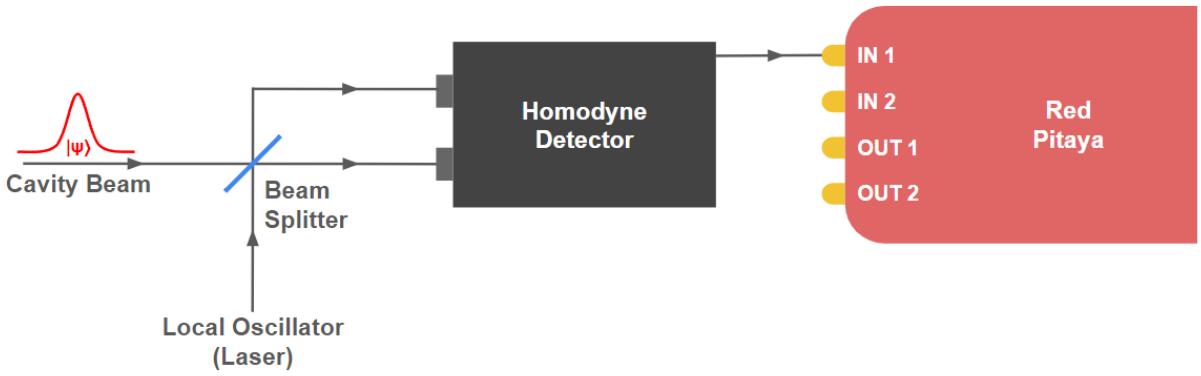


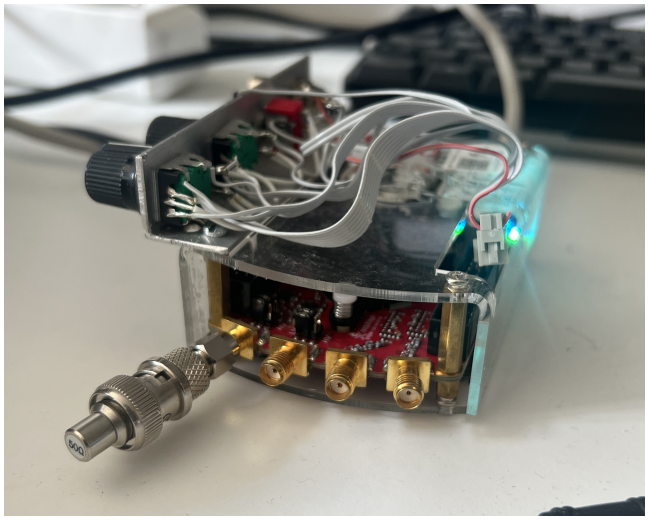
Figure 2.10: Illustration of homodyne detection system in the experimental setup.

allows the measurement of a quadrature of the electric field of the input signal [5]. As shown in Figure 2.10, the detection system consists of the beam coming from the cavity and a laser beam with a known phase serving as a local oscillator entering a beam splitter. The two outputs of the beam splitter enter the homodyne detector that measures the response difference, and the result is collected by an analog data acquisition device. In our hypothetical situation, the Red Pitaya replaces the NI 5761 as the input acquisition module.

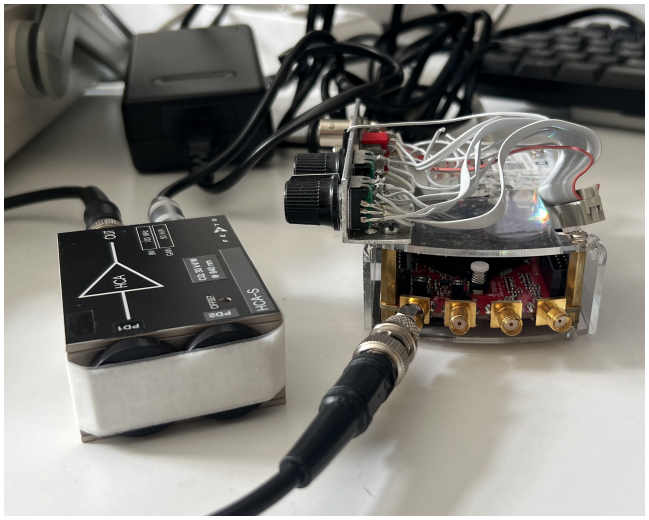
Shot Noise ~~The shot noise of such a detection system is limited by the~~ noise of the homodyne detector and the Red Pitaya:

$$\sigma^2_{\mathrm{SN}} \gg \sigma^2_{\mathrm{HDN}} + \sigma^2_{\mathrm{RPN}} \ , \tag{2.4}$$

where SN stands for shot noise, HDN stands for homodyne detector noise, and RPN stands for Red Pitaya noise. Ideally, we want the noise of the Red Pitaya to be significantly lower than that of the homodyne detector, such that the shot noise of the detection system can be limited at a minimal value.



(a) A 50 Ω resistor connected to input channel A of the Red Pitaya.



