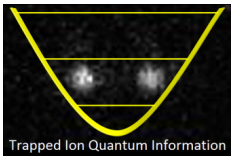
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



Enabling a quantum-gate-level interface with a trapped ion control system

Master's Thesis

Marco Erwin Stucki

marco.stucki@outlook.com

Institute for Quantum Electronics

Departement of Physics, D-PHYS

ETH Zürich

Supervisors:

Martin Stadler

December 9, 2021

Abstract

As quantum computing experiments grow in scale and become more robust to noise the complexity of applicable experiments grows as well. At this point, it is a natural step to switch from a low-level control of the experiments to a higher level. More precisely: switching from a pulse-level control to a control through a quantum programming language.

The first step of this transition is established through this master's thesis. After choosing a suitable quantum programming language (Qiskit), an interface between the trapped ion control system of the TIQI group (trapped ion quantum information group of the federal institute of technology zurich) and this language was established through a parser. A backend allows the transformation from a quantum circuit to a pulse schedule. This schedule is then translated by the parser to a JSON-string that follows a well-defined structure, can be read-in by the control system and executed on the hardware. In order to utilise the strengths of the control system, additional instructions were added to Qiskit. This way, the entire functionality of the control system can be utilised at a higher level.

This work enables a gate-level quantum program to be executed on a trapped ion setup of the TIQI research group.

Acknowledgments

Foremost I want to thank my supervisor Martin Stadler. During the time of this master thesis and already before during a workshop at the TIQI group he was a reliable guide, who would answer my questions even late in the evenings and give me constructive and much appreciated feedback in our weekly meetings.

I want to thank Professor Jonathan Home and the whole TIQI group for letting me take part in this project and contributing in this exciting research field. Unfortunately I was not able to interact with the group that much due to the nature of my project and the pandemic, which strongly encouraged working from home. Nevertheless you let me feel part of this group especially during the three days of the annual group retreat.

Finally I thank my family particularly my mother Evelyne, father Urs and step-mother Giusi wholeheartedly for their emotional and financial support and all the encouragement through the six years of studying at ETH. And a special thanks to my partner Lisa who was always next to me as a source of strength and inspiration during my master's studies.

Contents

‘

Chapter 1

Introduction

The Trapped Ion Quantum Information (TIQI) group of the 'Eidgenössische Technische Hochschule Zürich' (ETHZ) conducts research on many topics in the experimental and theoretical area of quantum information and quantum physics with trapped ions and related fields. Many projects have the final goal of developing a universal quantum computer on a trapped ion platform. A brief introduction to quantum computing in general and its motivation can be found in section 2.1. A control system is utilised to perform experiments on their multiple, independent trapped ion setups. This control system is called modular advanced control of trapped ions (M-ACTION, more information in section 2.2).

The goal of this master's project was to enable an interface to the M-ACTION system with a quantum programming language (see figure 1.1 for an overview). Qiskit was chosen as this language. An introduction to Qiskit can be found in section 2.3 and a comparison of Qiskit with other quantum languages as well as the reason for choosing Qiskit is given in section 3.1.

An interface between Qiskit and the M-ACTION system is created through a parser that translates a quantum circuit or a pulse schedule described in Qiskit to a JSON-string. This string follows a specific structure, such that it can be read-in by TIQI's control system. Detailed information about the JSON structure can be found in section 4.1 and about the parser in section 4.2.

At the end of the report an overview (see at the start of chapter 5) and an outlook (see section 5.2) on the next steps of this project are given to the reader.

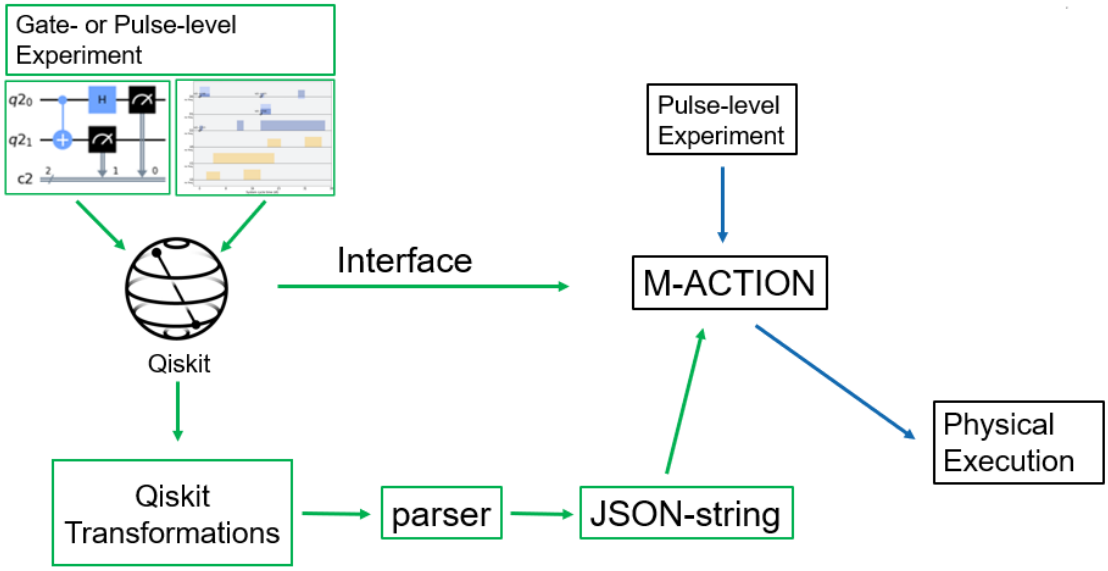


Figure 1.1: This figure shows a broad overview of the contents of this master's thesis (in green) and how it schematically interacts with the current control setup (in blue). With the interface created by this master's thesis, gate- and pulse-level experiments can be created in Qiskit and executed on the M-ACTION control system. This interface consists of the quantum programming language Qiskit and a parser function that translates an experiment defined in Qiskit to a JSON-string following a specific structure. This string can then be read in by the M-ACTION system and the defined experiment can be executed on the physical system.

1.1 Motivation for the Interface

The development of quantum computers is reaching a point where they are not only useful for quantum experiments, but, even though still subject to errors, can be used to run small quantum circuits[i]. The current stage of development is known as the NISQ (Noisy Intermediate Scale Quantum) regime [1].

In order to efficiently define experiments for quantum computing, there must be a transition from a low-level, architecture-dependent control of the experiment to a higher-level, architecture-independent control scheme. The motivation for this transition is that the gate-level description of quantum circuits is platform independent and therefore does

[i] IBM offers public access to their quantum computer through the cloud.

not require any specialised knowledge about the underlying physical system that is used for the quantum computation. At the same time the definition of a quantum circuit is much more concise compared to an equivalent pulse schedule (low-level description). This transition from lower to higher-level access is a natural technological step whenever a system becomes large and robust enough for more complex applications. This master's thesis is one of the first steps to enable a higher-level access to the quantum computers of the TIQI group. The higher-level access in this case is accomplished by the gate-level quantum programming language Qiskit. The interface between Qiskit and the control system of the TIQI quantum computers is created via a parser that translates experiments defined in Qiskit to a data structure (called the JSON structure) that can be directly read into the control system, which then executes the experiment.

This master's thesis is only one of the first steps towards the higher-level control. It enables the interface between Qiskit and the M-ACTION system conceptually, which means that a quantum circuit can be translated to a pulse sequence that can then be executed on the control system. However, this pulse sequence does not correspond to the actual physical implementation of the quantum circuit. To achieve this correspondence, backends must be calibrated, which was beyond the scope of this master's thesis and will be addressed by another master's thesis (more information in section 5.2).

Chapter 2

Fundamentals

This chapter gives a brief introduction of three topics that are important for understanding this master's project. The first section 2.1 covers the basics of quantum computing. Section 2.2 gives an overview of the control system of the TIQI group. The third section 2.3 acts as an introduction to the quantum programming language Qiskit.

2.1 Quantum Computing

The goal of this section is to give the reader a brief background on quantum computing and quantum programming on a conceptual level.

To highlight the difference between a quantum computer and a classical computer, we start by describing the working principles of classical computing. A classical computer stores information in the form of bits. A bit can be in exactly two states: a 0 state and a 1 state. Multiple bits are used together to encode information in a binary encoding. A classical computer acts on bits with classical gates. These gates are the well known logic gates (AND, OR, NOT, XOR, etc.). By applying gates on bits any classical circuit can be described.

A quantum computer on the other hand deals with quantum information, information that obeys the laws of quantum mechanics. This quantum information is stored in qubits (quantum bits). A qubit is a two-level system, a system that has two different energy states: a ground state and an excited state. The quantum nature of these systems allows the qubits to be in superpositions of both states (see subsection 2.1.1). A quantum

computer uses quantum gates (specific quantum operations) to act on the qubits and influence their state. In order to read out a quantum state the quantum information must be projected to classical information. This is achieved by measuring the qubits. A quantum measurement collapses the superpositions of qubits and therefore forces them into one of the two states. These two states are then interpreted as a 0 or a 1 and one bit of classical information can be extracted from them.

2.1.1 Basic Principles

In this subsection the working principles of a quantum computer shall be explained in a bit more detail.

Qubit As mentioned before a qubit is a two-level system. Two-level systems can be approximated by an anharmonic oscillator. Anharmonic oscillators have non-equidistant spacings of energy levels. If only drive frequencies close to a single energy spacing are applied an anharmonic oscillator will behave similar to a two-level system, because its other energy levels are not addressed by this drive. Atoms (and thus also ions) have inherently non-equidistant energy levels. Therefore, they are well suited as qubits. [2]
Mathematically, a qubit can be described using linear algebra. Each state of the qubit can be represented as a basis vector of a two dimensional Hilbertspace. [3]
Usually the two states are represented as the standard basis.

$$|0\rangle \rightarrow \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad |1\rangle \rightarrow \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{2.1}$$

Superposition Superpositions are an inherently quantum mechanical property. Quantum systems are described by wave functions. The square norm of a wave function describes a probability distribution of the possible quantum states. In the case of a qubit there are only two possible states and the wave function of a qubit can be described simply as the superposition of these two states (which are also wave functions). In a

superposition the wave function consist of both states simultaneously.
Mathematically a superposition of a qubit is expressed as a linear combination of these two basis states.

$$|\phi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{2.2}$$

Where the two coefficients are complex numbers and satisfy $|\alpha|^2+|\beta|^2 = 1$. The square norm of these coefficients represent measurement probabilities. Upon measurement of the state $|\phi\rangle$ the probability is $|\alpha|^2$ to measure the state $|0\rangle$, which corresponds to the bit value 0. The measurement collapses the superposition and the qubit ideally stays in the measured state. [3]

Entanglement Entanglement is a special case of a superposition of multiple quantum objects. At least two qubits are required to create an entangled state (more general at least two quantum objects). This state is a superposition of two pure qubit states that upon measurement yields correlated measurement outcomes. While the result of a measurement of a single qubit in an entangled state is still random, the outcome of the second qubit measurement is predetermined by the first result.
Two classical bits can take the following states: 00, 01, 10, 11. A system of two qubits has the states $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$. These four states are basis states of the now four dimensional Hilbertspace. Analogous to the one qubit example a four dimensional vector can be associated with each of these states in the following way:

$$|00\rangle \rightarrow \begin{pmatrix}1\\0\\0\\0\end{pmatrix},\ |01\rangle \rightarrow \begin{pmatrix}0\\1\\0\\0\end{pmatrix},\ |10\rangle \rightarrow \begin{pmatrix}0\\0\\1\\0\end{pmatrix},\ |11\rangle \rightarrow \begin{pmatrix}0\\0\\0\\1\end{pmatrix} \tag{2.3}$$

A superposition in the two qubit case looks like the following equation:

$$|\psi\rangle = \alpha\,|00\rangle + \beta\,|01\rangle + \gamma\,|10\rangle + \delta\,|11\rangle \tag{2.4}$$

The four complex coefficient once again must obey the normalisation condition: $|\alpha|^2 + |\beta|^2 + |\gamma|^2 + |\delta|^2 = 1$. As mentioned earlier, an entangled state is a state in a superposition that yields correlated measurement outcomes. In the example of $|\psi\rangle$ this is the case if, for example, $\alpha = \delta = \frac{1}{\sqrt{2}}$ and $\beta = \gamma = 0$. We can rewrite equation 2.4 as:

$$|\psi'\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle \tag{2.5}$$

A measurement of the first qubit of the entangled state $|\psi'\rangle$ projects the two-qubit system in either the $|00\rangle$ or $|11\rangle$ state depending on whether the result of the measurement is 0 or 1. From the two qubit case one can extrapolate to the n qubit case. A number of n classical bits can have 2^n different states. Analogously, n qubits have 2^n different states that are basis states of a 2^n dimensional Hilbertspace. [3]

As you can see the number of basis states doubles for each qubit, which means that for each additional qubit twice the amount of complex numbers are required to fully represent the quantum state. And multiple bits (depending on the accuracy) are required to represent one complex number. This is a reason why large quantum systems are very hard to simulate on classical computers.

Quantum Gates This paragraph is inspired by and based on reference Williams [4]. A quantum gate is a physical evolution of a qubit quantum system described by the Schrödinger equation. The solution of the Schrödinger equation shows that the time evolution of isolated quantum systems is performed by unitary operators. Quantum gates are therefore unitary operators (that can be represented by unitary matrices) as long as no measurements are conducted and no unwanted interactions with the environment occur. The unitary property implies that these gates are also reversible.

There are single and multi-qubit gates. The action of single qubit gates are explained most easily using the concept of the Bloch sphere. The degrees of freedom of a single qubit state (up to a global phase) can be represented by the surface of a sphere called the Bloch sphere. Any distinguishable qubit state is represented by a point on this sphere.

The action of single qubit gates can be expressed as rotations of the qubit state on this sphere. Therefore, any single qubit gate can be expressed as composition of rotations around the x, y and z axis of this sphere. In figure 2.1 one can see the action of a RY($\frac{\pi}{2}$) gate (rotation around the y-axis by 90°) on the state $|0\rangle$.

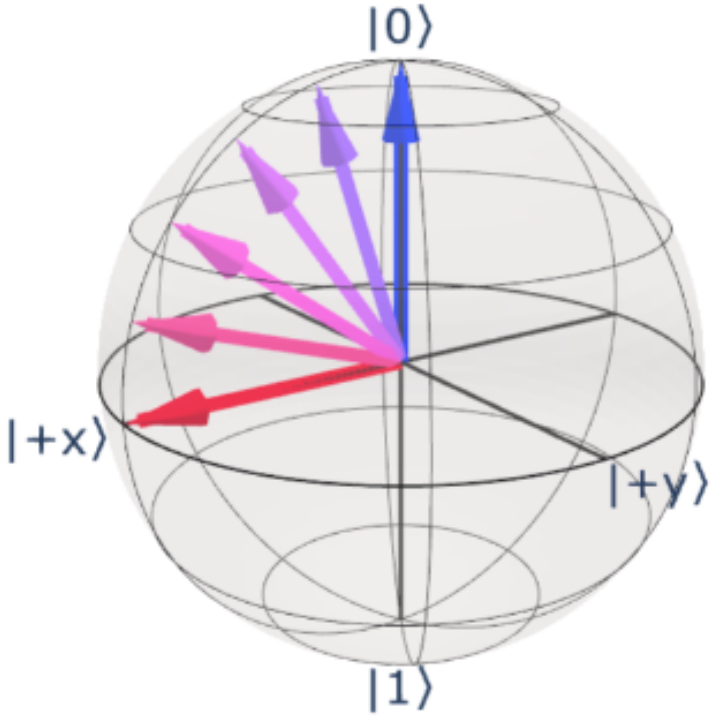


Figure 2.1: Actions of the RY gate visualised on the Bloch sphere. The blue arrow shows the single qubit state $|0\rangle$ and the red arrow the state $|+\rangle = \frac{|0\rangle+|1\rangle}{\sqrt{2}}$. The state of the red arrow is the results of a RY($\frac{\pi}{2}$) gate applied to the state of the blue arrow. The arrows in between are possible intermediate states for the rotation from the blue to the red arrow. This figure was created using the Python packages Qiskit and Kaleidoscope.

Multi-qubit gates can have much more complicated interactions on both qubits. In this brief introduction only two two-qubit gates will be mentioned as examples: the SWAP gate and the CNOT gate. The SWAP gate simply swaps the state of qubit 1 with the state of qubit 2. The action of the SWAP gate on an arbitrary two qubit state, as in equation 2.4, can be understood by replacing the qubit basis states in the following

manner:

$$|00\rangle \rightarrow |00\rangle\,, \quad |01\rangle \rightarrow |10\rangle\,, \quad |10\rangle \rightarrow |01\rangle\,, \quad |11\rangle \rightarrow |11\rangle \tag{2.6}$$

The CNOT gate (controlled-NOT gate) inverts the state of one qubit (target qubit) if and only if the other qubit (control qubit) is in the $|1\rangle$ state. The action of the CNOT on an arbitrary two qubit state, where qubit 1 is the control qubit and qubit 2 is the target qubit, can be understood by replacing the qubit basis states in the following manner:

$$|00\rangle \rightarrow |00\rangle\,, \quad |01\rangle \rightarrow |01\rangle\,, \quad |10\rangle \rightarrow |11\rangle\,, \quad |11\rangle \rightarrow |10\rangle \tag{2.7}$$

Quantum Circuit Quantum algorithms can be described as quantum circuits, which are ordered collections of quantum gates and measurements applied to qubits. A small example of such a circuit can be seen in figure 2.2.

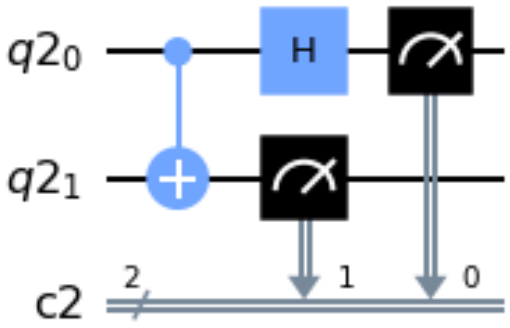


Figure 2.2: An example of a two qubit quantum circuit for visualisation. The circuit contains a CNOT and a Hadamard gate and measurements of the two qubits that are saved in classical registers. This figure was created using the Python package Qiskit.

