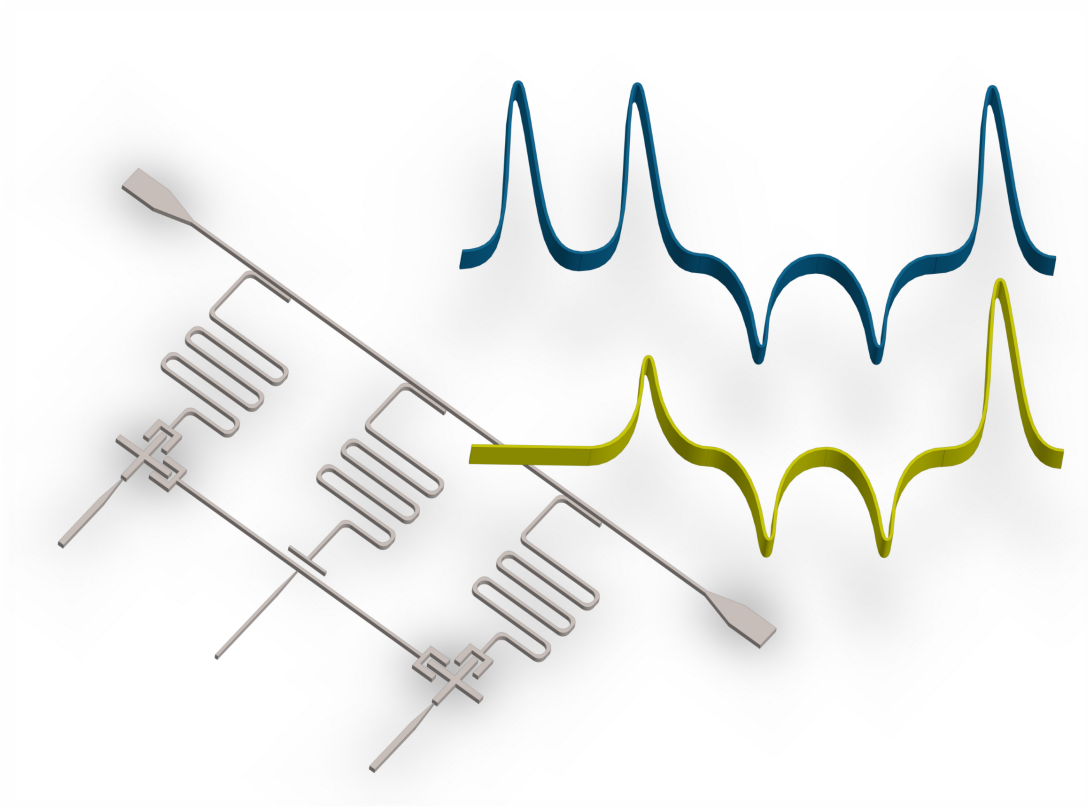
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Instrument and measurement automation for classical control of a multi-qubit quantum processor

Master's thesis in Embedded Electronic System Design

CHRISTIAN KRIŽAN

CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Department of Computer Science and Engineering
Gothenburg, Sweden, June 2019

Master's thesis

Instrument and measurement automation for classical control of a multi-qubit quantum processor

Christian Križan

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Computer Engineering division

Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Instrument and measurement automation for classical control of a multi-qubit quantum processor

Supervisors:
Philip Krantz, Ph.D. [1] and project coordinator [2]
Andreas Bengtsson, Ph.Lic. [1]

Examiner:
Lena Peterson, Ph.D. [3] and lecturer [4]

[1] Quantum Technology division, department of Microtechnology and Nanoscience
[2] Wallenberg Centre for Quantum Technology
[3] Electrical Engineering, Faculty of Engineering (LTH), Lund University
[4] Computer Engineering division, department of Computer Science and Engineering

Master's Thesis, 2019.
Department of Computer Science and Engineering
Computer Engineering division
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 (0)31 772 1000

Cover: *Two-qubit quantum processor core with RB IQ-waveforms.* © Christian Križan, 2019.
Microscope and SEM imagery: © Andreas Bengtsson, 2019. Reused with permission.

Typeset in LaTeX
Printed by Teknologtryck
Gothenburg, Sweden 2019

Instrument and measurement automation for
classical control of a multi-qubit quantum processor

CHRISTIAN KRIŽAN

Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

The recent field of quantum computing has seen great progress in the development of multi-qubit systems, with qubit usability lifetimes increasing, putting practical quantum computers on the near-horizon. As these systems take shape, scalability becomes of high priority as implementable algorithms require a multitude of qubits to function. Such systems will require precise timing control using instrument platforms for their development to continue. In this thesis, I develop and present such an instrument platform solution. This platform consists of automated instrument drivers written for Zurich Instruments' HDAWG arbitrary waveform generator and UHFQA lock-in amplifier i.e. classical electronic control systems. The drivers are integrated in the experiment-control software Labber, and verified by characterising a multi-qubit quantum processor loaded with a two-qubit DUT. One qubit is further characterised using the drivers, with extracted values of interest including: the qubit frequency, f_{01}, located at 4.302665 GHz ± 3.916 MHz; the π-pulse amplitude, Ω, of 721 mV ± 20 mV; and the energy relaxation time, T_1, of ~57.6 µs. The platform is then benchmarked in terms of duration times for typical events in a running experiment, such as Labber-to-instrument connection times (2 603 ms and 2 328 ms for the UHFQA and HDAWG respectively), compilation times (632 ms and 1 288 ms), and finally also waveform data upload times to the instruments (764 ms and 1 481 ms). The platform control was optimised in terms of upload speed, using a memory injection technique for the HDAWG. The upload time was reduced to 131 ms, typically demonstrating an averaged improvement of >91 % (131 ms vs. ≥1 481 ms). Finally, I discuss some observed potential for improvement, and speculate as to the onwards outlook regarding the future of the delivered instrument automation platform.

Keywords: quantum, processor, superconducting, Python, spectroscopy, Bloch, qubit, resonator, dispersive, Rabi.

Acknowledgements

I'd like to heartwarmingly acknowledge and thank my supervisors Philip and Andreas for putting up with my shenanigans. To Philip, thank you for taking your time to properly introduce me to this illustrious field, in particular for someone like me without prior experience in the quantum fairyland. Thank you for the lessons in all things theory, practice and in-between. To Andreas, thank you for showing me how the magic was done. Thank you for all discussions on how to properly assemble this platform, for letting me use your qubits, and especially thank you for putting up with all questions big, small and stupid.

I would also like to thank my master program responsible Per and your faith in scientific vagabonds growing into their jobs. As Händel would have put it: thank you for sparing me the cruda sorte of being stuck in Mölndal.

And finally I want to thank ETA. Because a man without wings is a man without fantasy.
Thank you for teaching me what Chalmers did not.

Christian Križan
Gothenburg, World Milk Day 2019

Contents

Terms and abbreviations

Term	**Written out**	**Brief explanation**
AWG	Arbitrary Waveform Generator	Instrument for generating output waveforms following a user-defined blueprint.
classical		In the field of quantum computing, the prefix *classical* denotes that a device is operating using classical physics. Loosely, it may be seen as synonymous with *non-quantum*.
driver		A program that enables a personal computer to control a connected device.
(c)QED	(Circuit) quantum electrodynamics	The (on chip) study of the interaction between atoms and photons.
DUT	Device Under Test	Device undergoing testing, commonly in a verified rig.
EESD	Embedded Electronic System Design	Field within electrical engineering, focusing on computational systems with dedicated purposes within electronics.
GUI	Graphical User Interface	User interface where the technical control is achieved using graphics, icons etc.
Labber		Laboratory instrument skeletal framework software that controls instruments, conducts measurements, and logs data.
lock-in amplifier		(AWG-based) carrier-wave generator combined with an oscilloscope, used for measuring signals in noisy environments.
micrograph		Microscope photography.
MSO	Mixed-signal oscilloscope	Oscilloscope combined with a logic analyser, allowing for mixed-signal observations.
Python		An interpreted object-oriented programming language.
QPU	Quantum Processor (Unit)	Processor manipulating information using quantum mechanical phenomena.

Term	Written out	Brief explanation
qubit	quantum bit	The discrete unit of information of a quantum two-level system, analogous with a bit in classical computing.
SEM	Scanning electron microscope	Microscope that produces images by scanning an electron beam, capable of relaying imagery at nanometre resolution.
TTL	Transistor-Transistor Logic	Used in this report to denote a defined voltage span for representing a high or low logical state.
VNA	Vector Network Analyser	Instrument for measuring electrical network parameters, commonly capable of measuring scattering parameters.
ZI	Zurich Instruments	Manufacturer of the UHFQA and HDAWG instruments.

1
Introduction

Acting as an introduction to the report, this chapter will begin by providing the reader with a background, explaining the current superconducting quantum computing scene and its recent need for instrument automation. Following this explanation, this chapter will describe the research question in greater detail and the intended accomplishments of this master's thesis. Finally, this chapter will conclude by outlining the executive methodology of how this thesis will attempt to answer the aforementioned research question and accomplish the goals as outlined.

1.1 Background

Quantum computing has the potential of vastly outperforming classical computing technology at certain computational tasks [1]–[6], with many algorithms executing at an exponential gain in computing efficiency [7]. A very important step towards making quantum computing practical, is extending the current potential of multi-qubit processing systems [2, 8, 9]. Superconducting microwave circuits can be seen as one of the most promising pathways of implementing a practical quantum computer [10]–[13], realising the aforementioned sought-for potential. The control of such quantum computers relies on classical electronics [14, 15], often separate from the quantum processor core, necessitating precise timings of different microwave instruments in a scalable and controllable manner. Because of this instrument control complexity, all settings in turn need to be managed via a program for instrument control and measurement automation. An example of such a program is the Python-based instrument control platform Labber [16].

It is imaginable that future experiments in quantum computing may benefit from having the quantum processor hardware abstracted, similar to how a programmer does not write machine code to run a website. This abstraction requires a software stack between the programmer and the quantum processor, an example of which can be seen with for instance the IBM Q Experience [17]. Striving to extend the potential of multi-qubit quantum computing, the Quantum Technology laboratory at Chalmers University of Technology is currently building a processor control platform suitable as a layer in such a software stack. This platform (controlled using Labber) is built using the Zurich Instruments HDAWG [18] and UHFQA [19] instruments, which as a platform has not been demonstrated before. By developing this instrument platform, we will provide researchers with a simplified method of controlling scalable multi-qubit processors and performing affiliated experiment measurements. The exemplified stack is illustrated for clarity in Fig. 1.1.

This master's thesis thus consists of developing drivers for the HDAWG and UHFQA instruments, and integrating these in the Labber instrument platform. A driver in this sense constitutes a program that allows a microcomputer to control an attached device. The final instrument setup will then be verified using both high-performance measurement equipment (such as an MSO), as well as a live quantum processor. Optimisation of the speed and quantum processor scalability is of high priority. The main application of this project thus lies in the research domain, with the overall objective of assisting scientific advancements in superconducting quantum computing.

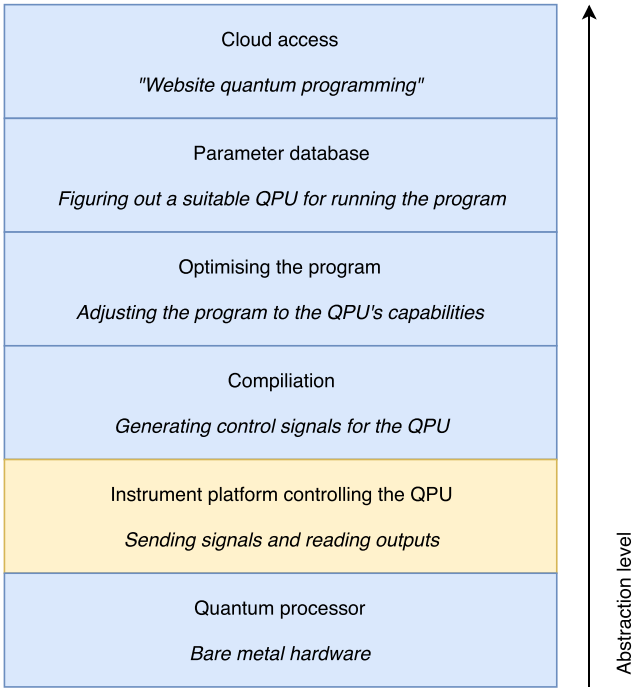


Figure 1.1: Illustration of a proposed software stack highlighting the perspective of this thesis. The Labber-based instrument platform will be at the second-lowest layer, driving the quantum hardware based on compiled quantum code. The need for a parameter database mainly stems from the fact that quantum computers are dissimilar to one-another by topology and capabilities.

1.2 Problem description

As outlined in the introductory background, the Quantum Technology laboratory is currently constructing a control system for a superconducting quantum processor using an instrument automation platform. The processor's qubits consist of microwave circuits, whose degrees of freedom (see subsection 2.1.2) are controlled using microwave pulses. In turn, this platform will control the classical generation of IQ-modulated pulses for qubit control and readout. The main research question is: how should one construct a scalable instrument-automation platform that can control a multi-qubit superconducting quantum processor setup, using arbitrary waveform generation and lock-in amplifier readout?

In the setup of such a processor we require the ability to generate qubit control waveforms and probe resonators for S_{21} responses [20] as shown in subsection 2.3.2. S_{21} represents the transmission coefficient [21] which in turn represents the amplitude received at the output port relative to the amplitude at the input port of a device. The resonator conveys information about the qubit state without having to probe it directly, which would degrade its quantum information properties. The readout should be done preferably using a lock-in amplifier in order to achieve a sufficient signal-to-noise ratio. The instruments required for the sought-for control and readout abilities will need to be controlled using an automation platform, for instance Labber. An important step on the path of multi-qubit computing is scalability [8, 9], which is of essence in any quantum processor architecture. A comparable example of a scalability issue could be imagined with a $100\,000-$qubit QPU bearing $100\,000$ cable feedlines, which is a non-scalable solution.

One must also consider the time required for uploading a specified waveform into an AWG (Arbitrary Waveform Generator) from the automation suite. For comparison, current Labber-based setups for quantum processor control and readout upload complex waveforms to AWGs by defining each discrete sample of the entire waveform curve. In contrast to the extended functionality available in the HDAWG and UHFQA instruments, the current method of AWG control sequence uploading arguably qualifies as a brute-force method which requires an increasing amount of time as processor scalability increases. The present system is prohibiting the platform from expanding in a scalable fashion as the mere upload process becomes very lengthy for even rather simple algorithms.

The instrument automation platform should also be verifiable. This verification can be done in steps by initially verifying that arbitrary waveforms from the HDAWG are correctly generated using oscilloscope readouts. The UHFQA can be verified by generating known test sequences from its AWG and verifying that these are read by and fed to the internal oscilloscope. Once the constituent parts are created, a final Labber implementation should be able to operate all constituent instruments, and thus also be verifiable on a real quantum processor. Expected readout from such an experiment is in turn well established in theory and from empirical experiments.

The implementation also exhibits additional problems to be tackled; even though the HDAWG can conditionally execute code and loop commands which in turn is a prerequisite for a Turing-complete machine [22] potentially allowing the user to execute anything, its memory is not runtime rewritable except for the samples constituting the currently played back waveform (see section 3.4). An automated program should not require the user to script waveforms for upload, implying that a standalone Python program (or similar) must be used to derive how to best generate the required waveform generation scripts for the instruments. The Python code may in turn be run as firmware in the Labber instrument platform. Based on this, one may tackle the problem of scalability within the Python program, as it can generate an execution script bearing as many waveform repetitions as one may require. Potentially parts of waveforms may be identified as repetitive and can in turn be called upon to mitigate instrument memory usage, similar to the very common Lempel-Ziv data compression techniques [23]-[25]. Such a method is further explored in section 5.1.

A drawback of conventional arbitrary waveform generators is that sequence tables cannot support control flow beyond simpler, repeated operations [26]. Furthermore, extended functionality is sought for in the form of conditional execution, arbitrary nesting subroutines and loop execution. To respond to these requests for extended functionality, an automated instrument platform should at least be able to add complexity into the repeated control execution. An additional requirement of such an instrument platform, is that control software running on classical hardware must also be able to specify waveform construction at the nanosecond level, i.e. typically a hundredth of the quantum system's coherence times [26], see section 2.4.

The reader has now been presented with a background to the problem at hand as well as its description and main research question. The upcoming sections will detail the aims of this project set in order to answer said research question.

1.3 Aim

Following the problem description in section 1.2, this project features a set of required accomplishments which will be presented as goals in subsections 1.3.1 and 1.3.2.

1.3.1 Core goals

This project will initially accomplish the following tasks:

- Implement waveform upload functionality for the Zurich Instruments HDAWG and UHFQA respectively in the Labber software, in order to realise qubit control- and readout waveforms.

  Through Labber, the operator shall be able to specify an arbitrary waveform to be uploaded into the HDAWG instrument, for instance by defining numerical values. Through Labber, values read by the UHFQA instrument shall be presented in a likewise suiting and observable manner, for instance by graphical representation of the read waveform demonstrating a mapping of a qubit's state to its respective resonator's response.

- Verify generator control- and lock-in amplifier readout waveforms, played back at room temperature using oscilloscopes and/or spectrum analysers.

  The HDAWG instrument is to be analysed at its outputs; waveform generation is considered implemented when an arbitrary waveform can be requested and played back using the instrument's driver. The output is to be verified by comparing signal amplitudes, component frequency contents, and overall shape. The goal is considered achieved when any differences in the aforementioned categories differ by at most a few percent between specified and measured waveforms. Larger differences may be acceptable provided that they can be reasonably shown to stem from specified hardware constraints or similar. The lock-in amplifier shall be able to read out S_{21} parameters from either actual or simulated qubit-affiliated readout resonators, simulated using either instruments or assemblies of simple electronic filters. The lock-in amplifier is considered verified when a pulsed carrier wave, passed through an actual or a simulated qubit-affiliated resonator, exhibits notch attenuation peaks of at least 10 dB at all frequency bands corresponding to the intended resonator frequencies as demonstrated from theory in subsection 2.3.2.

- Verify expected qubit feedthrough using the Zurich Instruments platform by demonstrating expected correspondence of input waveform versus resonator readout on a multi-qubit QPU.

  Through Labber, the read waveform from the UHFQA shall by visual inspection demonstrate harmonic matching to a control sequence uploaded to the HDAWG. Implying successful instrument automation for classical control of a multi-qubit quantum processor.

1.3.2 Auxiliary goal

Key optimisations of the core instrument driver can be seen in functionality, upload speed and scalability, desirable features of which are missing in the current instrument platform. As an auxiliary goal, the drivers are to be optimised in terms of upload speed to the instruments in order to circumvent the current method of individually defined samples uploaded to the device. This project thus also aims to accomplish the following task:

- Optimisation of the waveform generator upload process in terms of speed.

  This auxiliary goal is considered achieved when the waveform upload time is diminished by a relatively significant portion in time. The relativeness factor is given by perspective: a very complex and non-periodic waveform may require the software to define every discrete waveform sample, providing close to no upload speed reduction. A waveform with repetitive elements may however be defined as only one small set of discrete samples followed by a repeat-after-time value, greatly decreasing the upload speed. Another way to interpret the aforementioned relativeness factor could be via a change of uploading procedure altogether, provided a great upload speed increase is shown compared to what was used previously.

1.4 Limitations

A set of limitations have been identified as related although redundant for this project's execution. These will be presented and motivated in this section.

1.4.1 This project will not feature quantum programming

As the core project goals are centred about establishing an instrument platform necessary for scalable quantum processor control, this project will not include elements of quantum programming nor its optimisation for the scalable platform. During testing and verification, only known and verifiable test patterns and experiments will be executed on the instrument platform. Likewise, waveforms that shall be uploaded to the platform are to be pre-generated using existing virtual instruments. The topic of quantum programming can arguably be considered a covered topic, with work done in key areas such as suggested compilation toolchains [27], programming languages [28] and algorithms with corresponding improvements over their non-quantum counterparts [29]. Quantum programming is thus seen as out-of-scope for this project.

1.4.2 Final verification will be done on currently existing QPU hardware

Even though the intended software functionality includes scalability to an arbitrary number of qubits, no hardware modifications will be done on the existing quantum processors in the Quantum Technology laboratory in order to stretch the capabilities of the control drivers during functional verification. This implies that the final control scalability will be evaluated to an arbitrary amount of qubits currently loaded in the QPU used for final functional verification.

1.5 Method description and problem approach

A multi-qubit control setup must be able to also control a single qubit. The project will thus commence with theoretical groundwork as to how single qubit control and readout is performed at the Quantum Technology laboratory. The existing methods of single-qubit control and readout will be replicated using a currently existing quantum processor with an associated control setup, albeit using the new automation platform. A known and specified microwave waveform will be input into an arbitrary waveform generator and verified at its output using a high-performance oscilloscope located in the laboratory. The generated signal will then be verified by comparing it to the specified waveform, and adjusted accordingly if needed.

The waveform will be fed through a quantum processor setup using the HDAWG and have its corresponding resonator response analysed using the UHFQA lock-in amplifier. If the generated probing signals produce readout signals which imply corresponding (expected) qubit responses as known in theory (see section 2.3), then proof that quantum processor control and readout has been acquired by theoretic replicability.

The instrument drivers that control the platform will be optimised with respect to upload speed to the arbitrary waveform generators and scalability to multi-qubit setups. The upload speed is planned for optimisation by for instance discovering repeating patterns in the requested waveform, and requiring the waveform generator to replay these patterns. If this provides the same output as in the single-qubit verification, albeit at a more flexible and scalable uploading process, this would prove that Labber instrument drivers with higher flexibility has been created for the HDAWG and UHFQA instrumentation platform. Another plan is to investigate more optimised waveform uploading methods and attempt to implement these into the platform, using the same criteria for verification as just mentioned. The aforementioned optimisation plan brings forth the method description of the auxiliary goal as set by this project, which will be described below.

Method for the additional work

An auxiliary achievement has been outlined for this project; as this goal depends on the development workflow done during the core goal implementations, the exact execution method has been on purpose left flexible in comparison to the core goals' achievement criteria. It's problem approach has been outlined as the following:

- Optimise the UHFQA and HDAWG waveform upload speeds. Investigate how the upload process can be shortened, for instance by analysing waveform compression techniques, and attempt to implement these in the Python drivers for the AWGs. Or, swap upload method entirely if possible provided that it improves the upload speed of what currently is available. Verification is done continuously by measurement of upload times for different waveform uploads. See subsection 1.3.2 for the brief achievement criteria.

The reader has now been presented with the introduction to this master's thesis, presenting the field background, the current problem and how it's intended to be solved. As expected, the reader does run the risk of not understanding the problem's solution should adequate theory be left non-introduced. The next chapter will lay the theoretical groundwork needed to comprehend the implemented solution to the research question.

2
Theory

This chapter aims to introduce the reader to an adequate set of theory governing quantum computation. As is correctly assumed by the reader, this area is greater than what is relevant for this thesis, thus this chapter will mainly focus on an applied view of quantum computation. The upcoming sections have been written assuming that the reader has a background in embedded electronic system design, with the main goal being that the reader should understand why the implemented functions are required for successful quantum processor control and readout automation. Simply put, the reader should understand how a (superconducting, multi-qubit) quantum processor is constructed in essence, as well as how it is controlled.

In brief, this chapter is constructed from a bottom-up approach starting with an explanation as to why quantum computations are useful. As there are many different quantum computation methods, this chapter will delve into the superconducting topology as is used by the quantum processor in this master's thesis. The reader will be introduced to a circuit-level layout of the implemented qubits along with an introduction to dispersive readout techniques and qubit control methods using microwave pulse generation. Finally, the chapter will end with a theoretical explanation of the quantum analyser calibration procedure, why this process is important in terms of this thesis, providing linkage to chapter 3 (Implementation) which explains how the goals in this project were tackled.

2.1 Introductory quantum computing theory for electronics engineers

The goal of this section is to provide the necessary quantum theory required for understanding why the instruments included in this thesis have been automated the way they have. As noted, the intended background is within the area of EESD, thus the main outline will be comprehensible for readers of various technical backgrounds. It has however been assumed that the reader has a basic level understanding of algorithmic hardness, such as NP-completeness.

2.1.1 The benefits of quantum systems

This section will act as a brief inlet describing why quantum information processing is useful where its classical computing counterpart is not.

Imagine two computational entities, chips if you may. One operates using eight classical bits whereas the other one operates using eight quantum bits. For the electronics engineer, the classical entity is assumed to be represented by synchronous combinational logic and may thus assume at most 2^8 different output values (not counting driver metastates such as 'the wire is cut,' 'the wire is weakly driven' etc.). To acquire all of these output states with 100% certainty, assuming each single input state would generate a uniquely corresponding output state, one would be required to manipulate the bit input set 2^8 times by executing 2^8 known input combinations. The quantum

entity differs in that every input could represent a simultaneous state of both 0 and 1 [3, 30, 31]. This concept is known as *quantum parallelism*. However at this point in time, we simply do not know which of the 2^8 output values is present on the output port. And as is commonly known with Schrödinger's cat box [3], actually measuring the output port provides us with the one computed answer. As we now have spoiled the answer, the quantum property is lost, commonly known as *collapsing* the quantum system. For the electronics engineer, one may expand the analogy by claiming that it represents a MISO system with every possible input combination represented *at once* by the M.

The reader should be aware that the logical value of a multi-qubit setup is a multidimensional entity and is not losslessly equatable to a set of parallel bits. More specifically, the previous comparison has over-simplified the fact that n quantum bits represent an exponentially growing vector space [30]; n qubits represent 2^n mutually orthogonal quantum states in Hilbert space, whereas 2^n bits represent 2^n distinct values [2]. Subsection 2.1.2 will expand further on the major differences of bits versus qubits.

An example that demonstrates a clear benefit from having a quantum processor as opposed to a classical processor can be seen with the Bernstein-Vazirani problem [32]–[34]. The Bernstein-Vazirani problem is an example of an algorithm where a classical processor requires exponential complexity increase whereas a quantum processor only requires a logarithmic complexity increase [4], the electronics engineer would know this as a superpolynomial speedup. The Bernstein-Vazirani problem starts by defining a black box problem, i.e. a problem where some string of zeroes and ones enters a black box, some string of zeroes and ones leaves the box, and the box owner is assigned to figure out some property of that box (usually its transfer function) [34]. An additional piece of information about the box is known as a promise. In the Bernstein-Vazirani problem, we are promised that some input s yields an output as seen in

$$s \longrightarrow (r \cdot s + h) \mod 2, \tag{2.1}$$

where the task of the problem is to determine r and h [34]. We are also promised two boundary conditions regarding r and h as given by

$$r \in \{0,1\}^n, h \in \{0,1\}, \tag{2.2}$$

which dictates that r is an n-length string of ones and zeroes whereas h is either just a one or a zero. It can be shown that any optimal classical algorithm, deterministic or random, requires n attempts for finding r [4, 34] and one additional attempt for finding h [34]. Furthermore, a quantum algorithm can be shown to require two attempts only [34], while a classical computer would have to brute-force every single input bit until the correct answer is found with 100% certainty [4]. This solution is accomplished using so-called *Hadamard* quantum gates, which provide quantum entanglement thus enable the sought-for quantum parallelism via superposition. With some simplification, this parallelism is what enables such a brisk solution for finding r.

Do note that it is possible to construct algorithms for which increased parallelism does not reduce the required computational time sufficiently [2], meaning that neither quantum nor classical computers compute the answer in satisfyingly short timespans. Quantum computers are expected to be apex problem solving machines when the problem does not scale polynomially with input length, and the output is not verifiable in polynomially scaling verification time [5]. Relatively few natural problems fit these criteria (input not in P, verification not in NP) [5] but a substantial amount of quantum algorithms are still available [29]. The reader is encouraged to look at [6] should the concept of complexity theory in quantum systems be of interest; claiming that parallelism gives a quantum system its strength discards the verifiability hardness, with some claiming that the real potential in quantum algorithms stem from being able to cancel out probabilities in collapsed systems. With this subsection as motivation, we now understand why quantum computers provide advantages to classical logic in theory. As with any logic, quantum computing relies on descriptive states. The upcoming section will outline how to interpret the logical states of a quantum bit.

2.1.2 Computational logic states, from bits to qubits

The state of a single classical bit is commonly seen as 0 or 1 not counting metastates; these two discrete values usually represent that the observed wire features a relative potential to ground within two defined voltage ranges. TTL-logic levels for instance commonly define these intervals to span from 0 – 0.5 V to 2.7 – 5 V [35] for a 0 and 1 respectively (although the exact values differ depending on the implementation). We may imagine that the entire range of voltage values from 0 to 5 constitutes a continuous line. When we measure a given point on this line, we interpret the voltage value and discretise it as a logical 0 or 1, provided that the voltage was within the two defined logic-valued ranges. This discretised value is known as a bit; voltage-defined logic arguably comprises the vast majority of classic computation though not necessarily as seen with current-logic and many other methods. Similar to the bit, the full set of values available for a single qubit can also be seen as continuous in nature, and likewise the qubit is also discretised upon its interpretation. For quantum logic, the line is expanded into an opaque sphere, commonly known as a *Bloch sphere* shown for illustration in Fig. 2.1. The sphere is of unit radius, and is commonly used to model quantum two-level systems - in our case, the qubit [39]. While the classical bit could attain any value along its logical line, the single qubit may instead attain any value on and within this sphere. A vector is drawn from the absolute centre of the sphere to the point indicated by its coordinate values as given by a Cartesian coordinate system. This vector in turn represents the current logical state of the single qubit, and is known as a *Bloch vector* [39].

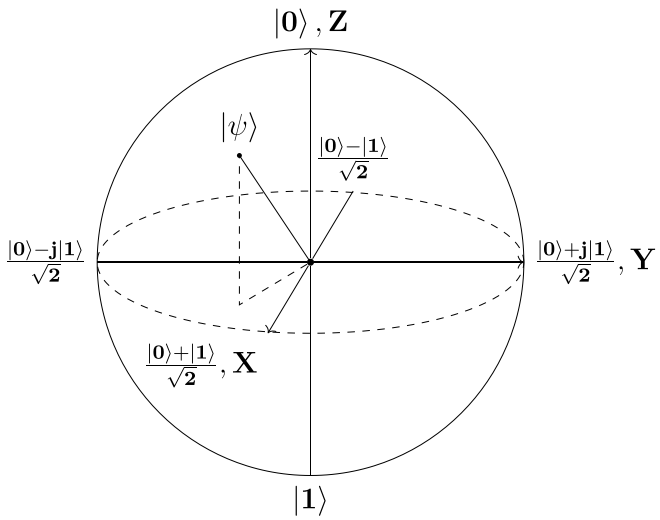


Figure 2.1: Illustration of the Bloch sphere logic model used for describing the logic state of the single qubit. The vector ψ describes the qubit state value, and is operated upon when the logic state is manipulated. The X and Y axes respectively represent superpositions between the states $|0\rangle$ and $|1\rangle$.

The discrete logic values of the single qubit are plotted on the surface of the sphere, with the north pole corresponding to $|0\rangle$ and the south pole corresponding to $|1\rangle$. This notation encapsulating 0 and 1 is expected to be alien for the electronics engineer; it is known as a bra-ket notation [36] and is common in the field of quantum mechanics. It is mainly used to describe quantum states, more particularly how quantum phenomena interact with each other using linear algebra. The notation $|0\rangle$ represents an orthogonal quantum state of the single qubit, likewise $|0100\rangle$ represents the state of a four-qubit register.