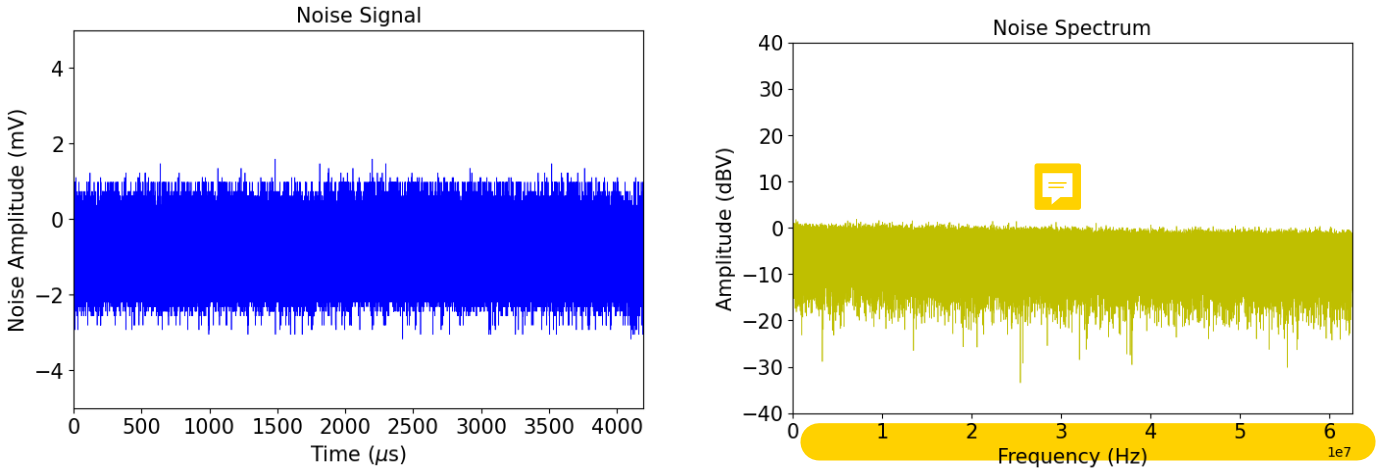
(b) A homodyne detector with inputs taped over connected to input channel A of the Red Pitaya.

Figure 2.11: Experimental setups for determining noise levels.

Methodology To determine the intrinsic device noise of the Red Pitaya, a resistor ~~with high resistance~~ (50 Ω) was connected to a selected input channel to block out noise coming from the external environment, as shown in Figure 2.11a. Similarly, to determine the ~~shot noise limit~~ of

the homodyne detection system, a homodyne detector with its two inputs (where the beam splitter outputs would enter) taped over was connected to a selected input channel of the Red Pitaya, as shwon in Figure 2.11b. For each test scenario, using Deep Memory Acquisition, N = 524,288 samples of noise signal data were extracted for processing and analysis via remote SCPI commands.

2.3.2.2 Data Analysis



(a) (b)

Figure 2.12: Signal and spectrum of Red Pitaya's intrinsic noise.

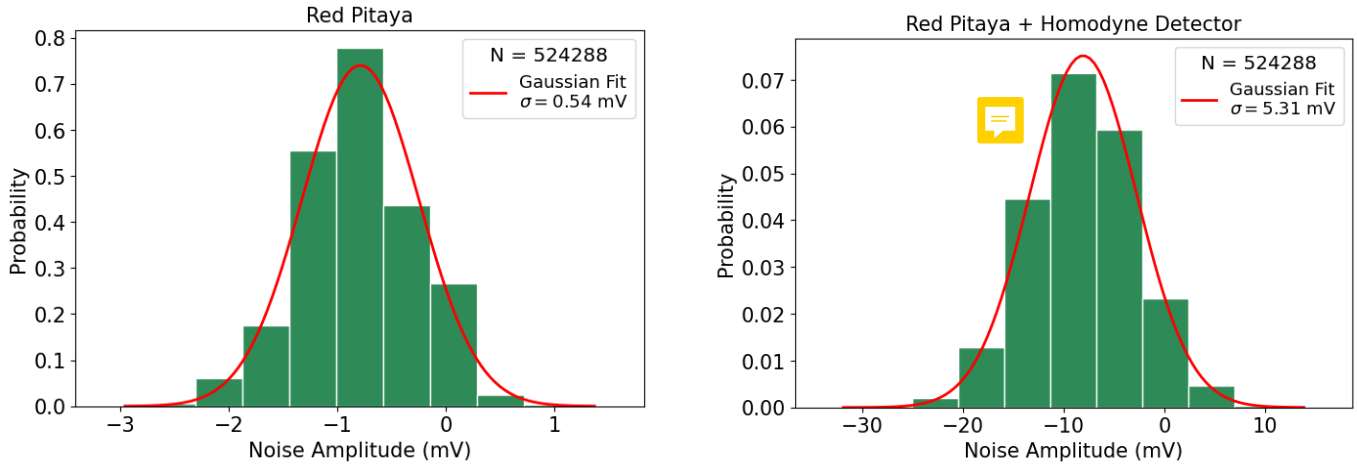
Figure 2.12a shows the intrinsic device noise of the Red Pitaya sampled over 4194 μs (max ADC rate of 125 MHz for N = 524,288 samples). This set of noise data was converted to noise spectrum as shown in Figure 2.12b using Numpy FFT (Fast Fourier Transform) functions. The noise spectrum was cut off at the Nyquist frequency and the amplitude converted to decibel-volts for better visualization. The absence of sharp peaks across the frequency range of the noise spectrum assured that the Red Pitaya is not intrinsically hypersensitive to any particular signal frequencies.