2.1.2 Motivation

There are many reasons for pursuing this interesting field. Quantum computers have an advantage over classical computer as they can explore a much bigger configuration-space with only a few qubits (because the dimension of the joint qubit state-space grows with 2^n, where n is the number of qubits. This corresponds to 2^n complex numbers, while n bits can only represent a n digit binary number). Quantum computing also offers new types of interactions between the qubits that allow the use of quantum effects, such as interference, to be utilised and grant an enormous speed-up for certain algorithms compared to their fastest classical counterparts. Two prominent examples of quantum computing applications are unstructured search (which makes use of Grover's algorithm) and quantum chemistry applications.

Grover's Algorithm Grover's algorithm enables a polynomial speed up for searching unstructured databases compared to classical algorithms. The algorithm starts with an equal superposition of all possible database entries. It uses a method called 'amplitude amplification', where the probability amplitude of the searched entry is increased. This method is applied repeatedly until a measurement will return the searched entry with a high probability. For a database with N entries, Grover's algorithm requires $O(\sqrt{N})$ steps, where classical algorithms require $O(N)$ steps. This is a quadratic speed up. It can be proven that Grover's algorithm is the best possible quantum search algorithm. For more information check the reference of this paragraph LaPierre [2].

Quantum Chemistry Simulating quantum systems on classical computers poses many different challenges, for example the scaling problem mentioned earlier (storing 2^n complex values for a n qubit system). Quantum computers are naturally more suited for simulating quantum systems and are therefore extremely useful for certain quantum chemistry problems. On quantum computers, wave functions can be encoded in multi-

qubit entangled states, which then can be used to find the eigenstates and eigenenergies of the simulated quantum chemistry system.

Fault tolerant quantum computers required for applications like Grover's algorithm are not available near term, but for quantum chemistry applications there are also algorithms that are designed for NISQ Devices. These NISQ algorithms are defined with the limitations of imperfect, noisy quantum computers in mind and therefore need to leverage the strength of both classical and quantum computing by allocating different task according to their strengths. Such algorithms are called HQC (Hybrid Quantum—Classical algorithms). The VQE (Variational Quantum Eigensolver) is an example of an HQC algorithm and provides approximate solutions to the time-independent Schrödinger equation. The energy estimation of a VQE is performed by a method called 'Hamiltonian averaging'. This technique makes up for the short qubit coherence times by sampling the quantum computer multiple times. For more information check the reference of this paragraph Cao *et al.* [5].

2.2 TIQI Control System

The goal of this section is to give the reader a rough overview of the current control system of the TIQI group and how it is used to create experimental sequences. The information for this section is taken from Vlad Negnevitsky's thesis [6].

2.2.1 Overview

The TIQI control system is called M-ACTION system. It enables trapped ion quantum experiments. These experiments can be understood at the lowest level as specifically configured laser or RF pulse sequences combined with changes in the trapping potentials of the ions. Trapped ion quantum experiments mostly rely on RF (radio frequency) pulses to drive acousto-optic modulators which are used to modify the physical parameters

(frequency, phase and amplitude) of laser pulses. The laser pulse sequences interact with the internal electronic and motional state of the ion and the changes in the trapping potentials are used to control the position of the ions. The electronic components required for these experiments must communicate to a central instance, which is the control system, to send the individual sequences to each hardware component and ensure their synchronized timing. The control system is also capable of feedback, which means that sequences can be adapted in real-time depending on outcomes during the execution of a sequence. During the execution of an experimental sequence all the pulses are physically realised on their hardware components and interact with the trapped ions. This leads to specific quantum mechanical interactions, which can represent, among other things, quantum gates. The M-ACTION system can be structured into three major parts.

1. The first part is a control PC with a GUI (graphical user interface), where experiments are started, modified and monitored.

2. The second part is a SoC (System on Chip) consisting of a dual-core ARM CPU (central processing unit) and an FPGA (field programmable gate array), where experimental sequences are processed, forwarded to the hardware components and controlled in real-time (with feedback).

3. The third part consists of multiple different hardware components that physically execute the experimental sequences, which includes DDS (direct-digital synthesiser) boards for RF signal generation (which are used to modify laser beams or trapping potentials).

The control system is a hybrid CPU- and FPGA-centric design. Together they circumvent the disadvantages of both individual architectures.
A problem with an FPGA-centric design is that experiments which require fast feedback based on more complex algorithms must have this functionality coded into the FPGA firmware. This is inflexible and much more error-prone than coding it into software.

Such functionality is coded into the CPU in the hybrid design, where it can be tested much more quickly and reliably.
An issue with a CPU-based design is that the CPU must remain unburdened to keep the pulse sequencer on the FPGA supplied with instructions, otherwise it can lead to timing errors. This problem becomes worse as the control system is scaled up and more peripherals are added that need to be controlled by the same CPU. In the hybrid design the burden of the CPU is distributed to FPGAs in the peripherals (in this case to the FPGAs on the DDS boards). The design philosophy behind the FPGA architecture is geared towards reducing the real-time communication (by making the individual components more independent) and saving BRAM space (block random access memory on an FPGA), which is a limited resource.

Experimental sequences are written in C++ and the compiled program is loaded into the CPU. It provides instructions and parameters to the FPGA, where the sequences are executed sequentially. It can run a given number of experimental shots independently from the GUI and calculate the occupation probability of the quantum states in the experiment. A newly compiled program must only be loaded infrequently because certain types of values and settings, called remote parameters, can still be controlled via the control PC. With the versatile CPU as control element, feedback and result analysis can be executed in software instead of the much less flexible FPGA.
To reduce the demand on the CPU and prevent consequent timing errors the M-ACTION system reduces its required data throughput by distributing instructions to standalone DDS boards. These boards have their own FPGA, which is utilised for the low-level control of the RF generation, and run independently once a pulse sequence is loaded from the CPU. Only parameter updates and fork path changes are sent from the main CPU in real-time.

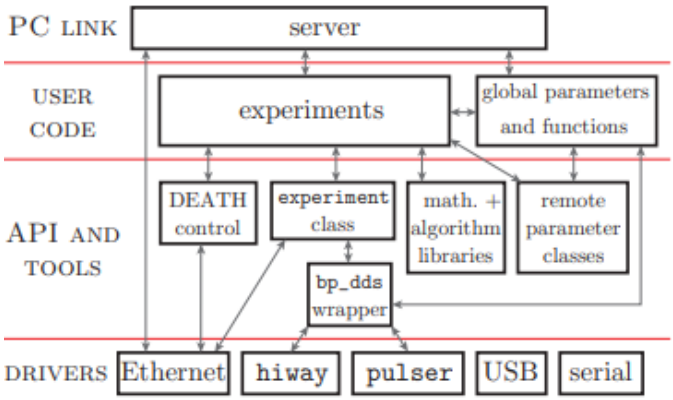


Figure 2.3: A schematic overview of the four abstraction levels of ionpulse mentioned in the continuous text. In the top level the server communicates with the GUI. It handles changes of remote parameters and requests to execute experiments. The experiments are programmed into ionpulse in the user code level. They build up on the experiment class and the M-ACTION API given by the second level. The lowest level contains the hardware drivers. This figure is taken from Negnevitsky [6].

ionpulse Ionpulse is the software written in C++ that runs on the CPU. It is structured hierarchically such that each stage builds on functions and classes from lower modules. There are four different abstraction levels (see figure 2.3). The lowest one consists of the drivers for the physical hardware. Two drivers are the interface between the CPU and the FPGA cores. One of the two generates instructions which will be forwarded through the FPGA to the DDS boards and the other one receives data through the FPGA from PMTs (photo-multiplier-tubes), laser lock status signals and other external triggers and generates digital outputs. There is an Ethernet, USB and serial driver as well. The second level acts as a wrapper for these drivers. It provides a unified API (application programming interface, see subsection 2.2.2) for all subsystems to create RF pulse sequences and also provides a framework for experiments. This framework hides the low-level interactions with the driver, which have to happen at the start and end of an experimental sequence. Through the third abstraction level the different subgroups of the TIQI group individually create their own specialised functions tailored to their

experimental needs. These functions include, among others, cooling, state preparation and qubit gates. Remote parameters are used to hold values which the user wants to change and store in the GUI (like transition frequencies, laser amplitudes and pi-pulse times). The third abstraction level also features the crystal classes. According to Marinelli [7] these classes are a code representation of a trapped ion chain and its objects contain information about the ion chain composition, the motional modes frequencies, and all possible sequences to manipulate a specific ion chain. The fourth level handles the requests from the control PC. For example, to execute experiments and modify remote parameters.

The ionpulse API features an experiment class which links the program on the CPU to the GUI of the control PC. On the control PC each experiment is displayed as a page, where its remote parameters can be modified. An experiment consist of multiple functions, which initialise the system, send the pulse sequences to the devices, run the experiments multiple times to acquire statistics, do read out measurements and also calculate the results from the raw measurement data, which can be user-defined.

2.2.2 Workflow

It will be explained how experiments for the M-ACTION system are created in ionpulse. The fundamental elements of an experimental RF sequence are the following pulse classes: `Edge`, `Cap`, `Shaped` and `Wait`. An `Edge` contains a wait time, after which a change in the RF frequency, phase and/or amplitude is realised. A `Cap` is two `Edges` in a row, where usually the first defines a rising edge of a pulse and the second a falling edge. A `Shaped` pulse is similar to a `Cap`, with the difference that the amplitude does not change abruptly at the start and end of the pulse but is shaped to narrow its frequency spectrum. A `Wait` pulse corresponds to a delay. Objects from these classes contain the `Settings` that they require to run. For an `Edge` these `Settings` consist of a frequency, phase, amplitude

and time parameter. These pulse objects will be physically realised by the DDS boards. Each parameter in a pulse's `Settings` has its own address and multiple pulses can use the same `Settings`. This reuse of values may enable longer sequences to be stored on the DDS boards and also allows modification of multiple pulses at once by changing the referenced parameter. The pulses are arranged in sequences, which can also store subsequences and instructions such as loops and forks. The firmware on the FPGA is capable of branching/jumping between different subsequences, which is required for forks, among other functions.

The M-ACTION system is capable of loops and conditional forks by changing parameter values stored in BRAMs in real time. A loop with iterated values is realised by sending the value of the iterated `Settings` object to the DDS cards FIFOs (first-in first-out buffer). The values from the FIFO are then written to the BRAM address of the iterated pulse object in sync with the loop execution, such that for each iteration the new value from the FIFO is applied. Forks are realised by changing a jump/branch instruction conditioned on an experimental outcome.

Function pulses are another feature on the M-ACTION system. These are pulses that can be reused without the need of sending the complete description of the pulse again to the CPU, because their sequences are saved in the BRAM on the FPGA. An example of a function pulse is the $\frac{\pi}{2}$-X-rotation.

Execution of Experiments Once an experimental sequence is written using the C++ API ionpulse will be recompiled and loaded onto the CPU. Every sequence is checked for validity and memory addresses are assigned to each of them, when the user calls the sequence through the GUI. All the instructions are then transferred to the DDS boards. This process has a duration of hundreds of microseconds up to milliseconds. After the successful transfer the experiment can begin. It should be stressed that the CPU is not processing the experimental sequence in real time, rather it constructs the sequences

which are then executed autonomously on each DDS board with little to no interaction with the CPU during execution.

During the construction of sequences on the CPU an organisation and check routine (named 'Boss') enforces equal pulse times globally on each DDS channel for synchronisation purposes (see figure 2.4). These enforced time slots were no issue for the size of the setup at the time the control system was developed. As up-scaling of quantum computers are a major research goal, these slots have become more and more impractical with the increasing size of the setup (more DDS channels to address more ions/qubits).

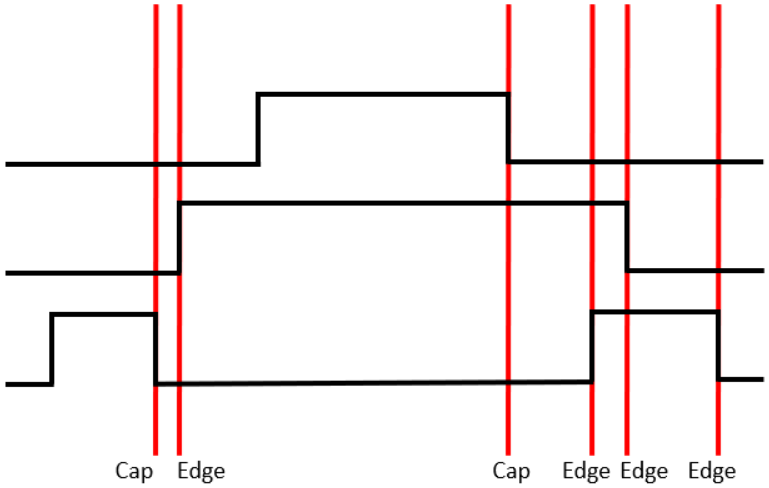


Figure 2.4: A schematic example of an experimental sequence. In this example one can see how the 'Boss' routine enforces time slots on all channels during the construction of sequences (red lines). While on one channel either an `Edge` or a `Cap` is defined a `Wait` instruction is introduced on all other channels. The bottom line describes if an `Edge` or `Cap` instruction is used for each slot.

After completion of the experiments the results and the raw data for each experimental run are returned to the control PC.

Instead of recompiling and reloading ionpulse to the CPU each time a new experiment needs to be executed, this master's thesis simplifies the process. Experiments can now be described in Qiskit Pulse (see subsection 2.3.2) instead of C++ and ionpulse does not

have to be recompiled to use the new experiment. Instead a JSON-string, describing the experiment, can be loaded directly into ionpulse. ionpulse will process the string and internally create the pulse sequences for the experiment circumventing the creation of the global slots depicted in figure 2.4. Additionally, Qiskit provides a direct link to the quantum gate representation of an experiment.

2.3 Qiskit

Qiskit is the open-source quantum software development kit (SDK) of IBM. It is the platform of choice to interface with the TIQI control systems, for many reasons (see section 3.1). Throughout this thesis Qiskit is often called a quantum programming language in the sense that it allows to describe quantum circuits, which can then be transformed to an assembly language.

In this section I want to give a brief overview on the relevant capabilities of the platform. Qiskit is split up in four elements: terra, aer, ignis and aqua. Each element has its own GitHub repository and provides modules from different areas in quantum computing. There are additional repositories beside the four elements, for example Qiskit Finance. One can check out all the repositories on Qiskit's GitHub page. In this project I worked almost exclusively with the qiskit-terra API, since qiskit-terra lays the groundwork of Qiskit. It contains, for example, all the Qiskit Circuit and Qiskit Pulse modules and important functions which assemble quantum objects and transpile quantum circuits. The information about the Qiskit functions and classes was taken from the Qiskit Terra API reference [8]. There is a glossary at the end of this section due to the high number of specific terms. The glossary has links to the sections, where these terms are introduced.

2.3.1 Qiskit Circuit

Qiskit Circuit could be called the main module of Qiskit. It contains all the tools to define and manipulate a quantum circuit (a collection of quantum gates and measurements applied to qubits, see at the end of section 2.1.1).

QuantumCircuit The `QuantumCircuit` class is the main class in the circuit module. It has more than 100 class methods, most of which are gate implementations. They are added to a `QuantumCircuit` instance upon the call of the method. There are also other useful non-gate instructions like `reset()`, which sets a qubit to a give state (1 or 0) and `barrier()`, which ensures, that all the gates before and after the barrier will not overlap after the circuit went through automated optimisation processes.

Qiskit Circuit gives the possibility to create conditional gates. That means gates which are only applied to a specified qubit, if a certain measurement result (stored in a classical register) has a certain value.

An important and useful method of the `QuantumCircuit` class is `add_calibration`. With this function a gate and parameter pair can be mapped to a `Schedule` object (see subsection 2.3.2). This `Schedule` object can be understood as the low-level instructions for the physical realisation of this gate. The mapping is then applied when a circuit is transformed to its pulse representation via the `schedule` function (see subsection 2.3.5). Such mappings can also be created through a `Backend` (see subsection 2.3.3). The difference to the `add_calibration` method is that the latter is more flexible and a specific calibration can be created on the fly for a gate without the need of instantiating a complete new `Backend`.

2.3.2 Qiskit Pulse

The Qiskit Pulse module features tools to represent a pulse schedule. These schedules can be generated from a `QuantumCircuit` object and therefore typically correspond to a

physical implementation of a quantum circuit. The schedules are intended to be played directly on an arbitrary waveform generator (AWG) to physically realise the quantum circuit as RF pulses on superconducting qubit devices (IBM's preferred quantum computing platform). This is not a problem for the purpose of this master's project, as trapped ion quantum experiments also mostly rely on RF pulses (see section 2.2.1). Qiskit Pulse offers therefore a low-level access for defining pulse sequences that can represent quantum circuits.

Schedule The `Schedule` class similar to the `QuantumCircuit` class lies at the heart of Qiskit Pulse. A schedule is a collection of timelines for different types of channels. More information about the different channels will be explained in its own paragraph. All pulse instructions (instructions that add pulses to a `Schedule` object or modify pulses, see in the coming paragraph 'Pulse instructions') need to be added to a `Schedule` or `ScheduleBlock` object (see in the next paragraph). The `Schedule` keeps track of the timing of the instructions (the time when the instructions will be executed and the duration of the instructions), on which channel they are defined and checks that no instructions overlap on the same channel. Figure 2.5 is made using the `Schedule.draw()` method and illustrates an example `Schedule` object.

`Schedule` objects insist on explicit duration information. The explicit duration information is necessary to order the pulse instructions chronologically and ensure that no pulses are overlapping. Parametrising durations is not straightforward. This was problematic for applications in the `Backend` (see 2.3.3) or when `Schedules` are added as calibrations to `QuantumCircuits`. Ideally one would have the possibility to parametrise durations in `Schedules` as well to have maximal freedom of representing the circuit gate parameters in the `Schedules`.