The electronics engineer may find that for many intents and purposes, the bra-ket encapsulation surrounding 0 and 1 may be ignored and simply seen as '1 or 0.' It should be noted that this

omission severely limits the potential of the quantum information contained within. As shown by [40], the general state of n qubits is described by a 2^n-dimensional vector where one dimension distinguishes a state of the n systems. Furthermore these states may show entanglement and may thus not be described as a product of the states of the n qubits. This information and more is lost by omitting the bra-ket notation.

Where the ranges on the voltage line in the previous paragraphs were known as 'logical low' and 'logical high' for the 0 and 1 discretised voltage ranges respectively, the single qubit in the superconducting topology is said to be at its 'ground state' at $|0\rangle$ and at its 'excited state' at $|1\rangle$. As will be seen shortly, an actual measured output from a quantum processor will infer a phase value θ. Similarly to a spherical coordinate system, this value corresponds to the angle between the vector and the XY-plane in the Bloch sphere. In the specific topology of this master's thesis, the discretisation from an analogue value (the phase) simply corresponds to the sign of θ, where $+\theta$ corresponds to $|0\rangle$ and vice versa.

The superconducting topology used in this master's thesis brings with it further modifications to our Bloch sphere model; as will be demonstrated in section 2.3, rotations of the Bloch vector about the X and Y axes are crucial for pointing it to specific coordinates within the Bloch sphere, i.e. realising particular logic states using our single qubit. What has been purposely left out until now is that the Bloch sphere is a rotating coordinate system, implying that the vector naturally rotates (or precess) about the Z-axis in time at the frequency of the qubit. The X/Y-operations can thus be realised by first rotating the vector to the equator, followed by idle waiting for a calculated amount of time for it to naturally rotate in the sphere. Rotations of the Bloch vector are in spirit not very indifferent to alternating the wire voltage of the classical logic entity from earlier, as both actions represent state transitions. A state transition in the Bloch sphere corresponding to π radians worth of Bloch vector rotation about a particular axis is commonly known as a π-pulse [41], likewise a rotation of $\frac{\pi}{2}$ radians is known as a $\frac{\pi}{2}$-pulse. These pulses have particular importance for this thesis as will be expanded upon later. Some of the initial QPU calibration is devoted solely to finding the shape and settings which create the π-pulse with as good precision (fidelity) as possible, as can be seen in subsection 4.1.2.

Bell states

In theory, the polar states of the Bloch sphere equates to the common digital states of 1 and 0. It can thus be speculated that a quantum computer is in theory at least as 'useful' as a classical one [30]. However, the non-polar locations of the Bloch sphere is what truly sets this state model apart from traditional logic. A rotation of $\frac{\pi}{2}$ radians for instance puts the Bloch vector on the equator, in which the logical state is in a 50-50 superposition of $|0\rangle$ and $|1\rangle$ [37]. Please observe Fig. 2.1: let's imagine that we start a computation with the Bloch vector being located in the ground state (which is the expected common case). We now rotate the vector $\frac{\pi}{2}$ radians about the Y axis heading down the positive X direction. The vector is now pointing straight at the pure X-axis. As the reader can see in the figure, our coordinates are now

$$|\psi\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle . \tag{2.3}$$

Let's imagine that we are monitoring a wire with a signal bearing (2.3). We as monitors are also capable of attaining such an exotic state if we wish. We also say that we signal $|0\rangle$ if we see a $|0\rangle$ and signal $|1\rangle$ if we see a $|1\rangle$. What do we then signal when we spot the superposition state? There is a $|0\rangle$ present, we thus signal $|0\rangle$, and there is a $|1\rangle$ present thus we also signal $|1\rangle$ at the same time. We have been put in what is known as a *Bell state* [37]. By applying a single quantum gate (a so-called CNOT-gate), we have opened up two branches of the computation. In one, we returned $|0\rangle$ and in the other one we returned $|1\rangle$. From a classical truth table point-of-view, we are treating two columns simultaneously using only one operation [37]. The state of our two-qubit system however, is now

$$\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle\,, \tag{2.4}$$

where our particular signal has been colour-coded red to show the reader our output. Please do note that what has been described is a CNOT gate. For reference, its (classical) truth table has been synthesised in Fig. 2.2. As we were originally in the ground state, the $|0\rangle$-monitoring branch of the equation output $|0\rangle$, while the $|1\rangle$-monitoring branch inverted its input ($|0\rangle$) and now outputs a $|1\rangle$.

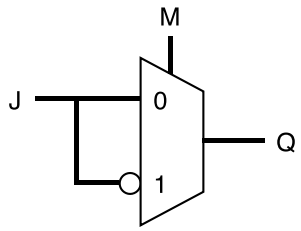


Figure 2.2: Synthesised Q <= NOT(J) WHEN M ELSE J; – ergo the logic of the CNOT gate as it would behave if it was classical.

Finally, what do we read out if we actually collapse 'the red signal?' At this point, we enter the realm of probability, and we get a 50-50 chance of the output being strictly either a $|0\rangle$ or a $|1\rangle$.

The reader has now been introduced to the logic model of the single qubit used for carrying out quantum logic. For this model to be practical, we require a component which has the ability to represent it. The last subsection of this chapter introduce the reader to the Josephson junction, along with a set of relevant circuit quantum electrodynamics theory.

2.1.3 Circuit quantum electrodynamics and the Josephson junction

The Josephson junction constitutes a crucial component of circuit quantum electrodynamics (cQED) [11, 15, 38]; as a circuit element, it features a particularly important role in superconducting quantum computing since it realises the theoretical Bloch sphere model outlined previously.

Circuit QED (more specifically the Josephson junction) allows us to separate the energy needed for forcing an artificial atom from $|0\rangle$ to $|1\rangle$, from the energy needed to force the artificial atom from for instance $|1\rangle$ to $|2\rangle$ (or some other energy band) [15]. This is accomplished via the junction's non-linear behaviour, as it features a sinusoidal potential well [39]. Familiar to the electronics engineer, a sinusoidal potential well requires different amounts of energy for traversing the distance L from the bottom up, than it would require traversing the same distance L from the middle up. There is now some method of detecting the state of the qubit, which can be applied to convey computed logic. Quantum signal processing using linear components is considered impossible [10]: by comparison, should the sinusoidal well be made quadratic, the energy needed to traverse the well would be distance-dependant only (linear to L) [31, 39]. Also, superconducting quantum circuits require the non-linear elements to be non-dissipative meaning that classical logic circuitry is unusable for quantum computation in the first place [10].

Since the Josephson tunnel junction is a circuit element that enables both non-linear [10, 15] and non-dissipative operation [10], it seems to be a natural choice for superconducting quantum computers. For clarity's sake, it should at this point be mentioned that a qubit is in essence any realisation of a quantum two-level system [31], such as two orthogonal polarisations of a photon, or more importantly an atom's ground state and its excited state [40]. By construction, the Josephson

junction is created by stacking a sub-nanometre insulator between two layers of superconducting material [10, 38] because this allows for sought-for tunnelling effects [10].

The qubit-based system logic has been realised using the Josephson junction. In the upcoming section, these junctions are now to be integrated into a superconducting transmission line circuit.

2.2 The superconducting embedded electronics

This section will cover how the specific quantum processor architecture is engineered, with a basis in the Josephson junction used to realise qubits using superconducting circuits. As this field of research is expanded upon constantly, this section will only cover the specific architecture used in this particular project and its interfaced quantum processor setup.

2.2.1 The qubit in a circuit diagram setup

Fig. 2.3 illustrates a Josephson element as would be seen in a circuit diagram [10]. Its equivalent circuit behaviour as a component constitutes a non-linear current-dependent variable inductor. Do note, 'tunable' inductance is a big topic in this field, which is not the property being described here.

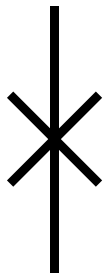


Figure 2.3: Schematic illustration of the Josephson junction. The symbol is unfortunately identical to the common schematic illustration of a circuit fault.

Manufacturing a Josephson junction produces an equivalent capacitor in parallel over the Josephson element, forming an LC-circuit as seen in Fig. 2.4. The added capacitance is known as the junction capacitance C_J, and will be of theoretical importance later. Fig. 2.4 shows the Josephson tunnel junction along with the equivalent subcircuit contained within [10].

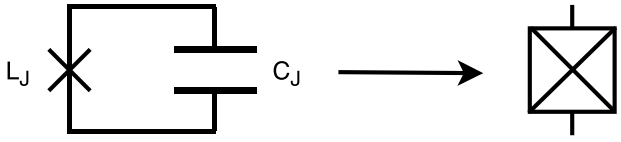


Figure 2.4: Schematic illustration of the Josephson tunnel junction along with its equivalent subcircuit.

This configuration is expanded further in Fig. 2.5, where the Josephson tunnel junction has been connected to a large shunting capacitance $C_s \gg C_J$; the circuit is now known as a transmon qubit. In practice, superconducting qubits are relatively large and are thus more easily susceptible to environmental disturbances [11]. As will be further delved into in section 2.4, this susceptibility to disturbances reduces the time for which the quantum properties are preserved in the system. It is argued that the transmon qubit has been designed the way it has been in order to reduce sensitivity to electric charge density fluctuations [11] and its sensitivity to charge noise [46], leading to increasingly longer preservation times of the system's quantum properties.

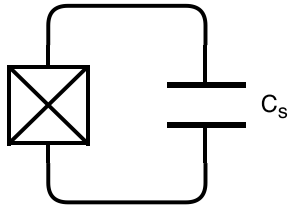


Figure 2.5: Schematic illustration of the transmon qubit.

The transmon qubit now constitutes a (weak) harmonic oscillator. As will be shown in subsection 2.3.2, its inherent resonant frequency is key to interacting with the logical state of the qubit. But, as noted in subsection 2.1.1, one should not measure the resonant frequency of this circuit directly as this collapses its quantum state. Instead, this qubit circuit is coupled to a resonator as shown in Fig. 2.6. Subsection 2.3.2 will explain why such a structure is preferred, for now it is sufficient to know that the resonator acts as the tool used for (more) safely probing the qubit to read out its logical state.

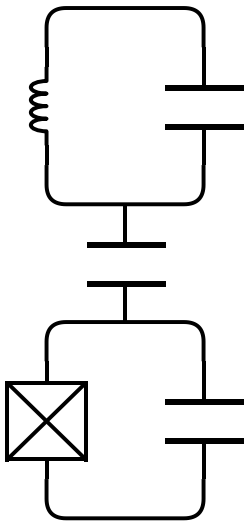


Figure 2.6: Schematic illustration of the transmon qubit (bottom) coupled to a readout resonator (top). The resonator schematic represents a quarter-wave waveguide resonator. In contrast to the Josephson junction, this resonator (and the transmon qubit) are macroscopic in size, measuring in the order of hundreds of micrometers.

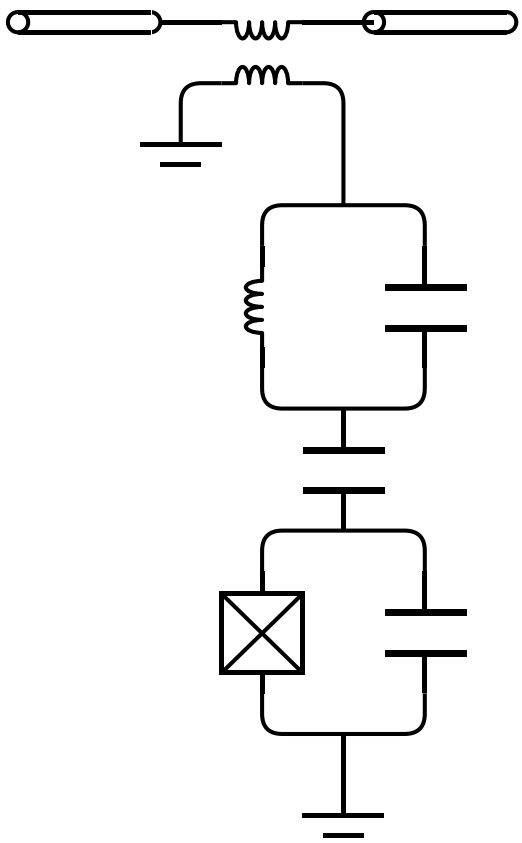


Figure 2.7: Schematic illustration of a transmon qubit coupled to a readout resonator, in turn coupled to a transmission line via inductive coupling.

Similar to two pendulums connected by a spring, the qubit resonance circuit is now able to alter the resonance frequency of the resonator, as will be of greater importance in subsection 2.3.2. The resonator now acts as a probe of sorts, thus all readout circuitry will from this point interface to the resonator. The resonator is thus connected to the outside world via an inductive coupling as shown in Fig. 2.7.

Measuring the logical state of the qubit is commonly known as a readout, and is done via a probing signal sent through the upmost transmission line in Fig. 2.7. Measuring the resonator voltage is in turn commonly known as dispersive readout [10]. This upmost node is connected to a transmission line, and the whole resonator-transmon circuit is repeated in parallel to it depending on the amount of qubits one desires. Fig. 2.8 illustrates a three-qubit setup done using superconducting topology.

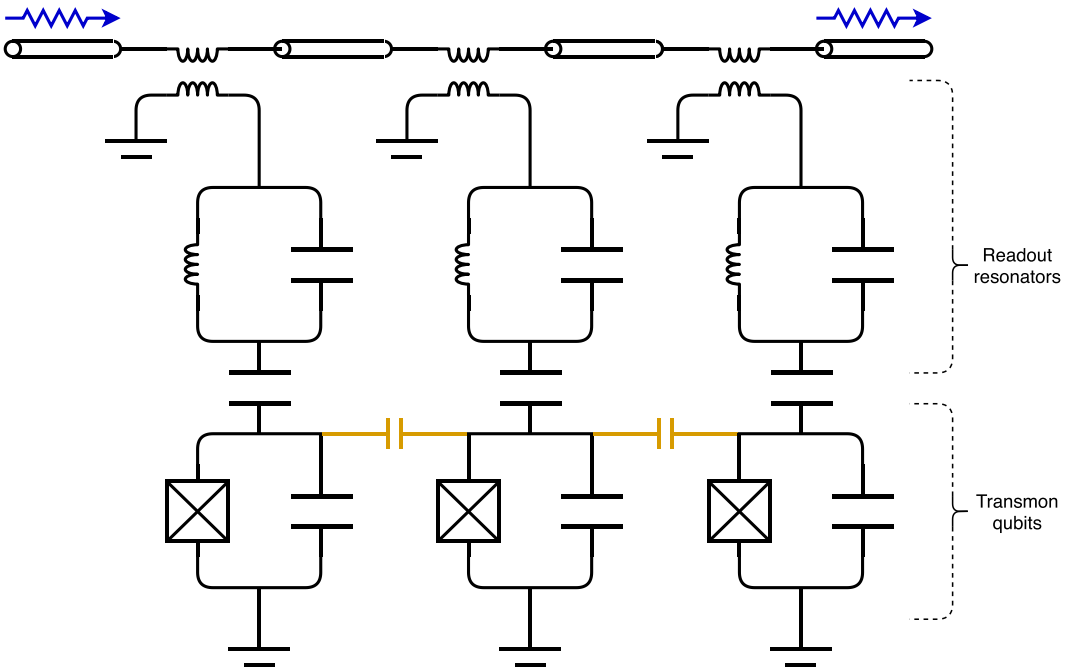


Figure 2.8: Schematic illustration of three qubit-resonator circuits connected to a common transmission line. It is important to note that the qubits are entangled thus affecting the states of each other. This is illustrated by including capacitances (yellow) between the individual transmon qubits. The transmission path has been illustrated using blue arrows at the top of the figure.

Selecting the value of the coupling capacitor between a resonator and its qubit constitutes an engineering trade-off. Larger coupling enables a stronger connection from the qubit to the resonator. As will be shown in subsection 2.3.2, a strong coupling lets through more of 'the probing signal' (the readout pulse) from the transmission line to the qubit, in turn implying that scoping the logic value of the qubit is less noisy. But, a strong coupling also implies that the qubit is more strongly coupled to its environment, in which the quantum properties of the system will be lost faster.

Fig. 2.9 depicts a micrograph of a typical sample constructed using the aforementioned schematic description. The equivalent components constituting the sample have been colour-coded in Fig. 2.10. Fig. 2.11 demonstrates a zoomed-in micrograph of the Josephson junction. Fig. 2.12 demonstrates an SEM image of said Josephson junction as connected in a so-called Xmon qubit configuration, characterised by the large cross-shape. Fig. 2.13 and 2.14 demonstrates clearer imagery of the Josephson junction along with appropriate dimensions.

The images and schematics throughout section 2.2 introduced the reader to how the Bloch sphere and related quantum information logic is integrated into a physical device. The next subsection will introduce the reader to the required circuitry for controlling the qubit, which is essential for executing quantum programs on the device.

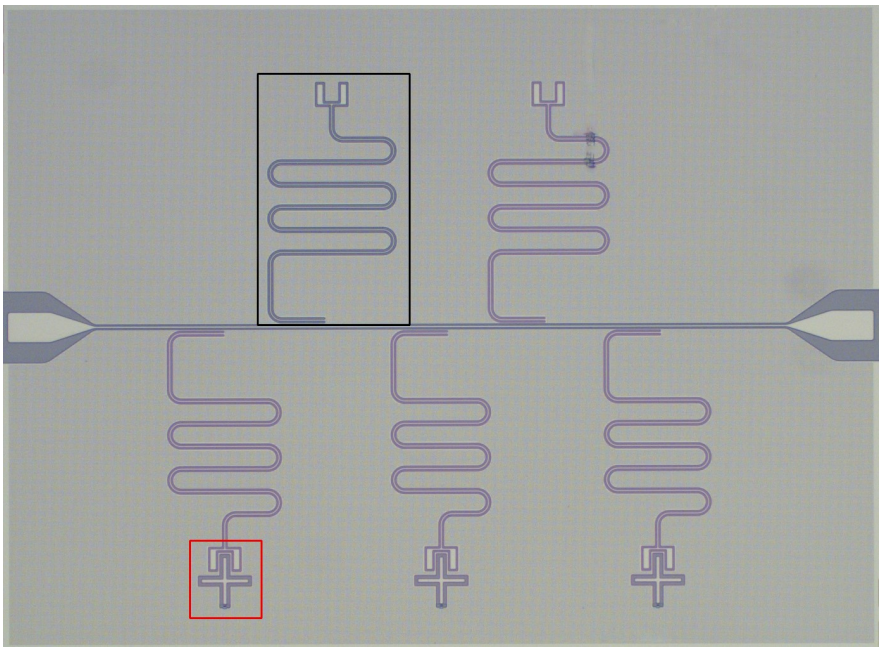
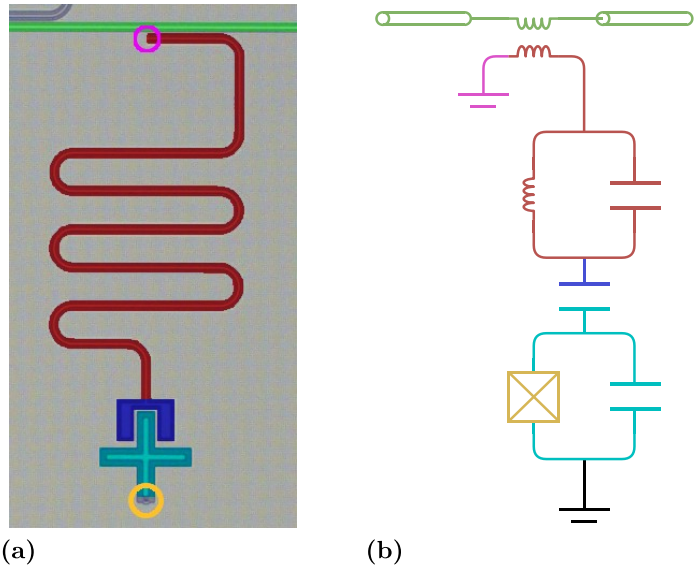


Figure 2.9: Micrograph of two bare resonators without transmons (top) and three resonators with attached transmon qubits (bottom). The resonator portion has been highlighted by a black box, the qubit portion has been highlighted by a red box. The left- and rightmost triangular shapes are connector pads. The particular type of transmons seen in the sample are known as Xmons, characterised by their large cross shapes.



(a) **(b)**

Figure 2.10: False-coloured, mirrored micrograph of a resonator **(a)** and its corresponding circuit elements using a lumped element model **(b)**.

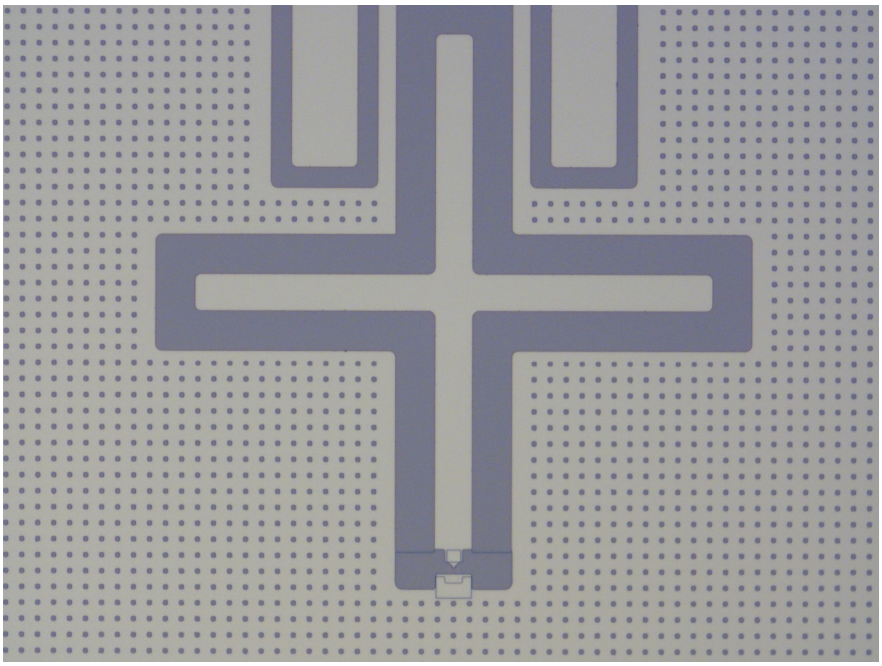


Figure 2.11: Micrograph of the Xmon transmon illustrating the Josephson junction (bottom) connected to the cross-shaped island.

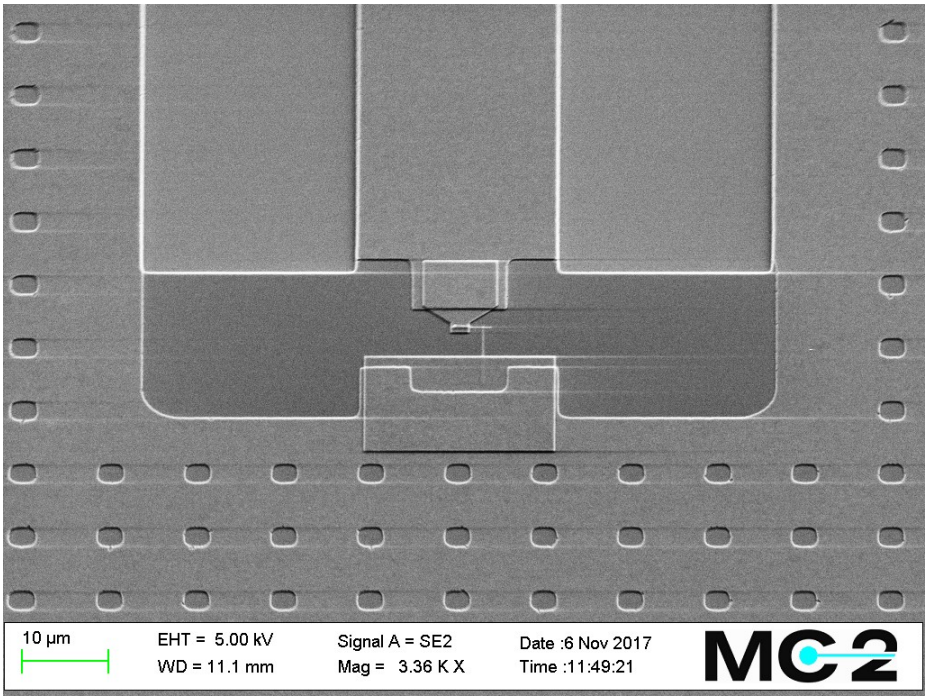


Figure 2.12: SEM image of a Josephson junction part of the Xmon. Note the tiny γ-like connection just right of the absolute centre of the image: this is the Josephson junction as will be made clearer in Fig. 2.13 and 2.14.

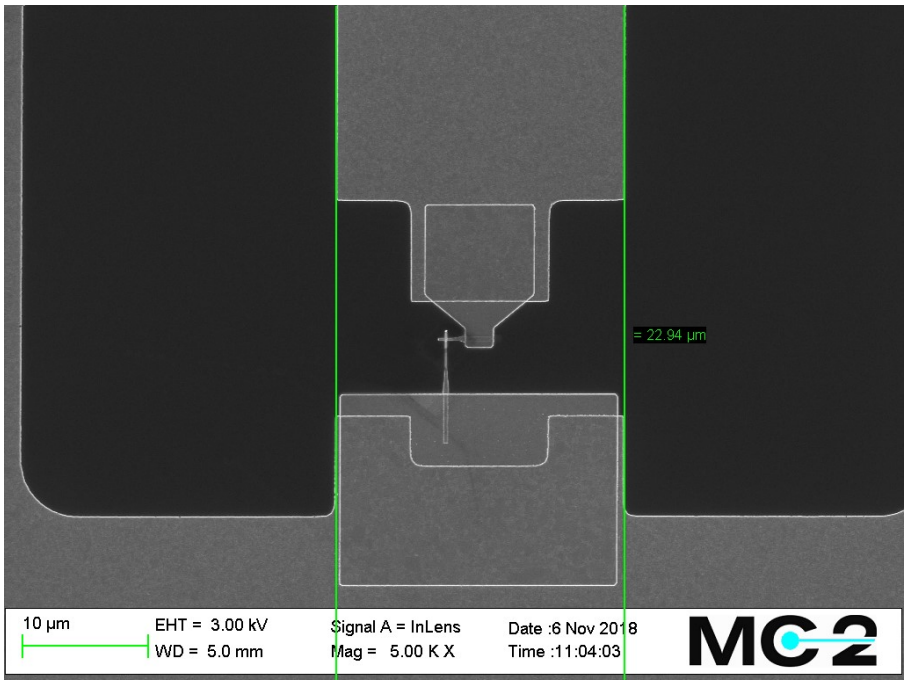


Figure 2.13: SEM image of the Josephson junction part of the Xmon, along with relevant length measurements.

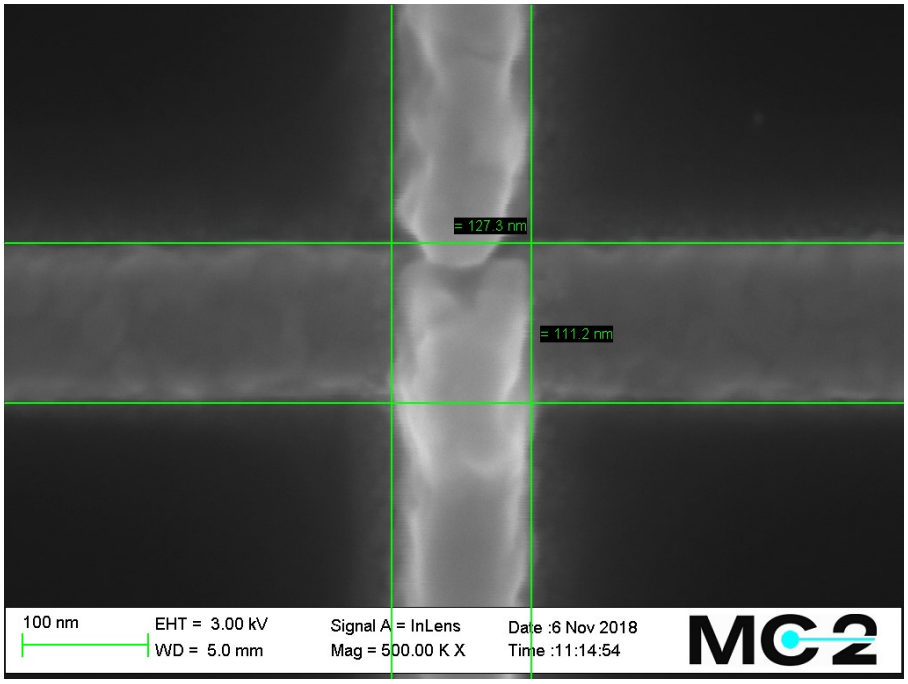


Figure 2.14: Zoomed-in SEM image of the Josephson junction itself, along with appropriate size measurements. The full junction in essence consists of two strands of overlapping aluminium (white and grey respectively) layered on top of each other, visible inside the centre green box.

2.2.2 Qubit control circuitry

The previous subsection explained how the sought-for qubit logic is realised using the Josephson junction connected to a resonator via a transmission line. Such a setup provides the user with the ability to probe (and thus listen to) the resonator for the current state of the qubit, however the important property of state manipulation is not yet realised. Qubit control can be generally represented as seen in Fig. 2.15.

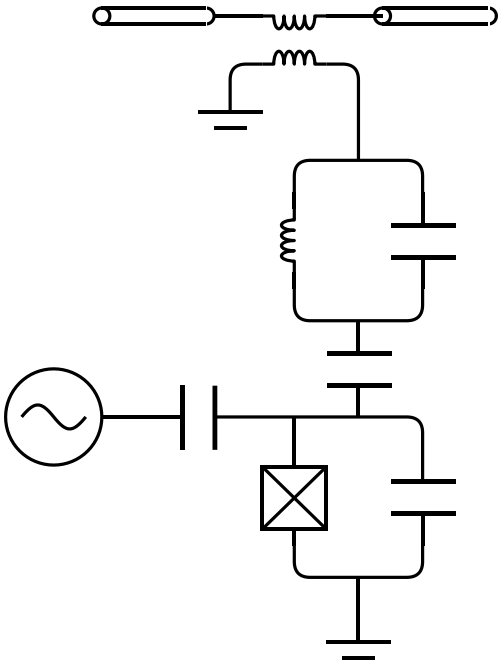


Figure 2.15: Schematic illustration of a transmon qubit coupled to a resonator with an additional control line, modelled as a capacitance seen to the left of the Josephson tunnel junction. The generator illustrates where the control signals enters the qubit setup, however omits any microwave circuitry used to attenuate/amplify the control signal. The control line is connected to the wire of the circuit known as 'the island' which in itself is a broader topic in quantum computing. Fig 2.14 demonstrates a micrograph of the actual hardware making up the island for the Xmon transmon, namely the cross shape.

The setup as depicted in Fig. 2.15 allows for logic using single qubit gates, typically by connecting the control line to an external waveform generator. Similar to classical logic, gates constitute the elements required for manipulating information through the execution of a program. A major difference to classical logic however, is that quantum gates are not bound to specific circuitry but instead to the shape of the microwave pulses used to control them. The upcoming section will leave the circuitry constituting the embedded electronic system, and instead enter the realm of quantum program execution on this device.