The noise signal histogram of the Red Pitaya and the homodyne detection system were also produced and are shown in Figure 2.13. The histograms are fitted with Gaussian distribution curves using SciPy and the variances were extracted from the fit, where $\sigma_{\text{RPN}} = 0.54$ mV and $\sigma^2_{\text{HDN}} + \sigma^2_{\text{RPN}} = (5.41\text{mV})^2$. This implies that $\sigma^2_{\text{HDN}} = 5.28$ mV $\gg \sigma_{\text{RPN}}$, i.e. the Red Pitaya is much less noisy compared to the homodyne detector, which is ideal for the experiment.

Conclusion The Red Pitaya noise signal study suggests that the Red Pitaya is a viable candidate for data acquisition within the homodyne detection system with regards to noise sensitivity. This is shown by the absence of sharp frequency spikes in the intrinsic noise spectrum of the Red Pitaya. On top of that, the fact that the Red Pitaya is ~ 10 times less noisy than the homodyne detector, means the shot noise limit of the entire homodyne detection system can be kept at an ideally low value.

2.3.3 Discontinuous Acquisition

Deep Memory Acquisition on the Red Pitaya is limited to a continuous mode, with the number of samples to be collected by the 125 MS/s ADC as the only defining parameter. Currently, there are no functions in the list of supported SCPI and API commands that can customize acquisitions beyond a simple continuous fashion. However, referring to Figure 2.3 as an example, we see that experimental data acquisition is put in a loop with roughly 1 μs between each run. Within this 1 μs experimental



(a) (b)

Figure 2.13: Noise signal distribution of (a) the Red Pitaya with a 50 Ω resistor connected to input and (b) the homodyne detection system with Red Pitaya.

pause, we would like DMA to pause as well, in order to avoid acquiring nonsensical data that will not be used. Therefore, it becomes crucial to understand how the DMA feature, specifically through its list of supported SCPI and API commands, configures the Red Pitaya to acquire data via the ADC. The goal here is to be able to configure the Red Pitaya with home-made SCPI and API commands that will allow customized, discontinuous data acquisition.

This experimental endeavor is still ongoing.

Conclusion

The effort to eliminate the experimental dependence on LabVIEW, the now costly proprietary programming language, is very promising. We determined that the NI PXI-6713 Analog Output module can be programmed with open source languages such as Python or C, and the NI 6581 Digital I/O can essentially be replaced by the Digilent Cmod A7-35T.

For the NI 5761 Analog Input module, which we hope to replace with the Red Pitaya, we determined that the Deep Memmory Acquisition feature is the best method for data acquisition. More research and work are currently underway to tweak the Red Pitaya so that it can acquire data discontinuously according to experimental needs.

Bibliography

[1] Marcelo Alejandro Luda. "Instrumentación y control con aplicaciones en óptica y metrología". PhD thesis. Universidad de Buenos Aire, 2021.

[2] National Instrument. *NI Software Subscription and Application Deployment*. Accessed 27 May 2024 from https://www.ni.com/en/support/documentation/supplemental/22/ni-software-subscription-and-application-deployment.html. 2024.

[3] National Instrument. *NI-DAQmx C Reference Help*. Accessed 20 February 2024 from https://www.ni.com/docs/fr-FR/bundle/ni-daqmx-c-api-ref/page/cdaqmx/help_file_title.html. 2023.

[4] National Instrument. *NI-DAQmx Python Documentation*.
Accessed 20 February 2024 from https://nidaqmx-python.readthedocs.io/en/latest/. 2024.

[5] Valentin Magro.
L'ingénierie quantique de la lumière dans un ensemble d'atomes de Rydberg en cavité.
Collège de France, 2021.

[6] Red Pitaya. *STEMlab 125-14*.
Accessed 31 May 2024 from https://redpitaya.com/stemlab-125-14/. 2024.

[7] Julien Vaneecloo, Sébastien Garcia, and Alexei Ourjoumtsev.
"An Intracavity Rydberg Superatom for Optical Quantum Engineering: Coherent Control, Single-Shot Detection and Optical π Phase Shift". In: *Physical Review X* 12.2 (May 2022). ISSN: 2160-3308. DOI: 10.1103/physrevx.12.021034.
URL: http://dx.doi.org/10.1103/PhysRevX.12.021034.