Pulse Builder and ScheduleBlock Qiskit offers a solution to the aforementioned problem with the Pulse `Builder` context and the `ScheduleBlock` class. One can either

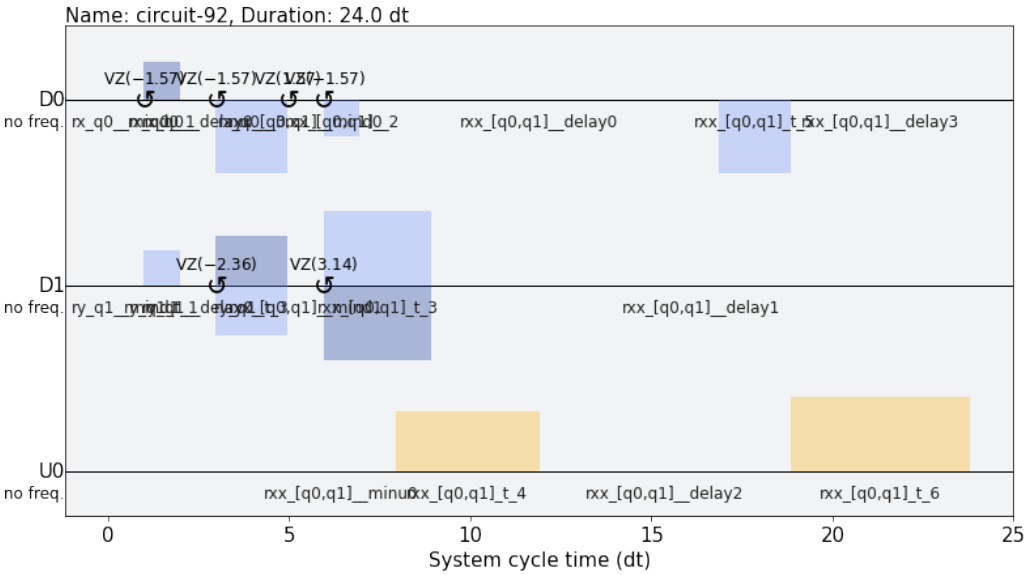


Figure 2.5: This figure shows an illustration of a `Schedule` object. It consists of three individual plots, one for each channel used in the `Schedule`. Each plot depicts pulses with a normalised amplitude (between 0 and 1) on the y-axis and the time in units 'dt' on the x-axis. The light blue colour shows the real part of the pulse and the dark blue the imaginary part. Pulses below the grey line have a negative real/imaginary part. The round arrows show changes in the phase for all future instructions on the corresponding channel. 'dt' stands for the minimum timescale given by the control system. This figure was created using Qiskit.

use the `Builder` context to create a `ScheduleBlock` object or use the `ScheduleBlock` class directly. Recent Qiskit examples promote the use of the first one. `ScheduleBlock` objects are more flexible in the relative order of pulses. A `ScheduleBlock` can define different alignment contexts (for example left aligned, where all pulses are played as soon as possible or right aligned, where all pulses are played as late as possible) and even use pulses with parametrised durations. A parametrised duration of in `ScheduleBlock` can be used to modify the duration of multiple pulses simultaneously and dependent on each other. For example, 'Pulse 1' and 'Pulse 2' may use the same duration parameter. This way one can create a `ScheduleBlock` where 'Pulse 1' has always twice the duration of

'Pulse 2'. Multiple different `Schedules` can be created by binding different values to the parameterized `ScheduleBlock`.
During the writing of my thesis I discovered that `ScheduleBlock` objects can, counter to what I have tried at an earlier point, now be very useful, because there is a way to add them to a `Backend` object (see the 'Commands' paragraph in subsection 2.3.3). Unfortunately, I found out about this too late. For this reason both the `Builder` context and `ScheduleBlock` class are not used in the scope of this master's thesis, rather the focus was put on the `Schedule` class as this is the underlying class that is needed in many different contexts.

Channels There are multiple channel classes for different purposes. A channel takes an index as an argument. The channel type and the index together would uniquely correspond to a physical channel, for example a specific RF source. There are four types of channels: `PulseChannels`, `AcquireChannels`, `MemorySlots` and `RegisterSlots`.

- `PulseChannel` – In an experiment a `PulseChannel` object corresponds to a physical device that can generate a pulse. In superconducting circuit experiments there is typically one channel to address each qubit individually (see `DriveChannel` below). In trapped ion quantum experiments there is usually no one to one mapping between channels and ions. Often multiple ions are addressed by a single laser. This makes the use of these channels, which are designed with a superconducting circuit setup in mind, more challenging for us.
  - `DriveChannel` – `DriveChannel` objects typically have a one to one correspondence to qubits, in a sense, that the index of the `DriveChannel` is the same number as the index of the qubit that this `DriveChannel` is driving. Therefore, the frequencies of `DriveChannel` objects are typically close to the resonance frequency of the qubits they are driving.
  - `ControlChannel` – `ControlChannel` objects are less specific than `DriveChannel`

ones. They represent remaining auxiliary channels, that can be used for example, in multi-qubit gates. Specifically, as these channels are designed for superconducting quantum circuits, these channels are used, among other things, to drive the flux lines that tune the frequency of the superconducting qubits.

  - `MeasureChannel` – These channels are used to readout qubit states, where typically the index of the `MeasureChannel` corresponds to the qubit which is read out.

- `AcquireChannel` – An `AquireChannel` is used for measurements. Different from the `MeasureChannel` it does not drive the qubit to readout, rather it activates the data acquisition process for a certain time window. In the example of a trapped ion experiment, it could activate a PMT and in a superconducting circuit experiment it activates the data acquisition on the readout channels of the circuit, which are only activated for a small time window to minimize interference of background noise.

- `MemorySlot` – A `MemorySlot` is a channel, where a measurement outcome can be saved and readout again at a later time. The outcome is calculated by a kernel and a discriminator specified in an `Acquire` instruction (see in the following paragraph). A kernel is a mathematical function that is used to integrate the raw data of each measurement shot. A discriminator is used to distinguish between possible measurement outcomes (1 or 0 for a single qubit measurement).

- `RegisterSlot` – `RegisterSlot` objects are very similar to `MemorySlot` objects, where the only difference is the timescale they are operating on. While a `MemorySlot` object is used for less time critical data acquisition `RegisterSlot` objects are used when a measurement outcome will be used very soon, eg. for a conditional gate. The distinction between these two slots makes most sense for a superconducting circuit platform, where timing is a much greater constraint than in a trapped ion platform.

Pulse instructions Qiskit Pulse offers multiple different instructions to define `Schedule` or `ScheduleBlock` objects. All these instructions take the `PulseChannel`, on which they act, as an argument. The pulse instructions are:

- `Play` – The `Play` instructions adds a pulse, which is either a waveform (a list of amplitude values saved under a reference in a pulse library) or a `ParametricPulse` (a class instance that defines a waveform, see next paragraph), to a `Schedule`.
- `Delay` – The `Delay` instructions can be used to define gaps between pulses on a channel. These gaps are time intervals in which the amplitude on the channel is zero.
- `ShiftPhase` – The `ShiftPhase` instruction shifts the phase on the channel it is applied to by a given amount. Shifting means that the phase is adjusted relatively to the current phase of a channel.
- `SetPhase` – The phase of a channel is set to the input of the instruction independent of its earlier value.
- `ShiftFrequency` – This instruction is the frequency equivalent to the `ShiftPhase` instruction.
- `SetFrequency` – This instruction is the frequency equivalent of the `SetPhase` instruction.
- `Acquire` – The `Acquire` instruction must be applied to an `AcquireChannel`. It activates the data acquisition on this channel for a given amount of time and stores measurement results in either a `RegisterSlot` or a `MemorySlot`. A kernel and a discriminator can optionally be supplied to calculate the measurement result.
- `Call` – This instruction can be used to call a subroutine (of type `Schedule` or `ScheduleBlock`) in a `Schedule`. This can allow for code reuse in the Qiskit session

and in the hardware if supported. The `Call` instruction is represented as a single instruction in a `Schedule`, but once this `Schedule` is assembled the `Call` instruction is replaced by the assembled contents of its subroutine.

- `Snapshot` – The `Snapshot` instruction does not have a physical meaning. It is used to collect a snapshot of, for example, a quantum state at a certain point in time in a `Schedule` during a simulation.

It is important to note that all phase and frequency changes affect channels and not pulses.

Parametric Pulses `ParametricPulse` is a base class in Qiskit for multiple classes, which can be used together with the `Play` instruction to add the corresponding parametrised pulse shapes to a `Schedule`. The parametric pulse library contains the following `ParametricPulse` types: `Constant` (a pulse with a constant amplitude), `Drag` (derivative removal by adiabatic gate, see in reference Gambetta *et al.* [9]), `Gaussian` and `GaussianSquare` (a `Constant` pulse with a gaussian shaped rising and falling edge). For example, a `Constant` object can be used together with a channel as arguments for the `Play` instruction.

2.3.3 Backend

The `Backend` is an object in Qiskit that stores important information about the physical setup. Important attributes are the configurations and the pulse defaults, which are elaborated in the paragraphs below. The `Backend` is technically not necessary at all, but having all the configurations in one place and easily accessible makes function calls much cleaner. For example it reduces the arguments of the `assemble` function from thirteen to just two. In other words, all functions that use the `Backend` could also be executed by supplying all the necessary members from the `Backend` directly.

Configurations The configuration part of the `Backend` (given by the `PulseBackendConfiguration` class) stores many values that don't change often. For example information about the channels of the hardware, the native gate set of the quantum computer, how many qubits the quantum computer has access to etc. The `Backend` configurations are needed, for example, in the `transpile` function and the `assemble` function (see subsection 2.3.5). This part of the `Backend` is needed for both the Circuit and the Pulse side of Qiskit.

Pulse Defaults The pulse defaults (given by the `PulseDefaults` class) define default settings, which are needed to translate a `QuantumCircuit` object to a `Schedule`. These are: the estimated qubit frequency, the estimated measurement frequency and the pulse library, where custom wave forms are saved that can not be defined via `ParametricPulse`s and the `Command` definitions.

Commands To translate Qiskit Circuit instructions to Qiskit Pulse, `Command` objects can be defined in the `Backend`. These `Command`s will each be transformed to a single entry in a map (called the `instruction_schedule_map`) that maps circuit instructions applied to a specific qubit to its pulse `Schedule`. In this thesis the term `Command` is used when the `Command` class or an object of that class is meant. On the other hand, the term 'command' is used when an entry of the `instruction_schedule_map` is meant, since the entries of this map can be created by `Command` objects. A `Command` (generally also a command) is defined by the circuit instruction name it is replacing, the qubits this instruction is acting on, and the `Schedule` that the instruction will be translated to. The `Schedule` of a `Command` is given as a sequence of `PulseQobjInstructions` (see subsection 2.3.4). These objects are initialised with all the information of the pulse instructions they will represent plus their channel and their start time relative to the beginning of the `Schedule`. During the initialisation of the `Backend` instance, the sequence of `PulseQobjInstructions` is converted to their pulse instruction equivalent. These

instructions with their channel and time arguments are then combined to a `Schedule` object[i]. It is possible to have parametrised entries in the command `Schedule`. These entries can either be filled in directly from the circuit instruction in the scheduling process (see subsection 2.3.5), in the case of phase arguments, or after the translation of the `QuantumCircuit` to the `Schedule`. The usage of frequency and phase arguments in `Commands` is straight forward. Parametrised pulse amplitudes can be used in the command `Schedule` as well, but it has to be done in a specific way. In the examples of `Backends` that I saw, `PulseQobjInstructions` were initialised by their `from_dict()` method, where a Python dictionary can be handed to the function and the concrete object is directly created. The problem with this implementation is that the amplitude argument of a `ParametricPulse` will run through a check that only allows numerical values and no parameters. This seems like an unnecessary restriction. Therefore, I created a Github issue to address this problem. From the answer in this issue I learned that using `Commands` is an old way to create the `instruction_schedule_map`. The new way seems to be developed around the time I was researching how to create a functioning `Backend`. This is a great example to show that Qiskit is very actively developed. The new way to create entries in the `instruction_schedule_map` is by using the `backend.defaults().instruction_schedule_map.add()` method once a `Backend` is initialized. This way even `ScheduleBlock` objects (see the paragraph 'Pulse Builder and ScheduleBlock' in subsection 2.3.2) can be used in these mappings, which means that also parameterized durations are allowed and `Commands`, which inherently rely on the `Schedule` representation, which does not allow parametrised durations, seem obsolete. This is great news for the future of this project. Unfortunately, I learned about it too late to make the necessary adaptions in the `Backend`. Therefore, the `Command` class is still very prominent in this project.

[i] Instead of mentioning the `PulseQobjInstruction` sequence of the `Command` it will be called command `Schedule` in the rest of this thesis, because this is simpler to understand and is what the `PulseQobjInstruction` sequence will be transformed into.

2.3.4 Quantum Objects

Quantum objects (`Qobj`) are Qiskits approach for standardised representations of actions on quantum hardware such as gates on the quantum circuit level and pulses on the pulse level. Quantum objects exist for the assembly languages OpenQASM (open quantum assembly language) and OpenPulse. OpenQASM defines a standardised gate-level description of quantum circuits in a JSON structure while OpenPulse does the same for quantum experiment with pulse level control. These two JSON structures are explained in the reference McKay *et al.* [10].

QasmQobj The QASM quantum object is created by applying the `assemble` function together with a `Backend` on a `QuantumCircuit` object. It contains the whole description of the experiment as a sequence of gates in a standardised structure.

PulseQobj The pulse quantum object (`PulseQobj`) is created by applying the same `assembly` function together with a `Backend` on a `Schedule`. It contains a sequence of `PulseQobjInstruction`s. The parser in this master's project will translate the `PulseQobj` of a `Schedule` to a representation, which is straight forward to read in and execute by the TIQI control system (specifically ionpulse).

PulseQobjInstruction It is a generic class for all kinds of pulse instructions. Its attributes are restricted to a set of quantities, but using these allows many different kind of instructions to be represented.

2.3.5 Important Functions

Some Qiskit functions are very important and their task in the Qiskit workflow is explained here briefly.

transpile The `transpile` function takes a `QuantumCircuit` object and the `Backend` as an input and translates all the quantum gates used in the circuit to an equivalent representation using only gates from the native gate set defined in the `Backend`. The transpilation process can be customized with many different optional inputs to optimize the resulting `QuantumCircuit`. For example different optimisation levels can be applied to reduce the complexity (the gate count) of the `QuantumCircuit` object. All other `transpile()` specifications can be looked up on the Qiskit documentation.

assemble The `assemble` function can be applied to either a `QuantumCircuit` or a `Schedule` object. It will translate these complex objects to the simplified quantum objects, where only the necessary information is kept. The `assemble` function makes use of the information in the `Backend` for this process. In the case of `QuantumCircuit` objects, a `QasmQobj` is created. In the Pulse case, there are specific converters for each instruction that define how the information is transferred from the instructions to the `PulseQobj`s (which correspond to a description of the pulse sequence in the OpenPulse structure). These converters are implemented in such a way that they can be easily overridden from outside the Qiskit module. This allows for a customised translation of the instructions if necessary (see the paragraph 'Converter Overwrite and Command labels' in subsection 3.2.1).

schedule The `schedule` function bridges the Qiskit Circuit and Pulse representations. The function takes a `QuantumCircuit` and a `Backend` as an input and applies the gate calibrations stored in the `QuantumCircuit` object (if any) to translate the gates to their corresponding `Schedules`. If no calibrations were added to the `QuantumCircuit` the commands in the `Backend` are used for this transcription process. The high-level quantum circuit can this way be transformed to a low-level pulse schedule.

pad Padding is a process where all gaps in a `Schedule` are filled with `Delay` objects of the same duration. The Qiskit `pad` function does exactly that and is useful for parsing purposes.

2.3.6 Qiskit Workflow

After explaining the use of many different Qiskit classes and functions what follows now is an overview of how these objects may be applied to end up with an executable experiment. To create an executable quantum experiment on the TIQI control system one needs a `PulseQobj`. There are three different ways to create one.

1. The user can start with the description of a quantum circuit purely in Qiskit Circuit. The next step is to transpile it via a `Backend` to a `QuantumCircuit` object that only contains native gates and then scheduling it to the Pulse representation. The resulting `Schedule` is then assembled to the `PulseQobj`. This way, the user may define an experiment utilising only a gate-level description without needing any knowledge of pulse-level instructions.

2. Alternatively the user can describe a `Schedule` solely by pulse instructions. Once the `Schedule` is finished it can then be assembled to the `PulseQobj`. This way the user has maximum freedom to define his experiment, but requires an in depth understanding of the low-level control.

3. Instead of directly assembling the scheduled `QuantumCircuit` in the first example the user can adapt the experimental `Schedule` by adding pulse instructions before and after the scheduled `QuantumCircuit` and assemble it then. This way the user can take advantage of predefined gates but still has the freedom to make custom adaptations.

A parser which was written as part of this master's project will then take the `PulseQobj` and transcribe it to a JSON structure that can be used to instantiate the corresponding

objects in ionpulse dynamically which then can be executed on the hardware.

2.4 Qiskit Terms

assemble The Qiskit `assemble` function is introduced in this paragraph. 21, 28, 29, 31–33

Backend The Qiskit `Backend` class is introduced in this paragraph. 22, 23, 28–33

Command The Qiskit `Command` class is introduced in this paragraph. 29, 30

instruction_schedule_map The Qiskit `instruction_schedule_map` class is introduced in this paragraph. 29, 30

ParametricPulse The Qiskit `ParametricPulse` class is introduced in this paragraph. 27–30

PulseQobj The Qiskit `PulseQobj` class is introduced in this paragraph. 31–33

QuantumCircuit The Qiskit `QuantumCircuit` class is introduced in this paragraph. 22, 23, 29–33

Schedule The Qiskit `Schedule` class is introduced in this paragraph. 22–25, 27–33

schedule The Qiskit `schedule()` function is introduced in this paragraph. 22, 32, 33

ScheduleBlock The Qiskit `ScheduleBlock` class is introduced in this paragraph. 23–25, 27, 30

transpile The Qiskit `transpile` function is introduced in this paragraph. 21, 29, 32, 33

Chapter 3

Quantum Programming Language

This chapter is focused on quantum programming languages in the perspective of ion trapping. This includes explaining why Qiskit was chosen as the language of this project (see section 3.1) and describing the work on and with Qiskit (see section 3.2).

3.1 Choosing a Quantum Programming Language

The choice of the quantum language is important for multiple upcoming projects in the TIQI group which will build on this work. The choice is also significant for the interface between this language and the control system. First of all, the specialised vocabulary shall be clarified. The following direct quotes stem from reference LaRose [11] (2019).