2.3 Quantum program execution and its setup

Subsection 2.1.3 described a hardware component capable of realising the Bloch sphere logic of the qubit. Section 2.2 in turn laid out a hardware setup which could apply the Josephson junctions in a controllable and measurable fashion. The upcoming section will describe the principles of program execution on the aforementioned superconducting quantum processor hardware.

Similar to a classical program, a quantum program equates to a list of instructions to be executed in sequence. A compiled quantum program in turn constitutes an instruction set of microwave pulses to be input in sequence onto the qubit island as seen in Fig. 2.15. As seen from theory, these pulses constitute rotations of the logic vector inside the Bloch sphere. By rotating the vector, and allowing the qubit to interact with its neighbouring qubits, one may effectively manipulate information according to a given set of instructions. In essence, one has gained the ability to execute algorithms which infers the ability to execute programs. The microwave pulses are generated by external (classical) waveform generators, containing their own configured waveform programs to be executed. Automating the waveform program construction and the executing firmware of these generators constitutes a crucial part of this thesis, and is covered in chapter 3.

The electronics engineer is familiar with the textbook conceptual computer as made up of a processing unit, memory and a connecting bus. Should we choose to store the program inside the memory, one has effectively equated the program memory. In a typical von Neumann architecture one would expect (among other specifications) the instructions of the program to be fetched, and a subset of the processor's gates manipulated depending on the instruction [42]. A superconducting quantum processor does not execute using these methods, instead the instructions themselves constitute the gates and the hardware is identical for every operation. The program memory would be equated by the waveform playback memory of external devices used for generating the pulses needed for rotating the vector inside the Bloch sphere. These waveforms may consist of a collection of sinusoidal pulses [41], where the lengths, amplitudes and phases determine the equated value sent onto the qubit. This thesis utilised mainly gaussian pulses for qubit control, as doubling the top amplitude of a gaussian distribution equates to also doubling its integral ergo the energy contained within the sent pulse. Each pulse in the sent signal's pulse train are part of (calculated) subsequences of pulses corresponding to particular quantum gates, as dictated by the compiled output of quantum program algorithm. It should be noted that many gates require at minimum two qubits; these are commonly known as two-qubit gates. As correctly assumed, two-qubit gates require two sets of pulse trains to execute the required gate. Simple quantum program algorithms can usually be drawn from left to right in so-called quantum circuits, an example of which has been illustrated in Fig. 2.16. The reader should note that the algorithm is performed from left to right however the calculations are performed right to left [39].

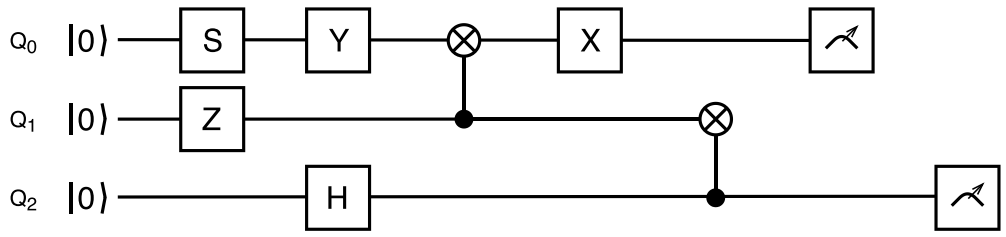


Figure 2.16: Cartoon of a fictional quantum program algorithm. The three qubits, given in the far left, begin in their ground states and are manipulated upon by control sequences illustrated in the quantum gates. The gates included in the circuit above are comprised of *Pauli-X* (X), *Pauli-Y* (Y), *Pauli-Z* (Z), *Hadamard* (H), *Phase* (S) and the CNOT-gate (crossed circles with interconnections). The Pauli gates for instance equates to rotating the logic state of the Bloch vector π radians about the corresponding axis. The CNOT gates for instance equates to inverting Q_1 if Q_0 is set (see Fig. 2.2). The rightmost dial-like boxes correspond to state value readouts.

The upcoming section will outline a set of common measurement techniques used for characterisation of a QPU. These values are commonly used in this field when comparing the apples and oranges of QPUs, and are in turn interesting side-effects resulting from the final verification stages as outlined in the goals of this master's thesis.

2.3.1 Quantum processor characterisation process

In order to efficiently compile a quantum program for a quantum processor, the compiler must typically know a set of parameters in order to be part of the software stack (Fig. 1.1). These parameters are shown in Tab. 2.1. The parameters marked red are further investigated and determined in the final verification stages of the thesis.

Table 2.1: Table of quantum processor parameters needed from a calibration run in order to compile a quantum program using the superconducting topology.

Parameter	Brief explanation
$f_{0,N}$	Bare resonator frequencies under minimal qubit influence.
χ	Dispersive frequency shift on a qubit's resonator.
f_q	Spectrum location of the qubit transition frequency, corresponding to transferring from state $\|0\rangle$ to $\|1\rangle$ (thus commonly denoted f_{01}).
Ω	Amplitude of a control pulse corresponding to a π-pulse. Please note that Ω usually denotes the Rabi frequency.
T_1	Energy relaxation time.
T_2	Dephasing time.
g	Coupling strength between qubit and resonator.
α	Anharmonicity given by DRAG pulsing.
f_{12}	Frequency corresponding to transferring from state $\|1\rangle$ to $\|2\rangle$.
η	Anharmonicity, i.e. the difference in frequency between f_{12} and f_{01} (f_q).
F	Single-qubit gate fidelity.

The calibration process for acquiring the parameters seen during the final verification of this thesis is further delved into in this subsection. The overall calibration procedure of the quantum processor is briefly described as the following:

- Frequency scan at swept power, in order to acquire resonator frequencies for which the attached qubits do/do not contribute to the resonance point. The qubits attached to the resonators skew their resonance frequencies, however at an inversely power-dependent rate. Should the scan be done at high power, the bare resonator frequency is shown. A low power frequency scan likewise corresponds to acquiring a spectral point for which the qubit may skew the resonance point in a dispersive shift. The low-power frequency value is in turn required for qubit control.

- Find the qubits using qubit spectroscopy and the previously acquired resonator frequencies.

- At the found frequency: calibrate the appearance (amplitude and/or duration) of the π-pulse, a Z-axis bitflip on the Bloch sphere. This in turn yields so-called Rabi oscillations by monitoring the output as the input amplitude/pulse duration is swept.

- Acquire coherence times; start by measuring the energy relaxation time T_1 by pulsing the qubit and sweep the readout delay until a usable output can no longer be seen. See subsection 2.3.4.

- Continue by acquiring the dephasing time T_2 using Ramsey spectroscopy. The qubit is put at the equator of the Bloch sphere, held there for some time, and flipped to a readout position.

- Apply DRAG pulsing in order to determine the coupling strength and a 'proper' π-pulse using the acquired parameters.

- Spectral locationing of the state transitions, and thus the qubit anharmonicity.

- Finally determine the gate fidelity of the system, typically using benchmarking methodology.

2.3.2 Dispersive readout, acquiring outputs from the QPU

The transmission line visible at the upmost position in Fig. 2.8 may be swept (and measured) in frequency using classical electronics, i.e. signal generators and oscilloscopes provided sufficient quality. Due to the very weak signal produced by the setup quantum electronic system, a readout is commonly performed using a lock-in amplifier. The lock-in amplifier constitutes one of the two major instruments paramount in this master's thesis (the UHFQA quantum analyser). The resonators are 'interrogated' by spectroscopy: a microwave frequency sweep is generated by the AWG inside the lock-in amplifier, and is put on the transmission line. The lock-in amplifier features an oscilloscope for monitoring the other end (the output) of the transmission line. Once the frequency sweep reaches a resonance frequency of a particular resonator, the signal amplitude will drop typically about 30 dB causing a noticeable dip in transmitted signal as much of it is routed to ground. A typical S_{21}-sweep is illustrated in Fig. 2.17, note that the values are near-arbitrary.

The resonance frequency of the transmon will contribute to the resonance frequency of the resonator as is demonstrated in (2.5). The reader should note the annotation used for the frequency shift addition χ, which is a common parameter in this field. The shift contribution is negatively dependent to the input signal power, thus the aforementioned frequency sweep was performed at a relatively high signal power in order not to be affected by the transmons connected to the resonators. This is done to acquire the spectral position of the resonators.

$$f_{\text{res}} = f_0 \pm \chi, \tag{2.5}$$

where f_0 is the bare resonance frequency of the resonator, χ is the dispersive frequency shift and f_{res} the observed resonance frequency when the resonator is coupled to the qubit. The qubit contribution χ is a desired property when scoping the resonator for the current logical state of the qubit. As is seen in Fig. 2.18 at low signal input powers, the transmon manages to shift the readout frequency's spectral location for which the signal is attenuated. This shift is known as a dispersive shift; in the superconducting quantum processor topology, it ultimately demonstrates whether the vector inside the Bloch sphere is pointing within the northern or southern hemisphere as was described in subsection 2.1.2.

Performing these readouts and measuring the logical state of the qubit constitutes a large portion of this master's thesis, namely the UHFQA readout. Acquiring the resonators' spectral positions constitutes only the very beginning of the quantum processor calibration process.

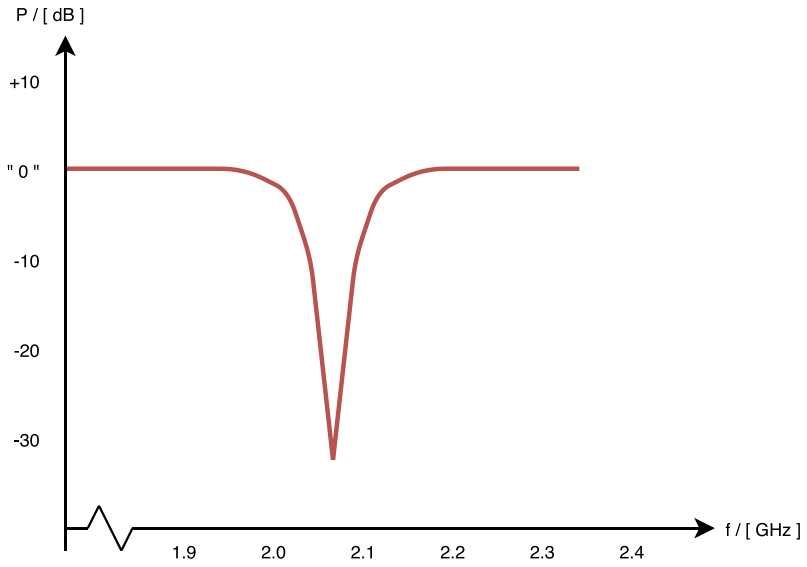


Figure 2.17: Illustration of a fictitious albeit typical resonator sweep using near-arbitrary values. This plot would be a typical output from a VNA during an S_{21} transmission characterisation. The dip in signal amplitude is characteristic of a QPU-affiliated resonator. The physical quantities of the axes are signal amplitude (P) and swept signal frequency (f).

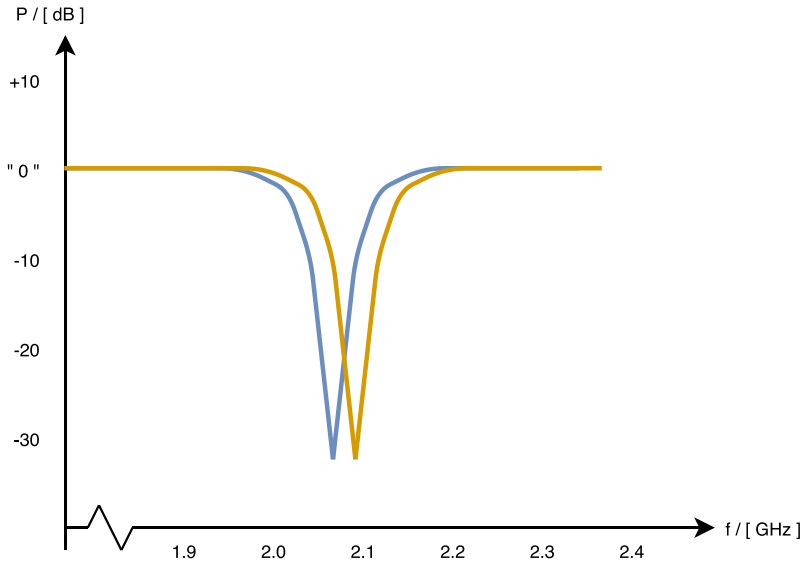


Figure 2.18: Illustration of a fictitious albeit typical dispersive shift. As the qubit is excited, the resonance frequency changes slightly. One of these lines correspond to the qubit being at its ground state, while the other one corresponds to the qubit being at its excited state. The shift is in turn known as a dispersive shift. The direction of the shift, i.e. if moving rightwards or leftwards in the spectrum corresponds to $|1\rangle$, is however uncertain from this plot alone. Thus, this plot is usually done while sweeping the readout power as well, forming a 2D-plot where the shift can be connected to $|0\rangle$ at high output power. Do note that the values chosen are near-arbitrary. The physical quantities of the axes are signal amplitude (P) and swept signal frequency (f).

2.3.3 Qubit spectroscopy, acquiring qubit frequencies

Following the previous experiments, we have acquired the spectrum location at which we may listen to a qubit's resonator in order to spot changes onto its state. It is now of interest to figure out the frequency f_q (f_{01}) we may stimulate the qubit with, in order to change its state from $|0\rangle$ to $|1\rangle$. This frequency is found using qubit spectroscopy; the experiment described in this subsection corresponds to performing a so-called two-tone spectroscopy. As the reader may have guessed, the main calibration parameter of interest following the experiment is f_q ($= f_{01}$).

In qubit spectroscopy, a two-tone experiment is performed in which the resonator is probed (and thus listened to) using one tone, while the qubit control line is swept using the other tone [44]. As the control tone is swept in frequency, we monitor the readout for a sudden spike in transmitted amplitude [20, 43]. At this point, the qubit has attained a blended state signalling that the drive tone is now matching the qubit frequency [44]. For a brief understanding as to why this happens, please observe Fig. 2.18: imagine that we are probing the DUT's readout line with a 2.094 GHz readout tone without driving the qubit (note: arbitrary values). We would expect to see a vastly diminished output as contrasted with most other points in the spectrum, because we are reading the resonator at its resonant frequency (a large amount of signal is shunted to ground). Should we however drive the qubit with a frequency corresponding to f_q, we would cause a dispersive shift in the resonance frequency. We now see a vast increase in transmitted power since the probing signal is not as strongly routed to ground. For comparison, please observe the figure again at 2.094 GHz and keep in mind that the qubit is now driven, implying that the blue line now represents the qubit+resonator transfer function. The expected output from a qubit spectroscopy experiment is thus a sudden increase in transmitted power at some drive frequency [20, 43, 45], as is illustrated in Fig. 2.19.

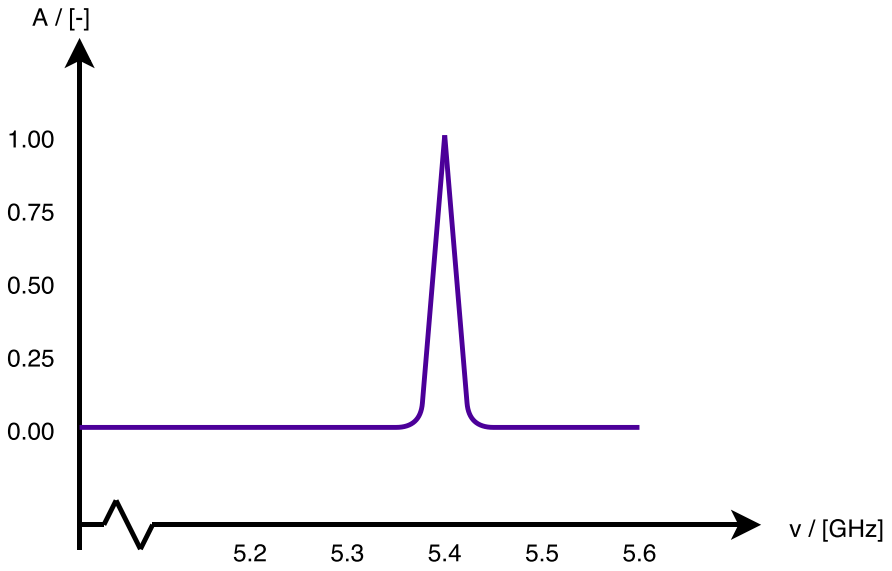


Figure 2.19: Illustration of a fictitious albeit typical output following qubit spectroscopy. As the actual amplitude may depend on a broad range of parameters, the y-axis is shown normalised. The peak's spectral width constitutes a clear spike in the ideal case. This width is in turn related to the power of which the qubit is being driven. Please note that the values used are near-arbitrary. The physical quantities of the axes are transmitted amplitude A and control-line tone frequency v.

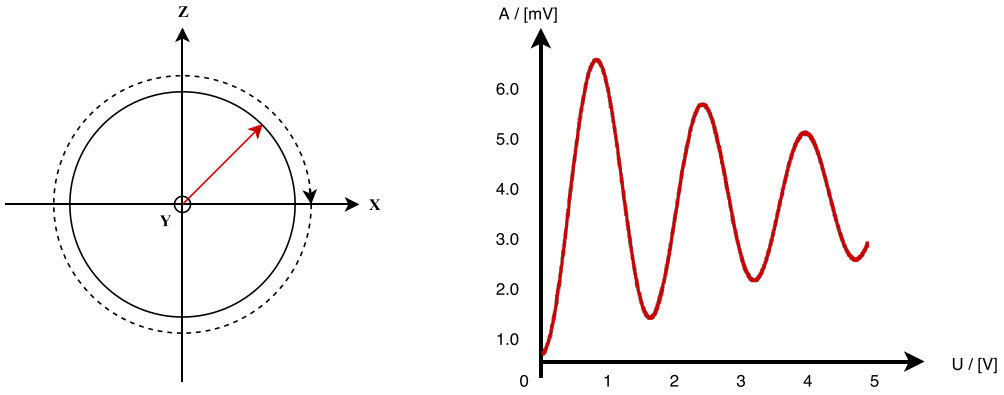
Given the qubit frequency, the next calibration process step is commonly to characterise the π-pulse for setting said qubit into the excited state. The next subsection thus introduces the reader to *Rabi oscillations*.

2.3.4 Rabi oscillations, acquiring the pi-pulse and T1

Of the electronics engineer, I ask to imagine a floating, undriven antenna. The antenna will absorb incoming electromagnetic waves, which excites the antenna. As it is excited, it emits electromagnetic waves of its own in a process you know as scattering. Similar to this is the qubit under the incident oscillatory stimulus, such as a microwave. The qubit will absorb photons from the stimulus, and re-emit them due to stimulated emission - dropping the particle in energy level. This oscillatory behaviour under incident stimulus is commonly known as Rabi oscillations.

These oscillations are a main staple in the field of quantum computing, as they provide sought-for information regarding the nature of the quantum computing system. For instance, this could include the amplitude corresponding to the control pulse (= incident stimulus) which would rotate the qubit's Bloch vector π radians about the Bloch sphere origin. Implying that we then know how much energy is required for flipping the qubit from the ground state $|0\rangle$ to its excited state $|1\rangle$. This value is commonly labelled Ω, and denotes the voltage of the control pulse.

Considering the theory as previously laid-out in this chapter, we may observe Fig. 2.20 and link the Rabi oscillations to the Bloch sphere in the following way: please observe subfigure **(b)**. It illustrates a fictitious Rabi oscillation run where the amplitude of a control pulse is swept as the amplitude of the readout pulse is read. As the input amplitude increases, the qubit rotates more and more in the θ-direction as shown in **(a)**. At U = 851 mV, we notice a change of direction of the oscillation. This corresponds to having put π-radians worth of energy onto the qubit, we note this value down and label it Ω. At this point in time, we now know the frequency of the qubit and the amplitude of the π-pulse.



(a) **(b)**

Figure 2.20: Illustrations of **(a)** the Bloch sphere seen from the negative Y-axis and **(b)** a typical (but fictitious) Rabi oscillation. The axes of **(b)** are the demodulated voltage A, i.e. the read-out and integrated qubit value; and input signal amplitude U of the control pulse, as is applied onto the qubit typically by a waveform generator. Please note that the values are near-arbitrary.

Having acquired the amplitude of the π-pulse, we have acquired the parameters needed for setting the qubit to $|1\rangle$ in a somewhat accurate manner. This allows us to measure the energy relaxation time T_1, a characteristic decay time [39] used as a merit for measuring the energy loss of the system [47].

T_1 is prompted from the system by setting the qubit in state $|1\rangle$, which is done by sending the π-pulse onto it. The readout resonator is then read immediately as the qubit has rotated to the excited state. The experiment continues by sweeping a delay time τ added after the control pulse end, and reading the resonator. This is shown in Fig. 2.21.

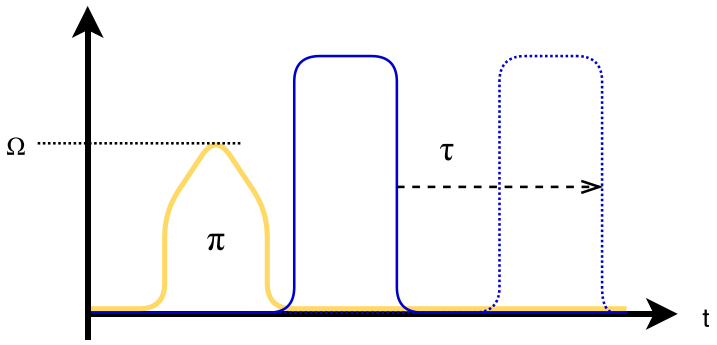


Figure 2.21: Illustration of a typical T_1 experiment. The yellow line corresponds to the control line stimulus onto the qubit, while the purple line corresponds to the readout pulse sent by the lock-in amplifier. The delay between the last sample of the π-pulse and the first sample of the readout pulse is made larger as the experiment progresses.

The read output from the procedure in Fig. 2.21 will look similar to an exponentially decaying curve, which demonstrates the rate at which the qubit is approaching its ground state. The energy relaxation time is thus modelled as a fitted decaying exponential function

$$e^{-t/T_1}. \tag{2.6}$$

Further, when normalising the Y-axis so that the maximum amplitude is one and the trend converges to zero, it is trivial to see that when t = T_1:

$$e^{-T_1/T_1} = e^{-1} = 0.368, \tag{2.7}$$

thus we easily locate T_1 on the x-axis as shown in Fig. 2.22.

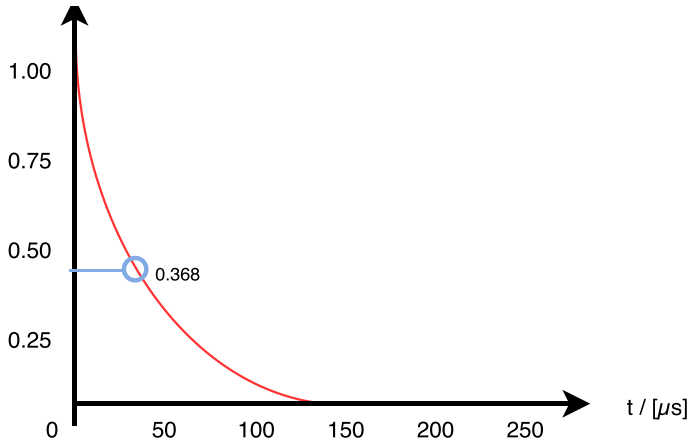


Figure 2.22: Fictitious illustration of a typical T_1 decaying function. As the energy relaxation follows an exponential decay, it becomes trivial to locate T_1 at 34.8 µs in this fictitious graph.

From now on, the reader is expected to have a basic and sufficient understanding of applied quantum computing theory in order to understand the terminology and modus operandi of the Implementation and Results chapters. Before this report delves into how the specific project was implemented, the reader is encouraged to look through section 2.4 in order to see some of the engineering caveats and most importantly the topic of coherence.

2.4 Engineering caveats and quantum coherence

A closed-off qubit with no stimulus from the outside world would be deterministic at any point in the future, provided we know its starting position and behavioural model (its Hamiltonian) [39]. Such a closed-off qubit would however be impossible to interact with, thus real-life qubits are expected to be exposed to environmental noise. Quantum computers are highly sensitive to environmental noise [48], because this outside interference interacts with the qubits - putting them in unknown, undefined or unwanted logical states. With time, the qubit's state will be very different from what is predicted by our models, and the computation is rendered unusable after what is known as the *decoherence time* (the length in time for which the system retains its quantum properties [11]). The system has to be kept coherent during the entire calculation for the output to be reliable [50]. Typical sources of noise leading to quantum decoherence can for instance be found in the microwave signals probing the qubits; these may for instance have amplitude or phase variations from some oscillator which adds thermal noise to the system [39]; or, the signals may contribute to the noise by adding photons with noise levels determined by their quantum properties [49]. Noise temperature in classical logic is a common foe in quantum electronics as well; along the computational transmission line path, a set of attenuators and/or amplifier stages are included which contribute with a vast array of noise properties known to the electronics engineer. The circuitry used in superconducting quantum computing also requires temperatures in the millikelvin-range; it follows naturally that quantum computers are operated in large fridges capable of millikelvin-levels of temperature, known as dilution refrigerators.

To isolate the qubits from the contributing noise, they are separated from their manipulative environment wherever possible. The act of trying not to disturb a qubit puts effort into engineering a trade-off between noise-cancellation while simultaneously probing the qubit resonators for their logical states. Superconducting qubits are typically also large, thus arguably hard to separate from their affecting environment [11].

Another engineering caveat of interest can be found on the topic of attenuation. As is shown in an actual quantum processor setup in Fig. 3.1, the reader might notice the heavy usage of attenuation in the signal path. A caveat of interest can be found here with attenuating the output of the instruments directly. Apart from being ultra-low voltage electronics, the typical demodulated voltage of a QPU readout is on the order of a few millivolts (as will be shown in chapter 4). However, a typical instrument is capable of supplying 1,000 times this value at minimum. It is imaginable that instruments may in turn even be optimised in measurement accuracy somewhere in the centre of their output span. By applying attenuation straight to the output of the instrument, one may use larger output voltages in their experiment. The inherent noise from the generated signal is relatively smaller, and the instrument may even be put in a more optimised voltage range. Albeit, this does not account from additional noise from the attenuator itself. The distributed attenuation in the different stages of the dilution refrigerator's cryostat, can in turn be argued to have a diminishing effect on thermal radiation propagating from the room-tempered equipment down onto the hypercold DUT [44].

The next chapter will enter the very specific domain of this master's thesis project by introducing the reader to the instrument platform's implementation.

3

Implementation

This chapter aims to introduce the reader to how the goals of this project were tackled, and by extent how the research question was answered. The initial part of the chapter will lay out a full quantum processor system description, mainly for attempts at reproducibility and experiment transparency. The experiments involving driver verification via quantum processor characterisation will likewise be presented in such a way that the eligible reader should be able to reproduce them on their own quantum processor using Labber and the ZI (Zurich Instruments) instrument platform. The sections following this part will constitute very implementation-specific descriptions of the instrument automation software.

The software portions of this chapter will start from the perspective of Labber, aiming to introduce the reader to how Labber operates the drivers. As there are parts of the drivers which are non-trivial in terms of simply setting API node values, these parts will be explained in subsections 3.2.2 and 3.2.3 while the API nodes needed for the experiments in this thesis is left as an appendix (see appendix A). Playing back waveform information using the instruments requires the construction of sequencer scripts for the AWGs; these scripts will be explained as well as their automatic generation by the instrument automation software. Finally, the implementation chapter will conclude by outlining how the auxiliary goals of this thesis were tackled, namely AWG upload optimisation and gate-fidelity benchmarking.

3.1 Quantum processor verification setup

The final driver verification as performed by this thesis was done on a multi-qubit quantum processor loaded with a DUT bearing two qubits, one control line and three resonators for the respective elements. Every component of the system setup has been outlined in Fig. 3.1.

All measurements were carried out using Labber's measurement editor, and logged using Labber's log browser. Via the drivers written for this master's thesis, I could use the measurement editor to control all required aspects of the instrument platform, described on individual-usage cases in section 4.1 for the final QPU verification. The user is free to assume that in every mentioning of an instrument being set to a particular value, it has been done using its Labber drivers through the Labber measurement editor (the experiment setup portion of the program). Specific details regarding the usage of Labber in combination with the UHFQA and HDAWG drivers for measuring is given in section 3.2.

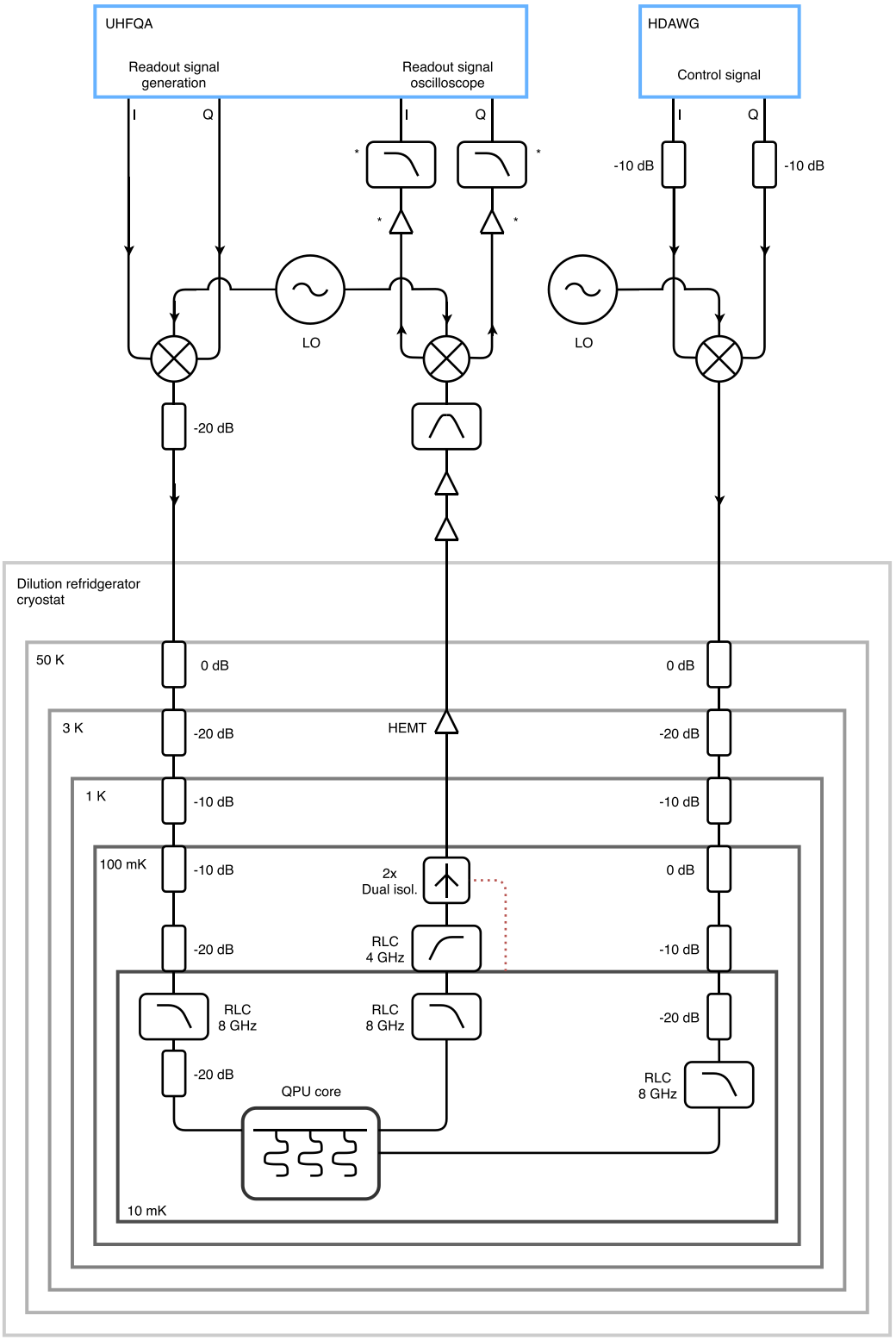


Figure 3.1: Multi-qubit quantum processor system setup used for readout- and control driver experimentation as well as the system characterisation seen in section 4.1. A component declaration is given in Tab. 3.1. All vector signal generators, the HDAWG and the UHFQA are synchronised using a 10 MHz rubidium reference clock. The asterisk-marked components close to the UHFQA's lower right are denoted *After Frequency Downconversion (AFD)* in Tab. 3.1.

