ETH zürich

Alexander Ferk

Integrated Real-Time Phase Modelling for Trapped Ion Quantum Information Processing

Master's Thesis

Eidgenössische Technische Hochschule Zürich

Trapped Ion Quantum Information

Supervisors:
Martin Stadler
Jonathan Home

Zürich, October 2022

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):

Integrated Real-Time Phase Modelling for Trapped Ion Quantum Information Processing

Authored by (in block letters):

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

Name(s):	**First name(s):**
Ferk	Alexander

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, 17.10.2022	

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

Abstract

With increasing complexity in quantum algorithms more qubits and gates are required. Gate errors must be minimized, otherwise accumulated errors render the computation results unusable. In trapped ion quantum computing, gates can be performed by optically driving transitions using laser pulses. For laser driven gates, the phase of the pulse relative to the qubit determines the axis of rotation. However, off-resonant coupling to other transitions shifts the qubit frequency. For accurate, coherent control of qubit transitions, the phase of the qubit must be modelled in order to adjust the phase of the laser pulse accordingly. We present an integrated approach to this problem, modelling the phase of the qubit in real-time on the current control system and demonstrate the phase adjustment of the RF laser modulation on the control hardware.

Contents

1 Theory

In this chapter, we will at first look at the interaction between the atom and the laser light starting from the Hamiltonian in the electric dipole approximation. Then we will see how we can use a $^{40}Ca^{+}$ ion to encode a qubit and drive gates for Quantum Information Processing (QIP) operations. Finally, from the interaction with the laser, we will derive the effects on the frequency of the qubit. This lets us determine the phase between the qubit and our phase reference (usually the laser).

1.1 Atom Light Interaction

This section shall summarize the theory of atom-light interaction as well as introduce the notation used in the following sections. We will mainly follow the derivation from Fischer [5], Malinowski [11], Cohen-Tannoudji et al. [3], and Loudon [10].

As a starting point, we introduce the electric dipole Hamiltonian [5, 11]. This Hamiltonian is obtained by starting from the interaction of the vector potential with the current density of the atom, then rewriting in the *Göppert-Mayer gauge* and applying the *long-wavelength*[1] and *electric dipole approximation*[2] [3]. This gives

$$H_{dipole} = \sum_{\alpha} \frac{\boldsymbol{p}_{\alpha}^{2}}{2m_{\alpha}} + V_{coul} - \boldsymbol{d} \cdot \boldsymbol{E}(t) \tag{1}$$

with the summation performed over all charge indices α with momentum $\boldsymbol{p}_{\alpha}$ and mass m_{α}. The Coulomb potential V_{coul} contains the electrostatic energy of the electrons and the core. The interaction takes the form $H_{AF} = \boldsymbol{d} \cdot \boldsymbol{E}(t)$, with the dipole operator $\boldsymbol{d}$ and the electric field at the atom $\boldsymbol{E}(t)$.[5]

Ideally, we want a two level system as our qubit, with ground state $|g\rangle$ and excited state $|e\rangle$. In the dynamics of this two level system, the dipole operator $\boldsymbol{d}$ only couples states of opposite parity and can be identified as [2, 5]

$$\boldsymbol{d} = \boldsymbol{\mu}^{*}\sigma_{+} + \boldsymbol{\mu}\sigma_{-}$$

where the dipole matrix elements are $\boldsymbol{\mu} = \langle g | \boldsymbol{d} | e \rangle$, and the Pauli operators $\sigma_{+} = |e\rangle\langle g|$, $\sigma_{-} = |g\rangle\langle e|$. For this two level system, we obtain the Hamiltonian

$$H = H_0 + H_1 = \hbar\omega_g |g\rangle\langle g| + \hbar\omega_e |e\rangle\langle e| - (\boldsymbol{\mu}^{*}\sigma_{+} + \boldsymbol{\mu}\sigma_{-}) \cdot \boldsymbol{E}(t) \tag{2}$$

[1] Assumes that the separation of charges (≈0.1 nm for an atom) is much less than the wavelength of the light (>100 nm)

[2] Neglects higher order terms when expanding the electric scalar and vector potentials.

where we identify the Hamiltonian of the bare system as H_0 and the interaction Hamiltonian as H_1. We further assume the interacting light to be a plane wave [2, 5]

$$\boldsymbol{E}(t) = \frac{E_0}{2}\left(\boldsymbol{\epsilon} e^{-i(\omega t+\phi)} + \boldsymbol{\epsilon}^* e^{i(\omega t+\phi)}\right)$$

with the polarization vector $\boldsymbol{\epsilon}$, amplitude E_0, frequency ω and phase ϕ. We obtain

$$H_1 = -\frac{E_0}{2}\left(\boldsymbol{\mu}^*(\boldsymbol{\epsilon} e^{-i(\omega t+\phi)} + \boldsymbol{\epsilon}^* e^{i(\omega t+\phi)})\sigma_+ + \boldsymbol{\mu}(\boldsymbol{\epsilon} e^{-i(\omega t+\phi)} + \boldsymbol{\epsilon}^* e^{i(\omega t+\phi)})\sigma_-\right).$$