Vocabulary regarding quantum programming languages

Quantum assembly/instruction language – such a language "instructs the quantum computer which physical gates to implement on which qubits". (p. 5)

Quantum programming language – allows "to manipulate quantum (assembly) languages in a more natural and readable way for programmers ". (p. 2)

Quantum software platform – stands for a "collection of a quantum programming language with other tools, such as compilers and simulators". (p. 2)

gate-level quantum software platform – stands for a quantum software platform, which is "designed around the circuit (gate) model of quantum computing". (p. 2, 4)

transpile/transpiler – refers to rewriting a given quantum circuit to a new universal set of gates and often times optimising the circuit in the process. This is commonly used to

translate an arbitrary circuit to a platform specific set of basis gates.

Many different languages were evaluated qualitatively after the following criteria:

- **Compiling and Optimisation** – The quantum programming language must be able to compile quantum code to a general format and ideally offer optional optimisations of the defined quantum circuit. This sort of compilation is necessary and therefore a hard criterion for the evaluation.

- **Open-source** – The quantum programming platform must be open-source. This is a hard criterion, because access to the source-code is necessary to directly interface with the language and possibly extend needed functionality. If a problem arises while interfacing with an open-source quantum language the problem can be examined thoroughly by looking through the source code of the language, check what is causing the problem and maybe circumvent it this way. And to extend the needed functionalities of a language it makes sense to use existing classes in the language as an orientation to create the extensions compatible with the existing framework. Additionally, it ensures long-term availability free of charge.

- **Abstraction level** – On which level does this programming language interact with qubits? The preferred interaction level is low, since in the NISQ regime only few qubits are available. One must be able to access them directly to be able to optimise processes and use the limited resources most efficiently. For this reason, quantum software platforms which support gate-level manipulation are preferred.

- **Documentation** – How well documented is the programming language? A detailed documentation is preferred, as group members will need to learn the language. This is easier if the language is well documented.

- **Community/Activity** – How well known is the programming language and how

active is its community? Also how actively is the language developed? Programming languages which are well known and serve an active community are preferred, because this increases the likelihood that the development of the language will continue in the future and that forums for exchange, discussions and questions exist.

- **Additional functionalities** – Does this programming language offer useful extras such as simulation or visualisation tools? These can be considered as upsides.

The GitHub page in reference Fingerhuth *et al.* [12] acts as a great overview for existing open-source quantum programming languages. From the following two sources, LaRose [11] and Heim *et al.* [13], information about some handpicked quantum programming languages was acquired. This selection was extended by a few choices from the aforementioned GitHub page. A complete list of which quantum programming languages were inspected (with varying depth) follows:

- **Braket – Amazon:**

  > "The Amazon Braket Python SDK is an open source library to design and build quantum circuits, submit them to Amazon Braket devices as quantum tasks, and monitor their execution."
  >
  > (Braket Documentation, last accessed 25.11.2021)

  Braket is an open-source gate-level quantum programming platform programmed in Python. It provides a documentation, a GitHub page with examples, simulators, and access to real quantum computers through the amazon web service[i].

- **Cirq – Google:**

  > "Cirq is a quantum programming library for Python with a strong focus on supporting near-term quantum hardware."
  >
  > (Heim *et al.* [13], 2020, p. 717)

[i] Braket features. Accessed July 1, 2021

Cirq is an open-source gate-level quantum platform programmed in Python. It provides a documentation, access to quantum computers of AQT, a compiler and according to Fingerhuth *et al.* [14] a simulator and a circuit drawer.

- **Forest – Rigetti:**

  > "Forest is a quantum software platform developed by Rigetti which includes pyQuil, an open-source quantum programming language embedded in the classical host language Python, for constructing, analyzing, and running quantum programs."
  >
  > (LaRose [11], 2019, p. 4)

  Forest is an open-source gate-level quantum software platform. It features a detailed documentation with examples, tutorials, a community slack channel, a simulator, access to a real quantum computer, and a compiler.

- **OpenQL – QuTech TU Delft:**

  > "OpenQL (is) an open-source high-level quantum programming framework. OpenQL is mainly composed of a quantum programming interface for implementing quantum algorithms independently from the target platform, and a compiler which can compile the algorithm into executable code for various target platforms"
  >
  > (Khammassi *et al.* [15], 2020, p. 1)

  OpenQL is programmed in Python and C++. It provides a detailed documentation, a compiler which uses the quantum assembly language cQASM (common QASM), a simulator and a instruction timing diagram generator.

- **ProjectQ – ETHZ:**

  > "We introduce ProjectQ, an open source software effort for quantum computing ... (featuring) a compiler framework capable of targeting various types of hardware, a high-performance simulator with emulation capabilities, and compiler plug-ins for circuit drawing and resource estimation."
  >
  > (Steiger *et al.* [16], 2018, p. 1)

  Also ProjectQ is an open-source gate-level quantum software platform written in Python. It provides a documentation with examples, a simulator, access to IBM's quantum computers, and a circuit drawer.

- **Qiskit – IBM:**

  > "Qiskit provides a Python-based programming environment that allows one to generate and manipulate OpenQASM programs. It provides powerful abstraction capabilities, such as the ability to synthesize gate decompositions for arbitrary isometries and certain unitary transformations"
  >
  > (Heim *et al.* [13], 2020, p. 717)

  Qiskit is as well an open-source gate-level quantum software platform programmed in Python, but also usable in JavaScript and Swift. Similar to Forest, it features a thorough documentation, tutorials, a community slack channel, simulators, access to real quantum computers (IBM Quantum), a compiler which uses the quantum assembly language OpenQASM (open quantum assembly language), and a circuit drawer to visualise programmed quantum circuits.

- **Quantum Development Kit (QDK) – Microsoft:**

  > "We present Q#, a scalable, high-level, domain-specific language for quantum programming ... offering a high level of abstraction"
  >
  > (Svore *et al.* [17], 2018, p. 10)

  QDK is an open-source gate-level quantum software platform. It uses the quantum programming language Q# written in C#, which is designed for higher abstraction. It provides a documentation, and a simulator.

- **Quipper – Artiste-qb:**

  > "It (Quipper) addresses the problem of describing quantum computations at a practical scale, and demonstrated by describing quantum circuit representations with up to trillions of quantum gates."
  >
  > (Heim *et al.* [13], 2020, p. 718)

  Quipper is a quantum programming language embedded into Haskell.

- **Scaffold/ScaffCC – Princeton:**

  > "Scaffold is designed for expressing quantum algorithms in a high-level format that can be compiled into low-level implementations whose properties can be studied."
  >
  > (Heim *et al.* [13], 2020, p. 719)

  Scaffold is a standalone quantum programming language which makes use of the open-source compiler ScaffCC to translate the instruction to QASM. One can find a user manual and code samples on the ScaffCC Github page.

- **Silq – EHTZ:**

  "Silq, a high-level quantum language enabling safe, automatic uncomputation."

  (Bichsel *et al.* [18], 2020, p. 287)

  Silq is a high-level quantum programming language, which does not use the context of any base programming language. It provides a documentation, and a proof-of-concept simulator.

Now that the evaluated quantum programming languages were briefly introduced, some can already be discarded.
As already mentioned in the direct quotations above, QDK and Silq are designed towards higher level programming, which is not so relevant on NISQ devices in the near future. Even though Q#, the quantum programming language of QDK, features a gate-level representation, according to reference Heim *et al.* [13], it is still targeted at enabling large-scale algorithms on future quantum computers.
Quipper falls into the same category. It addresses the problem of quantum computation at a, from the current context, impractical and immense scale ('trillions of gates', Heim *et al.* [13], 2020, p. 718), which is not applicable in the near future. Also according to Fingerhuth *et al.* [14], Quipper is no longer actively developed. This rules it out completely.

This leaves seven suitable quantum programming languages (Braket, Cirq, Forest, OpenQL, ProjectQ, Qiskit, Scaffold). The direct comparison makes apparent that these languages differ quite a bit in the community criterion and in the documentation. ProjectQ and Braket lack a detailed documentation with examples and tutorials and even though Scaffold features a manual and some example scripts, it is in my opinion clearly

inferior compared to the excellent documentation of the remaining languages.
To find the best suited language out of the final four, one can compare them by development activity and community activity. To quantify this aspect, the number of commits during last year on their GitHub pages are compared.

activity comparison	pyQuil (Forest)	Qiskit Terra	Cirq	OpenQL
total number commits	1029	**5709**	2327	2934
total number commits in last year	45	**1047**	649	863
number of contributors	81	**304**	135	23
ratio of open issues to closed issues	0.30	**0.21**	0.31	0.25

Table 3.1: Comparison of remaining four suitable languages in terms of activity and community. Total number of commits last year is in the interval July 5. 2020 – July 04. 2021. All the data is collected from the corresponding GitHub pages, which are linked in the names of the languages in this table. Out of all the different Qiskit projects, Terra is the most relevant regarding this master's thesis, therefore the information regarding Terra is listed. Qiskit Terra excels in all four disciplines.

Tabular 3.1 reveals, that Qiskit is the most actively developed quantum programming language out of the remaining four. Reference LaRose [11] documents that IBM was very successful in building an active community of student and researchers which use Qiskit. The reference also stresses how Qiskit stands out with its large numbers of tutorial notebooks on a broad rage of topics including even didactic quantum games.

To summarise, Qiskit is the clear winner out of the initial set of languages as it more than satisfies all of the previously described criteria.
Qiskit is even more convincing, when we take additional features into account, in particular Qiskit Pulse. Qiskit Pulse is an extension of Qiskit, which allows compiling a quantum circuit not only to the QASM assembly language, but directly to a pulse schedule. These pulse schedules are intended to be easily mappable to the arbitrary waveform generators of a superconducting circuit control system. Even though the TIQI group doesn't work with superconducting circuits, an experiment on a trapped ion setup is still defined through a schedule of RF and transistor-transistor logic (TTL) pulses on many

channels, which lines up almost perfectly with the capabilities of Qiskit Pulse.
Choosing Qiskit as a quantum programming language for this project and for the TIQI group seems justified. Qiskit not only excels in the key requirements, but offers even more, like the Qiskit Pulse functionality.
In this thesis Qiskit will be called a quantum programming language for readability. It may be a bit imprecise as Qiskit is rather a gate-level quantum software platform or a 'SDK (software development kit) for working with quantum computers' as it is called on the Qiskit website (last accessed 03.12.2021).

3.2 Qiskit Configurations and Extensions

This section is focused on Qiskit configurations and extensions to fit the needs TIQI group and enable the full potential of ionpulse through Qiskit. The configurations include mostly a usable `Backend` with `Command` instances (see subsection 3.2.1). The Qiskit extensions signify additional pulse instructions (called extension instructions, see subsection 3.2.1) which enable all the Qiskit functionalities and additional parametric pulse types (see subsection 3.2.3).

3.2.1 Backend

What the `Backend` is and how it is used is explained in section 2.3.3. Here, the focus lies on how the `Backend` was configured and extended for the purpose of this project. Most specific configurations, for example all the commands, are only for demonstration and testing purposes and do not have an explicit physical meaning.
In subsection 2.3.3 in the 'Commands' paragraph was mentioned that a better way to add entries to the `instruction_schedule_map` was developed, which maps circuit instructions to pulse schedules in the `Backend`. This makes the `Command` class practically obsolete. Because this was discovered only during the writing process of the thesis,

the project still makes heavily use of the `Command` class and word 'command'. Not everything regarding commands is obsolete, only the way to add them to the `Backend`. A lot what is written in this thesis about commands still applies to the entries of the `instruction_schedule_map` (which what is referred to when 'command' instead of `Command` is used).

Configuration The `Backend` configuration dictionary stores a lot of different information. The most notable entries are:

- the number of qubits. In the example `Backend` the number of qubits was chosen to be three.
- the basis gates and their QASM definition, which describes the effect of the gate on its qubits in terms of CNOT gates and single qubit rotations. The QASM definition is necessary for the Qiskit simulators and `transpile` function to understand the effect of a potentially arbitrary gate in the basis gate set. The `Backend` in this project uses the following operations: Rotations of arbitrary angles around the x- and y-axis (RX and RY gates), plus a two-qubit rotation of an arbitrary angle for qubit pairs (RXX/MS gate).
- the available channels, their purpose and their default frequency. The `Backend` created for this project uses three drive channels and also three measure channels, one for each qubit, six control channels and two acquire channels.

Gate Commands Each of the three basis gates in the `Backend` configuration require a command (see the 'Command' paragraph in subsection 2.3.3) for each different set of qubits that it can be applied to. These commands enable the translation of a `QuantumCircuit` object to a `Schedule` object. Therefore, three commands for the RX gate (one for each qubit), three for the RY gate and up to six for the two qubit gate RXX, but since this gate is symmetric on both qubits (Sørensen & Mølmer [19]), only

three commands are required. So together there are nine gate `Command` examples stored in the `Backend`.

Non-gate Commands Commands in the `Backend` can also include non-gate instructions. The most common one would be a `measure` instruction for each set of qubits that are measured simultaneously. Other instructions can be defined and added as a command as well. These commands could describe, for example, an initialization sequence for qubits or any other non-gate instruction. In this project there are two additional non-gate command examples. One is a simple `cooling` instruction, the other is called `test-command`. The `cooling` command has a circuit instruction counterpart. This enables it to be added to a `QuantumCircuit` object. When this circuit is transformed to a `Schedule` the `cooling` `Schedule` (that is defined in the `Command` as a sequence of `PulseQobjInstructions`) will be inserted in place of the `cooling` instruction analogously to how gates are replaced. The `test-command` does not have a circuit counterpart as it was only used for testing the Qiskit extensions, `Loop`, `Fork` and `Sync`, in the `Command` framework and to parse them inside `Command`s. One can call a command that does not have a circuit representation directly from the `Backend` instance to utilize the its `Schedule` as a template and customize it by binding parameters.
Because the `cooling` command has a circuit counterpart, `cooling` must be added to the basis gate set in the `Backend`. Otherwise the `transpile` function will raise an error, as this instruction would be unknown by the `Backend`, even though `cooling` is not a gate at all. The `measure` instruction is an exception and does not need to be declared in the basis gates. The `test-command` is not needed to be added to the basis gates, since it does not have a corresponding circuit instruction. Therefore it will never run into the transpilation problem.
Following the `cooling` example any kind of useful `Schedule`s can be added to the `Backend` and then used already in `QuantumCircuit` objects. These self-made instructions are

opaque to the Qiskit Circuit framework, which means that they will not interact with in simulations or circuit optimisations.

Command instruction and custom instruction In this paragraph I want to explain the usage of word 'custom' paired with the words 'instruction', 'pulse' or similar. Usually custom in this case means manually added to a `Schedule` object. It is therefore the exact opposite of a command instruction, which would be an instruction that is part of a command `Schedule` (sequence of command instructions), because command instructions are usually not added to a `Schedule` rather a `Schedule` made out of command instructions is created at once by applying the `schedule()` function to a `QuantumCircuit` object. If this `Schedule` object is then manually customized with pulse instructions these instructions are called 'custom instructions'.

Usage of Commands in the JSON structure Command instructions ought to stay grouped in the JSON-string (that communicates a Qiskit `Schedule` to ionpulse). A particular command `Schedule` has a one to one correspondence to a circuit instruction plus its parameters. Therefore, if in one experiment a circuit instruction with the same parameters is used multiple times and the corresponding command `Schedule` is grouped as one cohesive object this particular command `Schedule` needs to be represented only once in the JSON-string but can be referenced any number of times. By preserving these common (command) instruction blocks space in the DDS cards BRAMs can be saved, which allows for longer experimental sequences.

Command labels One problem with trying to maintain command `Schedule`s as one cohesive object is that a `Schedule` object is simply a time-ordered list of pulse instructions (and a `PulseQobj` is a time-ordered list of `PulseQobjInstruction`s). This means that a `Schedule` object has no record that it contains instructions that stem from a command

`Schedule`. For this reason labels are added to command instructions to mark them as being part of a command. These labels also contain information about which command the instruction is part of and more. By looking for these labels, a parser function can distinguish the command instructions from custom instructions and group command instructions which stem from the same command. These labels are formatted in the following way: command-type_qubits_parametrised-variable_instruction-number.

- The first section of the label specifies the command type by name, for example 'rxx'.
- The qubits are indicated with a 'q' followed by the qubit index. For multi-qubit commands the qubits are enclosed by square brackets and separated via commas.
- Commands may contain parametrised instructions. If they do, the type of the parametrised variable is indicated by one of these letters: 'f', 'p', 'a' and 't'. They stand for parametrised frequency, phase, amplitude and duration. This indication is used (by the parser) to read out the value of the parametrised variable. For example, if a circuit instruction applies a frequency parameter to a pulse instruction in the command `Schedule`, the modified pulse instruction will have an 'f' in its label. Due to this label the parser will read out the frequency value of the instruction and insert it in the reference/name of the corresponding command sequence in the JSON-string. The parameter binding must be saved in the reference because parameters influence the `Schedule` of a command. Then, if the same command is parsed multiple times with the same parameters it is only saved once in the JSON-string. On the other hand, if the command is parsed multiple times, each time with different parameters, each command will be represented individually in the JSON-string with unique references. The references are unique because they contain the applied parameters.
- The last section of the label is the instruction number. The enumeration starts

with zero.

Here is an example of a command label: 'rxx_[q1,q2]_p_1'.

Converter override Qiskit Pulse utilises converter functions to transform between pulse instructions and `PulseQobjInstructions` (which is used in the parsing process). These converters define which parameters are extracted from the pulse instructions and put into the `PulseQobjInstruction` dictionary and vice versa. Unfortunately, instruction labels are not preserved during the transcription process. This means the command marker would be lost in the conversion. Fortunately, Qiskit features a rather simple way to override these converters. The override functionality was used in this project to enable instruction labels to be conserved through these conversions.

PaddedCommand The `PaddedCommand` class replaces the `Command` class for the definition of commands in the `Backend`. `PaddedCommand` instances fulfill the same role as `Command` instances with four differences:

1. The `Schedule` of a `PaddedCommand` will be automatically padded. 'to pad' means filling the gap intervals in a Schedule with delays (see subsection 2.3.4). `Delay` instructions are added to the end of channels as well, such that the sequences of all channels have the same duration. This creates a `Schedule` which can be more easily translated to instructions for ionpulse.