Table 3.1: Component declaration for the quantum processor system layout as seen in Fig. 3.1. AFD: After frequency downconversion.

Layout symbol	**Component**
Room temperature amplifiers	Pasternack PE1522
HEMT amplifier	Low Noise Factory LNF-LNC4_8C
Dual junction isolators	Low Noise Factory LNF-ISISC4_8A
Band pass filter	Mini Circuits VBFZ-6260-S+
Low pass filters AFD	Mini Circuits VLFX580
Amplifiers AFD	Mini Circuits ZFL-100LN+
Rubidium reference clock	SRS Model FS725
Vector signal generators (IQ)	Rohde & Schwartz SGMA SGS100A 6GHz IQ
Vector signal generator (LO)	Rohde & Schwartz SGMA SGS100A 12.75GHz CW
VNA	Keysight ENA E5080A 9kHz - 9GHz

3.2 Instrument automation using Labber

Labber's role in the experimental setup is that of an umbrella program, encompassing a broad setup of drivers and sending data vectors between the different instruments under its control. Each driver controls its own instrument; Labber's purpose is to communicate to the different drivers via its own Python API [51]. The API in principle consists of a Python package which monitors specific key functions that are expected to exist in the drivers. The full Labber interface in turn operates by *Opening the instrument connection* → *Set/Get variables* (→ *Close the instrument connection*). Interfacing to the UHFQA and HDAWG instruments is no different, and is done through the Labber API.

However, the communication from Python to the ZI instruments is done using another API namely the ZI API. To the ZI API, a user may send data commands to address strings similar to URL-addresses. These URLs are instances of the ZI API as an object, fetched by Labber during the opening of the instrument connection. Since Labber always performs an instrument initialisation when connecting to a new instrument [51], it is only suiting that the ZI API object is fetched during this time. Similarly to the ZI API object, the two ZI instruments have an AWG object each which the driver may use to command their respective AWGs. The UHFQA also features a Scope object, used for controlling its oscilloscope functions. Waveforms generated in the AWGs are defining in turn using program scripts for the playback sequencers, known as SeqC-code. This waveform generation will be further presented in section 3.3.

In order to automate the instruments needed for the final quantum processor verification, a user must be able to configure the instruments to a broad range of input setups, sampling ranges, and similar. The instruments do however offer much functionality beyond what has been used to complete the tasks set by this thesis. Thus, only a subset of the full functionality offered has been implemented and is visible as appendix A to this thesis. The reader should further note that not all functions are readily available as simple API commands, for instance executing Rabi-oscillations or a T_1 measurement. Such procedures would be arduous to implement as ready-made functionality in the stock instrument since they depend on multiple other instruments. The upcoming subsections will describe the more complex functions designed for this driver by this master's thesis project, starting with how the automation suite generates control and readout pulses. This is followed by the non-API Python functions written for the UHFQA, and ends with the HDAWG functions respectively.

3.2.1 Labber readout- and control pulse generation

The Labber automation suite features a multi-qubit pulse generator. This generator is a virtual instrument used for generating arrays of waveform samples, constituting pulses used for a broad variety of quantum applications. For this thesis, this multi-qubit pulse generator has been used exclusively for generating the applied microwave waveforms on par with the limitations following subsection 1.4.1.

Using the aforementioned measurement program (*Measurement editor*) part of the Labber instrument automation suite, a user may operate the pulse generator to for instance procure a gaussian pulse. The output of the pulse generator (the gaussian pulse) may then be linked to the input of the HDAWG (the vectors which the user wishes to play back) graphically in Labber's measurement editor. As Labber executes an experiment, the measurement editor will fetch and load the gaussian pulse into the HDAWG. This procedure is identical for any cross-instrument data sent through Labber. In future applications, it is not unimaginable that a quantum program compilation stack generates the waveform data fed into the automation platform instead of the pulse generator.

The reader is now expected to understand that every time a pulse has been generated out of thin air, it has been calculated by the pulse generator depending on what the user has requested in their measurement editor instrument settings.

3.2.2 Non-API Labber functionality for the UHFQA

The main functionality needed for operating the UHFQA not covered by the simple API functions is demonstrated in Tab. 3.2. These functions are expanded upon in this subsection, presented by descriptions and algorithmic flowcharts.

Table 3.2: Non-API custom UHFQA driver functions for execution on the personal computer.

Function	**Brief purpose**
get ScopedVector	Allows Labber to execute scope acquisitions, keeps track of enabled channels, formats scoped data to a Labber-compatible format.
runScopeDataAcquisition	Handles the UHFQA oscilloscope, pushes relevant API commands to configure and acquire data from the oscilloscope ports on the UHFQA.
generateLocalAwgProgram	Generates the SeqC program depending on user-selected options.
loadLabberVectorIntoProgram	Inserts fetched waveform vectors from Labber into the SeqC program.
localProgramPlayback	Sets a playback repetition rate; the waveform will repeatedly be played back after a given amount of seconds.
compileAndUploadSourceString	Compiles the SeqC program and uploads it to the instrument. Keeps track of the compilation process.

get ScopedVector:
Acquire and send scoped data array to Labber

Expects: Labber quant object
Returns: Formatted array of scoped data

As Labber's measurement program runs an experiment, the user may choose to log an arbitrary amount of data variables. Two of these variables are the arrays ScopedVector1 and ScopedVector2, which contain scoped data from the two inputs of the UHFQA. Should ScopedVector1 or ScopedVector2 be marked for logging, then upon each iteration throughout the experiment Labber will trigger this function as a ScopedVector is get.

Since the UHFQA acts as the sole readout point for observing outputs from the QPU, it follows naturally that the ScopedVector variables will be fetched on each and every iteration of the experiment run by the measurement program. Thus, this subfunction contains calls to loadLabberVectorIntoProgram, compileAndUploadSourceString and finally runScopeDataAcquisition - in order to upload and scope the output following the upload of a new readout waveform. As seen in section 2.3, altering the readout waveform and scoping for new data is common procedure for a large subset of QPU experiments, necessitating this function's design.

The algorithm for this function is illustrated in Fig. 3.2.

---

runScopeDataAcquisition:
Acquire data array from the oscilloscope

Expects: Self instance
Returns: Averaged, unformatted acquired data (dict of lists of floats)

The lock-in amplifier, consisting of an arbitrary waveform generator and an affiliated oscilloscope, must be able to use said oscilloscope for acquiring the readout pulses sent through the quantum processor. The runScopeDataAcquisition function serves as the main oscilloscope method for the UHFQA. Bear in mind, it does not run the AWG portions of the lock-in amplifier.

During a normal run of the oscilloscope in the UHFQA, a number of records are collected almost asynchronously in comparison to the PC communication. Thus, the ZI scope module [19] must be polled to acquire the current status of the scope. Via a tunable variable in Labber (RecordAmountToAverage), the user may request how many records the user wishes to scope. In practice, this sets a duration limit for the current scope run, which will break when the oscilloscope has been re-run sufficiently for acquiring the requested amount of records. The main purpose of acquiring more records for the driver is to average the measurements using element-wise averaging (mainly to reduce noise).

Parallel to this, the scope module also has a progress metric, which also signals when the scope is finished. Together with the amount of records gotten, these constitute the main criteria for halting the oscilloscope run. At this stage, should the amount of records be inadequate, the entire function is re-run. These criteria are checked under the influence of timeouts and attempt iterators, in order to avoid stuck-at-looping faults. Such faults commonly and easily occur when the user has misconfigured the oscilloscope trigger in the experiment setup, meaning the function will loop as no records are acquired.

The quickest poll to the scope module from the driver returns about 30 records. Should averaging not be an issue with respect to the manipulation of the acquired data, the user is encouraged to use about 30 records as minimum averaging, as this adds no delay in comparison to even single-shot measurements. Records fetched in excess of the selected averaging variable are simply discarded otherwise. Likewise, the poll to the scope module returns a full status dict containing data about the measurement, which is also discarded. The algorithm of this function is visible in Fig. 3.3.

---

generateLocalAwgProgram: Generate SeqC program

Expects: Self instance
Returns: (None)

The generateLocalAwgProgram function creates a single string, with appropriate tags at various locations. These tags are replaced by different functions throughout the driver, resulting in a finished SeqC program. In order not to upload non-compilable code to the UHFQA, these tags are left as commented lines. The string, known as a source string [18, 19], is stored in the self instance for later usage by the compileAndUploadSourceString function.

Because the generateLocalAwgProgram function generates the entire source string template, it also serves as a SeqC-program reset at specific instances in the driver.

Illustrating this function in an algorithmic plot is unnecessary: its only action is to generate a string. A typical finished source string (a SeqC program) is illustrated in Fig. 3.12.

---

loadLabberVectorIntoProgram: Load data vector from Labber into the SeqC program

Expects: Self instance, Channel selection (int)
Returns: (None)

This function modifies the source string (SeqC program) as generated by generateLocalAwgProgram for the UHFQA. The appropriate code lines are uncommented depending on for instance which channels the user has activated. This function also marks the channels as valid when they are loaded, which is used to configure the oscilloscope, since the returned data dict is altered depending on which combination of input channels received data.

This function also does the main insertion of the data previously loaded from Labber in the get ScopedVector function. This is done by flushing the currently loaded vector data in the source string and inserting the fetched data anew. The finished SeqC program is stored in the self instance.

The function has been illustrated in Fig. 3.4.

localProgramPlayback:
Internal waveform playback repetition

Expects: Self instance, Current playback status flag (int)
Returns: Current playback status flag (int)

In certain measurements, such as sweeping for resonator locations using the UHFQA, the HDAWG is superflous as it would merely act as a trigger generator. The localProgramPlayback function takes care of this by inserting a local playback repetition into the SeqC program. This enables for instance doing a resonator sweep using only the UHFQA instrument.

Should the rate be set to 800 ns for instance, then a loaded waveform into the UHFQA waveform memory is to be played back at a given UHFQA output every 800 ns. This necessitates that the loaded SeqC program is to be rewritten from the instrument driver, which this function also does.

The internal playback algorithm at its current state provides a playback resolution close to the theoretical maximum (1/sample playback rate) as is shown in subsection 4.2.3.

The currently implemented algorithm is demonstrated in Fig. 3.5.

---

compileAndUploadSourceString:
Compile and upload SeqC program to the instrument

Expects: Self instance
Returns: (None)

The compile and upload source string function is identical for both the UHFQA and the HDAWG. Provided that the generateLocalAwgProgram function ran, the compileAndUploadSourceString function attempts to fetch a defined source string from the self instance.

This function initiates the compilation procedure, continuously fetching the status from the AWG module's compiler. The compiler in turn has a set of status flags for indicating its current status [18, 19]. Likewise the compiler also provides compiler-specific errors; these are fetched if any and simply fed onto a raised exception from the function. Warnings are fed onto Labber's instrument log, similar to a console where the user can see driver printouts. After the compilation stage, there have been no identified usage cases where the compiled output could not instantly be uploaded to the devices, hence the upload functionality. The upload procedure continuously fetches status information as well as upload progress and relays this onto the instrument log.

As the AWG defaults to an off-state, some additional code fetches a status package from the UHFQA, derives whether the AWG was playing before compile-and-upload, and resets it to this value after the function has been executed.

The algorithm corresponding to this function is illustrated in Fig. 3.6.

3.2.3 Non-API Labber functionality for the HDAWG

Similar to what was mentioned in subsection 3.2.2, the main functionality needed for operating the HDAWG that is not covered by the simple API functions is demonstrated in Tab. 3.3. These functions are likewise expanded upon in this subsection, presented by descriptions and algorithmic flowcharts.

Table 3.3: Non-API supported custom HDAWG driver functions for execution on the personal computer.

Function	Brief purpose
generateLocalAwgProgram	See Tab. 3.2. The function itself differs from the UHFQA version in content albeit not in purpose.
loadVectorsFromLabber	Fetches waveforms from Labber, calculates whether the sequencer program requires recompilation and reuploading depending on loaded vectors. Sets which vectors are to be injected into the HDAWG sequencer memory.
compileAndUploadSourceString	See Tab. 3.2.
writeWaveformToMemory	Prepares injectable interleaved packages of waveform data, enables the HDAWG to accept said package, injects the package accordingly.

generateLocalAwgProgram: Generate SeqC program

Expects: Self instance, Internal repetition rate (double), Buffer length (int)
Returns: (None)

The generateLocalAwgProgram for the HDAWG has a similar purpose to that of the same function in the UHFQA. A key difference is that the HDAWG driver's function contains semi-automated characterisation options which assist in setting up the measurement program in Labber. Should the user request a characterisation run, the driver will default to HDAWG channels 1 and 2 for running the measurements. The difference in SeqC output from the function can be seen in Fig. 3.10 versus Fig. 3.11.

The function supports an internal repetition rate similar to what is done in the UHFQA, which is received as an input parameter. The same code is utilised for calculating the delay needed during characterisation runs such as Rabi oscillations or a T_1 measurement. Because the HDAWG uses memory injection for uploading waveforms, the SeqC program will be different in that a predefined zero vector will be allocated for the memory injection. For this, the function requires a buffer length input parameter, which consists of how many samples are needed for storing the longest (active) waveform that will be uploaded.

The finished product of the generateLocalAwgProgram is a SeqC skeleton stored in the driver's self instance, constructed according to variables as set in the measurement program. The algorithm generating the SeqC program for the HDAWG is illustrated in Fig. 3.7.

loadVectorsFromLabber: Load data vectors from Labber

Expects: Self instance
Returns: (None)

During HDAWG startup, a set of vectors are tagged by Labber as the initial set procedures take place after connecting to ('opening') the instrument. This procedure tags a set of channels which will be used during the experiment. The loadVectorsFromLabber function goes through all tagged channels, stores the previous waveform, fetches new waveforms from Labber, and if finding the waveforms to be different - signals the driver to update the waveform requested by the measurement program. This is in turn done through the writeWaveformToMemory function.

loadVectorsFromLabber also administrates all flags in the driver which control whether a vector triggers a SeqC program compile-and-upload. If the requested waveform (is valid and) differs in length in comparison to the currently loaded waveform in the HDAWG memory, a new SeqC program must be uploaded to the sequencer. Also, this function administers the flags that set which waveforms are to be uploaded by the writeWaveformToMemory function.

Fig. 3.8 illustrates the algorithm depicting how vectors are loaded from Labber for the HDAWG.

---

compileAndUploadSourceString: Compile and upload SeqC program to the instrument

Expects: Self instance
Returns: (None)

See compileAndUploadSourceString in subsection 3.2.2: this function is identical.

---

writeWaveformToMemory: Inject waveform vector data into allocated memory

Expects: Self instance, Current playback status flag (int)
Returns: Current playback status flag (int)

The writeWaveformToMemoryFunction serves as the primary method of actual waveform data uploading for the HDAWG, using direct sample injection into a pre-allocated memory space via the sequencer program. It replaces sequencer-defined vector data (as done in the UHFQA) in order to optimise the upload speed to the device.

The algorithm is illustrated in Fig. 3.9. The function itself is described thoroughly in section 3.4.

3.3 AWG sequencer code generation

The Zurich Instruments UHFQA and HDAWG both require waveform playback for conducting experiments; the HDAWG must play back control waveforms for the quantum gates whereas the UHFQA generates the readout pulses that probe the readout resonators (see subsection 2.3.2). All but the simplest of sinusoidals must be generated using so-called sequencer programs in order to run on the ZI instruments. These programs are typically used to specify any arbitrary waveform to be played back, or the location of a CSV file containing waveform data to be loaded and played back. The resulting output of a sequencer program consists of compiled code for an FPGA soft-core. The user scripts these programs using the proprietary language *Sequence C* (SeqC), which is compiled and uploaded to the selected unit upon request. The language draws heavy inspiration from C-like syntax, although puts heavy restrictions on variable usage.

As noted in section 1.2, an automated system should not require the user to script waveform programs. Thus, the drivers produced for this thesis have automated the process of SeqC generation. As the user controls the instruments (and experiment) through Labber, the SeqC programs are automatically generated depending on the requested functionality by the experiment. A Rabi-oscillation experiment will for instance generate one SeqC program for execution on the HDAWG, while an unspecified waveform playback experiment will trigger the default SeqC program generation. As can be seen in section 3.2, these programs are also automatically compiled and uploaded to the instruments when deemed necessary by the drivers.

For further reference, the reader should be aware that waveforms defined in the SeqC programs may also consist of explicitly defined vectors of data. Another method of waveform definition consist of allocating memory (by defining vectors of zeroes) and filling these with waveform data packages during runtime using the personal computer. The main advantage of the latter method is that the program does not require recompilation nor reuploading to the devices, thus this method is usually the fastest possible waveform upload method [18, 19]. These latter two methods were used for the final experiments of this thesis; the UHFQA used data vector definitions while the HDAWG used memory allocation with runtime data vector writing. See section 3.4 for a brief mention as to why the UHFQA was not configured to use the most optimal method of waveform uploading ergo runtime data injection.

The following subsections will present the SeqC program skeletons as automatically generated by the Python drivers.

3.3.1 HDAWG driver SeqC skeletons

A typical output of a default SeqC program skeleton is given in Fig. 3.10. The algorithm implemented for generating this skeleton is presented in Fig. 3.7. As can be seen in Fig. 3.10, a specified number of waveforms are defined as vectors of zeroes, waveform memory allocation if you may. These are rewritten at runtime using vector data memory injection, the algorithm of which is presented in Fig. 3.9. Even though the instrument offers sequencer rerun functionality mitigating the usage of a while loop, it is still recommended to run sequencer programs using while loop methodology as this improves on the playback jitter greatly [18, 19].

```
// Waveform definitions
wave w1 = zeros(1000); // End of w1 definition.
wave w2 = zeros(1000); // End of w2 definition.

while(1){
   playWave(1, w1, 2, w2);
   waitWave();
   wait(100);
} // End of while loop
```

Figure 3.10: SeqC skeleton as automatically generated by the HDAWG driver during default mode (no preset QPU characterisation set). The wave w# definitions are expanded depending on how many waveforms the user has requested to play back in the Labber measurement program; in this example the user specified two waveforms for playback. The zeros(1000) are automatically expanded to the amount of samples needed to accommodate the longest specified waveform. In this example the user requested to play back two waveforms with the longest one consisting of 1000 samples. playWave specifies what ports the waveforms should be played back on, and will expand to accommodate all specified waveforms. In this example, the user requested some internal waveform playback repetition, presumably about 350 ns at the default instrument settings. wait(100) corresponds to 100 ticks of the internal sequencer clock; the tick amount is calculated by the driver depending on the currently loaded settings in the instrument. Should no such delay be requested, this line is removed. This SeqC-code lacks synchronisation with the UHFQA, which is further discussed in subsection 5.2 as this is a potential improvement over the current experiment methods.

The HDAWG features ready-made functionality for executing some QPU characterisation routines, such as Rabi-oscillations and T_1 measurements. These routines are requested by setting their respective flag (boolean) inside the Labber measurement editor. This will trigger an if-statement inside the SeqC-generating function within the driver, causing it to instead generate the SeqC-program skeleton as is visible in Fig. 3.11.

```
// Waveform definitions
wave w1 = zeros(1000); // End of w1 definition.
wave w2 = zeros(1000); // End of w2 definition.

playWave(1, w1, 2, w2);
wait(100 + getUserReg(0));

playWave(1, w1, 2, w2);
wait(100);

setTrigger(1); setTrigger(0); // Mark for the UHFQA
waitWave();
```

Figure 3.11: SeqC skeleton as automatically generated by the HDAWG driver during QPU characterisation mode. This mode is triggered when the user request a Rabi or T_1 measurement. In contrast to the default mode depicted in Fig. 3.10, this mode only generates two waveform definitions and one marker for the UHFQA to use as a readout trigger. The zeros(1000) are automatically expanded to the amount of samples needed to accommodate the longest specified waveform. During characterisation, this waveform will typically be a gaussian pulse. In this example the default setting of two waveforms were defined with the longest one consisting of 1000 samples. playWave specifies what ports the waveforms should be played back on. getUserReg(0) fetches a value from the AWG instrument's user register 0 as this is among the only methods of executing arbitrary variables in the sequencer; typically this value expands in a characterisation measurement in increments set by the user using the Labber measurement program. Expanding this waiting time is normal procedure during for instance a T_1 measurement. The driver has support for a so-called Ramsey measurement, used for characterising T_2 albeit not done for this thesis. Should a Ramsey measurement be requested, another waveform will be played before the readout trigger is sent (using setTrigger). Should a Ramsey measurement not be requested, the green lines are omitted. wait(100) corresponds to 100 ticks of the internal sequencer clock, this tick amount is calculated from the required playback length of the allocated waveform space by the driver. This way, the second phase of the program (setTrigger if Rabi, playWave if Ramsey) is executed at the very finishing sample of the initial wave. At the start of any characterisation, user register 0 will be 0 implying no delay between the first waveform and second waveform / trigger. This method puts a minimum delay resolution in any Rabi and Ramsey measurement corresponding to typically 3.3 ns, depending on the current sequencer clock rate settings.

3.3.2 UHFQA driver SeqC skeletons

Identical to what was shown in subsection 3.3.1, the UHFQA driver also generates SeqC code automatically albeit less dependent on the chosen measurement experiment. The major difference between the UHFQA and HDAWG in terms of the SeqC skeleton stem from the fact that the HDAWG uses vector memory injection while the UHFQA loads and defines waveforms into the SeqC code. Fig. 3.12 demonstrates the SeqC program skeleton as automatically generated by the UHFQA driver. Do note the waitDigTrigger call at the top of the skeleton: a similar call is missing in the HDAWG code which is arguably inflexible. This is discussed further in section 5.2.

```
const RSC = 1/0.75; // Range scaling

// Waveform definitions
wave w1 = RSC * vect(0.45, 0.55, 0.65); // End of w1 definition.
wave w2 = RSC * vect(0.45, 0.55, 0.65); // End of w2 definition.

while(1){

   // if (t == 0) {
      waitDigTrigger(1,1)
      setTrigger(1);
      playWave(1, w1, 2, w2)
      waitWave();
      setTrigger(0);

   // t = t + 1;
// } // End of t-swap

   // if (t == 1) {
      // wait(100);
      // t = 0;
   // } // End of t-reset

} // End of while-loop
```

Figure 3.12: SeqC skeleton as automatically generated by the UHFQA driver during default operation. Note the vect() functions: in a typical measurement, the user requests the I and Q waveforms corresponding to the readout pulse. These waveforms are loaded via Labber's measurement editor, and the vect contents are replaced with each individual sample of the two waveforms. The user may select to only apply one waveform, in which the second definition will be omitted. The parts marked in green correspond to the user setting an internal waveform playback rate. Should such a playback repetition be requested, the green sections will uncomment, and the waveform will be replayed at a calculated interval present at wait(100). The vectors w1 and w2 are in turn zero-padded with the intent that the playback repetition rate reaches near sample-rate resolution, as is visible in subsection 4.2.3. The seemingly interesting way of resetting t stems from the fact that 'setting a wait' inside the main loop does not execute as expected, this is presumed to be due to compiler optimisation or similar phenomena. Thus, the t variable is used in order to acquire a playback repetition delay. The range scaling parameter may be changed using a custom response to the range setting command; should the user request the instrument to change output voltage range, a condition is triggered in which the 0.75 factor in this SeqC skeleton is modified accordingly. This range scaling exists in order to match the output voltage of the UHFQA to that defined by the multi-qubit pulse generator. The waitDigTrigger function exists in order to await signalling from the HDAWG, as often the readout measurements must occur as soon as possible when an algorithm has executed on the qubits.

3.4 AWG waveform upload speed optimisation

As explained in 1.3.2, this project proclaimed an auxiliary achievement in the form of waveform upload optimisation in the created drivers. This section will outline how the solution to this auxiliary goal was implemented.

As explained in section 3.2, both the HDAWG and UHFQA instruments contain AWGs for generating control- and readout pulses respectively. These AWGs are controlled through a sequencer program as explained in section 3.3. Generation of the waveforms may be done through the program directly as shown in subsection 3.3.2, however this solution is very inflexible: the sequencer

program is not real-time rewritable, it requires compilation and device uploading before any waveform data can be output. As shown in section 4.3, this process is very lengthy in comparison to the time expected for simply uploading packages of playable waveform data. Both device specifications feature memory pre-allocation for data injection, meaning that provided the user has defined a set memory allocation - playable vector data may be injected from the personal computer during runtime. Such an upload method is expected to greatly improve on the waveform upload speeds, as no recompilation of the sequencer program nor SeqC uploading is required.

The memory allocation was done in the SeqC generation stages, as shown in section 3.3. The vector to be uploaded is fetched from Labber and analysed for its length. The SeqC program is automatically written bearing an arbitrary amount of zero vectors, as visible in Fig. 3.10. Using the algorithm in Fig. 3.9, the program uploads the vector data more or less during runtime, foregoing the need for SeqC compilation and upload as long as the vectors' sizes remain identical. A natural question to ask following this is why one may not simply allocate all of the available memory space and simply upload what sizes of waveform is needed. The simple answer is due to how the waveforms are played back. Observe the waitWave() command in Fig. 3.10: should the pre-allocated vector be larger than the loaded vector, the final product will contain a large zero-padding in the end. For many categorisation experiments such as Rabi oscillations, such a delay would be greatly detrimental to accurately measuring the QPU performance parameters as is for instance shown in subsection 2.3.4. Through experiment design, the need for SeqC updating has been kept to a minimum and thus a great optimisation in terms of upload times has been accomplished as is shown in section 4.3. The memory uploading procedure has been further optimised by using a feature known as interleaving [18]. Essentially the waveforms to be uploaded are separated two-and-two, and appended one onto the other. For eight waveforms, the driver now only has to upload four packets of two.

The AWG for readout waveform generation inside the UHFQA also supports memory injection according to the documentation of the device [19]. However in a personal written conversation, Philip Heringlake at Zurich Instruments verified on the 14th of May that this feature is currently bugged and thus this auxiliary goal was not achievable for the UHFQA. The main established method of waveform upload optimisation consisted of foregoing the compilation and upload stages of a typical upload, which none of the three other upload options [19] do. The upload time bottleneck was instead bypassed via experiment design, such as sweeping the local oscillator of the mixers instead of defining new waveform vectors.

We have reached the end of the Implementation chapter. The upcoming section will present the results as acquired by this thesis.

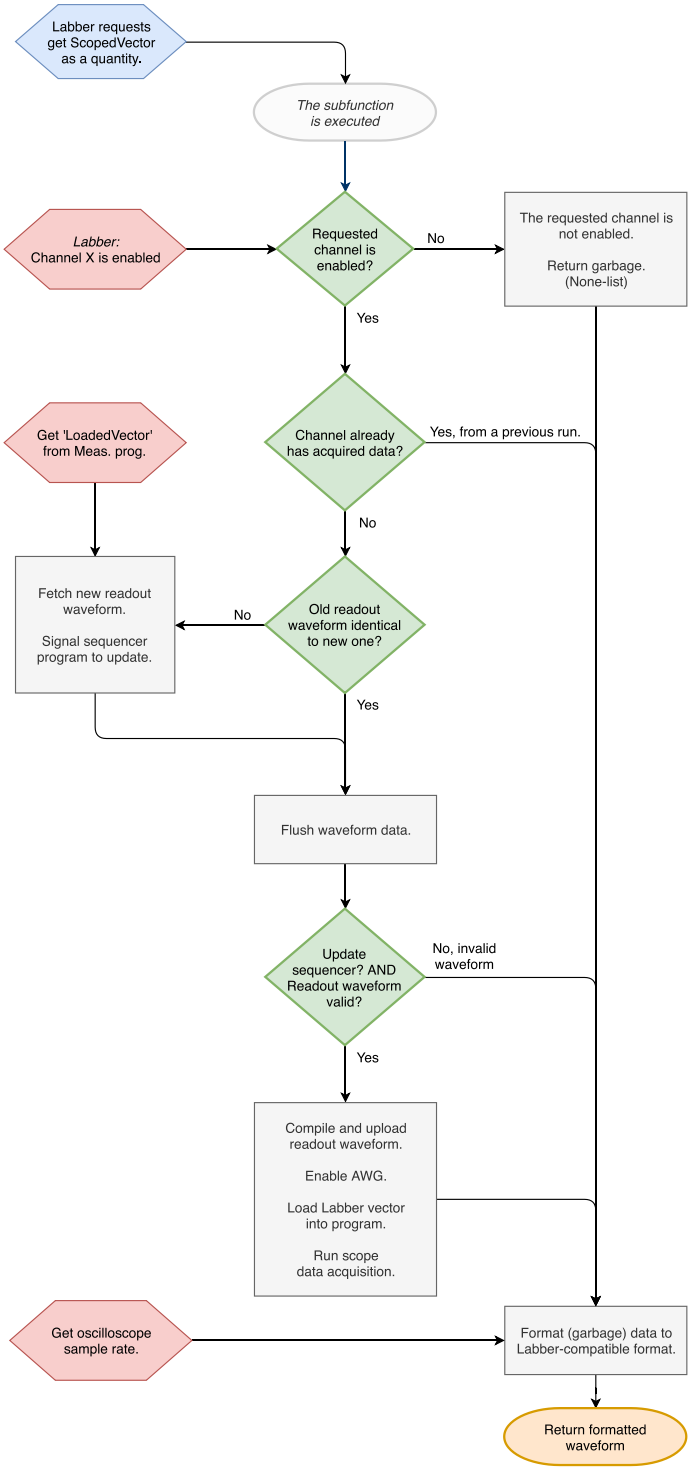


Figure 3.2: Algorithm of the main function for acquiring data when performing a measurement using the UHFQA. Upon Labber requesting a new datapoint from the UHFQA, this function manages and calls the loadLabberVectorIntoProgram, compileAndUploadSourceString and runScopeDataAcquisition subroutines. This function is run on every single iteration in a typical QPU experiment.