2. Consecutive delays on the same channels are combined to one long delay.

3. The command labels are checked whether they are consistent with the label format.

4. The durations of all instructions are verified to be longer than the minimum wait time, which is given by restrictions of ionpulse. No parameter change on a channel can be applied with zero delay. Therefore, a minimum wait time is defined.

dd_instruction Multiple different approaches were tried to enable parametrisable durations for instructions in `Commands`. In order to enable some way of duration parametrisation, and allow `Commands` that have duration dependent arguments, a workaround was developed. The function `dd_instruction` (duration dependent instruction) from this project takes a command from the `Backend` as a template, adapts the duration of the specified instructions and submits the adapted `Schedule` to the `QuantumCircuit` as a calibration of a corresponding gate. This has some limitations: Because only durations are adapted, the start time of the pulses are fixed. This defines a maximum duration bound for pulses, which are followed by other pulses on the same channel. This method was developed before I learned that duration-parametrisable `ScheduleBlock` objects can be added to the `Backend` instead of `Schedules`, which makes `dd_instructions` obsolete.

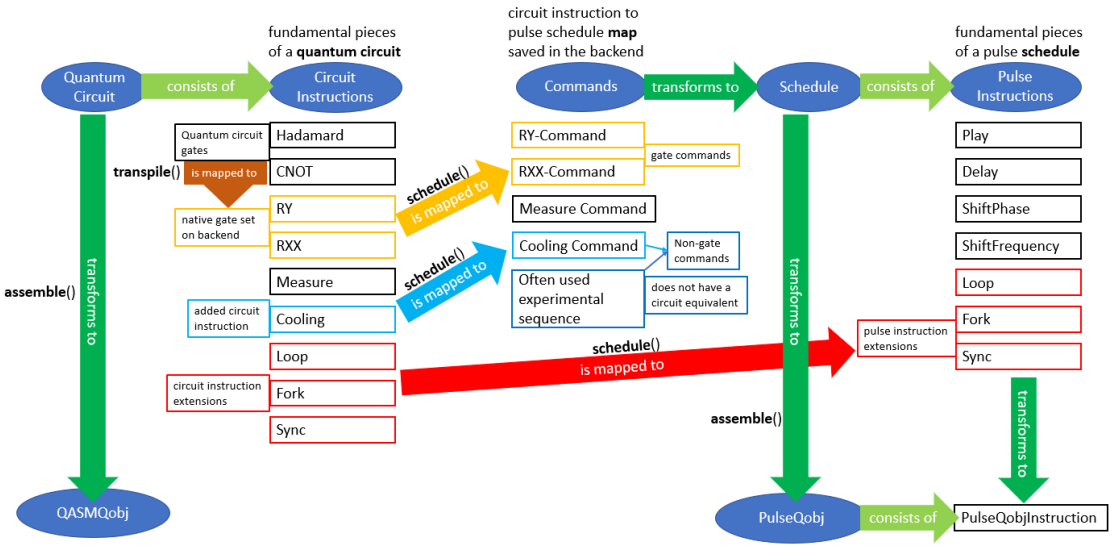


Figure 3.1: Overview of the different instruction and command types and their relations. The three important functions `transpile()`, `schedule()` and `assemble()` are shown as well. The names in the blue ellipses represent important classes. It is important to notice that the `transpile` and `schedule` function take `QuantumCircuit` objects as input and not circuit instructions as the origins of the 'is mapped to' arrows might suggest, but because different circuit instructions inside a `QuantumCircuit` are mapped to different objects this representation was chosen.

3.2.2 Extension Instructions

Qiskit already offers a lot of different tools and instructions, but to represent all the functionality of ionpulse some additional instructions had to be developed, namely the `Loop`, `Fork` and `Sync` instructions.

After trying multiple different approaches and looking through the Qiskit source-code and the internet for examples, an issue on the qiskit-terra Github page was found, which covers integration of third party instructions. Using the information from the issue combined with implementations of instructions from the Qiskit source-code (mainly the measure instruction) and a lot of testing, functioning extension instructions were developed.

These extension instructions are fundamentally different from the previously discussed added circuit instructions (for the non-gate commands, like `cooling`). Figure 3.1 gives an overview on how the different instructions are connected. In the figure the extension instructions are marked red. The difference between an extension instruction and a circuit instruction like `cooling` is that `cooling` has a command representation in the `Backend`, while the extension instructions do not require a command representation. They are rather pulse instructions to begin with that can be used in a `QuantumCircuit` as well. That means the `Loop`, `Fork` and `Sync` instructions are comparable to the `Play` or `Delay` instructions (with the difference that `Play` and `Delay` cannot be added to a `QuantumCircuit`). The bottom line is that a `cooling` instruction added to a `QuantumCircuit` will be translated to a `Schedule` when the circuit is scheduled, while an extension instruction inserts directly its corresponding pulse instruction, which was added to the circuit as a calibration. This was the most efficient solution to have a one to one correspondence between the loop, fork and sync circuit objects and their pulse objects. On the circuit side, the extension instructions are, similar to `cooling` type instructions, opaque objects that do not interact with things like simulations or circuit

optimisations.

Loop Loops are useful tools to make code more compact. They can be implemented in ionpulse but are not supported out of the box in Qiskit. To enable this feature, a loop instruction was developed in Qiskit which contains the necessary information about the loop. The arguments of the `Loop` pulse instruction are:

1. operands: A list or tuple with three entries
   1.1. A `Schedule` or `QuantumCircuit` object that describes the loop body.
   1.2. A looped value dictionary with the JSON representation: `{'Name of swept parameter': [value_iter1, value_iter2,...]}`.
   1.3. A number of iterations of this loop. Usually this is equal to the length of the looped values (from the second entry).
2. label: An optional label for the loop

The circuit `Loop` instruction uses the same arguments, but requires also a list of the involved qubits.
It is possible to create nested loops. In a nested loop the outer loop's looped value dictionary may influence also instructions of the inner loop. This means that if the outer loop has three iterations and the inner loop four, a parameter of an instruction in the inner loop can have twelve (three times four) independent values.
At the initialisation of a `Loop` object, the type of the loop body is determined. If the loop body is a `QuantumCircuit` it will be scheduled and the looped value dictionary will be transformed to match the new format of the loop body. Next, the value dictionary is thoroughly checked that all the keys have the correct format, reference an instruction that is contained in the loop body and that the length of the parameter list is equal to the number of iterations.
The duration of a pulse is a parameter which can be adjusted via the value dictionary.

The duration of the `Loop` instruction will take the modified durations for each iteration into account and, when queried, return the correct total duration of the `Loop` instruction.

Fork Similar to loops, forks too are useful tools for a quantum programming language. In Qiskit, there exists the possibility to create gates that are conditioned on a measurement outcome. Multiple instructions can be conditioned on the same measurement outcome, but a fork is not just one or multiple conditional instructions, it rather allows us to define independent paths depending on the outcome of a measurement. These paths could then contain additional forks as well. To enable this functionality, a custom `Fork` instruction was developed for Qiskit. The arguments of the `Fork` pulse instruction are:

1. operands: A list or tuple with three entries
    1.1. A list or tuple of conditions of the form: `{ "readout channel": 0, "state": 0x2 }`. The conditions describe which measurement result must have which result for a condition to be true. If a condition is true, the path with the same index will be executed. If no condition is true the first path in the list will be executed. This is the default path.
    1.2. A list or tuple of paths. Paths are `Schedule`s or `QuantumCircuit`s. There must be one path more than conditions as the first path counts as the default path, which is chosen when no condition is met.
    1.3. An optional tuple of passive channels. On passive channels, delays will added to each path, which are equal to the path's duration. This way, all the passive channels stay synced to the active channels in a `Fork`. Per default, all channels defined on the `Backend` will be treated as passive channels.
2. label: An optional label for the fork

The circuit `Fork` instruction uses the same arguments, but requires also a list of the involved qubits. The duration of a `Fork` is the duration of its shortest path. The definition

of the fork duration does not matter for ionpulse, as instructions on the hardware are scheduled relatively and not with respect to the beginning of the sequence. Therefore, no matter the actual length of the `Fork` it will be followed directly by the next event/instruction.
At the initialisation of a `Fork` object, the type of each path is determined. If a path is a `QuantumCircuit`, it will be scheduled. All paths are padded next. Then the active channels of the paths are determined. Active channels are all channels that have an instruction assigned. It will be checked if all channels have the same set of active channels, otherwise an error will be raised.

Sync A synchronisation instruction is used to ensure that the instructions that come after the `Sync` instruction will be executed parallel in time. This is useful after any process that does not have a predetermined duration. At this point, this is only true for `Fork` instructions. As soon as a channel reaches a `Sync` instruction in its sequence, it will not execute further instructions until all other channels listed in the `Sync`'s argument reach the same `Sync` instruction as well. From this point on, the channels will continue with the following instructions synchronised. The arguments of the `Sync` pulse instructions are:

1. operands: A list or tuple with two entries
    1.1. A list of `DriveChannel`s that will be synchronised.
    1.2. A list of non-`DriveChannel`s that will be synchronised.
2. label: An optional label for the `Sync`

The circuit `Sync` instruction separates the operands list into two arguments. Instead of the involved `DriveChannel`s, the corresponding qubits have to be handed to the instruction. The second argument is the list of non-`DriveChannel`s, as non-`DriveChannel`s are

not represented in the circuit model.

There is a tutorial in this projects Gitlab page (tutorials for TIQI / additional instructions tutorial.ipynb) that shows examples and gives an explanation on how to apply these extension instructions.

3.2.3 Pulse Types

Qiskit Pulse features parametric pulses (see at the end of section 2.3.2). The RF pulses in the TIQI experiments usually have a constant amplitude. This can be achieved with the `Constant` parametric pulse provided by the Qiskit library. The `Constant` pulse type receives an amplitude and a duration as arguments. Additional pulse types can be added to this library via external modules. To complement the `Constant` pulse, three additional pulse type were added:

- `Logic`: This pulse type is just a special case of the `Constant` pulse. A `Logic` pulse has an amplitude of 1. The pulse type has only a duration and optionally flags as an argument, since the amplitude is fixed. It can be used to define TTL pulses and add them to a `Schedule`.

- `QuadTone`: This pulse type is an extension of the `Constant` pulse. It still has a constant amplitude, but additionally it features four frequency tones. Its arguments are duration, amplitude, four frequencies and optionally flags. These frequencies are understood as frequency offset to the base frequency of the channel. It targets the successor of the current RF generation card which will be capable of playing back such a pulse.

- `Static`: The `Static` pulse type is also related to the `Constant` pulse. A `Static` pulse features a constant amplitude and its duration is defined such that it will end at the end of the `Schedule`. This `Static` pulse is equivalent to a single `Edge` on

ionpulse, which has no fixed duration (see section 2.2.2). The `Static` pulse has an amplitude argument and also an optional flags argument.

How to use these new pulse types, especially the Static one, is explained in more detail in the tutorial on the GitLab project page.

```
from qiskit.pulse import Schedule, Play
from qparser.features import Logic, Play_Static, QuadTone

pulse_example = Schedule(name='pulse_example')
pulse_example += Play_Static(0.1, DriveChannel(0), pulse_example)
pulse_example += Play(Logic(3), DriveChannel(1)) << 2
pulse_example += Play(QuadTone(5, 0.5, (200, 1230, 232, 212)),
    DriveChannel(2)) << 1

pulse_example.draw()
```

Listing 3.1: Qiskit Pulse code to create an example `Schedule` of the pulse type additions: `Logic`, `QuadTone` and `Static`

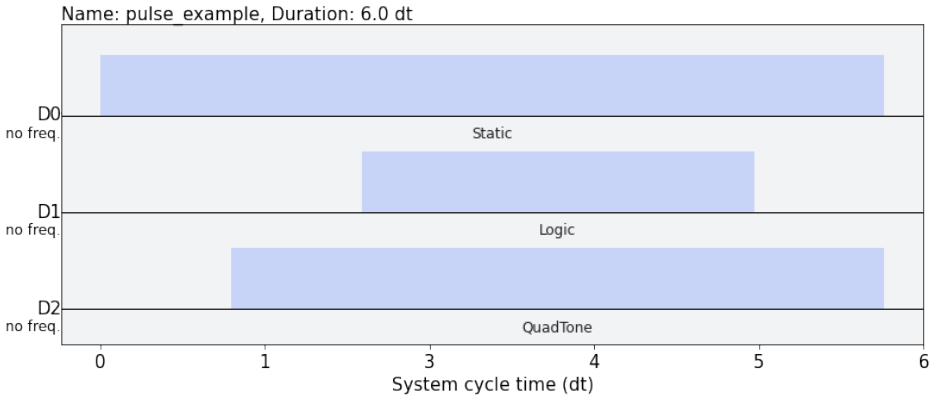


Figure 3.2: An example `Schedule` of the three pulse type additions: `Static`, `Logic` and `QuadTone`. More information in section 3.2.3. This schedule is the result of the code in listing 3.1. This figure was created using Qiskit.

Chapter 4

JSON Structure & Parser

First, the JSON structure will be explained in this chapter (section 4.1), followed by the design of the parser (section 4.2) that transforms an experiment defined as Qiskit `PulseQobj` to a JSON-string that obeys the structure. This enables the bridge between Qiskit as the quantum programming language and the TIQI control system.

4.1 JSON Structure

A certain JSON structure was defined during the course of this project. Once a `QuantumCircuit` or `Schedule` is defined in Qiskit and transformed to a `PulseQobj`, it can be converted to the JSON structure. The resulting string can then be read in by the control system and the experimental sequence can be constructed from the string. There are two different JSON structures explained in this section. The initial JSON structure is explained in subsection 4.1.1, the problem with the initial structure is illustrated in subsection 4.1.2 and the updated, current JSON structure is described in subsection 4.1.3.

4.1.1 Initial JSON Structure

The initial definition of the JSON structure is schematically described in figure 4.1 and 4.2. From this overview, we continue to explain the definition of the `Sequence_objects` structure and use of the different sequence types, followed by the rules on how to translate a given pulse sequence into this structure. Thereafter comes a description of a problematic

type of arrangement, which cannot be represented with the initial definition of the JSON structure.

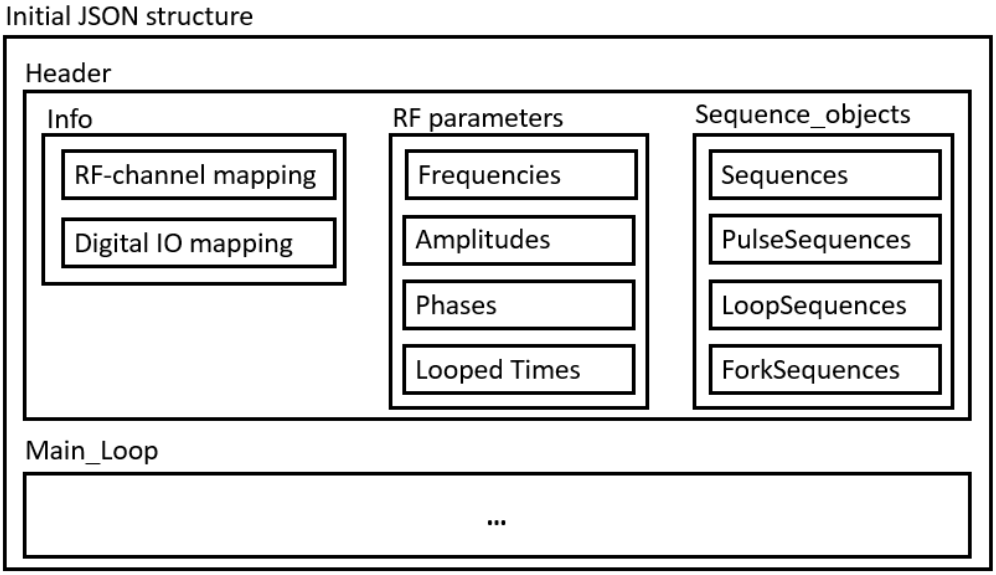


Figure 4.1: Schematic depiction of the initial JSON structure. A JSON-string contains a `Header` and a `Main_Loop` section. The `Header` is split into three parts: `Info` (optional), `RF parameters` and `Sequence_objects`. The `Info` subsection contains the RF-channel mappings and Digital IO mappings. The RF-parameters subsection contains the numerical values of the parameters used in the pulses: `Frequencies`, `Amplitudes`, `Phases` and if there is a loop over time, the `Looped Times` in a list. The `Main_Loop` section defines the whole experimental sequence. The `Sequence_objects` subsection contains all the sequence objects which are featured in the `Main_Loop`.

Explanation of the Header Figure 4.1 conveys the basic structure of the `Header`.

- **Info** – The `Info` section of the `Header` contains information about the various channel and digital IO mappings. This is not necessary for running the experiment, but to make the experiment human comprehensible in a selfcontained manner even if the physical setup changes significantly over time.

- **RF Parameters** – The `RF parameters` section contains objects for all unique RF parameters used in the experiment. There are four different types of objects to

encode: frequency, phase, amplitude and time. Each object contains the numerical value of the parameter and a channel index or mask that encodes on which hardware channel this parameter is used. Each of these objects corresponds to a `Settings` object in ionpulse, which can be reused by multiple pulses (for example the $\frac{\pi}{2}$ time).

- **Sequence_objects** – In the `Sequence_objects` section all the pulses and sequences which are used in the experimental sequence are defined. They are referenced by name in the `Sequence` object with the name 'main loop', which is the root sequence and represented in the `Main_Loop` section.