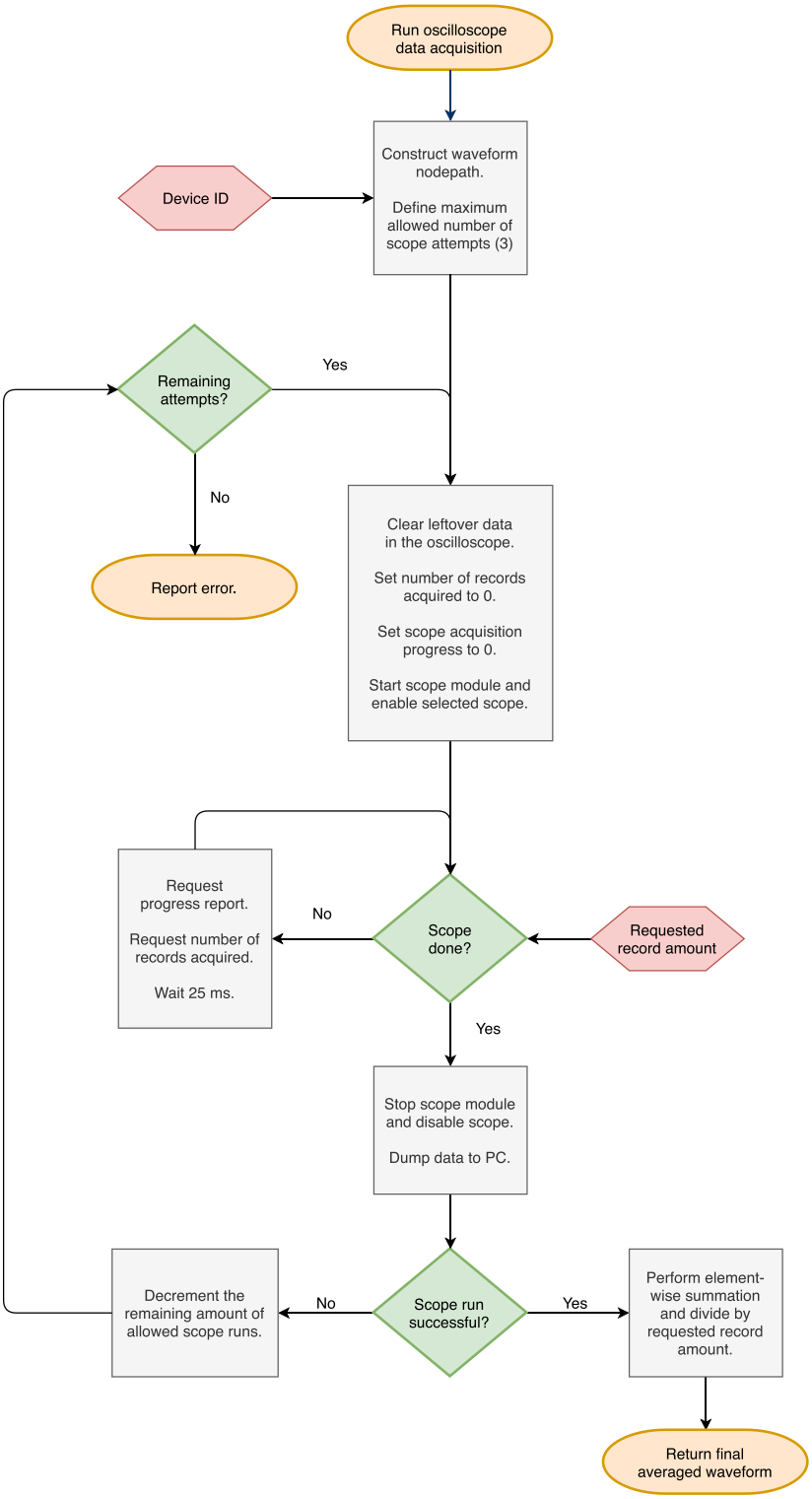


Figure 3.3: Algorithm flowchart of the UHFQA oscilloscope function. Visible in the lower right is also an averaging stage, which performs element-wise averaging if requested by the user. The wave nodepath construction in the beginning is used for sifting through the returned data for a successfully scoped vector.

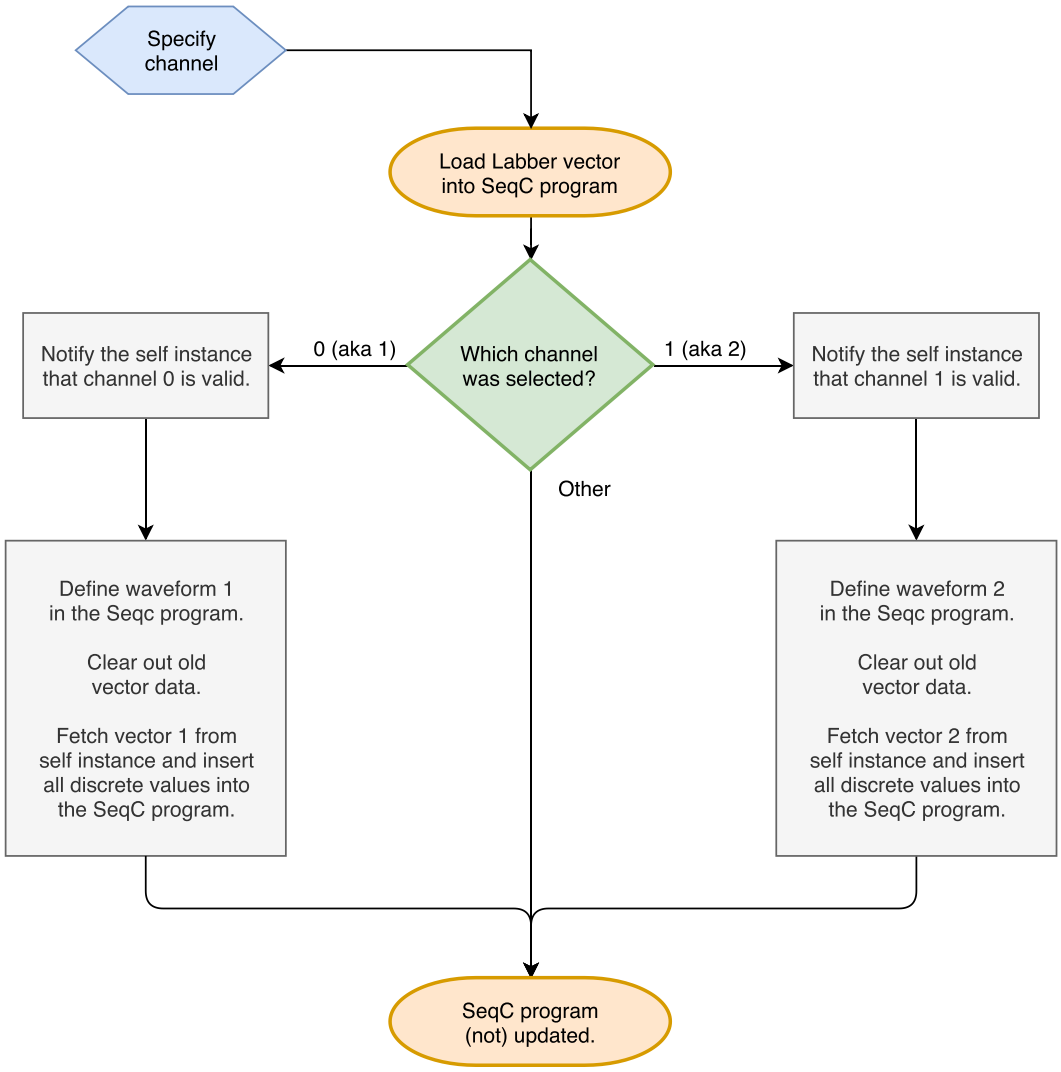


Figure 3.4: Algorithm flowchart of the UHFQA's function for loading Labber vectors into the SeqC program. The definition stages near the end represent source string rewrite functions, implying altering the SeqC code.

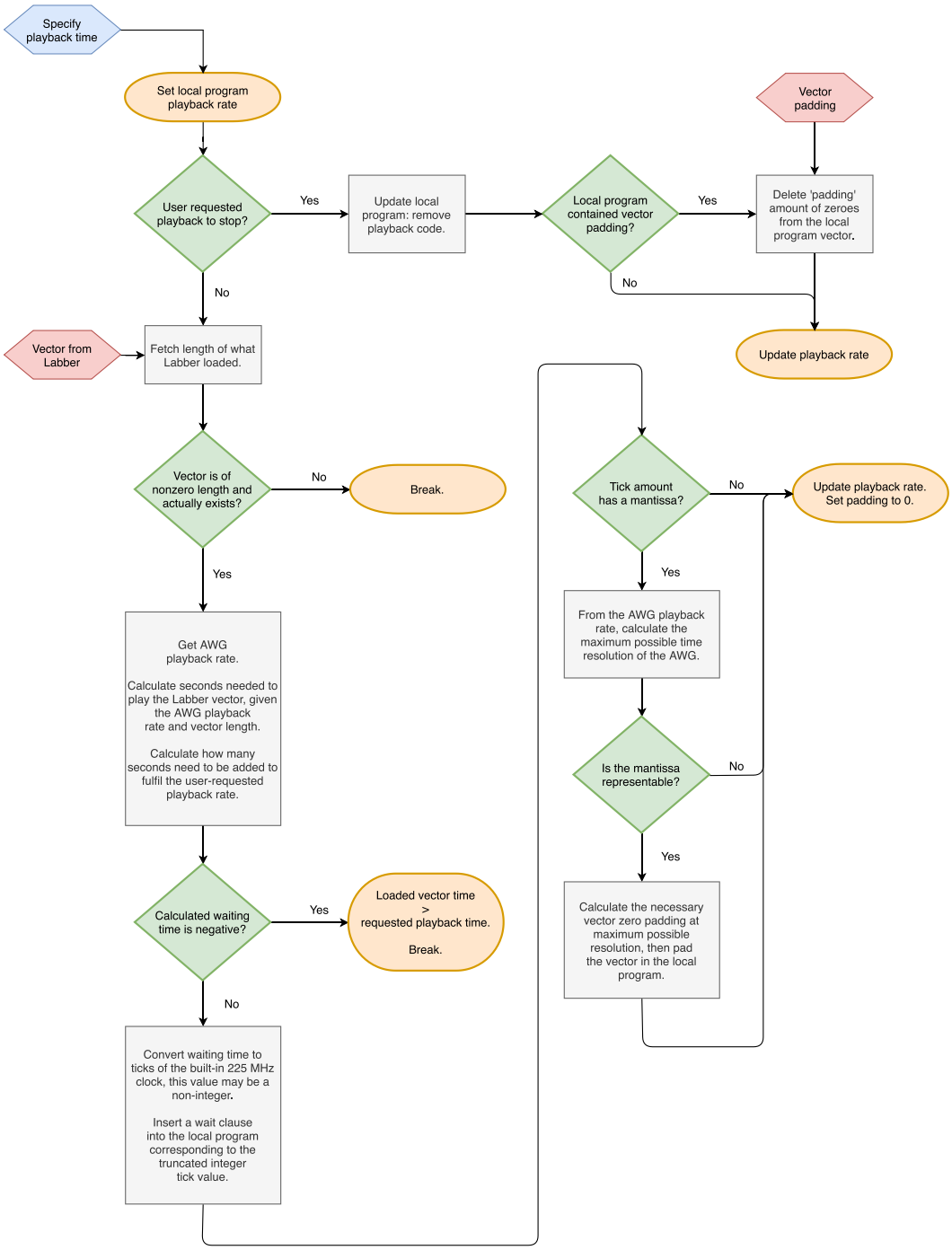


Figure 3.5: Algorithm flowchart of the currently implemented waveform playback function. The latter stages of the algorithm consist of time/sample playback conversions as well as padding until no more resolution accuracy can be gained due to the minimum time needed to play back a sample.

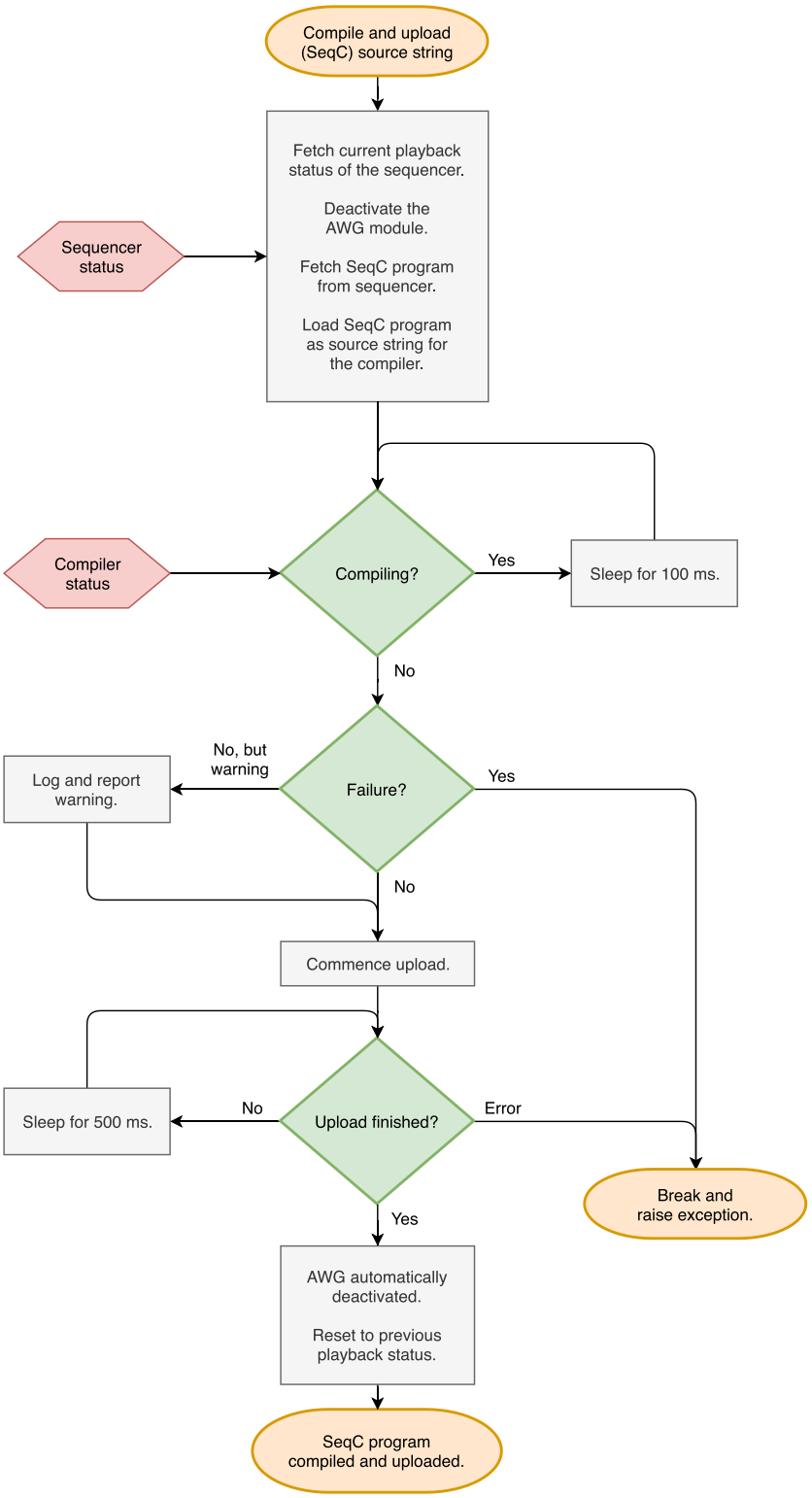


Figure 3.6: Algorithm flowchart of the compilation and upload procedure as used by both the HDAWG and UHFQA instruments.

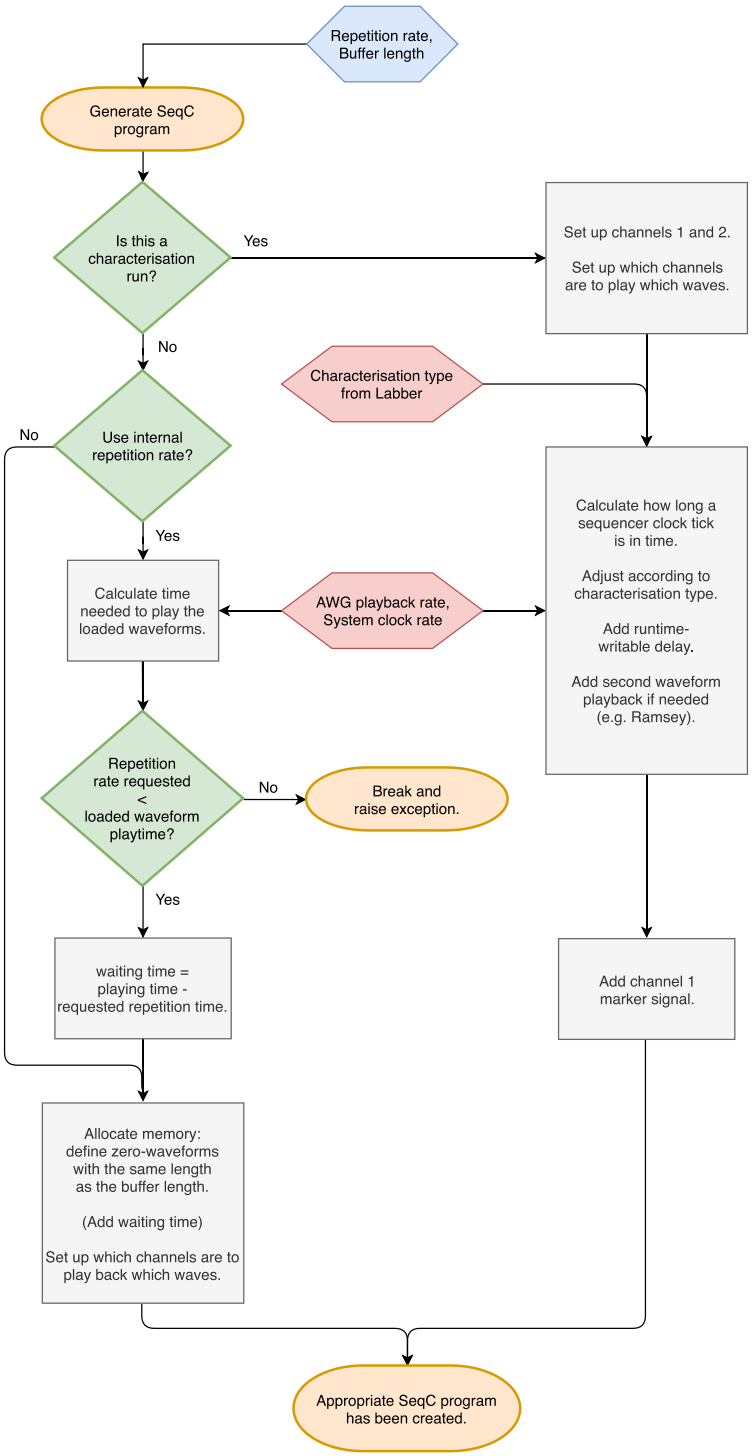


Figure 3.7: Algorithm flowchart of SeqC program generation for the HDAWG. As the HDAWG driver includes some assisting functionality in terms of QPU characterisation, the right path of the algorithm will result in a noticeably different SeqC program than the left one as can be seen in Fig. 3.10 versus Fig. 3.11.

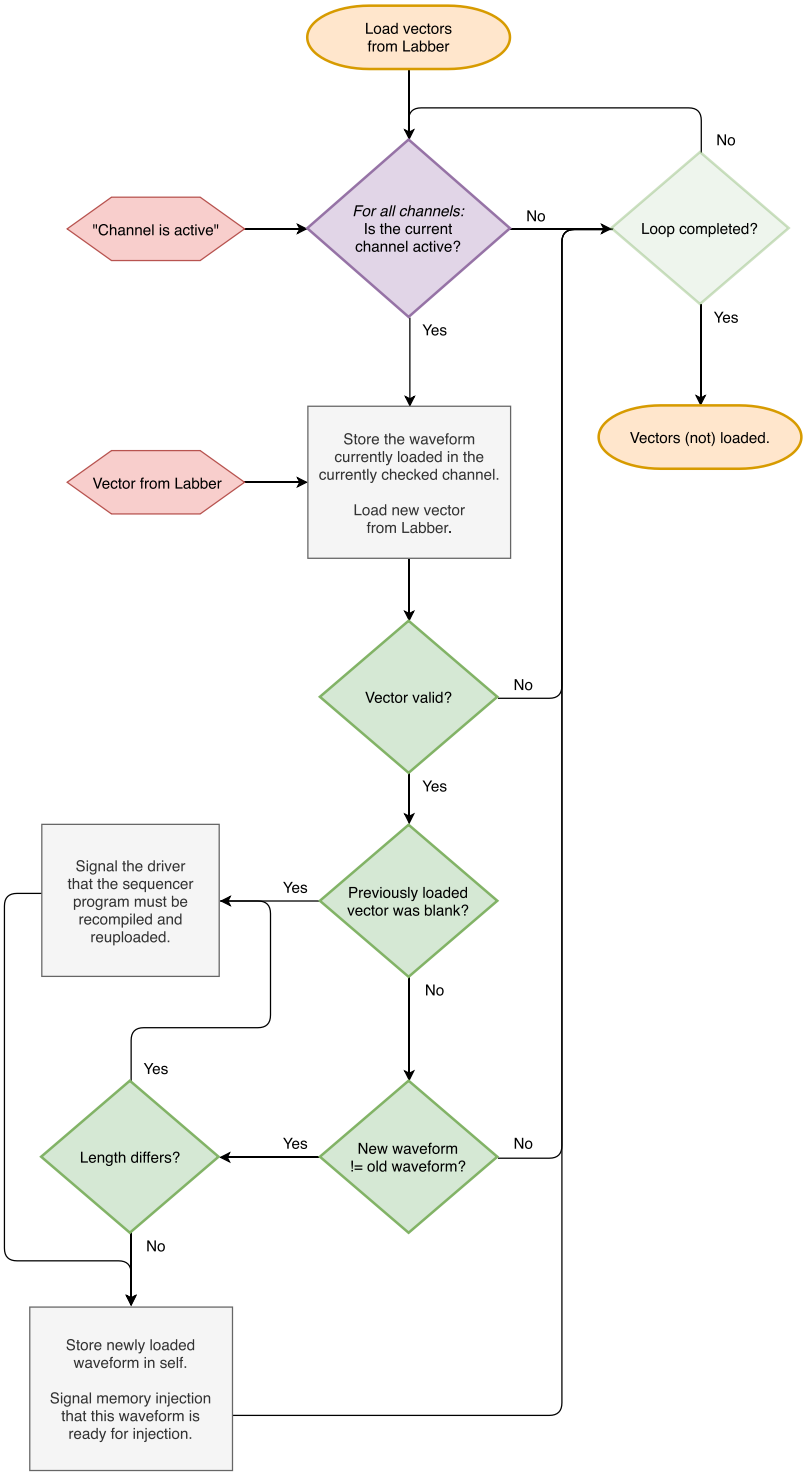


Figure 3.8: Algorithm flowchart of the HDAWG function used to load specified vectors from Labber into the driver. Do note the four inquisitive stages near the end of the algorithm: these make sure that no invalid vectors are loaded, no duplicate waveforms are uploaded, and tries to avoid a sequencer compilation and upload. This is done to minimise waveform upload times, as the SeqC program is only updated when necessary.

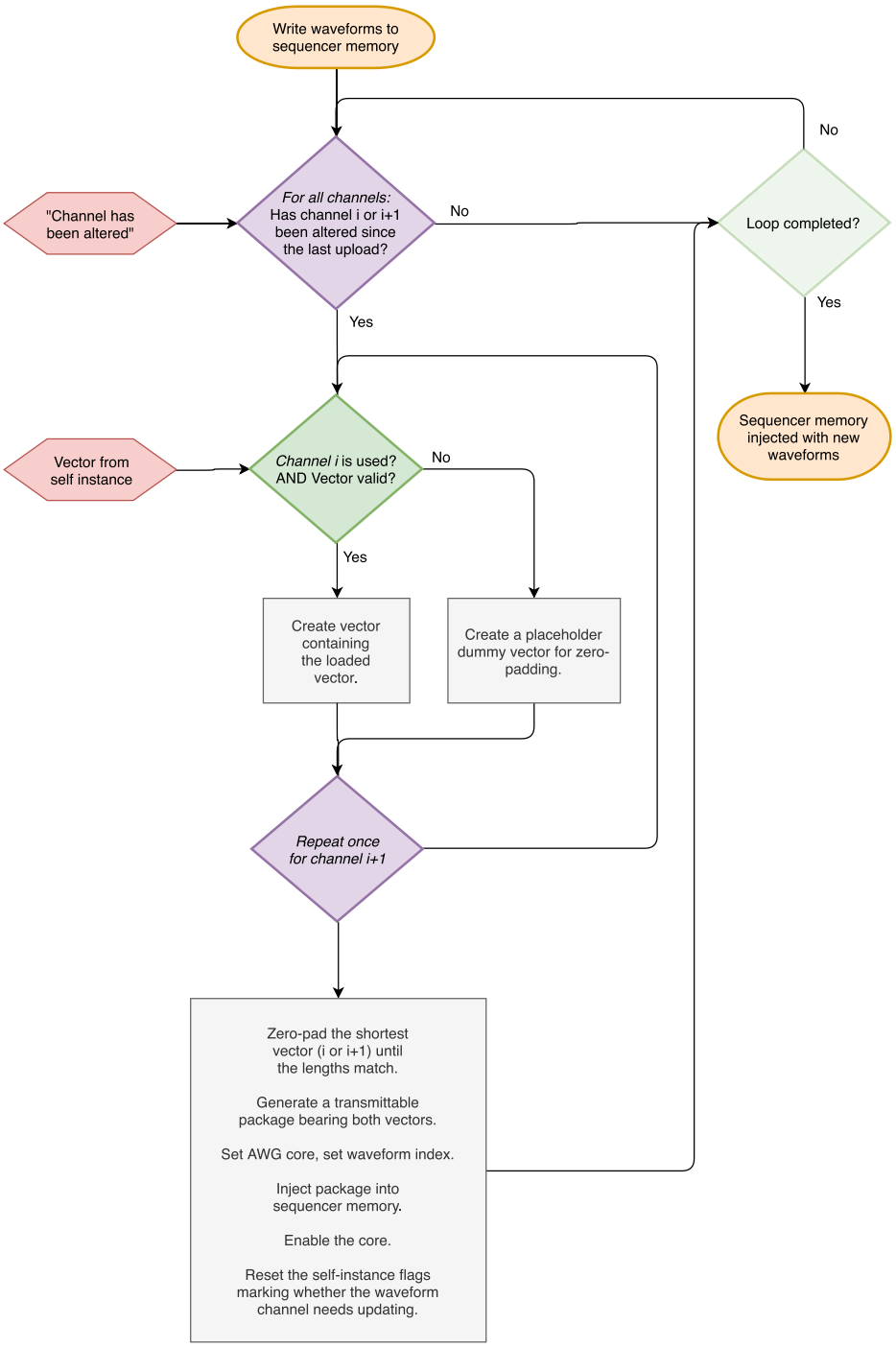


Figure 3.9: Algorithm flowchart of the waveform data memory injection function for the HDAWG. Do note the fetching of vector data from the self instance: the instance has a dict storing all channel waveforms, which is manipulated by the loadVectorsFromLabber function. The fetch thus consists of a dict lookup. Likewise the self instance provides information on what channels have changed since the function was last called, in order not to unnecessarily update identical waveforms.

4

Results

This chapter will present the outcome of the problem solving process as laid out by this thesis. The initial chapters will focus on the quantum processor verification, presenting the results of the QPU characterisation done with the thesis' drivers in order to acquire the marked parameters from Tab. [2.1]. The QPU-related experiment setups are outlined in chronological order in their respective subsections. All QPU experiments have taken place using the setup as shown in section 3.1 apart from the initial VNA sweep in which the cryostat readout line up to the mixer stages was connected to the VNA directly. After the QPU verification, this chapter will continue by presenting Labber-related results such as demonstrations of the implemented functionality and some driver-related characteristics such as driver timing benchmarks.

4.1 Driver verification by QPU characterisation

This section will demonstrate the results of the QPU characterisation, following a broad variety of experiments done in order to acquire typical figures of merit of the QPU core. As was outlined in sections 1.2 to 1.5, the drivers are to be seen as verified should a list of objectives be fulfilled. Among these included final QPU verification, which this section aspires to treat.

The verification procedure is seen as the following: the QPU core was previously characterised by one of my supervisors using the same hardware setup as is given in Fig. 3.1 although with stock solutions and otherwise different drivers operating the instruments. The setup was then loaded with the drivers as procured by this thesis in order to constitute the automated platform. Verification success is measured on basis with how similar my measurements match those done on the same hardware with an arguably experienced operator.

A full list of acquired parameters is given in Tab. 4.1. The remainder of this section will present detailed information regarding these values. Do note that all error bars have been taken using tool-assisted visual estimation based on acquired data. The comparable values have also been extracted by me using the data collected by my supervisor. The experiments are described in a chronological order, meaning that in case no other information regarding a certain parameter is supplied, it has not changed since it last was mentioned in this section.

Table 4.1: Table of acquired QPU characteristics versus target values taken using the same measurements on the same instrument platform, albeit using different instrument drivers. The top and middle values for f_0 and χ correspond to qubits 1 and 2 respectively. The third value corresponds to the resonance frequency of the interconnecting coupler's resonator between the two transmons, the coupler of which is connected to its resonator using two Josephson junctions in a so-called SQUID configuration. Implying that the coupler also expresses similar behaviour as the other transmons. The dilution refrigerator temperature were on the order of 7.5 mK in the 10 mK stage.

Parameter	Acquired values	Comparable values
f_0	6.1717500 GHz ± 125 kHz	6.1717500 GHz ± 85 kHz
	6.0372855 GHz ± 68 kHz	6.0372875 GHz ± 88 kHz
	6.716 GHz ± 1 250 kHz	6.716 GHz ± 235 kHz
χ	1.159 MHz ± 0.235 MHz	1.19 MHz ± 0.127 MHz
	1.830 MHz ± 0.171 MHz	1.871 MHz ± 0.096 MHz
	8.00 MHz ± 1.24 MHz	7.60 MHz ± 1.59 MHz
$f_{q,\ QB2}$	4.302665 GHz ± 3.916 MHz	4.300000 GHz ± 4.146 MHz
$\Omega_{\ QB2}$	721 mV ± 20 mV	740 mV ± 15 mV
$T_{1,\ QB2}$	~57.6 µs	~62 µs

4.1.1 Resonator spectral detection and qubit spectroscopy

The theory underlying this subsection is outlined in subsections 2.3.2 and 2.3.3. This subsection establishes preliminary values for the resonators' spectral locations, preliminary values for the qubits' dispersive shifts and the qubit resonance frequencies. Tab. 4.2 shows characterised values.

Table 4.2: Table of acquired QPU characteristics in this subsection. The method is outlined in the section itself, additional information in Tab. 4.1. Do particularly note $f_{q,\ QB2}$: this value requires both successful operation of the UHFQA and HDAWG instruments to be acquired, effectively proving that the platform as created by this thesis can be used to acquire realistic values given the verification method is trusted.

Parameter	Acquired values	Comparable values
f_0	6.1717500 GHz ± 125 kHz	6.1717500 GHz ± 85 kHz
	6.0372855 GHz ± 68 kHz	6.0372875 GHz ± 88 kHz
	6.716 GHz ± 1 250 kHz	6.716 GHz ± 235 kHz
χ	1.159 MHz ± 0.235 MHz	1.19 MHz ± 0.127 MHz
	1.830 MHz ± 0.171 MHz	1.871 MHz ± 0.096 MHz
	8.00 MHz ± 1.24 MHz	7.60 MHz ± 1.59 MHz
$f_{q,\ QB2}$	4.302665 GHz ± 3.916 MHz	4.300000 GHz ± 4.146 MHz

For acquiring initial estimations for the spectral locations DUT resonators, one could in theory use the UHFQA to sweep (S_{21}) an extreme bandwidth to observe all resonator dips that follow from such an experiment. In the interest of life quality, one usually considers the designer's word regarding where the DUT's resonators are located within the spectra; to gain a somewhat sufficient readout of said locations, one may use a VNA to characterise the first parameter of the QPU core, namely the spectral locations of the resonators. The DUT loaded in the final verification setup contains three resonators, coupled to two qubits and a separate coupler between said qubits. Fig. 4.2 demonstrates a VNA sweep of the DUT and the spectral locations of the DUT resonators.