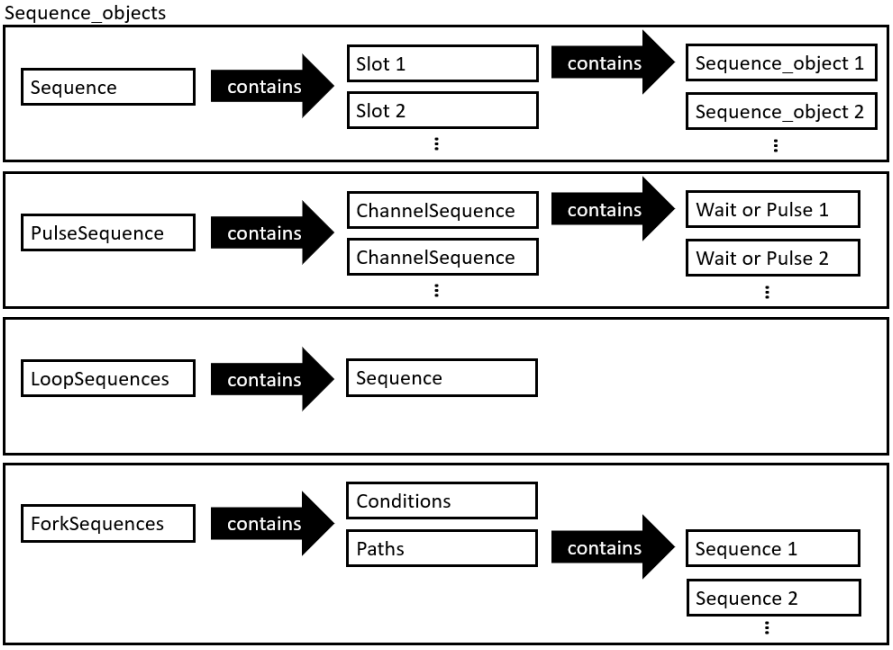


Figure 4.2: This schematic overview explains the structure of the different sequence types. There are four different sequence types: `Sequence`, `PulseSequence`, `LoopSequence` and `ForkSequence`. A `Sequence` contains at least one `Slot`, each `Slot` contains at least one sequence object. A `PulseSequences` contains at least one `ChannelSequence` and each `ChannelSequence` contains at least one `Wait` or `Pulse` object. A `LoopSequence` just contains one `Sequence`, which it will loop over a given number or times. A `ForkSequence` contains a list called `Conditions` and a second list of `Paths` which correspond to the `Conditions`. Each entry in `Paths` is a `Sequence`.

Explanation of the pulse and sequence types This paragraph will explain the structure of the sequence and pulse types, what they represent and which information each of them contain. Figure 4.2 conveys the basic structure of the four sequence types. All the sequence types can either be defined on just one or on multiple channels. Independent of the number of channels, sequence objects always have a 'rectangular shape'. That means, a sequence object starts on all of its channels at the same time, stops at the same time and has a constant set of channels (no channels are added or removed during the sequence). This makes it much easier to arrange and define such objects, but it also makes sense from a physical standpoint, as it helps preventing unintended interference with qubits, while a multiqubit gate is still active for example. The following itemization explains the sequence object types.

- **Sequence** – A `Sequence` is the most general object of the four. It is merely a chronologically structured container for sequence objects. The chronological structure arises from the `Slot`s. `Slot`s have a corresponding start time and stop time. One `Slot` follows after another without leaving time gaps. A `Slot` therefore represents a time interval. Inside a `Slot`, all sequence objects must start at the beginning of the `Slot`. The `Slot`s in this structure originate from the check routine described in 2.2.2 that enforces equal pulse times. Only sequence objects, which are defined on a subset of channels of the `Sequence`, are allowed to be added to this `Sequence`. Inside a `Slot` there can be only one sequence object per channel, therefore all sequence objects inside a `Slot` have a set of mutually exclusive channels, which are all subsets of the channel set of the parent `Sequence`. The channel set of a `Sequence` remains constant after its creation.

- **PulseSequence** – A `PulseSequence` can be of two types, either it corresponds to a command or a custom pulse. `PulseSequence`s which are of the command type represent reoccurring instructions (see the 'Commands' paragraph in sub-

section 2.3.3) that are saved in the `Backend` such as gates or cooling procedures. Therefore, commands are directly translated to `PulseSequence`s of the command type. Custom pulses, on the other hand, are also translated to `PulseSequence`s, but without the command flag. Each custom pulse will be translated to its own `PulseSequence`, even if there are multiple custom pulses in succession. Each custom `PulseSequence` contains only one pulse, which is realized as two pulse edges (one rising edge and one falling edge). In general `PulseSequence`s may contain one or multiple `ChannelSequence`s. Custom `PulseSequence`s only contain one pulse in a single `ChannelSequence`.

  - **ChannelSequence** – A `ChannelSequence` is a list of `Pulse`s and `Wait`s on a single channel. Each entry of the list will be executed sequentially.
    - **Pulse** – A `Pulse` object corresponds to an edge (single change of state) on a channel. It contains the pulse parameters: frequency, phase, amplitude, a wait time (wait duration before the edge is executed) and a flags argument.
    - **Wait** – It contains the information on how long to wait and an option to set flags.

    As `Pulse`s and `Wait`s do not contain information about the channel they run on, `Pulse`s and `Wait`s can only exist in the framework of a `ChannelSequence` which provides the channel context.

- **LoopSequence** – A `LoopSequence` consists of two things, a `Sequence` which acts as the loop body and the number of iterations.
- **ForkSequence** – A `ForkSequence` contains two lists: one which defines `Conditions` and the second defines `Paths` (in form of `Sequence`s), which is executed if the corresponding condition is met. Ideally, these conditions should be defined mutually exclusive, such that exactly one condition is fulfilled in every scenario.

Parsing rules After discussing the JSON structure and the sequence types, the rules on when and how to create these objects can be explained. All sequence objects and also the `Slots` inside a `Sequence` have always a well-defined start time, stop time and set of channels which they run on. The main loop `Sequence` object, represented in the `Main_Loop` section, is the direct or indirect parent sequence of all other sequence objects. As the experiment is parsed chronologically, the first parsed object will be placed in the first `Slot` inside the main loop. For each following sequence object, there are multiple different scenarios, which depend on the earlier parsed objects, the start time of the sequence object, type of the object and the channels on which the object is defined. For a `Sequence` to be parsed correctly, the right scenario must be deduced and the suitable instructions applied. These instructions include the creation of a new `Slot`, closing of an existing `Slot`, placing sequence objects in `Slots` and the creation of new `Sequences`, which in turn can be divided into multiple `Slots`. Although the structure definition seems simple, the parsing rules implied by the structure can become quite complex.

4.1.2 Problem with the Initial JSON Structure

While exploring the parsing rules implied by the JSON structure, problematic arrangements of sequence objects were found that can lead to unsolvable scenarios. One example can be seen in figure 4.3. The depicted `Schedule` is still possible to parse, but it creates an arrangement of sequence objects, which cannot be closed easily and can lead to an unsolvable error.

Explanation of the unsolvable scenario

The following subsection explains the process of parsing the `Sequence` depicted in figure 4.3, which will result in an arrangement of sequence objects as depicted in figure 4.4. The parser will start by adding command rxx_q0q2 to the first `Slot` in the main loop `Sequence`. The first `Slot` will be extended by adding 'Custom Pulse 1' to it. Next, either

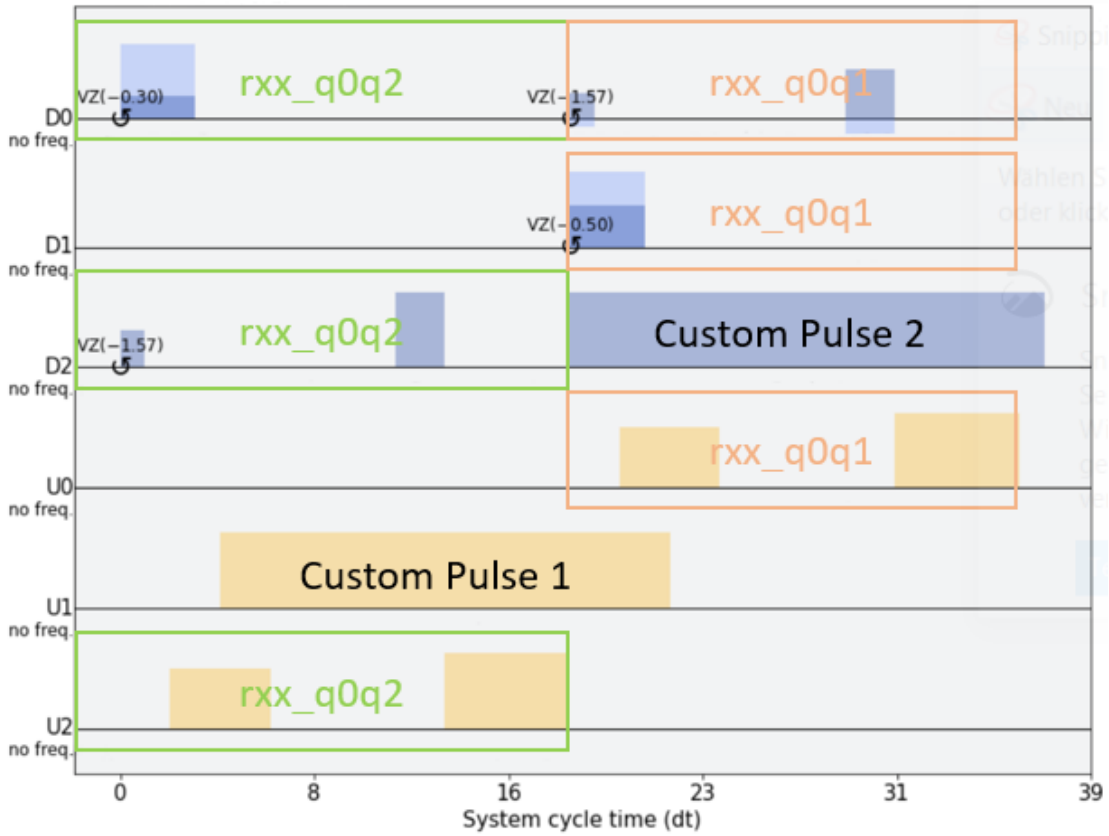


Figure 4.3: The depicted Qiskit `Schedule` consists of two commands (rxx_q0q2, rxx_q0q1) and two custom pulses (1 & 2). This specific arrangement of sequence objects could lead to a scenario, which is impossible to parse following the JSON structure rules at that time (see subsection 4.1.2 and figure 4.4). This figure was created using Qiskit.

rxx_q0q1 or 'Custom Pulse 2' will be parsed, in both cases we will end up with a nested `Sequence` inside main loop `Slot` 1, because a `Slot` can only contain one sequence object per channel. The nested `Sequence` runs on the channels D0, D1, D2, U0, and U2 and it has two `Slots`. The first `Slot` contains only rxx_q0q2 and the second `Slot` contains rxx_q0q1 and 'Custom Pulse 2'.

The problem arises when after this `Sequence` a new command, `Loop`, `Fork` or `Sync` instruction follows without delay and is defined on channel U1 and at least one channel of the following: D0, D1, U0, U2. The problem lies in the fact that the active `Sequence`

on channel U1 is still the main loop, while a new `Sequence` was created on the other channels. Usually one would just close the new `Sequence` and then the extra command could be parsed without issue, but the new `Sequence` can only be closed at the stop time of 'Custom Pulse 2'. If this stop time is later in time than the start time of the additional command, the closing of the `Sequence` will lead to an overlap, which results in an error. This problem cannot be circumvented as long as sequence objects can only be rectangular. As explained earlier, rectangular means that the start and stop time on all channels of a sequence object must be the same and the set of channels on which a sequence object operates must remain constant through the whole duration of the object.

Possible adaptions to the JSON structure

1. A solution for this problem would be to relax the constraints on the `Sequence` definition such that not the latest stop time of all the pulses inside a `Slot` defines the stop time of the `Slot`, but the earliest stop time. This way we get rid of the constraint, that the stop time on all channels of a `Sequence` must be simultaneous (`Sequence`s would not be rectangular anymore).

2. Another possibility would be to represent the main loop as a chronologically ordered list of sequence objects, completely eliminating `Slot`s. This simpler representation loses the notion of simultaneous timing and it removes the information about which channels a child object belongs to from the parent object, which makes it harder to parse the structure on the control system.

3. An alternative is to completely discard `Sequence`s and, similar to Qiskit, simply create an ordered list of pulses for each channel independently. This has the upside of a simpler implementation and the downside that objects which are run on multiple channels will lose the information that a synchronized operation is running on multiple channels.

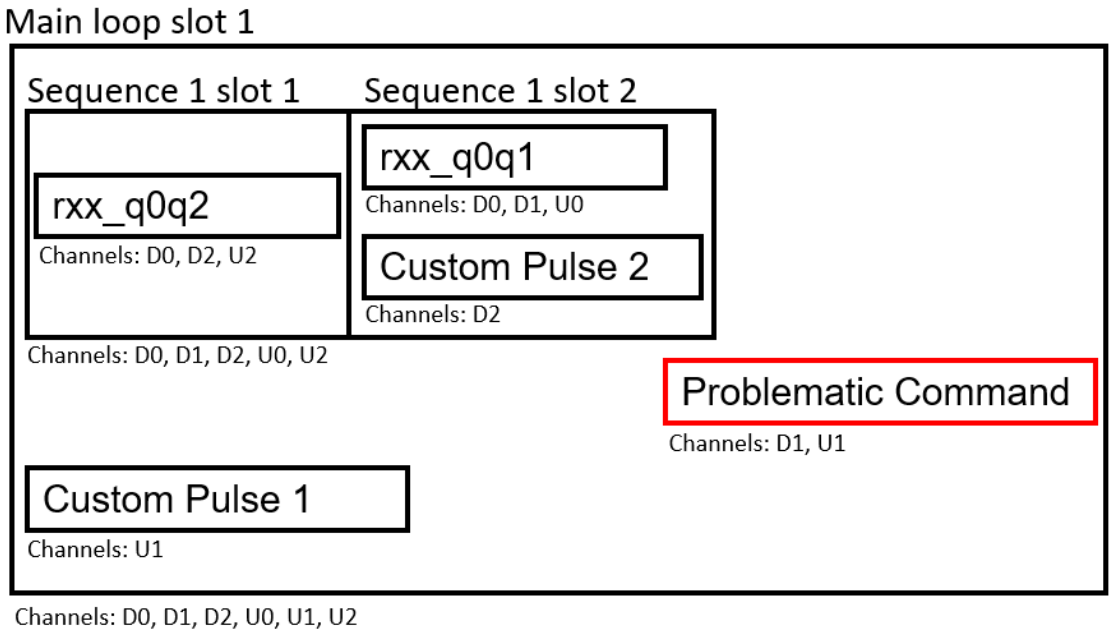


Figure 4.4: This figure is a schematic representation of the pulse arrangement in figure 4.3 plus an additional 'Problematic Command'. The black or red boxes represent different sequence objects. Below each sequence object one can read the channels on which the sequence object is defined. Subsection 4.1.2 explains how it comes to this arrangement. The 'Problematic Command' is added to help visualizing the problem. In this example the 'Problematic Command' is defined on the channels D1 and U1. The active `Sequence` on channel D1 is 'Sequence 1', while for channel U1 it is the main loop. The 'Problematic Command' cannot be defined in both 'Sequence 1' and the main loop. 'Sequence 1' cannot be closed, as it would create an overlap with the 'Problematic Command' on the channel D1 (due to 'Custom Pulse 2' ending after the start of the 'Problematic Command').

A variant of this solution would be to add a reference to a multi-channel object to each of the concerned channels. These references will then point to one single object with the information on which channel which channel sequence is executed. Commands on multiple channels still appear cohesive when using this variant.

4.1.3 Improved JSON Structure

Unfortunately, the parser for the old JSON structure was already implemented to a major part. Nevertheless, a parser for a structure which cannot represent any possible pulse sequence is at minimum questionable. This made the decision clear, that the JSON structure must be redefined and the parser adapted.

The new JSON structure is essentially a well-defined version of the third idea in the subsection 4.1.2. The JSON-string is structured in six sections: `Info`, `RF parameters`, `RF pulses`, `Digital IO`, `Readouts` and `Sequences`.

Info This section corresponds to the `Info` section in the initial JSON structure (see subsection 4.1.1). It contains information about the channel and digital IO mappings. One can define which units are used by the RF parameters and additional information about the experimental sequence can be added.

RF parameters In this part of the JSON structure, which is also still the same as in the old JSON structure, the numerical values of all pulses/instructions of the experiment are listed. There are four different types of `RF parameters`: frequency, phase, amplitude and time. The first three parameters are represented identically in this structure. Each parameter has a unique name as an identifier (the values are referenced inside the `RF pulses` section by this name). The frequency, phase and amplitude parameters then contain two entries: `ch_idx` and `value`. The first represents the index of the channel on which this parameter is used. The second contains the value of the parameter. The units of these values are Hz or MHz (not fixed yet) for frequencies, degrees [°] for phases and a normalized value between 0 and 1 for amplitudes, where 0 stands for minimum/no amplitude and 1 for maximum amplitude. Time parameters are represented very similarly to the ones before with the only difference that a time entry does not contain a `ch_idx`

but rather a `ch_mask`. This stands for channel mask and is a binary mask for the channels on which this time parameter is used. The unit of the time value is μs. If a parameter is iterated inside a loop, the value of the parameter is a list, and not just a single entry.

RF pulses The `RF pulses` section lists all instances of fundamental pulse objects that are required to describe a given experiment. These objects are closely related to the ones described in section 2.2.2, which makes sense as the JSON-string will be translated to instances of these classes. These objects are: `Edge`, `Wait`, `Quadedge` (an edge with four frequency components) and `Sync`. They reference RF parameters from the previous section in some of their entries. Here is a detailed overview of the structure these pulse objects:

- `Rf_edge`: describes a single change in frequency, phase and/or amplitude. Therefore, it describes a rising or a falling edge of a pulse on a given channel.
  - `type`: the type of sequence or pulse object. In this case this is always `rf_edge`.
  - `ch_idx`: the channel on which the `Rf_edge` is defined
  - `freq`: the name of the frequency parameter. The name acts as a reference to an entry in the `RF parameters`.
  - `phase`: the name of a phase parameter similar to `freq`
  - `amp`: the name of an amplitude parameter similar to `freq`
  - `time`: the name of the time parameter similar to `freq`
  - `flags`: an integer value that is used as a flag, default is 0. A shaped pulse as described in section 2.2.2 can be achieved by handing a specific flag.
- `Rf_wait`: describes a delay of a duration given by its time argument. During the wait time no instructions will executed on the corresponding channel. The `Rf_wait` contains the following information:

  - `type`: `rf_wait`
  - `ch_idx`: the channel on which the `RF_wait` is defined
  - `time`: the name of the time parameter
  - `flags`: an integer value that is used as a flag, default is 0

- `Rf_quadedge`: describes an edge of a `QuadTone` (see 3.2.3). The `Rf_quadedge` contains the same dictionary entries as the `Rf_edge` with the difference that the value for the `type` key is `rf_quadedge` and the value to the `freq` key is a list of four frequency tones.