The sweep is done in power as well as in frequency. Keep in mind that the dispersive shift is inversely power-dependent as mentioned in subsection 2.3.2. This implies that at 'high power,' the χ-factor will be low yielding the so-called bare resonance frequency of the resonators. The dispersive shift is however present in the lower power ranges. As we see in Fig. 4.1 and 4.2, the χ-factor is visible as the S_{21} dip moves in frequency (blue). It should be mentioned that the power sweep could have been initialised at an even lower value to guarantee that the dispersive shift had yet to take effect. Same is inversely valid for the upper sweep bound.

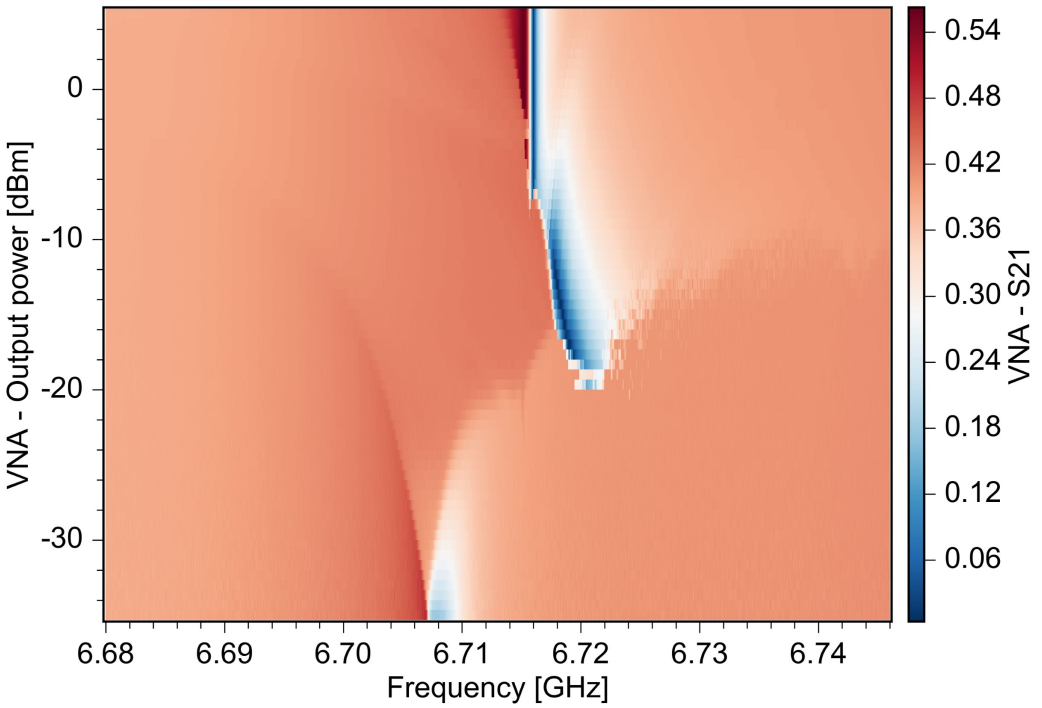


Figure 4.1: VNA frequency versus output power sweeps of the coupler resonator located between the two qubits. The IF bandwidth was 1 kHz, 100 averages per sweep, swept from −35 dBm to 5 dBm. As opposed to Fig. 4.2, the two Josephson junctions (SQUID) mounted to the resonator from the coupler line demonstrate a leftwards dispersive shift. All acquired values are presented in Tab. 4.2. A suggested improvement to the sweep would be to extend the output power limits, as this usually better guarantees that the qubits are / are not affecting the measurement. This is further discussed in Fig. 4.4.

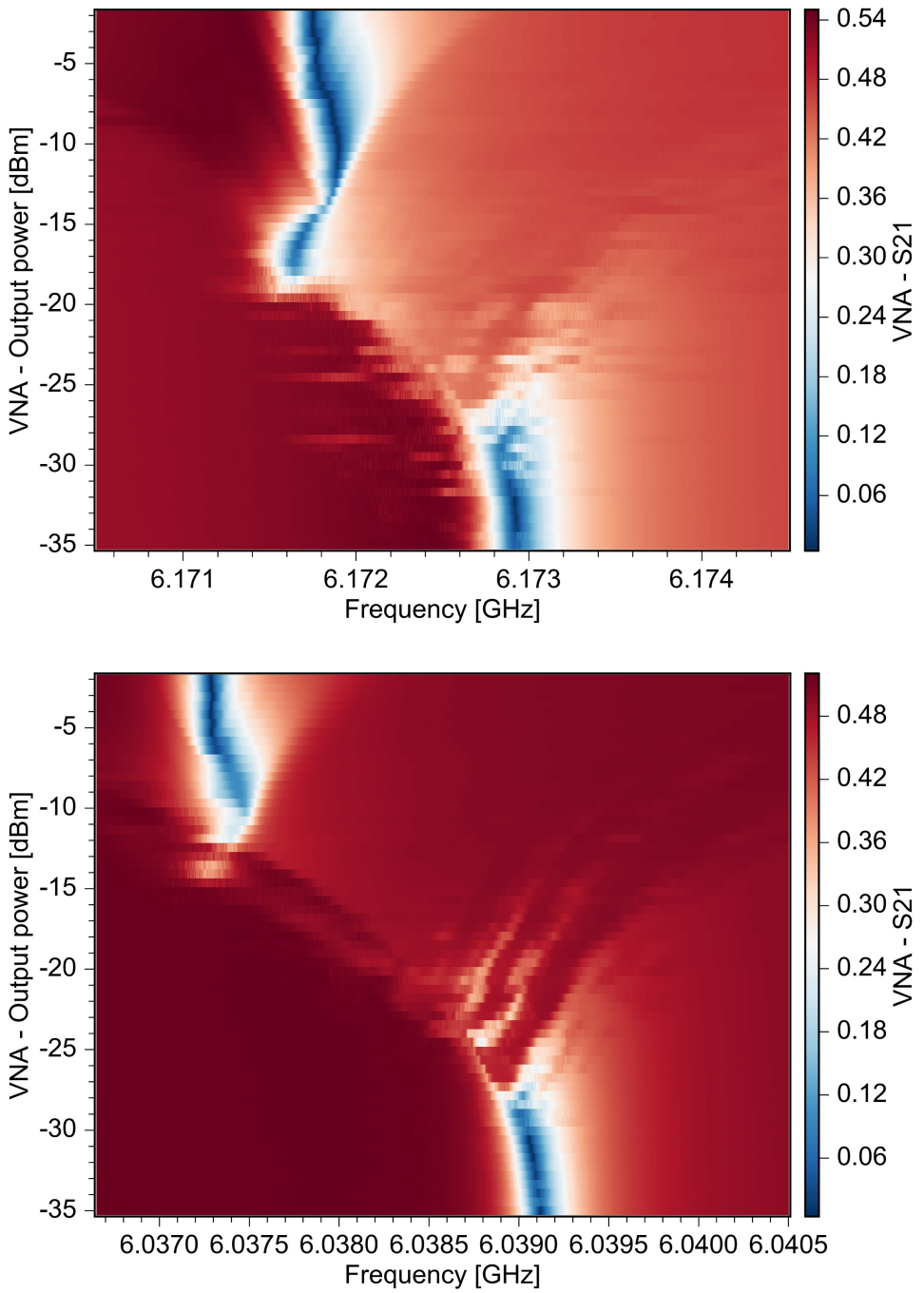


Figure 4.2: VNA frequency versus output power sweeps of qubits 1 (top) and 2 (bottom). The IF bandwidth was 1 kHz, 100 averages per sweep, swept from −35 dBm to −2 dBm. The blue portions of the images demonstrate where relatively low power is transferred through the system from the VNA, implying that the readout line is stricken into resonance. These regions are used to locate preliminary values for the resonator frequencies. We also see that the dispersive shift χ is rightwards for both DUT qubits. All acquired values are presented in Tab. 4.2. As in the case with Fig. 4.1, these sweeps show clearer potential for improvement by extending the output power limit. This is seen in the blue lines connecting to the borders of the figure: more accurate values would be expected should they be allowed to straighten out. As of now, some qubit-dependency can be suspected from the image (some curvature is present).

To verify the UHFQA driver, we may now use the UHFQA to locate the spectral locations of the resonators while under the influence of the transmons. Due to the limit in output power and the differences in spectroscopy using continuous wave measurements vs. the pulsed nature of the UHFQA, these measurements are unsuitable to repeat at high output power. The UHFQA is dependent on external mixers in order to generate frequency content at the correct spectrum for the DUT. A readout pulse sweep was thus constructed using

$$f_{\text{readout}} - f_{\text{LO}} + f_{\text{offset}}, \tag{4.1}$$

where: f_{readout} was 6.17302 GHz and 6.03914 GHz, for qubits 1 and 2; f_{LO} was 6.393 GHz, for both qubits; and $f_{\text{offset}} = \pm$ 2 MHz, in 101 steps. This sweep was fed into Labber's multi-qubit pulse generator, which constructed a 100 mV cosine pulse to be loaded into the UHFQA. The reason for (4.1) stems from a decision made to sweep the LO instead of sweeping the readout pulse, since this would create new waveform values to be loaded into the UHFQA for every measurement. This in turn would imply lengthy measurement times. Thus, to keep the waveform value to be loaded into the UHFQA constant, the LO was configured to sweep from

$$6.393\,\text{GHz} + f_{\text{offset}}. \tag{4.2}$$

The UHFQA driver in turn correctly detected duplicate upload attempts of a non-changed waveform, which optimised the measurement time noticeably. The UHFQA was set to 512 samples averaging. Fig. 4.4 demonstrates the resulting outcome of the experiment. Fig. 4.3 demonstrates an identical experiment for the coupler's resonator, apart from the sweep interval (-3 to -10 MHz in 501 steps), LO frequency (6.932 GHz) and sample averaging (32 vs. 512).

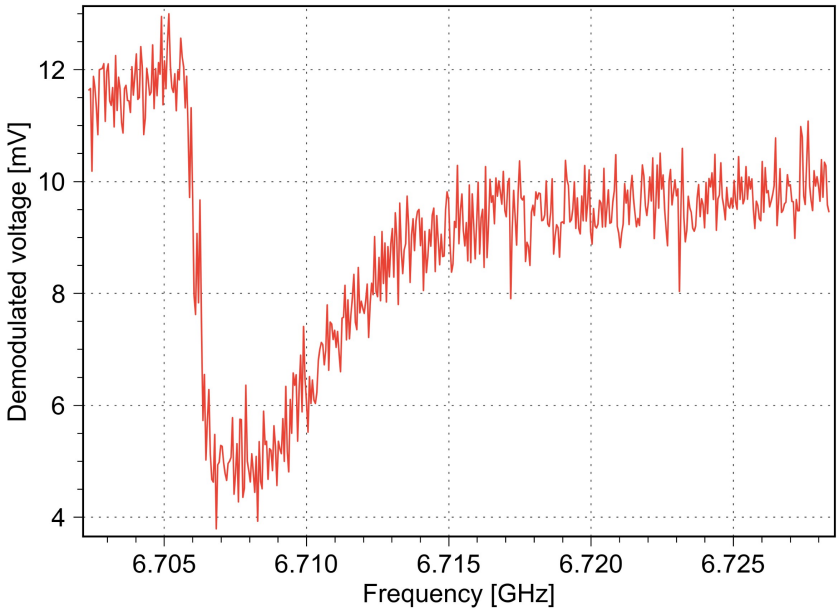


Figure 4.3: UHFQA readout of the coupler resonator. The symmetry of the resonator is very low, of which I have been assured was replicable. The main purpose of this graph is thus to verify that the coupler's resonator is working in the DUT. The noise level could have been mitigated by increasing the sample averaging, likewise some curve could have been fitted to the plot although this would not serve a useful purpose. Do note that the resonator dip in this image does not match the value in Tab. 4.2. Probable explanations could be the nature of the VNA's CW sweep versus the UHFQA's pulse sweep, but it is more likely that the UHFQA is outputting a signal power in which the qubits are affecting the result. This suspicion is emphasised following Fig. 4.4, in which this influence is clearly visible.

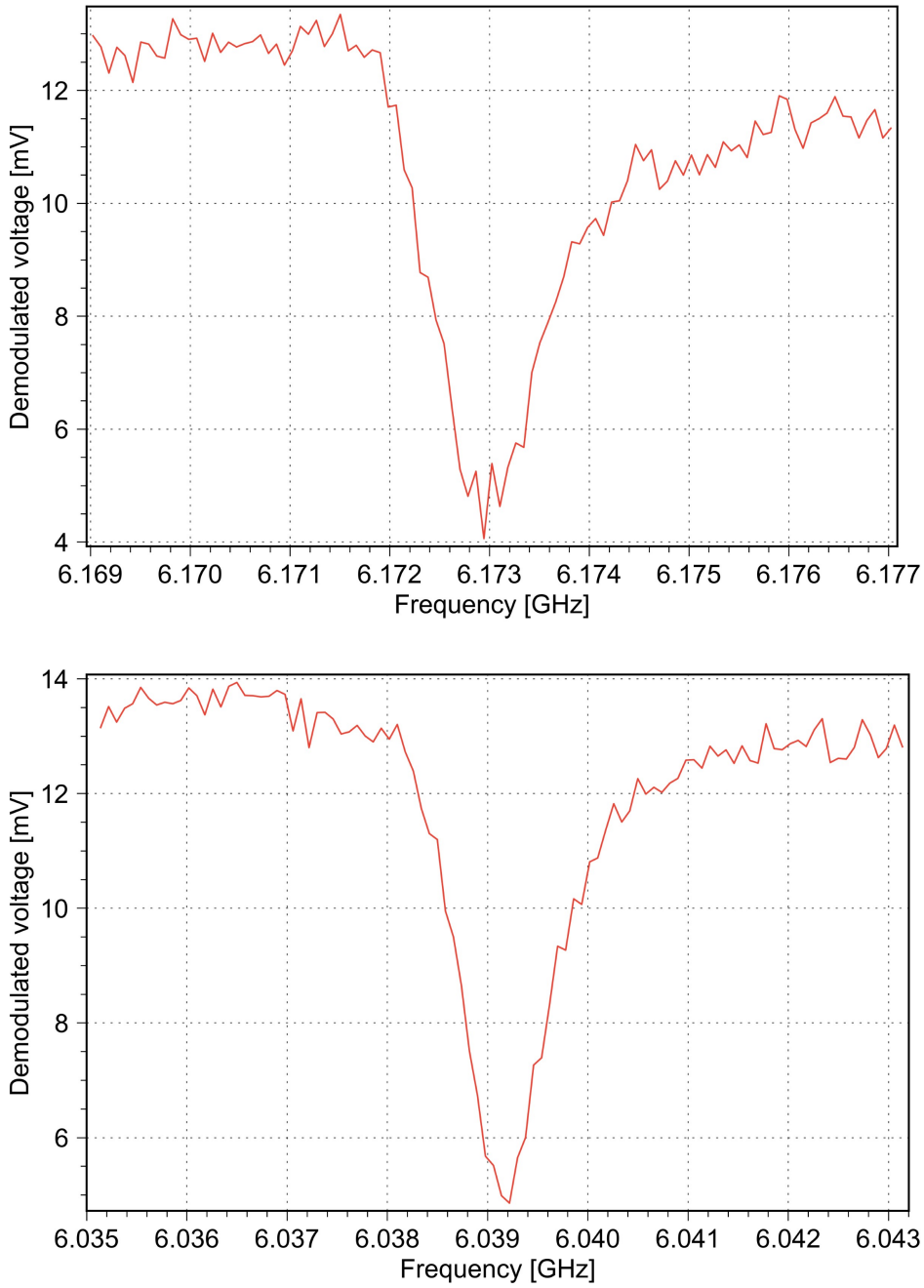


Figure 4.4: UHFQA-sweep demonstrating successful load-and-scope functionality of the UHFQA. Likewise, the existence of two resonators has been confirmed. The UHFQA was configured to use a 0% scope reference window, 0 µs hysteresis - in order to initiate the scoping at the trigger. Trigger holdoff was set to the lowest possible (20 µs). The integration time was set to 5 µs, which is longer than the 4 µs pulse length. Do note the differences in resonator frequency values in the notch vs. Tab. 4.2. A probable source of error could be the relatively 'high' output power from the UHFQA in this figure. It is visible that the shape of the notch is not as symmetrical as shown in Fig. 2.17, which indicates that the qubits are affecting the results by shifting the resonance frequency. Another probable source of asymmetry could be due to impedance mismatch within the QPU core with regards to the transmission line [52].

As can be seen in Fig. 4.5, the UHFQA can demonstrate a notch peak attenuation from −37.49 dB to −47.83 dB, as gauged in the top graph of Fig. 4.4. The goal achieved (>10 dB) made quantified by asserting a number to it does however not prove the point: using the UHFQA, the user can register notch peaks which clearly match the overall theoretical expectations (Fig. 2.17) which in turn imply that the UHFQA's scoping capabilities appear to be functional. A suggested improvement to the verification methodology could instead be to compare the acquired value to a more trusted source, such as a commercial oscilloscope with averaging capabilities, and then inspect suitable differences.

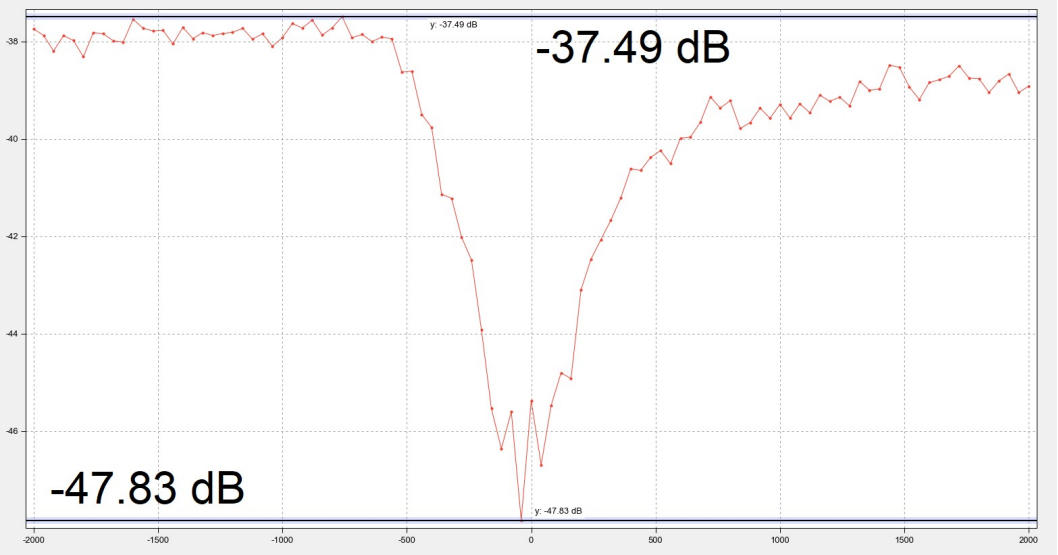


Figure 4.5: Snapshot from Labber's logging browser using Labber's marker gauge tools, with values added for clarity.

Following locating the spectral positions of all resonators, we may continue by running both the UHFQA and HDAWG together in order to find the qubit resonance frequencies. The verification will from now on be done on a single qubit, more specifically qubit 2. The next experiment is commonly referred to as qubit spectroscopy, described by theory in subsection 2.3.3. Fig. 4.6 demonstrates the qubit driven by the HDAWG output, and read out using the UHFQA.

The readout was performed as done previously, using relative parameters with a swept frequency offset. The sweep was done from 4.2 GHz to 4.6 GHz in 301 steps. $f_{readout}$ was fixed at 6.03914 GHz, and the UHFQA averaging was set to 2048 samples per data point. The HDAWG was added into Labber's instrument server along with another mixer, calibrated and pre-configured during the comparable characterisation run. Since signal power affects the measurement (the f_q spike widens for instance), a somewhat careful value of -15 dBm was chosen. The HDAWG's internal clock was set to match that of the UHFQA at 1.8 GHz. Likewise the UHFQA was set in the measurement setup to trigger its AWG using a trigger input from the HDAWG, since the readout pulse should execute as soon as possible after the control pulse has been completed. As shown in the SeqC implementation, the AWG in turn triggers the scope function via an internal signal call.

As is seen in Tab. 4.2 versus Fig. 4.6, the value $f_{q,\,QB2}$ extracted from this experiment match the expected value well, which in turn verifies the control and readout functionality of the platform.

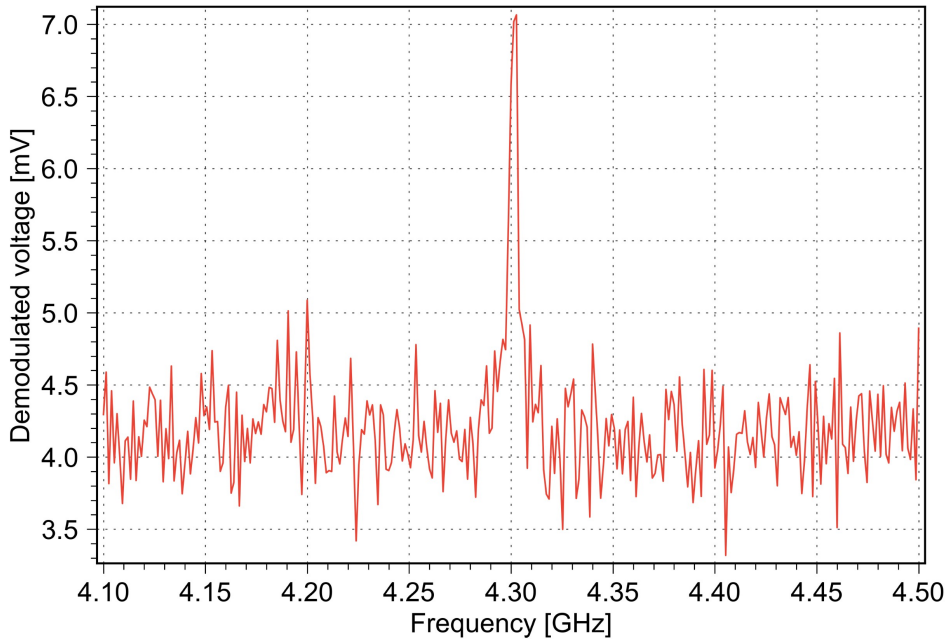


Figure 4.6: Qubit spectroscopy of qubit 2 in the multi-qubit QPU. Using the HDAWG and UHFQA, qubit 2 is located to 4.302655 GHz. Do note the second-highest peaks at about 4.20 GHz, these are heavily suspected to be remnants of LO leakage, since these provided similar signal strength when the spectrum was scoped while the qubit was left non-driven.

4.1.2 Rabi oscillations and energy relaxation times

The theory underlying this subsection is outlined in subsection 2.3.4. This subsection establishes preliminary values for the π-pulse amplitude Ω, as well as the energy relaxation time T_1. Tab. 4.3 shows the values characterised in this subsection.

Table 4.3: Table of acquired QPU characteristics in this subsection. The method is outlined in the section itself, additional information in Tab. 4.1.

Parameter	**Acquired values**	**Comparable values**
Ω_{QB2}	721 mV $\pm$ 20 mV	740 mV $\pm$ 15 mV
$T_{1,\text{ QB2}}$	~57.6 µs	~62 µs

Qubit spectroscopy allowed us to acquire the frequency used for controlling qubit 2. We also know at what resonator we expect to see a response from it following stimulus. The multi-qubit generator was reconfigured via Labber to output a mere waveform primitive in the form of another cosine pulse. Its frequency was changed from -100 MHz to -225 MHz following analysis of a planned Ramsey oscillation experiment, this will be delved into at the end of this subsection. The LO was accordingly set to 6.395 GHz, and the generated readout waveform in the multi-qubit generator followed accordingly. This pulse was as before fed into the HDAWG. A waiting time was set up for the HDAWG of 10000 clock cycles in order to ensure that the qubit has sufficient time to relax into the ground state after excitation. The amplitude of the HDAWG was set to sweep from 100 mV to 5 V in 101 datapoints, while the UHFQA was set to 512 samples averaging per acquired datapoint.

Adjustments to the vector signal generators had to be made: the output power was set very low at −60 dBm, at a new frequency of 4.525 GHz as the control waveform was changed. The expected outcome of the experiment was a coarse, converging sine wave which was also the case as is visible in Fig. 4.8. A finer sweep was promptly set up from 0 mV to 2 mV in 201 points, 4096 datapoints averaging, −40 dBm output power. The results matches the theoretically expected output as shown in subsection 2.3.4. Using Fig. 4.8, we may now estimate a preliminary value for Ω, which is seen at 721 mV. A suggested improvement for further applications would have been to apply a fitted converging sinusoidal to the analysed data, which lessens on having to estimate a fitted level inside a noisy output. Another improvement in methodology would be to set the UHFQA to signal the HDAWG when a measurement was completed. This improvement is discussed further in section 5.2.

Since we now know the preliminary amplitude Ω for setting the qubit to $|1\rangle$, we may now study its energy dephasing time T_1. The driver was set to run a T_1 experiment which is identical to the Rabi experiment apart from bearing a fixed amplitude and sweeping the delay at which a readout pulse is ordered. The SeqC program allows for a user-settable delay without any code re-uploading, in which user register 0 was used in a wait statement (see Fig. 3.11). User register 0, seen as τ in (4.3), was swept from 0 to 90,000 which corresponds to delaying the readout pulse with

$$\tau \cdot \frac{8}{1.8 \cdot 10^9} = \text{delay} \in [0, 400\,\mu\text{s}], \tag{4.3}$$

albeit neglecting the time required for calling getUserReg from the sequencer, and the time required for triggering the UHFQA from the HDAWG. Investigating these values and including them to the calculation could change this value with a nontrivial amount; improving on this methodology would likely infer more accurate results in this experiment. The outcome of the T_1 sweep is visible in Fig. 4.7, matching the expected curve from theory as is shown in Fig. 2.22.

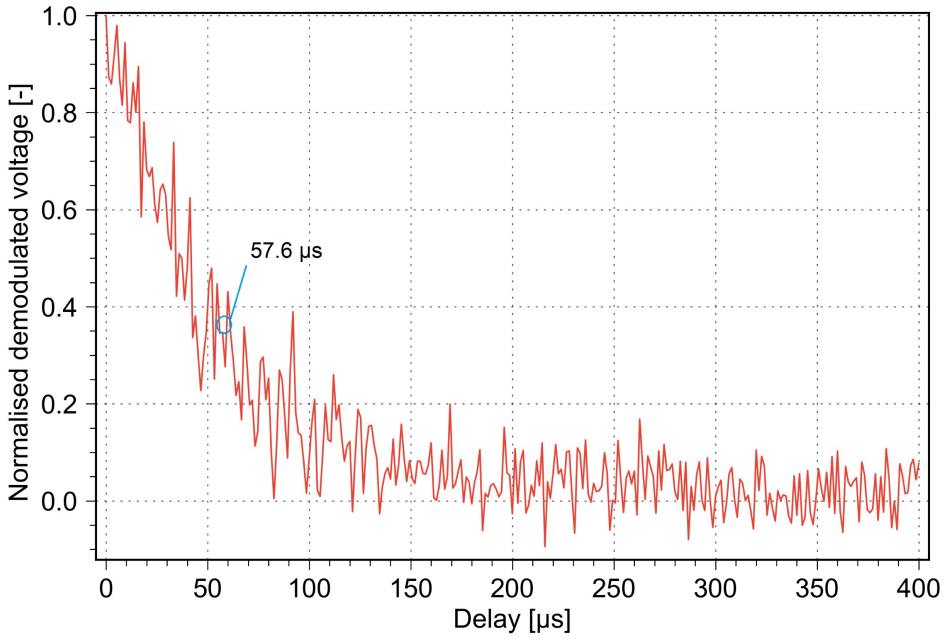


Figure 4.7: Energy dephasing time experiment at 721 mV input amplitude. The curve is presented normalised to the maximum amplitude as well as the estimated convergence level, yielding at the y = 0.368 point $T_1 = 57.6\,\mu$s. The experiment method is outlined in subsection 4.1.2. Note the choice of y-axis normalisation, in particular the convergence line to the bottom right. As just mentioned, T_1 is likely slightly lower since some delay should be added. As is well-known in this field, T_1 may vary substantially over time - even for shorter timespans. No filtering was applied.

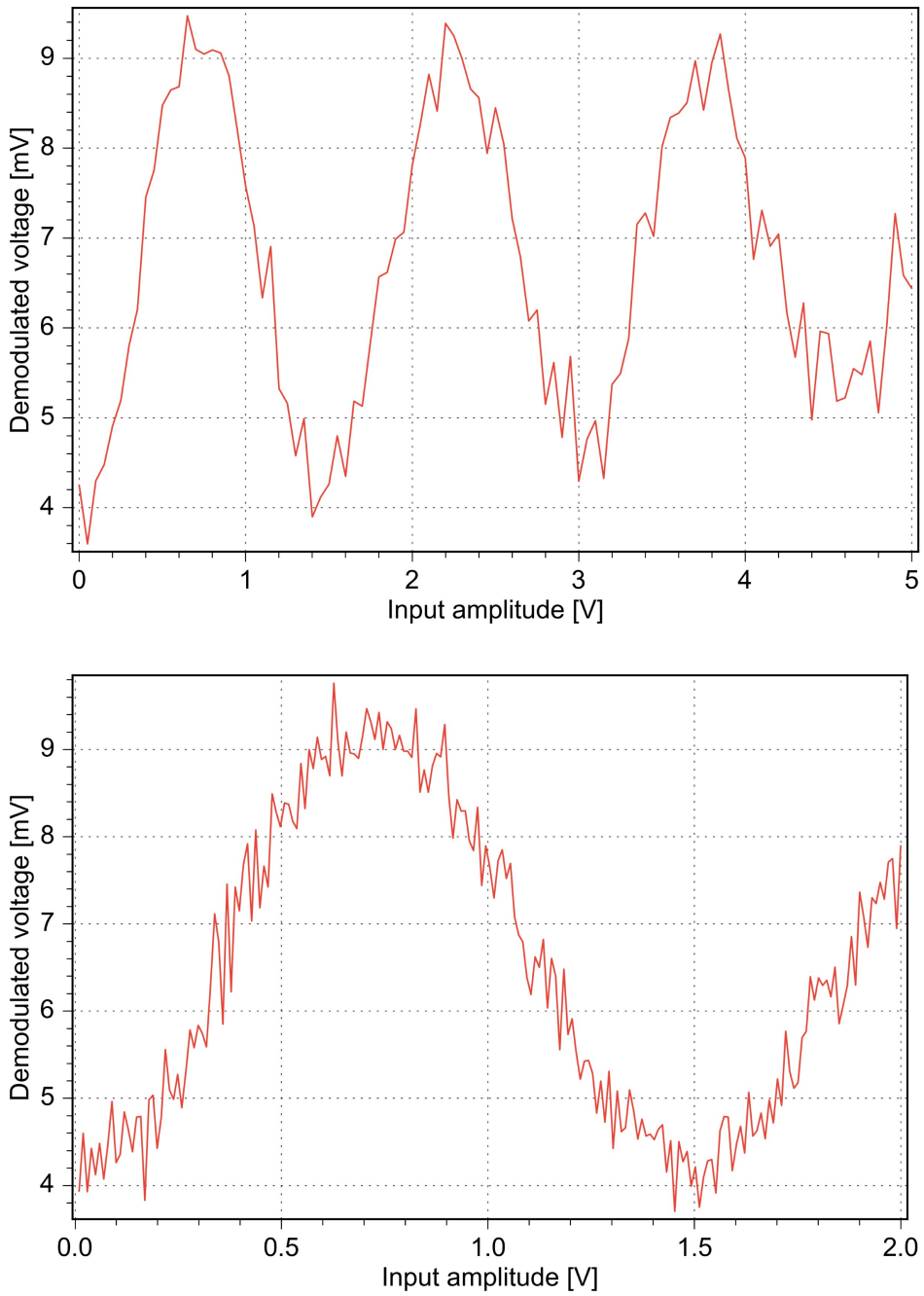


Figure 4.8: Illustrations of two rounds of Rabi oscillation experiments on qubit 2, one done for rough estimation (top) and the other one for finer data acquisition (bottom). The control pulse input amplitude, shown on the x-axis, is comparable to the current Z-axis position of the Bloch vector, implying pointing straight at $|1\rangle$ at 721 mV as seen in the lower figure. Ω was acquired at the sine wave peak, with slight noise filtering applied post-measurement.

The next calibration step would normally be to acquire T_2 using so-called Ramsey spectroscopy. Some issues were had with the drivers in this regard: for a Ramsey spectroscopy experiment, two $\frac{\pi}{2}$-pulses are played back with a variable delay from 0 to τ between them. A readout is then performed immediately after the second control pulse finishing. However, the loaded waveform

primitives typically consist of a pulse envelope bearing a subtone of some modulation frequency. This frequency is to remain constant, while the envelope is supposed to be swept as described. Using the HDAWG SeqC code as shown in Fig. 3.11, this was not the experimental outcome, and thus no T_2 value was acquired. Since the main sample clock of the sequencer in the HDAWG ran at 225 MHz, it was thought that for this very specific frequency the envelope would correspond to the correct one. Despite matching the generated waveform to the sequencer clock, no usable data was acquired. Suggestions for future implementations would be to utilise the built-in oscillators of the HDAWG to apply the required subtone, and use built-in functionality for modulating a Ramsey spectroscopy experiment. Fetched waveform primitives may then in fact be identical to the one used in the Rabi oscillation experiment, albeit halving the input amplitude Ω acquired from the Rabi experiment since Ramsey spectroscopy is done using $\frac{\Omega}{2}$-pulses.

We may thus conclude the QPU characterisation experiments as done on the QPU in order to verify proper functionality of the HDAWG and UHFQA drivers. The next section will treat the operative properties of said platform, mainly from a perspective of the Labber framework itself.

4.2 Instrument control and automation using Labber

This section intends to demonstrate the functionality of specific programmed functions in the HDAWG and UHFQA drivers. This demonstration will initialise with a simple demonstration of what is being loaded versus what is played back by the drivers, continued by showing Labber's generated GUI along with the playback resolution accuracy, and finally demonstrating some typical timestamp measurements done within Labber's log functionality.

4.2.1 Waveform definition, playback, and scoping

As can be seen in Fig. 4.9, the drivers for the UHFQA can load and play back arbitrary waveforms. The figure demonstrates an experiment where two complex waveforms were loaded one after each other, one consisting of a 100 mV amplitude gaussian pulse and a 500 mV amplitude gaussian pulse respectively. Subfigures **(a)** and **(b)** in Fig. 4.9 demonstrates the target loaded I and Q waveforms, while **(c)** and **(d)** demonstrates the recorded output from the UHFQA's AWG. The scope data was captured using the same UHFQA driver. As can be seen in the image, the peaks of the I waveforms were both defined to be located at 325 ns whereas the scoped input (synchronised to the waveform playback using a digital marker) shows that the waveform peaks are located at 461.1 ns and 455.5 ns for the 100 mV and 500 mV I-curves respectively. This result implies a delay from playback to scope in the order of 135 µs within the UHFQA. UHFQA-generated and scoped waveforms undershoot when returning to 0 V: the 100 mV waveform undershoots with 11.3 mV while the 500 mV waveform undershoots with 50.1 mV. As briefly noted in Fig. 4.9, the reason of the observed undershoot is currently not understood, but speculated to stem from general non-ideal variability within the setup.

Visible in Fig. 4.10, the drivers for the HDAWG can load and play back arbitrary waveforms. An operator error was made as the scope was set to 50 Ω reference while the HDAWG was not coupled this way, resulting in a distorted scoped amplitude.

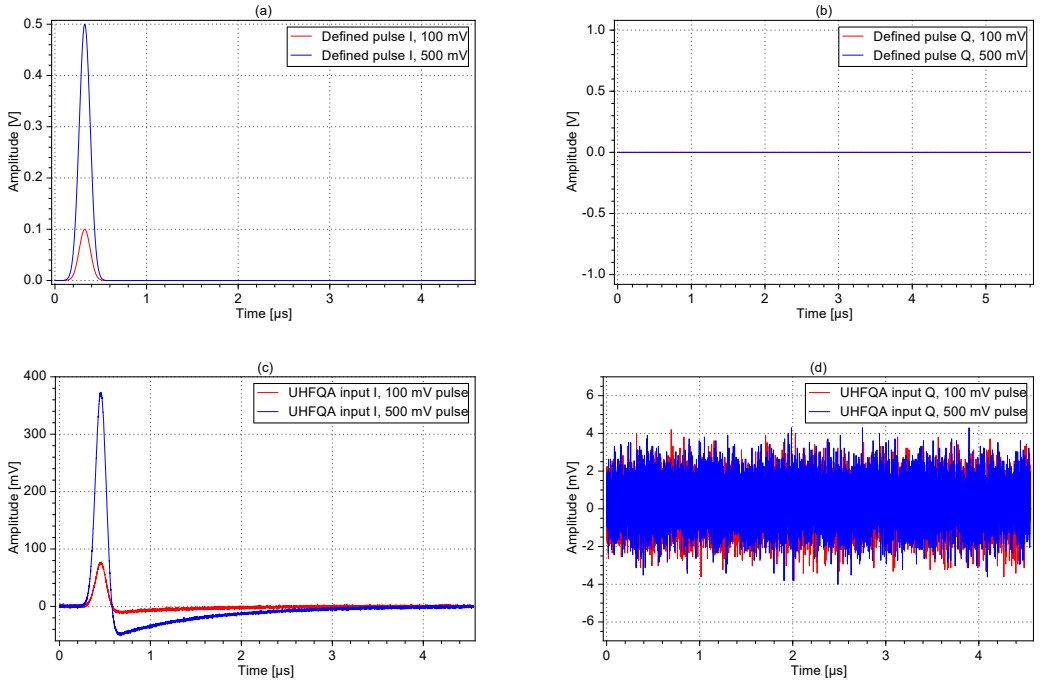


Figure 4.9: Graphical demonstrations of defined **(a, b)** versus actual waveforms as played back **(c, d)** by the UHFQA driver. The defined pulse constitutes an (arbitrary, fictional) π-pulse lacking a Q component. The played back Q component as would then be expected consists of noise. Do note the differences in peak amplitude of the 100 and 500 mV I waves, this behaviour is not understood at this time. Also note the undershoot of the measured 500 mV I wave, another currently non-understood phenomenon albeit speculated to stem from general non-ideal (dynamic) variability in the setup. Overall shape and amplitude of the specified and measured waveforms correspond adequately.

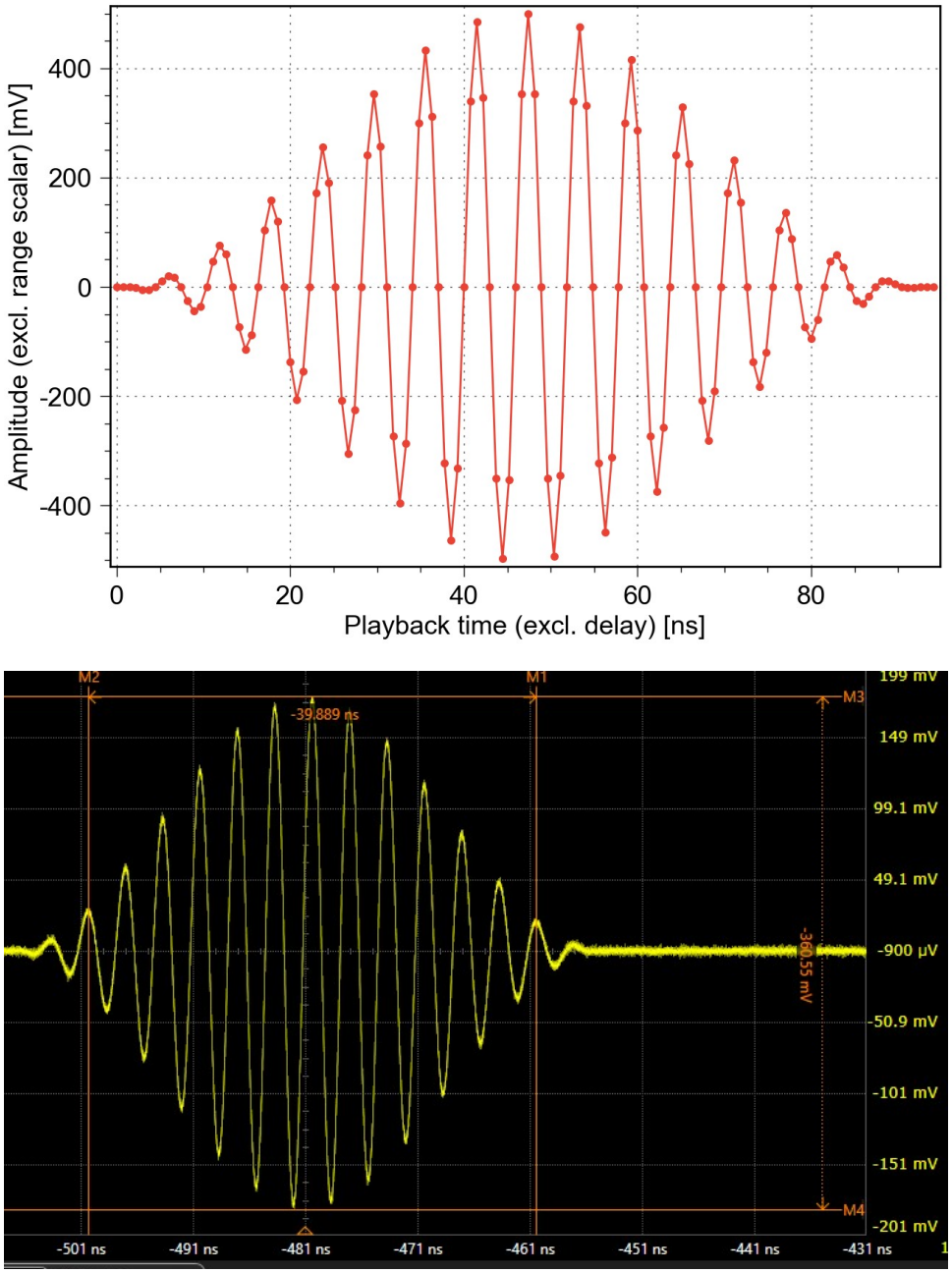


Figure 4.10: Scoped output of the HDAWG using an MSO. Captured at 8.40 GHz, 20.0 GSa/s, 2.00 kpts. The markers have been set following visual inspection. The yellow trace corresponds to the I signal as defined in the multi-qubit pulse generator (Labber). The division is set to 50 mV/div in both shots. The time reference (x-axis) is to the HDAWG's output marker's rising edge, as generated by the SeqC program. A choice was made to gauge the signal widths from more easily identifiable wave edges, i.e. two sine wave crests, as the waveform start/end is embedded in the noise floor and thus not as easily noticeable by eye. The MSO was set to 50 Ω reference even though the HDAWG was not set to 50 Ω coupling, resulting in distorting the scoped output amplitude. The relative amplitude from one point to the other within the waveforms do however coincide well for both waveforms, implying that the HDAWG has loaded the vector data correctly.

4.2.2 Manual instrument interface via Labber's instrument server

Integrating the Python drivers into the Labber instrument framework automatically generates a graphical user interface along with set/get functionality for individual parameters as is shown in Fig. 4.12. Using normal operation, all interface to the instruments would be done in the measurement program using node values (see appendix A); this interface is only visible should the user wish to engage in manual operation. The functions have been assorted into sections as is visible to the left, and groups as is visible in the subwindows just right of the *Sections* column. The assortment was done based in part on the ZI LabOne GUI interface, with the intent of easening on operator confusion: an operator would only have to learn specific setting names and functions in one of the interfaces, as the other one would feature similar naming.

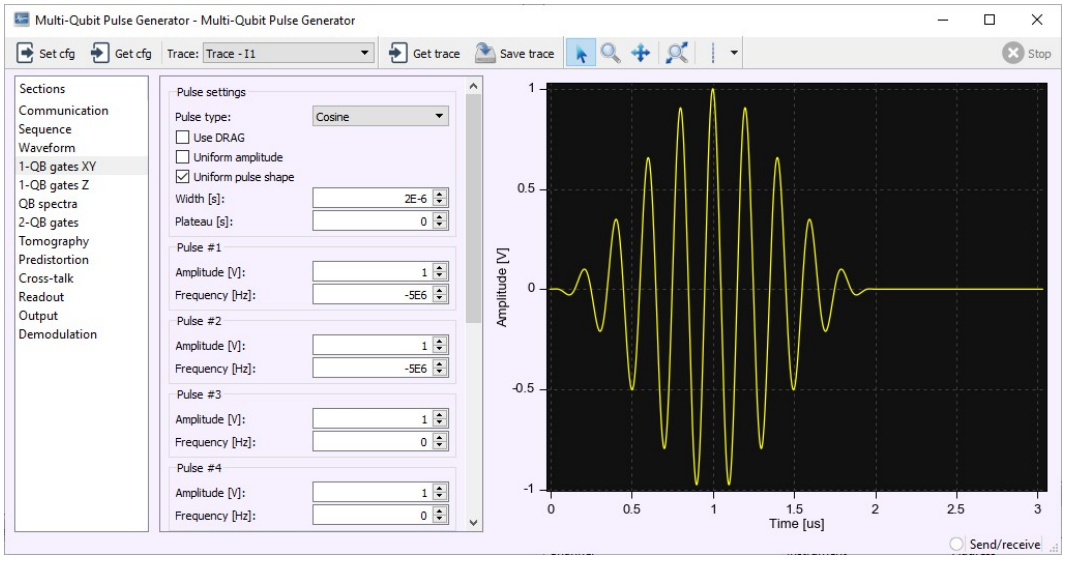


Figure 4.11: Screenshot from Labber's Measurement editor, currently seen defining a $-225\,\mathrm{MHz}$ control pulse to be fed into the HDAWG.

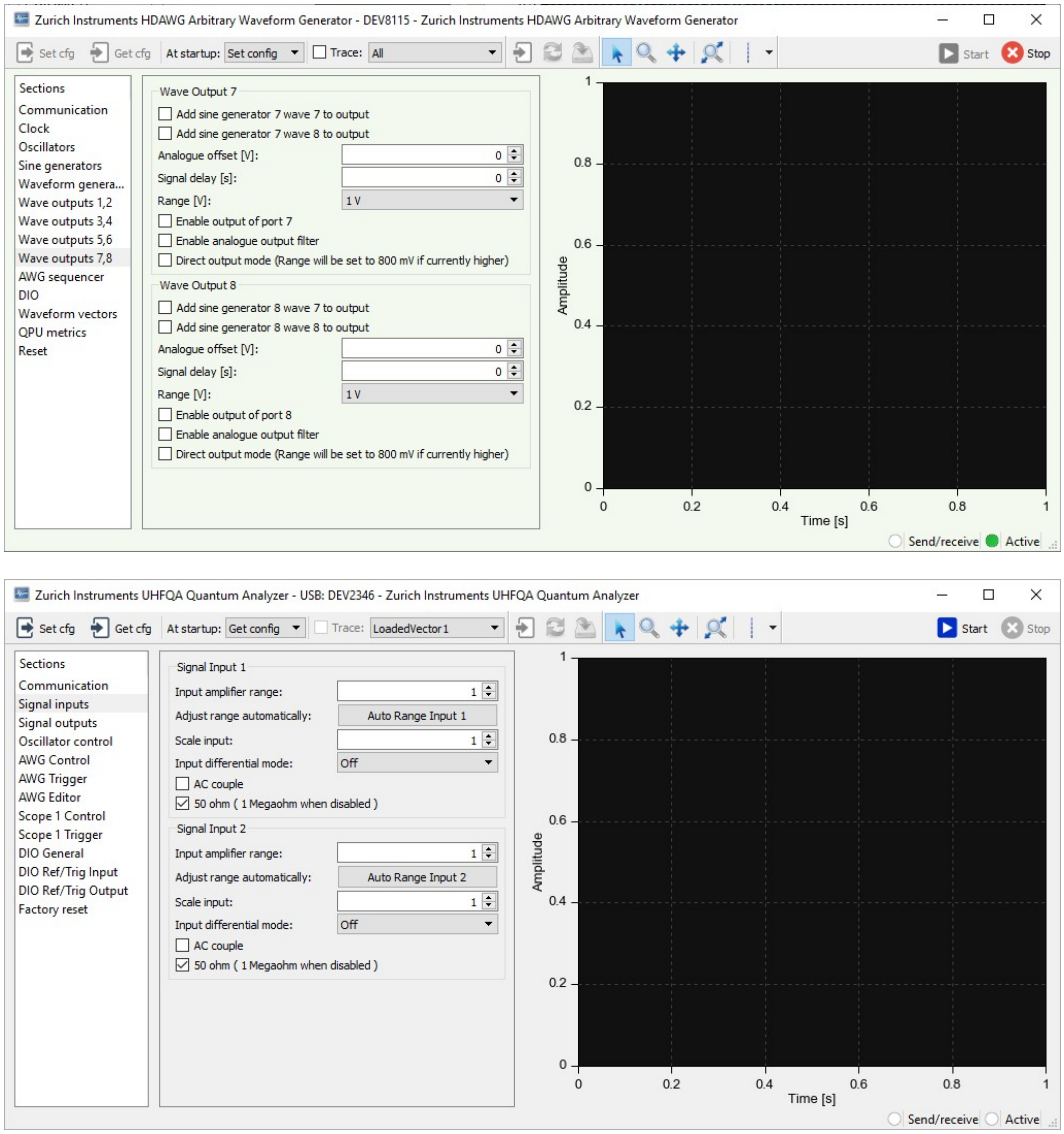


Figure 4.12: Screenshots from Labber's Instrument server, demonstrating manual control over the HDAWG (top) and UHFQA (bottom). The HDAWG interface is demonstrated while bearing an active connection to the instrument. The graphical window demonstrates loaded vectors related to the instrument in Labber, and typically demonstrates the loaded or scoped waveform(s). Fig. 4.11 demonstrates a screenshot from within the measurement editor where the waveform is defined in the multi-qubit pulse generator (virtual instrument).

4.2.3 Internal playback repetition accuracy

This subsection demonstrates the internal waveform playback resolution of the AWG inside the UHFQA, as was used for scoping results when no external trigger was provided by the HDAWG.

As can be seen in Fig. 4.13, the internal waveform playback resolution is not at its theoretically maximum level ($0.\overline{5}$ ns accuracy). This issue has been identified to be related to the zero-padding as done in the algorithm as seen in Fig. 3.5. Experiments show a consistent $4.\overline{4}$ µs accuracy which corresponds to one period of the UHFQA clock related to the wait command as seen in stage *Convert waiting time to ticks of the built-in 225 MHz clock...* in Fig. 3.5. Following this analysis, a future improvement has been identified as trying to increase the repetition accuracy in the algorithm. However, since most experiments are done using another ZI instrument as a 'master' (in this case, the HDAWG), thus not relying on the internal repetition accuracy - some thought should be given as to the purpose of increasing the accuracy beyond $4.\overline{4}$ µs when there is no trigger master available.

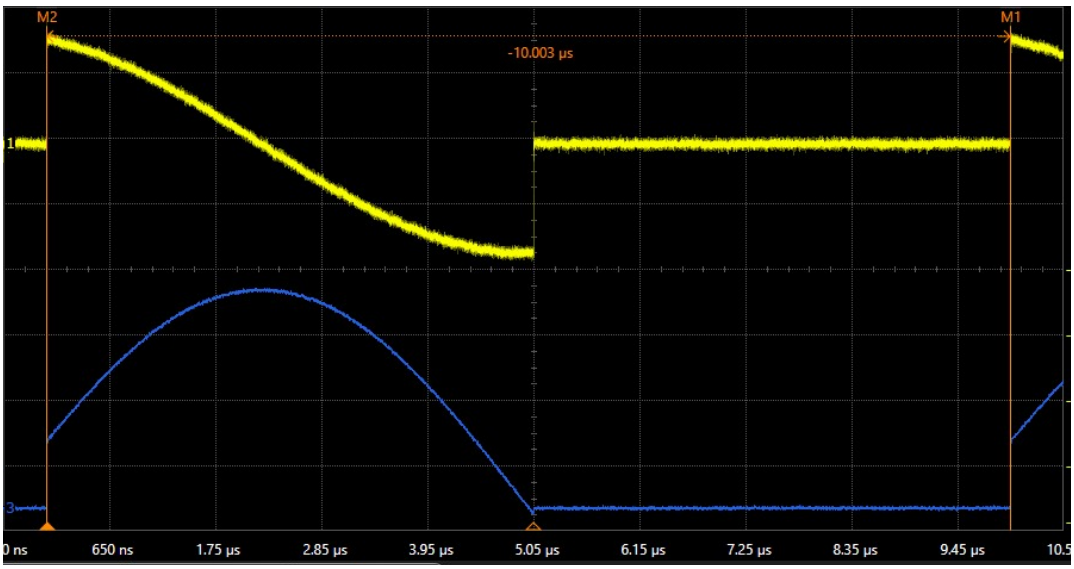


Figure 4.13: Screen capture of a mixed-signal oscilloscope demonstrating the playback resolution accuracy of the internal playback function as described in Fig. 3.5. The loaded waveforms are two arbitrary parts of a sinusoidal, played by the I (yellow, top) and Q (blue, bottom) channels. The flank markers have been manually set at a zoomed-in state. The playback rate was specified to be 10 µs, note the 0.003 µs offset in the measurement: this accuracy rate is not at the theoretically maximum internal playback rate accuracy ($0.\overline{5}$ ns), as outlined in subsection 3.2.2. The MSO was set to 20 GSa/s sample rate, 500 MHz bandwidth and 220 kpts memory depth.

4.3 Platform runtime measurements and upload times

This section presents runtime measurements of specific durations within the constructed instrument platform, such as the average time required for establishing an active instrument connection, uploading a waveform etc. The HDAWG measurements were done on waveforms bearing 7472 samples while the UHFQA measurements were done on samples bearing a swept range, however in similar orders of magnitude. The measurements are summarised in Tab. 4.4.

Table 4.4: Table of acquired runtime measurements. The HDAWG measurements have been coloured red as the data illustrations in Fig. 4.14–4.16 use a red curve for illustrating the HDAWG data and a black curve for the UHFQA data.

Runtime logged	**Average [ms]**	**Median [ms]**	**Hi./Lo. [ms]**	**Samples**
UHFQA Connect	2,603	2,609	2,670 / 2,510	21
UHFQA Compilation	632	641	721 / 498	250
UHFQA Upload	764	701	1,452 / 701	250
UHFQA Scope	1,673	1,639	2,324 / 1,466	240
HDAWG Connect	2,328	2,329	2,391 / 2,211	25
HDAWG Compilation	1,288	1,325	1,454 / 1,018	274
HDAWG Upload	1,481	1,487	1,582 / 1,201	274
HDAWG Injection	131	134	190 / 58	273

An interesting observation during an earlier memory injection in the HDAWG was that every twelfth waveform uploaded to the instrument had a substantial increase in upload time, for reasons which are not understood. The time increase were in the order of seven times longer versus the average. The three highest upload times recorded were then 3 315 ms, 3 018 ms and 2 801 ms. The three lowest were 30 ms, 31 ms and 31 ms (n = 36 in this separate run).

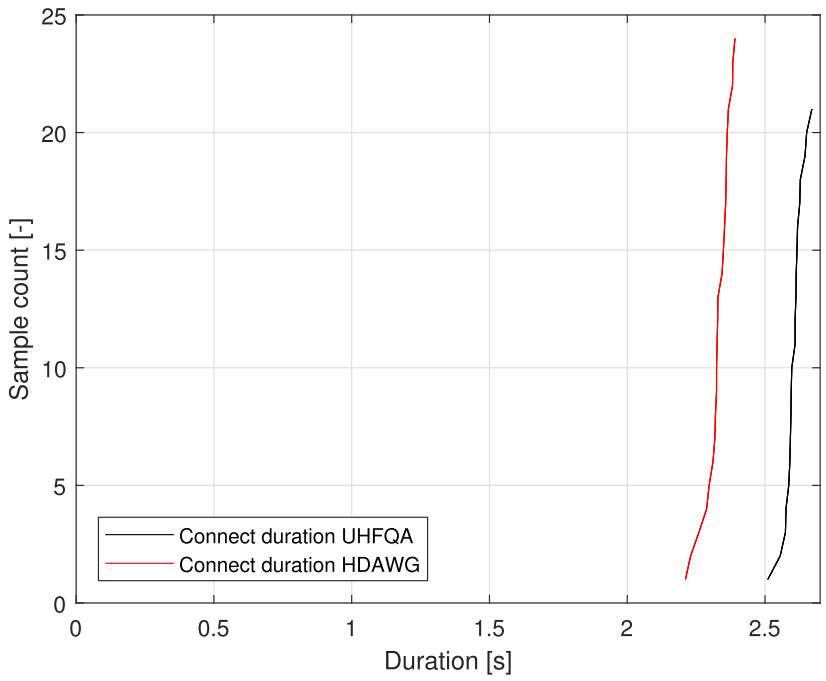
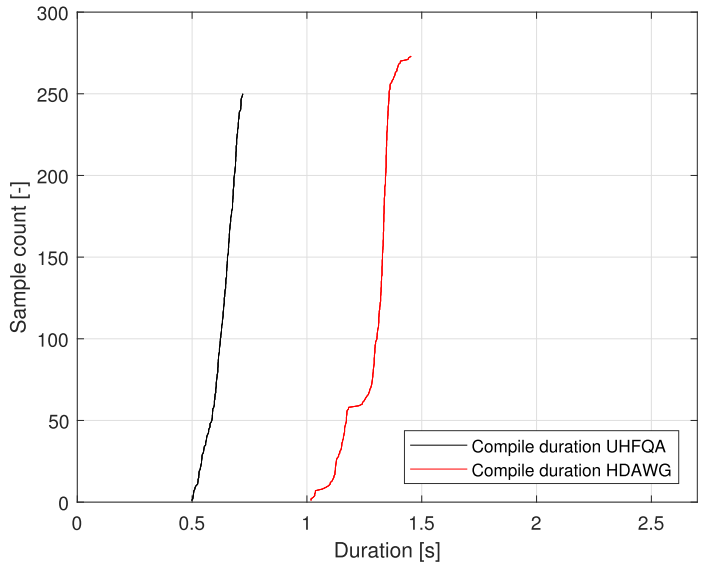
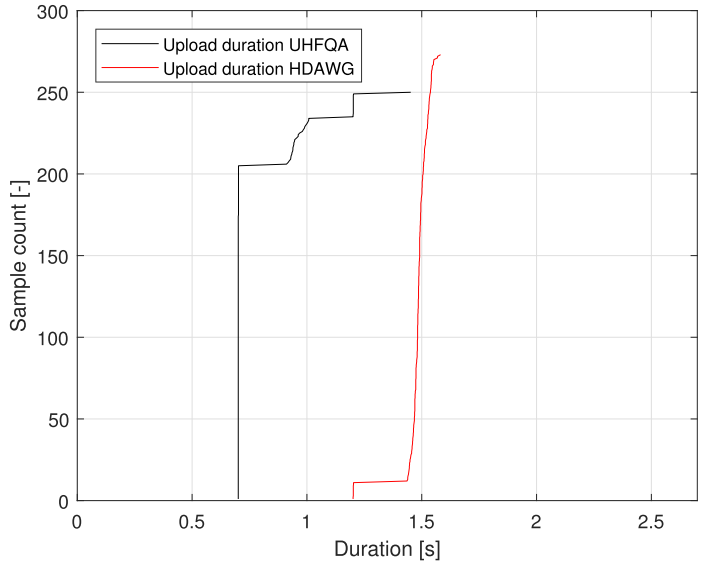


Figure 4.14: Sample-duration graph of the connection times required for Labber to initiate communication with the UHFQA (black) and HDAWG (red). As can be seen in comparison with Fig. 4.15 and 4.16, the connection times are the lengthiest instances (measured) in the drivers. Fortunately, these execute only once per measurement setup, which in turn also serves to explain the lower sample counts as seen on the y-axis. The upward straightness of the lines further imply that the connection times are rather consistent. Although as can be seen in Tab. 4.4, we notice a difference of 160 ms between the shortest and longest established Labber connections in the UHFQA and 180 ms for the HDAWG.