- `Sync`: describes a `Sync` instruction. A `Sync` contains the following information:
  - `type`: `sync`
  - `ch_mask`: a binary mask, which describes on which channels the `Sync` is active.
  - `time`: the name of the time parameter
  - `flags`: an integer value that is used as a flag, default is 0

Sequences The `Sequences` section of a JSON structure lists all the more complex objects in an experimental sequence. All sequence objects contain a `subsequences` section with a dictionary entry for each channel that the sequence object acts on. This dictionary entry is an ordered list of references to objects in the `RF pulses` section and/or previously described sequence objects. For each channel, each list entry will be executed one after another on the physical setup. These objects are ordered in such a way that more complex objects, which reference other sequence objects, are listed after all of their dependencies. This way, a function that reads in a JSON-string from top to bottom can efficiently unpack and combine the sequence objects, as all references are already defined once a more complex object is read in. The different sequence objects are:

- `Linear_sequence`: describes general container for any kind of pulse and/or sequence objects. A `Linear_sequence` contains the following information:
  - `type`: `linear sequence`
  - `ch_mask`: a mask, which describes on which channels the `Linear_sequence` object is active.
  - `subsequences`: a dictionary that has channel indices as keys and a list of references to pulse and/or sequence objects as values. The list contains in order all the objects that will be executed on the given channel.
- `Loop`: describes a loop instruction. `Loop`s, `Fork`s and `Sync`s are directly translated from their Qiskit counterparts (see 3.2.2). A `Loop` contains the following information:
  - `type`: `loop`
  - `ch_mask`: see `Linear_sequence`
  - `iterations`: the number of times the loop will be iterated
  - `subsequences`: the value to this key value has the same form as the `subsequences` in the `Linear_sequence` objects. Here the `subsequences` act as the loop body, which is the sequence of instructions that will be repeated in the loop.
- `Fork`: describes a Fork instruction. A `Fork` contains the following information:
  - `type`: `fork`
  - `ch_mask`: see `Linear_sequence`
  - `conditions`: a list of conditions
  - `subsequences`: a dictionary that has channel indices as keys and a list of references to `Path` objects as values. The lists contain in order all the `Path`s. The first `Path` in these lists is the default `Path` and any other `Path` will be executed

by the `Fork` if the condition with the `Path`'s index minus one is fulfilled. The lists of `Path`s must therefore contain one additional entry compared to the list of conditions.

- `Path`: has the exact same form as a `Linear_sequence` with the only difference that the value to the `type` key is `path`. `Path`s are separate objects to discern them from `Linear_sequence` objects.

The pulse and sequence objects combined are the building blocks to define arbitrarily complex experimental sequences (see a schematic example in figure 4.5) and make use of all of the capabilities of ionpulse.

The main loop, which defines the root object of the experimental sequence, is simply a `Linear_sequence` and is listed at the end of the `Sequences` section in a JSON-string, as it either directly or indirectly references all other sequence objects.

While the Qiskit `PulseQobj` (and also `Schedule`) instances keep track of an absolute time and the instructions usually contain a start time and a duration, the JSON structure utilizes a so-called 'event based approach'. In this approach each `Edge`, `Wait`, `Sync`, `Loop`, `Fork` or `Linear_sequence` is an event. Events do not have a fixed start time. A new event starts directly after the last event on the same channel has occurred. The time argument of an event is the wait time until the instruction (mostly `Edge`s, because `Linear_sequence`s, `Loop`s, `Sync`s, and `Fork`s do not require time arguments) is realized on the hardware. Due to hardware limitations, a minimum wait time exists between consecutive `Edge`s on the same channel.

The new JSON structure does not run into the same problem as the initial JSON structure. Although `Linear_sequence` objects are still rectangular, they do not contain

`Slots` as `Sequence` objects did in the original structure. This JSON structure even circumvents the creation of `Slots` on the control system, which is not the case with the former way of defining experiments through the ionpulse API. The check routine mentioned in 2.2.2 uses time slots to enforce a simple structure. This routine is not applied to the JSON-string. A `Linear_sequence` contains a separate sequence for each channel. These channel sequences do not have a need for `Slots`, because no instructions run in parallel on a channel sequence and only one instruction is active at a point in time. The instructions of a channel sequence are simply executed in order. One needs to pay attention that multi-channel instructions start at the same time on all channel sequences. The parser ensures that this is the case.

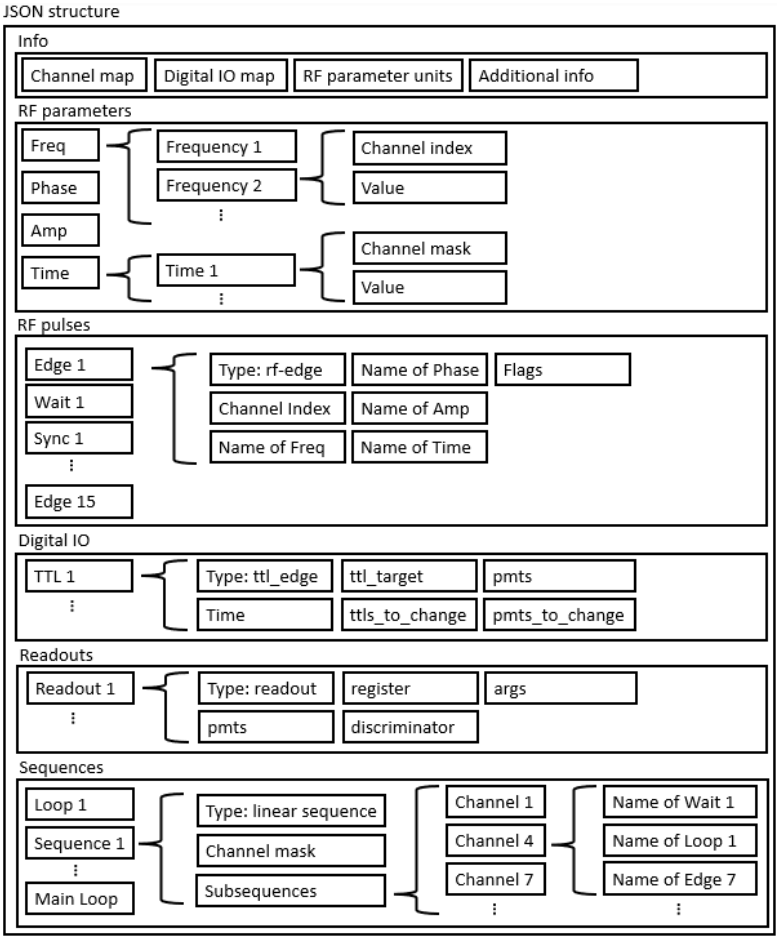


Figure 4.5: A visual representation of the new structure. Each box represents a key-value pair. Curly braces indicate additional contents in the boxes. The fourth and fifth part of the structure consist of the `Digital IO` pulses and `Readouts`. These two sections were added recently before submission of the thesis and are therefore not yet implemented in Qiskit nor in the parser and are shown here only for completeness.

4.2 Design of the Parser

The development of the parser is a major part of this master's project. It can be understood as the bridge between Qiskit and ionpulse. The parser translates the Qiskit `PulseQobj` to the JSON structure. A simplified overview of the structure of the parser can be seen in figure 4.6. The arguments of the parser function are the following:

- qobj: The first argument is a Qiskit `PulseQobj` generated by assembling an experiment described as a `Schedule`.
- backend: The second argument is the `Backend` that holds information on all channels used in the `PulseQobj` and also the command definitions for the instructions.
- verify: The third argument is optional and per default True. The boolean-type 'verify' argument decides if the resulting JSON-string will be checked against the `PulseQobj` after parsing. In the process of checking, any mismatches will be printed to the console. It is recommended to verify every time as confirmation of the integrity of the JSON-string.
- filename: The fourth argument is optional as well and per default 'parsed_experiment.json'. This argument defines the filename used for the JSON-string. One can also input a relative path in front of the filename.

The parser function will return the resulting JSON-string and at the same time create a file with the supplied name and path that contains the JSON-string.
The core of the parser (the `parse_instructions` function) is structured in a way that allows it to be used in a recursive manner. This is necessary because the loop body of `Loop` instructions as well as `Paths` of `Fork` instructions can be understood as complete experiments themselves. If such an instruction is encountered the parser will recursively parse them and all nested `Loops` and `Forks` if there are any. When the `parse_instructions`

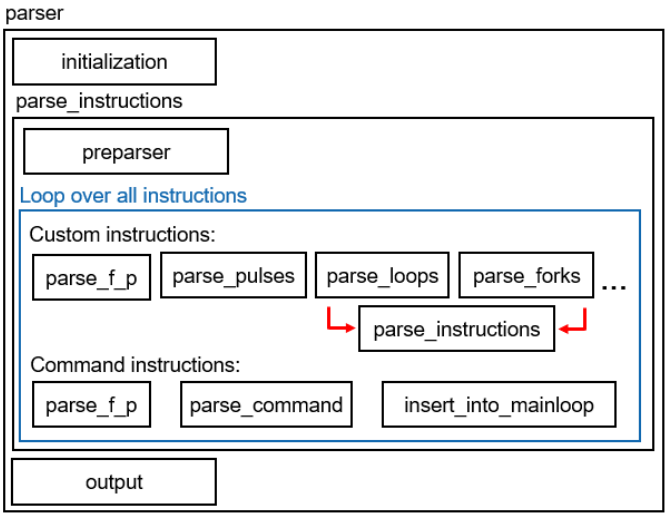


Figure 4.6: A simplified overview of the internal structure of the parser function. The parsing process starts with an initialization routine that prepares the important objects. Next, the subfunction `parse_instructions` is called, where the main parsing process happens. This starts with the `preparser` that rearranges the pulse instructions. A loop over all instructions is executed, where each instruction is parsed depending on if it belongs to a command and which kind of instruction it is. After the loop, the parsing process is done and the JSON-string is generated.

function is called recursively the created pulse and sequence objects are not added directly to the main loop. They are stored in a dictionary and are combined with the main loop only in the original call of the `parse_instructions` function. This is necessary to enforce the order of the objects in the JSON-string as discussed in 4.1.3 at the start of paragraph 'Sequences'.

Sequence objects reference other pulse or sequence objects by name. The parser creates names for both types of objects that define them uniquely, as its name carries all the relevant information about a certain pulse or sequence object. This enables savings in memory if multiple times the exact same object is created, as it will only appear once in the JSON-string. The names are constructed differently for `Loop` and `Fork` instructions, because they carry too much specific information in their loop body and paths argument to be condensed into a meaningful name. If necessary, the object names can be hashed

or a unique ID can be used. The memory saving capabilities would still apply, but it would not be easily human-readable anymore.

Due to the change in the JSON structure after months of development on the original parser, the parser had to be adapted to a major part as well. In that process as much code as possible was reused for time saving purposes. There may be legacy structural components which could be written more straightforwardly and make the parser more streamlined. For example, the preparser (see subsection 4.2.2) could be adapted.

The parser will ignore `Acquire` instructions. Different to all other Qiskit instructions the `Acquire` instructions do not carry a channel argument, although they are defined on `AcquireChannel`s. Another problem with `Acquire` instructions is that, even after many different approaches, no possibility was found to conserve the label of an `Acquire` instruction through the conversion into a `PulseQobj`. This is necessary for `Acquire` instructions to be recognized as part of a command. After realizing these issues, not too much time was wasted to enforce the correct parsing of `Acquire` instructions. The functionality of an `Acquire` instruction can be replicated by sending `Logic` pulses on `ControlChannel`s to PMTs to activate the data acquisition.

The main goal of the master's project is to conceptually create a bridge from a gate-level quantum programming language to the control system of the TIQI group. With the parser and the Qiskit extensions this is possible. The absence of `Acquire` function does not hinder this goal.

4.2.1 Important Objects

In the parser function, some objects are required by each parsing subroutine. These objects either accumulate parsed objects or keep track of information that is updated throughout the parsing process and is used as arguments in the pulse and sequence objects.

main loop The main loop is a `Linear_sequence` object. It holds the description of the whole experiment. During the parsing process, all the other objects are added to the main loop either directly or indirectly by first adding them to other sequence objects, which are eventually added to the main loop.

channel_dict The channel dictionary keeps track of the phase, frequency, absolute time and delays for each channel. Qiskit describes pulses not in a single instruction, rather the phase and frequency are adapted via their own instructions. The absolute time tracker is used to check if additional `Waits` must be created and how long the wait times of pulses are. The time is updated whenever an instruction with a non-zero duration is parsed. As `Schedules` are allowed to have gaps, which are implicit delays, the parser must notice these gaps by comparing the absolute channel time to the start time of instructions and act accordingly. This ensures the correct conversion from the timetable-based representation of the experiment in Qiskit to the event-based one in the JSON structure.

Another important dictionary is the `wip_command_dict` (short for work in progress command dictionary). It is only used for parsing commands and therefore explained in the subsection 4.2.4.

4.2.2 Preparser

The main goal of the `preparser` is to arrange all the Qiskit instructions from the `PulseQobj` into a dictionary with the name `timed_instructions`. The keys of this dictionary are the start or stop times (integers) of the instructions in the `PulseQobj`. The values associated with the time keys are again dictionaries and contain the following entries:

- 'start_time': It contains all the instructions that start at the time indicated by

the parent key. These instructions are arranged in this lower dictionary by the following three keys:

  - 'custom': all the instructions which are not part of a command. These custom instructions can be any type of instruction including extension instructions like `Loop` or `Fork`.

  - 'starting_inst': all instructions that belong to a command and are the first instruction of that command, i.e. all instructions of a command that have a start time of zero in the command. The starting instructions are important to recognize when a command starts.

  - 'cmd_body': all other instructions of commands that are not starting instructions.

- stop_time: a list of command names that end at the time indicated by the parent key. This stop time flag is used in the parser to signal that the assembly of the command is finished and that it can be inserted into the main loop. This is only needed for instructions that belong to commands, because commands need to be first combined to a `Linear_sequence` before they can be added to the main loop. All custom instructions can be added directly.

4.2.3 Custom Instructions

As mentioned before, custom instructions are all instructions that do not belong to a command. More information in the paragraph 'Command instruction and custom instruction' in subsection 3.2.1. Custom instructions are parsed individually depending on their type. For each type a paragraph describes what is noteworthy when parsing this kind of instruction.

frequency and phase The instructions that modify the frequency or phase on a channel are the following four: `ShiftFrequency`, `SetFrequency`, `ShiftPhase`, `SetPhase`. No individual object is created when one of these instructions is parsed. They only modify the phase and frequency entries in the `channel_dict` and have a duration of zero. It is important that all the changes to phase and frequency are made before any other instruction types are parsed. If this would not be the case and, for example, a `ShiftPhase` instruction and a `Play` instruction have the same starting time a problem would occur. If the `Play` instruction is parsed before all the changes to phase and frequency are applied, the phase and frequency arguments of the pulse, described by the `Play` instruction, are not up to date. This is unintended behaviour. Whenever a phase or frequency modifying instruction has the same start time as a instruction that describes a pulse, the phase or frequency modification must be applied first.

delay and gap A custom `Delay` is not directly parsed at the point when the `Delay` instruction runs through the parser. It is rather saved in the `channel_dict` as custom delay time. Depending on the next instruction on the same channel, a `Wait` object will be created with a duration of the custom delay time. This happens when the next instruction is a `Loop`, `Fork` or `Sync` or belongs to a command. If the next instruction is a custom pulse, a part of the custom delay time may be used as the time argument of the rising edge of this pulse. All `Edge` objects require a time argument of at least the minimum wait time defined by the hardware. For this reason, the minimum delay time must sometimes be taken from custom delay times.
Gaps are implicit delay instructions and will be translated to `Wait` objects if the next instruction is not a custom pulse (but a `Loop`, `Fork`, `Sync` or part of a command). If a gap is followed by a custom pulse, the duration of the gap is used as the time argument of the rising edge of this pulse if its duration is longer than the minimum wait time, otherwise an error is raised.

pulse Custom pulses are translated to either one (if the pulse is of type `Static`) or two `Edge` objects. The phase and frequency argument of these `Edge`s are taken from the `channel_dict`. The amplitude and the channel is given by the instruction itself. The time argument will be calculated depending on the custom wait time, the channel time from the `channel_dict` and the start time of the instruction. Once the `Edge`s are created, they are combined to a `Linear_sequence` object and added to the main loop or parent object.

Loop When a `Loop` is parsed, the start time of the `Loop` instruction is compared to the current channel time stored in the `channel_dict`. If necessary `Wait` objects are created and added to the main loop. At the same time the phase and frequency entries of the `channel_dict` is reset to the default values for all channels involved in the `Loop`. This is necessary for the JSON structure to function properly, otherwise each iteration of a loop could have the same pulse with different phase or frequency arguments. If this is intended it can be achieved with the looped value dictionary.
Next, the looped value dictionary of the `Loop` is expanded into a nested dictionary that carries the same information, but is easier to use for the parser.
The `parse_instructions` function is then called once recursively to parse the loop body. In this function call the restructured looped value dictionary is handed to the `parse_instructions` function, which is then used to replace the parameters of the instructions specified by the looped value dictionary with the corresponding list of looped parameters. After this, the `Loop` object is assembled and added to the main loop or other parent object.

Fork When a `Fork` instruction is parsed its start time is compared to the current channel time of the relevant channels and `Wait` objects are created if necessary. At the same time, similar the `Loop` parsing process, the phase and frequency entries of the `channel_dict` are reset to their default values at the start and end of the `Fork`

instruction. The reset is necessary for `Fork` instructions because it is undefined how the phase and frequency are modified during the different paths. The pulses after the `Fork` would then depend on which path was taken. This is not possible with the current JSON structure, but also not necessary. The paths can simply be expanded to include the concerned instructions, which then enables this behaviour.
Next, each `Path` is parsed individually by calling the parser function recursively. The `Fork` object is then constructed and added to the main loop or other parent object.