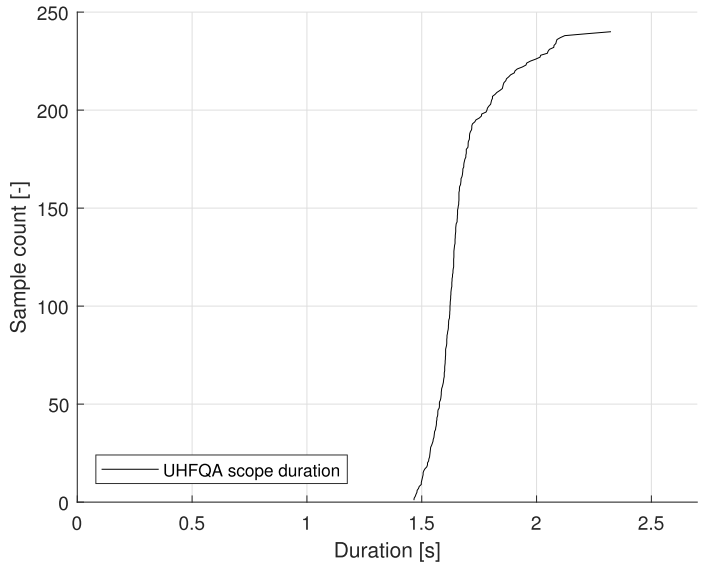


(a)

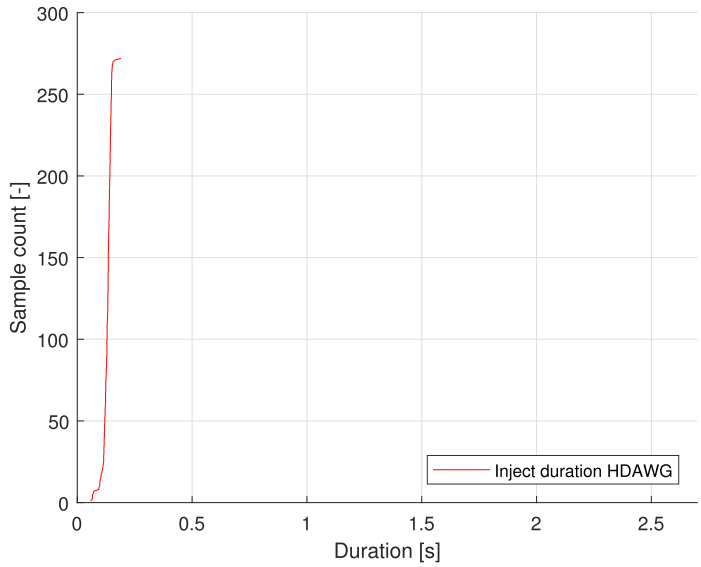


(b)

Figure 4.15: Sample-duration graphs of the **(a)** compile duration times and **(b)** upload duration times for the UHFQA and HDAWG respectively. As can be seen in **(a)**, the compilation time is not as easily estimated as for instance the connection times in Fig. 4.14, fortunately the skew in the case of the HDAWG seems to be towards a shorter duration rather than longer ones as visible by the non-linear break in the lower parts of the red curve. As is seen in **(b)**, we notice a particularly strange duration time set for the UHFQA: the perfectly straight portion implies that the compilation always takes 701 ms, however there is a ladder-like behaviour skewing the set towards longer compilation times. Code which may cause this structure could for instance be a while-loop running in parallel with some function which checks for function completion at some interval of delay. Should the first pass of the loop fail the break-case, albeit the second one (done after some fixed delay) pass - then this would be the expected duration set. Although, please observe the non-stairlike behaviour at sample ~225, which implies that there is still some variation should the speculated loop structure be a feasible hypothesis. The sudden skew towards lower times in the HDAWG upload times is also of interest, visible near the bottom.



(a)



(b)

Figure 4.16: Sample-duration graphs of the **(a)** UHFQA scope functionality and the **(b)** HDAWG memory injection functionality. As can be seen in **(a)**, the oscilloscope duration times vary substantially between runs. The differences between the shortest and longest scope runs span 858 ms, with a skew towards longer durations. The scope was set up in single-shot mode, meaning in turn that for instance applied averaging would only serve to increase these times further. We see in **(b)** when compared to Fig. 4.15 that there is a major improvement in waveform upload times for the HDAWG when using memory injection as opposed to using SeqC code uploading. Also, the upload times using injection shows slightly less variance, as seen in the vertical straightness of the curve.

5
Discussion & Conclusion

Following the execution of this master's thesis project, a conclusion has been reached as will be presented at the end of this chapter. Before this, there are noteworthy points of discussion that should be addressed regarding various details throughout this thesis. This chapter assumes to some extent that the reader is now acquainted with various topics described in the thesis content.

The first section will treat a different (considered) problem-solving approach to how the drivers play back data. The next section will discuss observed shortcomings in the synchronisation between the HDAWG and UHFQA, and how these could be solved in future implementations. Finally, this chapter will briefly discuss social impacts etc. of this thesis and finally conclude with the answer to the research question.

5.1 Compression-inspired sequencer programs

As was shown in section 4.3, the time required for uploading constitutes a significant portion of the experiment time in comparison to other durations of an experiment.

Borrowing from techniques used in various standard compression algorithms such as LZ77/78, one could imagine a future setup where waveform primitives (such as π-pulses) are defined in the SeqC and played back when needed according to a list. A waveform primitive could be defined using memory injection for fastest possible upload, then stored as an available waveform in the memory ready to be played back according to some SeqC-program written following the sequence list. The program itself would be automatically generated by the driver, with the overall goal of mimicking compression lookup tables as given in the very common Lempel-Ziv algorithm [23]-[25]. This would be done mainly by spotting identical subcomponents in a given set of data and re-using them instead of redefining duplicate data. Even better would be if this algorithm could be controlled using the User registers, allowing for potentially great waveform playback flexibility. For instance, a 3 in User register 0 could correspond to playing a π-pulse, while a 100 in register 1 could correspond to waiting 100 ticks at playback finish. This solution variant might however bring with it unwanted side-effects, as calls to User registers are sluggish and cumbersome from the sequencer. 'Slow' would imply that a single call to fetching the content of a user register is comparable to a large portion of the energy relaxation time, effectively making the qubit useful for less time (which is already at a premium).

For future projects, it is hereby instead suggested to split the multi-qubit generator in Labber into a waveform primitive generator and a sequencer generator. The sequence generator could take a custom algorithm to execute as a parameter, and fetch waveform primitives from the primitive generator as needed. The output could then be interpreted by the drivers, which would generate a SeqC program following the generators' specifications.

Perhaps with the combination of User registers at suitable locations, this solution might forego the need of sequencer code uploading altogether. This is speculated to keep the optimised runtime of a measurement intact while allowing for custom algorithm construction.

5.2 Synchronising the HDAWG to the UHFQA

On the 17th of April 2019, Bruno Küng of Zurich Instruments recommended that synchronisation between the HDAWG and the UHFQA should be done using the triggering functionality, as the intended intra-device communications link using ZI's proprietary device connection is not yet implemented for the HDAWG and the UHFQA combination. The user is thus tasked with designing suitable SeqC code for intra-device communication, as would be needed in a vast number of experiments.

Currently, there is no implemented functionality in the HDAWG for listening to the UHFQA's state of readiness after a scope run. Instead, the user has the option to set the amount of time in which the HDAWG will cease operation after a played-back waveform set. For successful operation, the operator must estimate how long time is needed for the UHFQA to complete its measurements before a new waveform can be played, in which case the user likely sets unnecessarily long waiting times to be safe. A suggested improvement for the future would be to include another triggering channel, in which the UHFQA can signal the HDAWG that a scope run has been completed. However simple modification to the SeqC code will not suffice as the averaging is performed on its master PC. The wiring solution can however remain intact with simple modifications to the HDAWG's driver, as the triggering ports are bidirectional and trigger port directional switching is easily implemented in the drivers.

5.3 Social, economic and environmental factors

With sufficient modification, the instrument platform following this master's thesis project could fit inside a quantum-processor software stack. This would enable software developers to develop code for generating compatible compilations into said platform, and hardware developers to implement newer QPU hardware. This easening of development in turn implies a step closer to quantum supremacy, which has a vast array of foreseen effects on everyday life in topics such as secure communication and encryption breaking. Social and economic gains are however foreseen in for instance the vast potential of natural quantum simulations enabled by quantum computing. It is imaginable that such simulator systems could for instance improve our current understanding of the environment. Many materials used in the production of quantum computing systems akin to the one used in this thesis require rare-earth metals and conflict minerals for electronics production. Procuring these materials is in turn by common knowledge known to have a social, economic and environmental impact without need for further explanation.

5.4 Conclusion of this master's thesis

In this master's thesis, I demonstrated an instrument and measurement automation platform created using Python for implementation in Labber. The platform is currently operational in a multi-qubit quantum processor located at the Quantum Technology laboratory, at the Department of Microtechnology and Nanoscience, Chalmers. The platform created has automated much work normally required to run the ZI instruments, and allowed for simplification as the data acquired is now handled by instrument control software. Some parameters were acquired using these drivers for the verifying QPU, specifically the spectral locations of its resonators, qubit frequency of one of its qubits, said qubit's π-pulse amplitude and energy relaxation time. The platform was benchmarked in timestamps, revealing a large improvement depending on the choice of waveform upload method.

The introduction laid out the current research field of quantum computing, and emphasised the need for control and readout automation in order to increase quantum-computer scalability. From this need arose the research question: how should one construct a scalable instrument-automation platform that can control a multi-qubit superconducting quantum-processor setup, using arbitrary waveform generation and lock-in amplifier readout?

The answer to the research question is: by implementing the required instruments in a control platform framework where ...

- The user-required interactivity should be minimised as much as possible as not to restrict on experiment design.
- The drivers should (with supporting quantum theory) be optimised for low latency, for instance minimising instrument intercommunication and optimising sequence tables in order to reduce on lengthy function calls.
- The drivers should allow for arbitrary user-input, preferable calculated in the instrument handling software.
- The automation platform should be verifiable on real hardware by replicating existing and theoretically known experiments.
- The associated waveform upload times should be minimised.

For future implementations, I primarily expect the instrument and measurement automation platform to improve in terms of SeqC code generation, upload times and known verified QPU experiments.

The goals as laid out by this master's thesis project were completed, which in turn implies that the thesis resulted in instrument and measurement automation for classical control of a multi-qubit quantum processor.

Bibliography

[1] T. D. Ladd, F. Jelezko, R. Laflamme, Y. Nakamura, C. Monroe and J. L. O'Brien, "Quantum computers", *Nature*, vol. 464, no. 7285, pp. 45-53, 2010 [Online]. Available: https://dx.doi.org/10.1038/nature08812.

[2] A. Steane, "Quantum computing", *Reports on Progress in Physics*, vol. 61, no. 2, p. 118, 1998 [Online]. Available: https://dx.doi.org/10.1088%2F0034-4885%2F61%2F2%2F002

[3] I. L. Chuang and Y. Yamamoto, "Simple quantum computer", *Physical Review A*, vol. 52, no. 5, pp. 3489-3496, 1995 [Online]. Available: https://dx.doi.org/10.1103/physreva.52.3489

[4] R. Krishna, V. Makwana and A. P. Suresh, "A Generalization of Bernstein-Vazirani Algorithm to Qudit Systems", arXiV e-prints, 2016 [Online]. Available: https://ui.adsabs.harvard.edu/abs/2016arXiv160903185K. [Accessed: 10- Apr- 2019]

[5] P. W. Shor, "Progress in Quantum Algorithms", in *Experimental Aspects of Quantum Computing*, 1st ed., H. O. Everitt, Ed. Research Triangle Gate, NC: Springer Science and Business Media Inc., 2005, pp. 5-12 [Online]. Available: https://link.springer.com/content/pdf/10.1007%2F0-387-27732-3.pdf

[6] M. Dobšíček, "Quantum computing, phase estimation and applications", Faculty of Electrical Engineering, Czech Technical University in Prague, Prague, Czechia, 2008 [Online]. Available: https://arxiv.org/pdf/0803.0909.pdf.

[7] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*, 10th ed. Cambridge: Cambridge University Press, 2018, pp. 1-11.

[8] G. Ciaramicoli, I. Marzoli and P. Tombesi, "Scalable Quantum Processor with Trapped Electrons", *Physical Review Letters*, vol. 91, no. 1, 2003 [Online]. Available: https://dx.doi.org/10.1103/PhysRevLett.91.017901.

[9] C. Monroe et al., "Large Scale Modular Quantum Computer Architecture with Atomic Memory and Photonic Interconnects", Physical Review A, 2014 [Online]. Available: https://dx.doi.org/10.1103/PhysRevA.89.022317.

[10] M. H. Devoret, A. Wallraff and J. M. Martinis, "Superconducting Qubits: A Short Review", cond-mat/0411174, 2004 [Online]. Available: https://ui.adsabs.harvard.edu/#abs/2004cond.mat.11174D. [Accessed: 27- Mar- 2019]

[11] G. Andersson, "Circuit quantum electrodynamics with a transmon qubit in a 3D cavity," M.Sc. thesis, KTH Royal Institute of Technology, Sweden, 2015 [Online]. Available: https://www.diva-portal.org/smash/get/diva2:813846/FULLTEXT01.pdf

[12] A. Wallraff et al., "Sideband Transitions and Two-Tone Spectroscopy of a Superconducting Qubit Strongly Coupled to an On-Chip Cavity", *Physical Review Letters*, vol. 99, no. 5, pp. 1-2, 2007 [Online]. Available: https://dx.doi.org/10.1103/PhysRevLett.99.050501.

[13] J. Majer et al., "Coupling superconducting qubits via a cavity bus", *Nature*, vol. 449, no. 7161, p. 443, 2007 [Online]. Available: https://dx.doi.org/10.1038/nature06184.

[14] Y. Salathé et al., "Low-Latency Digital Signal Processing for Feedback and Feedforward in Quantum Computing and Communication", *Physical Review Applied*, vol. 9, no. 3, pp. 034011, 2018 [Online]. Available: https://dx.doi.org/10.1103/PhysRevApplied.9.034011.

[15] D. J. Reilly, "Engineering the quantum-classical interface of solid-state qubits", *Npj Quantum Information*, vol. 1, no. 1, 2015 [Online]. Available: https://dx.doi.org/10.1038/npjqi.2015.11

[16] "Labber – Software for Instrument Control and Lab Automation", Labber.org, 2019. [Online]. Available: http://labber.org/. [Accessed: 12- Jan- 2019]

[17] "IBM Q Experience", research.ibm.com, 2019. [Online]. Available: https://www.research.ibm.com/ibm-q/technology/experience/. [Accessed: 25- May- 2019]

[18] *HDAWG User Manual*, 2nd ed. Zurich, Switzerland: Zurich Instruments AG, 2018 [Online]. Available: https://www.zhinst.com/sites/default/files/ziHDAWG_UserManual_54600.pdf. [Accessed: 24- Jan- 2019]

[19] *UHFQA User Manual*, 1st ed. Zurich, Switzerland: Zurich Instruments AG, 2018 [Online]. Available: https://www.zhinst.com/sites/default/files/ziUHFQA_UserManual_58300.pdf. [Accessed: 18- Jun- 2019]

[20] A. Wallraff et al., "Strong coupling of a single photon to a superconducting qubit using circuit quantum electrodynamics", *Nature*, vol. 431, no. 7005, pp. 162-167, 2004 [Online]. Available: https://dx.doi.org/10.1038/nature02851

[21] S. J. Orfanidis, *Electromagnetic Waves and Antennas*. Piscataway, NJ: Rutgers University, 2016, p. 664 [Online]. Available: https://www.ece.rutgers.edu/~orfanidi/ewa/ch14.pdf. [Accessed: 11- Feb- 2019]

[22] H. Dang et al., "P4xos: Consensus as a Network Service", Università della Svizzera italiana, Faculty of Informatics, Lugano, Switzerland, 2018 [Online]. Available: https://pdfs.semanticscholar.org/b146/9ab10b80f36cef02832bfe12d91f5058bb99.pdf.

[23] R. A. Bedruz and A. R. F. Quiros, "Comparison of Huffman Algorithm and Lempel-Ziv Algorithm for audio, image and text compression", in *2015 International Conference on Humanoid, Nanotechnology, Information Technology, Communication and Control, Environment and Management (HNICEM)*, Cebu City, Philippines, 2015, pp. 1-6 [Online]. Available: https://dx.doi.org/10.1109/HNICEM.2015.7393210.

[24] A. R. Saikia, "LZW (Lempel–Ziv–Welch) Compression technique", *GeeksforGeeks*, 2017. [Online]. Available: https://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique/. [Accessed: 02- Jun- 2019]

[25] M. K. Tank, "Implementation of Lempel-ZIV algorithm for lossless compression using VHDL", in *Thinkquest 2010: International Conference on Contours of Computing Technology*, Mumbai, India, 2010, pp. 275-278 [Online]. Available: https://dx.doi.org/10.1007/978-81-8489-989-4_51.

[26] C. Ryan, B. Johnson, D. Ristè, B. Donovan and T. Ohki, "Hardware for dynamic quantum computing", *Review of Scientific Instruments*, vol. 88, no. 10, pp. 104703, 2017 [Online]. Available: https://dx.doi.org/10.1063/1.5006525.

[27] T. Häner, D. Steiger, K. Svore and M. Troyer, "A software methodology for compiling quantum programs", *Quantum Science and Technology*, vol. 3, no. 2, p. 020501, 2018 [Online]. Available: https://dx.doi.org/10.1088/2058-9565/aaa5cc

[28] S. J. Gay, *Quantum programming languages: survey and bibliography*, 2nd ed. Glasgow, United Kingdom: Department of Computing Science, University of Glasgow, 2006, pp. 581–600 [Online]. Available: https://www.cambridge.org/core/services/aop-cambridge-core/content/view/80E4ECC8AE770B625A48F2EE28358BA6/S0960129506005378a.pdf/quantum_programming_languages_survey_and_bibliography.pdf.

[29] S. Jordan, "Algebraic and Number Theoretic Algorithms", *Quantum Algorithm Zoo*, 2018. [Online]. Available: http://quantumalgorithmzoo.org/. [Accessed: 11- Feb- 2019]

[30] S. Lloyd, "Universal Quantum Simulators", *Science*, vol. 273, no. 5278, pp. 1073-1078, 1996 [Online]. Available: https://dx.doi.org/10.1126/science.273.5278.1073

[31] R. J. Schoelkopf and S. M. Girvin, "Wiring up quantum systems", *Nature*, vol. 451, no. 7179, pp. 664-669, 2008 [Online]. Available: https://dx.doi.org/10.1038/451664a

[32] S. Aaronson, *Lecture 18, Tues March 28: Bernstein-Vazirani, Simon*, 1st ed. Austin, TX: University of Texas at Austin, 2019, pp. 1-2 [Online]. Available: https://www.scottaaronson.com/qclec/18.pdf. [Accessed: 10- Apr- 2019]

[33] S. G. Menon, *Lecture 08: More on Algorithms*, 1st ed. Chicago, IL: The University of Chicago, 2018, pp. 1-3 [Online]. Available: http://people.cs.uchicago.edu/~ftchong/33001/lecture08.pdf. [Accessed: 10- Apr- 2019]

[34] K. Rudinger, *Lecture 4: Elementary Quantum Algorithms*, 1st ed. Madison, WI: University of Wisconsin-Madison, 2010, pp. 1-5 [Online]. Available: http://pages.cs.wisc.edu/~dieter/Courses/2010f-CS880/Scribes/04/lecture04.pdf. [Accessed: 10- Apr- 2019]

[35] T. R. Kuphaldt, *Lessons in Electric Circuits Vol. IV - Digital*, 4th ed. 2007, ch. Logic signal voltage levels [Online]. Available: https://www.ibiblio.org/kuphaldt/electricCircuits/Digital/DIGI_3.html. [Accessed: 28- Mar- 2019]

[36] M. Sullivan, *Quantum mechanics for electrical engineers*. Hoboken, NJ: Wiley - IEEE Press, 2012, pp. 44-46. ISBN: 978-0-470-87409-7

[37] A. C. Lierta, T. Demarie and E. Munro, "Quantum Computation: a journey on the Bloch sphere", *Quantum World Association*. 2019 [Online]. Available: https://medium.com/@quantum_wa/quantum-computation-a-journey-on-the-bloch-sphere-50cc9d73530. [Accessed: 03- Jun- 2019]

[38] M. E. Ware, "Flux-tunable superconducting transmons for quantum information processing," Dissertation, Syracuse University, Syracuse, NY, 2015 [Online]. Available: https://surface.syr.edu/cgi/viewcontent.cgi?article=1249&context=etd.

[39] P. Krantz, M. Kjaergaard, F. Yan, T. Orlando, S. Gustavsson and W. Oliver, "A Quantum Engineer's Guide to superconducting qubits", *Applied Physics Reviews*, vol. 6, no. 2, p. 021318, 2019 [Online]. Available: https://dx.doi.org/10.1063/1.5089550.

[40] D. P. DiVincenzo, "The Physical Implementation of Quantum Computation", *Fortschritte der Physik*, vol. 48, no. 9-11, pp. 771-783, 2000 [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/1521-3978%28200009%2948%3A9/11%3C771%3A%3AAID-PROP771%3E3.0.CO%3B2-E

[41] A. Bengtsson, "Degenerate and nondegenerate Josephson parametric oscillators for quantum information applications", Chalmers University of Technology, Sweden, Gothenburg, 2017, ISSN 1652-0769. [Online]. Available: http://publications.lib.chalmers.se/records/fulltext/500028/500028.pdf. [Accessed: 03- Jun- 2019]

[42] J. von Neumann, "First Draft of a Report on the EDVAC", Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia, PA, 1945 [Online]. Available: https://sites.google.com/site/michaeldgodfrey/vonneumann/vnedvac.pdf?attredirects=0&d=1. [Accessed: 03- Jun- 2019]

[43] Zurich Instruments, "Superconducting Qubit Characterization", *AZO Materials*, 2016 [Online]. Available: https://www.azom.com/article.aspx?ArticleID=13264. [Accessed: 04- Jun- 2019]

[44] L. A. Tuan, "Frequency- and time-domain characterization of a transmon qubit in a transmission line", Technische Universität München, Munich, Germany, 2017 [Online]. Available: https://www.wmi.badw.de/publications/theses/Le%20anh,Tuan_Masterarbeit_2017.pdf.

[45] N. K. Langford, "Engineering the quantum: probing atoms with light & light with atoms in a transmon circuit QED system", *16th Asian Quantum Information Science Conference*, 2016 [Online]. Available: http://aqis-conf.org/2016/wp-content/uploads/2015/12/20160828langford_aqis_tutorial.pdf

[46] J. Koch et al., "Charge-insensitive qubit design derived from the Cooper pair box", *Physical Review A*, vol. 76, no. 4, pp. 1-3, 2007 [Online]. Available: https://dx.doi.org/10.1103/PhysRevA.76.042319.

[47] I. Chuang, "Lecture 19: How to Build Your Own Quantum Computer", 2003 [Online]. Available: https://ocw.mit.edu/courses/mathematics/18-435j-quantum-computation-fall-2003/lecture-notes/qc_lec19.pdf

[48] J. Gambetta and A. Kandala, "Noise Amplification Squeezes More Computational Accuracy From Today's Quantum Processors", *IBM Research Blog.* 2019 [Online]. Available: https://www.ibm.com/blogs/research/2019/03/noise-amplification-quantum-processors/. [Accessed: 01- Apr- 2019]

[49] F. Yan et al., "Distinguishing Coherent and Thermal Photon Noise in a Circuit Quantum Electrodynamical System", *Physical Review Letters*, vol. 120, no. 26, pp. 1-2, 2018 [Online]. Available: https://dx.doi.org/10.1103/PhysRevLett.120.260504.

[50] W. Unruh, "Maintaining coherence in quantum computers", *Physical Review A*, vol. 51, no. 2, pp. 992-997, 1995 [Online]. Available: https://dx.doi.org/10.1103/physreva.51.992

[51] "Labber Python API Documentation", Labber.org, 2016. [Online]. Available: http://labber.org/online-doc/api/index.html [Accessed: 27- May- 2019]

[52] A. Megrant et al., "Planar superconducting resonators with internal quality factors above one million", *Applied Physics Letters*, vol. 100, no. 11, pp. 113510-3, 2012 [Online]. Available: https://dx.doi.org/10.1063/1.3693409.

A

Implemented ZI API commands

This appendix lists the implemented Zurich Instruments API functions for the UHFQA and HDAWG respectively. Even though their API has support for a substantially larger number of commands, the ones presented here were implemented as they constituted necessary functions for combating the problems as laid out by this thesis.

The instruments feature lists of many identical calls, for instance the call to activate signal output 1 is the same as activating signal output 2 apart from the number used in the call. The drivers thus iterate a received command to figure out what particular output (or similar) was activated. These iterables have been colour-coded depending on how many options of the call may be iterated through in the driver.

- Black commands are not iterable, this call is unique to one function in the instrument.
- Blue commands are iterable through 1 to 2.
- Red commands are iterable through 1 to 4.
- Green commands are iterable through 1 to 8.
- Pink commands are iterable through 1 to 16.

To demonstrate, *SigOut1On* is blue meaning that there are two functions in the driver named *SigOut1On* and *SigOut2On* respectively. They in turn control the ports *Signal out 1* and *Signal out 2*. Some functions feature numbers as non-iterable parts of their call name, such as the HDAWG command *SineGen1EnableWave2*. In these cases, it is the depicted 1 that iterates while the 2 remains identical for all eight iterations.

A.1 Driver ZI API commands for the UHFQA

The implemented ZI API commands for the UHFQA are visible in Tab. A.1 and A.2.

Table A.1: Table of implemented booleans (top) and floats (bottom) for the UHFQA Labber driver. The %s is replaced by the device ID of the instrument as set by the user when adding the device to Labber's instrument server.

Command	Brief function	API node name
SigOut1On	Output on	/%s/sigouts/0/on
ImpedanceFifty1On	Apply 50 Ω to output	/%s/sigouts/0/imp50
EnableScope	Activate the scope	/%s/scopes/0/enable
Force Scope	Force the scope	/%s/scopes/0/trigforce
SingleShotScope	Single shot mode	/%s/scopes/0/single
TriggerEnabledScope	Use scope trigger	/%s/scopes/0/trigenable
Auto Threshold Input 1	Automatically set threshold for trigger input	/%s/triggers/in/0/autothreshold
Auto Range Input 1	Automatically set voltage range for trigger input	/%s/sigins/0/autorange
ACSigIn1	AC couple signal input	/%s/sigins/0/ac
FiftyOhmSigIn1	Apply 50 Ω to input	/%s/sigins/0/imp50
HysteresisMode1	Alter trigger hysteresis from range percent to fixed voltage	/%s/scopes/0/trighysteresis/mode

Command	Brief function	API node name
TriggerVoltageScope	Scope trigger voltage	/%s/scopes/0/triglevel
AmplitudeOutput1AWG	Scale output waveform of the AWG	/%s/awgs/0/outputs/0/amplitude
RangeSigIn1	Set voltage range of the signal input	/%s/sigins/0/range
SampleLengthScope	Amount of samples in a scope shot	/%s/scopes/0/length
ScalingSigIn1	Input voltage scalar for the signal input	/%s/sigins/0/scaling
Oscillator1	Output oscillators	/%s/oscs/0/freq
TriggerDelayScope1	Time needed at valid trigger level	/%s/scopes/0/trigdelay
TriggerHoldoffScope1	Turn off the trigger after valid trigger	/%s/scopes/0/trigholdoff
AmplitudeOutput1	Output scalar	/%s/scopes/0/trigholdoff
UserRegister1	Runtime-settable user registers	/%s/awgs/0/userregs/0

Table A.2: Table of implemented combinational lists (top) and complex functions (bottom) for the UHFQA Labber driver. The functions in the complex subtable feature extended functionality beyond simple API get/set calls. These contain several ZI API calls with various purposes.

Command	Brief function	API node name
TriggerFlankScope	Trigger flank type	/%s/scopes/0/trigslope
SignalSourceChannel1Scope	Select what input to scope	/%s/scopes/0/channels/0/inputselect
SamplingRateScope	Scope sampling rate	/%s/scopes/0/time
TriggerSourceScope	Scope trigger source	/%s/scopes/0/trigchannel
DiffSigIn1	Set up digital input differential modes	/%s/sigins/0/diff
ModeOutput1AWG	Select AWG output mode, e.g. modulation	/%s/awgs/0/outputs/0/mode
TriggerSourceAnalogue1AWG	AWG analogue trigger input source	/%s/awgs/0/triggers/0/channel
TriggerSourceDigital1AWG	AWG digital trigger input source	/%s/awgs/0/auxtriggers/0/channel
SlopeDigital1AWG	AWG digital trigger slope type	/%s/awgs/0/auxtriggers/0/slope
OutputSamplingRateAWG	AWG sampling rate	/%s/awgs/0/time

Command	Brief function
RangeSigOut1	Find impedance, update SeqC program RSC scalar
EnableAWG	Activates output for the AWG
EnableRerunAWG	Activates AWG rerun
SampleLengthScope	Set scope sample rate, using appropriate datatype
ManualThresholdRefTrigInput1	Manual input voltage threshold, force Labber update
OffsetSigOut1	Manual output DC level, force Labber update
TriggerHysteresisScope	Set absolute hysteresis mode, set value, force Labber update
RelativeTriggerHysteresisScope	Set relative hysteresis mode, set value, force Labber update
TriggerReferenceScope	Convert percent to decimal, set trigger reference
I messed up...	Reset using ZI Utils 'disableEverything()'

A.2 Driver ZI API commands for the HDAWG

The implemented ZI API commands for the UHFQA are visible in Tab. A.3 and A.4.

Table A.3: Table of implemented combinational lists (top) and other ZI API functions (bottom) for the HDAWG Labber driver. The %s is replaced by the device ID of the instrument as set by the user when adding the device to Labber's instrument server.

Command	Brief function	API node name
SineGenOscSelect1	Set oscillator source for this signal	/%s/sines/0/oscselect
ModulationOutput1	Modulation type for the selected output	/%s/awgs/0/outputs/0/modulation/mode
DigitalTrigger1	Link digital trigger to corresponding port	/%s/awgs/0/auxtriggers/0/channel
SlopeDigitalTrigger	Digital trigger slope	/%s/awgs/0/auxtriggers/0/slope
ChannelGrouping	Set channel grouping, for instance 1x8	/%s/system/awg/channelgrouping
AWGSequencerSampleRate	Sequencer sample playback rate	/%s/awgs/0/time
RangeOutput1	Maximum voltage at port	/%s/sigouts/0/range

Command	Brief function
MarkerOutSignal1	Assign signal to selected marker output
I messed up...	Reset using ZI Utils 'disableEverything()'
Disable all outputs	Disable all output ports
LoadedVector1	Marks the channel as used in measurement

Table A.4: Table of implemented booleans (top) and floats (bottom) for the HDAWG Labber driver. The %s is replaced by the device ID of the instrument as set by the user when adding the device to Labber's instrument server. Do note that the SequencerEnable commands all have identical ZI API node strings, which has caused severe workarounds in implementation and also the iterability of each of these functions.

Command	Brief function	API node name
SineGen1EnableWave1	Amplitude of the first sine generator output wave	/%s/sines/0/enables/0
SineGen1EnableWave2	Amplitude of the second sine generator output wave	/%s/sines/0/enables/1
DirectOutput1	Bypass the output amplifier	/%s/sigouts/0/direct
FilterOutput1	Add analogue output filter	/%s/sigouts/0/filter
EnableOutput1	Enable output	/%s/sigouts/0/on
HoldOutput1	Hold last output sample constant even after the wave finishes playing	/%s/awgs/0/outputs/0/hold
SequencerEnable1x8Group1	Enable sequencer group in 1x8 mode	/%s/awgs/0/enable
SequencerEnable2x4Group1	Enable sequencer group in 2x4 mode	/%s/awgs/0/enable
SequencerEnable4x2Group1	Enable sequencer group in 4x2 mode	/%s/awgs/0/enable
EnableAWG	Play sequencer program	awgModule/awg/enable

Command	Brief function	API node name
UserRegister1	Runtime-settable user registers	/%s/awgs/0/userregs/0
Oscillator1Freq	Oscillator frequency at given port	/%s/oscs/0/freq
SineGenHarmonic1	Multiplier for the oscillator's reference frequency	/%s/sines/0/harmonic
SineGenPhase1	Relative sine signal phase	/%s/sines/0/phaseshift
SineGenAmplitude1Wave1	Amplitude of the first sine generator output wave	/%s/sines/0/amplitudes/0
SineGenAmplitude1Wave2	Amplitude of the second sine generator output wave	/%s/sines/0/amplitudes/1
DelayOutput1	Delay of the output signal	/%s/sigouts/0/delay
OffsetOutput1	Analogue offset voltage in amplified mode	/%s/sigouts/0/offset
AmplitudeScalingOutput1	Digital scalar applied to the output	/%s/awgs/0/outputs/0/amplitude
ConfigClockReference	Set system clock frequency	/%s/system/clocks/sampleclock/freq