Sync A `Sync` instruction is parsed by comparing the channel times of each of the instructions channels to the start time of the instruction and creating `Wait`s if necessary. This sounds counterintuitive as it is the function of the `Sync` instruction to synchronize the channels. For the parser these `Wait`s are necessary to correctly keep track of the absolute time. Even though a `Fork` has a duration that is not predetermined, it still has a well-defined duration for the parser and if there is a time gap before any instruction (even `Sync`s), the parser must fill these with `Wait`s. Next, the `Sync` object can be created as only the channels argument is required and this object is then added to the main loop or other parent object.

4.2.4 Command Parser

In this subsection the process of parsing commands is explained. Before that, it is important to clarify why it makes sense to parse commands as one sequence object and not parse every instruction of a command independently. This has two reasons.
Firstly, a command usually represents a higher-level instruction, often gates. If a command is represented in the JSON-string as a cohesive object, the file is more human-readable.
Secondly, because commands represent a higher-level instruction, there is a high likelihood that a command would be reused in the same `Schedule` with the same arguments.

In this case the command's `Linear_sequence` object is described only once in the JSON-string and multiple references point to that same object. This makes the file more memory efficient and allows the control system to transfer these memory savings to the memory limited FPGA that executes the experiment.

Commands cannot be parsed in a single function call like custom instructions. A command needs to be built instruction after instruction. Intermediate results a saved in the `wip_command_dict`.

wip_command_dict The work-in-progress command dictionary is essential to the command parsing process. Disregarding the information in the `channel_dict`, the `wip_command_dict` stores all the information necessary to create a command instruction by instruction.
Multiple commands can be running in parallel, so the dictionary has an entry for each command that is currently being built. The most relevant entries of the dictionary are the following:

- 'complete_sequence': the complete sequence of the command saved as a list. A new pulse or sequence object is added to the list whenever an associated instruction is parsed. The complete_sequence is used to create the `Linear_sequence` which corresponds to the command once all its instructions were parsed.

- 'parameter_binds': A command usually requires parameters as arguments. These parameters are handed to instructions and influence the behaviour of the command. To differentiate between commands with different arguments, these arguments must be captured in the parsing process and featured in the name of the command, which is also its reference. Only this way, commands can be reused correctly.

- 'initial_p_f': This key in the dictionary serves a similar need as the parameter

bindings before. The sequence of a command is influenced by the phase and frequency of its channels. Therefore, if non-default phase and frequency values exist on any of the commands channels, the channel and the phase and/or frequency value is displayed in the name of this command as well to ensure that commands can be reused correctly.

- 'next_wait_time': A next wait time entry exists for each of the commands channels. The wait time is needed as a time argument of the rising edge of pulses that follow a previous pulse without any delay. In this case, the first pulse will not have a falling edge, which turns the amplitude on the channel to zero. The duration of this first pulse will become the wait time of the rising edge of the next pulse. This way some unnecessary edges can be omitted and pulses can be used in a command back to back.

The `preparser` already separated the custom instructions, the first instructions of a command and the rest of the command instructions. The parsing of a new command begins when a starting instruction of a command is parsed. The start time of the whole command is given by the start time of the starting instructions. The name and the qubits of the command is extracted from the label of the starting instructions. With this information the whole command can be retrieved from the `instruction_schedule_map` of the `Backend`. This gives access all of the command channels. The channel time from the `channel_dict` of each of the command channels is compared to the start time of the command and `Wait`s are generated accordingly. A new entry for this command is added to the `wip_command_dict` and the initial phases and frequencies for each of the channels are saved. Then all instructions of the command will be parsed and added to the corresponding entry in the `wip_command_dict`, since the command name is written in the label of each of these instructions. The parsing of the individual instructions happens very similarly to how custom instructions are parsed with the difference that they are

not added to the main loop yet but to the `wip_command_dict`. Once every instruction is parsed and the stop time of the command is reached, the command will be finalised in the function `insert_into_main_loop`. Final `Wait`s or turn off edges at the end of each channel sequence are generated and added to the command sequence. Then, the duration of each channel sequence is compared to the duration of the command to ensure that no instruction was missed or added multiple times. The name of the command is generated containing all the information to define the parsed command sequence in a reproducible manner. Finally, the `Linear_sequence` object of the command is generated and added to the main loop or parent object.

4.2.5 Verification

After the experiment is parsed, the resulting JSON-string can optionally be compared to the original `PulseQobj`. This happens through the `parser_verificator` function that is defined in a separate module. During the verification process the sequence of each of the channels is compared one instruction at a time. For each instruction the JSON-description and the Qiskit `PulseQobj` description are transformed to an intermediate description that makes comparison simple. Every single argument of all the instructions is compared. When a mismatch is detected, an error message is prepared, containing as much detail as possible about the type of mismatch and the position in the experimental sequence. This error message is printed into the console after the verification process and the verification function returns the boolean value False.
The verification function is programmed with recursive capabilities to correctly verify loop bodies and the paths of forks (and all nested varieties) as well.

4.2.6 Emulation

A proof of concept `Schedule` was created, assembled and parsed. The resulting JSON-string was loaded into an ionpulse emulation and executed to verify that the parser

correctly translates an experiment encoded in a `PulseQobj` to an equivalent experiment on ionpulse (via the JSON-string). Figure 4.7 compares the result of this test to the original `Schedule` (generated from the code in listing 4.1).

```
from qiskit.pulse import Schedule, DriveChannel, Play, Constant
from qparser import min_wait_time
from qparser.features import Fork, Loop

loop_body = Schedule()
loop_body += Play(Constant(3, 0.8), DriveChannel(0)) << min_wait_time
loop_body += Play(Constant(4, 0.5), DriveChannel(1)) << 4
loop_body += Play(Constant(2, 0.2), DriveChannel(1)) << min_wait_time
loop_body += Play(Constant(3, 1), DriveChannel(2))

looped_values = {'play_2_d0_dur': [2,3,4,5], 'play_4_d1_dur': [7,3,9,5]}
iterations = 4

path1 = Schedule()
path1 += Play(Constant(min_wait_time, 0.5), DriveChannel(0)) << (
    min_wait_time + 4)
path1 += Play(Constant(4, 0.1), DriveChannel(1)) << min_wait_time

path2 = Schedule()
path2 += Play(Constant(3, 0.4), DriveChannel(0)) << min_wait_time
path2 += Play(Constant(min_wait_time, 0.2), DriveChannel(1)) <<
    min_wait_time

paths = [path1, path2]
conditions = [{"readout channel": 0, "state": 0x2}]

test_sched = Schedule()
test_sched += Play(Constant(2, 0.2), DriveChannel(0)) << min_wait_time
test_sched += Loop([loop_body, looped_values, iterations])
```

```
28 test_sched += Play(Constant(3, 0.5), DriveChannel(1)) << min_wait_time
29 test_sched += Fork([conditions, paths, []])
30 test_sched += Play(Constant(4, 0.6), DriveChannel(0)) << min_wait_time
31 test_sched.draw()
```

Listing 4.1: Qiskit Pulse code to create the example `Schedule` shown in figure 4.7. The call of the `draw()` method in line 31 generates the top most plot in figure 4.7 without the insets. The insets need to be plotted separately, because the `draw()` method cannot represent the extension instructions (`Loop`, `Fork` and `Sync`) correctly.

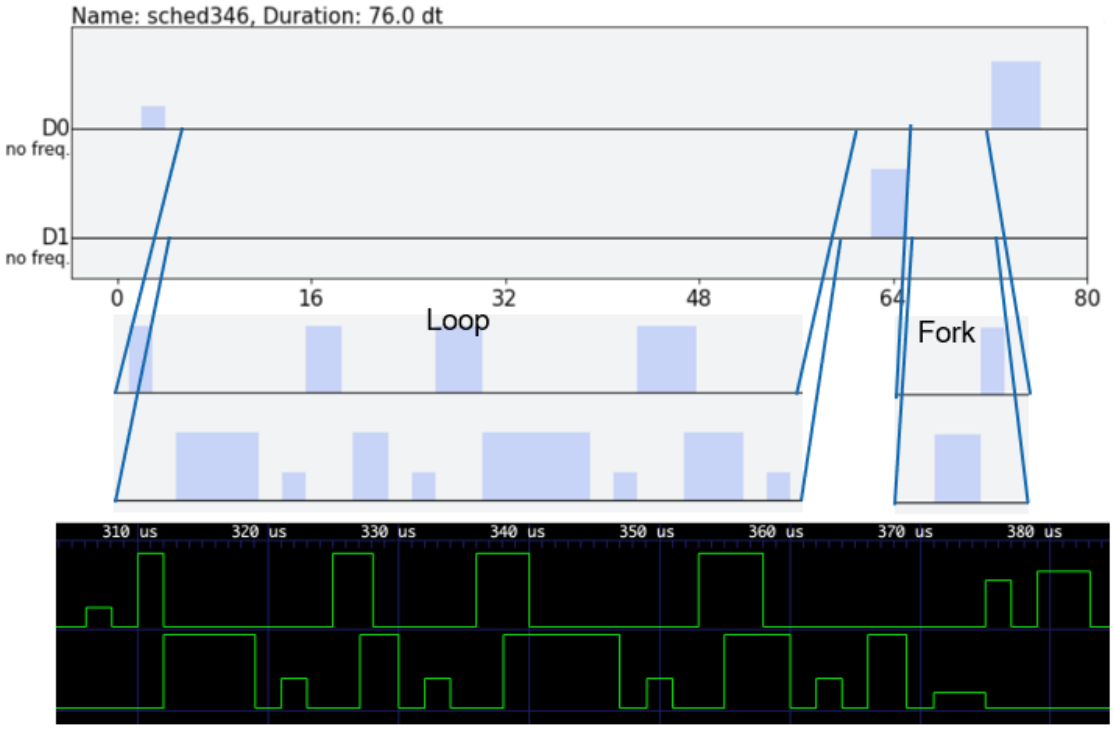


Figure 4.7: An example `Schedule` drawn by Qiskit (upper plot) with insets showing the loop and the fork. The lower plot shows the output of the ionpulse emulator provided with the JSON-string generated from the same `Schedule`.

Chapter 5

Conclusion & Outlook

After getting familiar with the current TIQI control system, the first step of this master's thesis was to find a suitable quantum programming language to interface with the system. Qiskit is the language of choice out of many reasons. The most prominent is that it features a low-level language for writing pulse sequences. This enables the choice between a higher, gate-level definition of experimental sequences or a lower, pulse-level control.

A proxy `Backend` was configured for testing purposes and to serve as a conceptual example. The function of a Qiskit `Backend` instance is to store information about the hardware setup and a map between gate instructions and pulse schedules, which allows to transform a Qiskit `QuantumCircuit` to a pulse `Schedule`.

Qiskit was extended with additional modules to enable all the functionality of ionpulse. These modules define a loop and a fork instruction on both Qiskit Circuit and Pulse.

After months of work on the parser between Qiskit and the JSON-string, a problem with the original JSON structure was found, which set the work back significantly. This hurdle was overcome by defining an improved JSON structure that allowed me to develop a parser with a much cleaner design. The parser also features a verification function to verify that no errors during the parsing process occurred.

The Gitlab repository of this project is named Qiskit ionpulse API and features two tutorial Jupyter notebooks to give examples on how to use the parser, the Qiskit extensions and the functionality around them.

Noteworthy Comments The parser ignores Qiskit `Acquire` instructions (instructions that define a data acquisition time window during a measurement process). There are multiple ways to create a `Schedule` that can be translated to a measurement acquisition instruction in hardware without the use of `Acquire` instructions. The reason for this and a simple alternative is mentioned in section 4.2.

At a very late stage during this master's thesis, it was discovered that certain parts of the `Backend` were implemented according to a now obsolete example. Right at the time when I was looking for example `Backend` instances, a new way to define the `instruction_schedule_map` in the `Backend` was getting introduced. This map is used to transform `QuantumCircuit`s to `PulseSchedule`s. The new method to create this map, explained in subsection 3.2.1, solves the former problem that duration arguments of pulses in the `instruction_schedule_map` could not be parametrised. Unfortunately, this method was found at such a late stage that it was not possible to utilise it in the `Backend` in the scope of this master's thesis. This means that durations cannot be parameterised in the example `Backend` of this thesis, but new `Backend`s can be defined without this limitation.

5.1 Conclusion

The goal of this master's thesis is, as the title implies, to enable a quantum-gate-level interface with a trapped ion control system. This is achieved by the conceptual `Backend` and the parser, which allows the transformation from a Qiskit `QuantumCircuit` or a pulse `Schedule` object to an experiment described in a well-defined JSON structure. This JSON-string can be read in by ionpulse and executed without the need of recompiling and reloading ionpulse to the CPU of the control system, which is an improvement to the former way of defining pulse experiments. This was demonstrated by running an experiment defined in a JSON-string, created by the parser, on the emulated control sys-

tem (see subsection 4.2.6). To verify the output of the parser only on an emulated system and not on the actual hardware is sufficient. This is because for this master's thesis we want to test if the parser generates the correct JSON-strings and not if the parser on ionpulse, which recreates the pulse experiment from the JSON-string, functions correctly.

5.2 Outlook

This master's project is a first step towards running arbitrary quantum circuits on trapped ion quantum computers of the TIQI group. What is missing at this point is an accurately calibrated `Backend` and additional functionality, which makes the `Backend` more flexible. Then, the `Backend` will not just translate a `QuantumCircuit` instance to a more or less arbitrary pulse sequence, but to a pulse sequence that physically corresponds to the quantum circuit.

Liberto Beltran will continue working on this project as part of his master's thesis with the final goal of running a small variational quantum eigensolver algorithm on a TIQI setup programmed in Qiskit. To achieve this goal a setup-dependent calibration of a backend for TIQI's eQual project is required. Ideally the calibration procedure can be developed in a general fashion such that this procedure can easily be implemented for other setups of the TIQI group.

Liberto's and my master's thesis also play a role in the efforts of the division for trapped ions of the ETHZ-PSI Quantum Computing Hub. One goal is to make quantum computers commercially available and also provide cloud access. The Qiskit ionpulse API will play a crucial role in these endeavours.

Bibliography

1. Preskill, J. Quantum Computing in the NISQ era and beyond. *Quantum* **2,** 79. ISSN: 2521-327X. https://doi.org/10.22331/q-2018-08-06-79 (Aug. 2018).
2. LaPierre, R. *Introduction to Quantum Computing* chap. Grover Algorithm. https://doi.org/10.1007/978-3-030-69318-3_12 (Springer, Cham, 2021).
3. Chang, W.-L. & Vasilakos, A. V. *Fundamentals of Quantum Programming in IBM's Quantum Computers* https://doi.org/10.1007/978-3-030-63583-1 (Springer, Cham, 2021).
4. Williams, C. P. *Explorations in Quantum Computing* https://doi.org/10.1007/978-1-84628-887-6 (Springer, London, 2011).
5. Cao, Y. *et al.* Quantum Chemistry in the Age of Quantum Computing. *Chemical reviews* **119,** 10856–10915. ISSN: 1520-6890. https://doi.org/10.1021/acs.chemrev.8b00803 (19 Oct. 2019).
6. Negnevitsky, V. *Feedback-stabilised quantum states in a mixed-species ion system* PhD thesis (ETH Zurich, 2018).
7. Marinelli, M. *Quantum information processing with mixed-species ion crystals* PhD thesis (ETH Zurich, 2020).
8. Qiskit Development Team. *Qiskit Terra API reference* https://qiskit.org/documentation/apidoc/terra.html (2021).

9. Gambetta, J. M., Motzoi, F., Merkel, S. T. & Wilhelm, F. K. Analytic control methods for high-fidelity unitary operations in a weakly nonlinear oscillator. *Phys. Rev. A* **83,** 012308. https://link.aps.org/doi/10.1103/PhysRevA.83.012308 (1 Jan. 2011).

10. McKay, D. C. *et al. Qiskit Backend Specifications for OpenQASM and OpenPulse Experiments* 2018. arXiv: 1809.03452 [quant-ph].

11. LaRose, R. Overview and Comparison of Gate Level Quantum Software Platforms. *Quantum* **3,** 130. ISSN: 2521-327X. http://dx.doi.org/10.22331/q-2019-03-25-130 (Mar. 2019).

12. Fingerhuth, M., Jayasinha, P. & Roy, A. S. *Open-Source Quantum Software Projects* https://github.com/qosf/awesome-quantum-software. Accessed June 30, 2021.

13. Heim, B. *et al.* Quantum programming languages. *Nature Reviews Physics* **2,** 709–722. ISSN: 2522-5820. https://doi.org/10.1038/s42254-020-00245-7 (Dec. 2020).

14. Fingerhuth, M., Babej, T. & Wittek, P. Open source software in quantum computing. *PLOS ONE* **13** (ed Mueck, L. A.) e0208561. ISSN: 1932-6203. http://dx.doi.org/10.1371/journal.pone.0208561 (Dec. 2018).

15. Khammassi, N. *et al. OpenQL : A Portable Quantum Programming Framework for Quantum Accelerators* 2020. arXiv: 2005.13283 [quant-ph].

16. Steiger, D. S., Häner, T. & Troyer, M. ProjectQ: an open source software framework for quantum computing. *Quantum* **2,** 49. ISSN: 2521-327X. https://doi.org/10.22331/q-2018-01-31-49 (Jan. 2018).

17. Svore, K. *et al.* Q#: Enabling scalable quantum computing and development with a high-level domain-specific language. *Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL2018.* http://dx.doi.org/10.1145/3183895.3183901 (2018).

18. Bichsel, B., Baader, M., Gehr, T. & Vechev, M. *Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics* in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Association for Computing Machinery, London, UK, 2020), 286–300. ISBN: 9781450376136. https://doi.org/10.1145/3385412.3386007.

19. Sørensen, A. & Mølmer, K. Quantum Computation with Ions in Thermal Motion. *Physical Review Letters* **82,** 1971–1974. ISSN: 1079-7114. http://dx.doi.org/10.1103/PhysRevLett.82.1971 (Mar. 1999).

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

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

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

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

Title of work (in block letters):

Enabling a quantum-gate-level interfacewith a trapped ion control system

Authored by (in block letters):

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

Name(s):	**First name(s):**
Stucki	Marco Erwin

With my signature I confirm that

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

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

Place, date	**Signature(s)**
Zürich, 08.12.2021	M. Stucki

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