Here we apply a unitary transformation U into the rotating frame of ω yielding

$$H_1 = -\frac{E_0}{2}\left(\boldsymbol{\mu}^*(\boldsymbol{\epsilon} e^{-i\phi} + \boldsymbol{\epsilon}^* e^{i(2\omega t+\phi)})\sigma_+ + \boldsymbol{\mu}(\boldsymbol{\epsilon} e^{-i(2\omega t+\phi)} + \boldsymbol{\epsilon}^* e^{i\phi})\sigma_-\right)$$

where we apply the *rotating wave approximation*, neglecting the fast rotating terms with a frequency of 2ω [5]. This yields for the full Hamiltonian

$$H \approx \hbar\frac{\delta}{2}\sigma_z - \hbar\frac{\Omega}{2}\left(e^{-i\phi}\sigma_+ + e^{i\phi}\sigma_-\right) \tag{3}$$

Here we identify the detuning $\delta = \omega - \omega_{eg} = \omega - (\omega_e - \omega_g)$ and the *Rabi frequency*

$$\Omega = \frac{E_0}{\hbar}\boldsymbol{\mu}^*\boldsymbol{\epsilon} = \frac{E_0}{\hbar}|\langle g|\,\boldsymbol{d}\cdot\boldsymbol{\epsilon}\,|e\rangle|$$

where we have chosen the phases of $\boldsymbol{\mu}$ and $\boldsymbol{\epsilon}$ such that Ω is real.[5, 11]

1.2 Atomic Level Structure

Trapped ion experiments begin with a (neutral) atom source (e.g. an oven[3] or an ablation target[4]). The neutral atoms are then ionized with a photo-ionization laser, cooled (with e.g. Doppler cooling) and trapped in an electric field. One common atom to employ is ^{40}Ca, which due to its electron configuration of [Ar]4s2 yields one valence electron after ionization (spin quantum number S =1/2).

Figure 1 shows the structure of a $^{40}Ca^+$ atom with the electronic levels denoted as L_J (total orbital angular momentum L and total angular momentum J). In addition, Zeeman effect splits each level L_J into sublevels of $m_j = (-J, -J+1, ..., +J)$.[11]

Dipole transitions can be driven according to Equation 3. These transitions follow the *dipole selection rules*:

$$\Delta L = \pm 1;\ \Delta J = 0, \pm 1;\ \Delta m_j = 0, \pm 1$$

[3] A piece of the chosen element is heated until atoms are emitted

[4] A powerful laser pulse ablates the atoms from a target made of the chosen element which can also ionize the atoms directly.

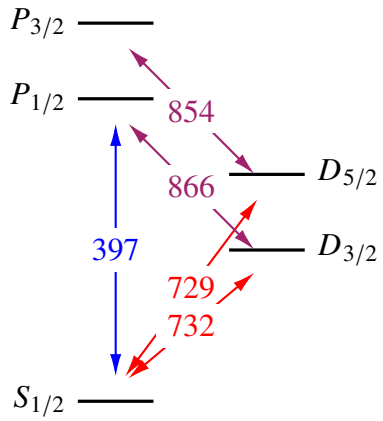


Figure 1: Electronic structure (states denoted as L_J) of the Ca+ atom with laser wavelengths (in nm). Dipole transitions are marked in ■ and ■, quadrupole transitions are marked in ■. Figure partially from [11]

Hence, the $S_{1/2} \leftrightarrow D_{3/2}$ and $S_{1/2} \leftrightarrow D_{5/2}$ transitions are dipole forbidden, which leads to longer lifetimes of ≈ 1.1 s, making these levels suitable for storing information. [11]

However, these transitions couple to the next order expansion of the field (see [8] for a derivation). These quadrupole transitions follow the selection rules:

$$\Delta L = 0, \pm 2;\ \Delta J = 0, \pm 1, \pm 2;\ \Delta m_j = 0, \pm 1, \pm 2$$

These allow us to access the longer lived $D_{3/2}$ and $D_{5/2}$ levels. But they do not allow $D_{5/2} \leftrightarrow P_{1/2}$ decay. This makes the $D_{5/2} \leftrightarrow S_{1/2}$, together with the cycling transition $S_{1/2} \leftrightarrow P_{1/2}$ driven by a 397 nm laser for readout (state dependent fluorescence), ideal to encode a qubit. [11]

1.3 Single Qubit Gates

For optical qubits[5], we have seen that we can use the the states $|e\rangle = D_{5/2}$ and $|g\rangle = S_{1/2}$, with the $S_{1/2} \leftrightarrow P_{1/2}$ transition for readout. Initialization is performed by optical pumping into the ground state[6].

For the purposes of quantum computing, we want to implement a rotation[15, 6]

$$R(\theta, \phi) = e^{i\frac{\theta}{2}(e^{i\phi}\sigma_+ + e^{-i\phi}\sigma_-)} = \mathbb{1}\cos\theta/2 + i(\sigma_x \cos\phi - \sigma_y \sin\phi)\sin\theta/2 \tag{4}$$

5 Other options to encode a qubit in an atom/ion exist, e.g. Zeeman qubits, Rydberg states, ...

6 Population from slow decaying levels may be pumped into faster decaying levels to improve the initialization time. Furthermore, the ion is trapped in a (harmonic) electrical potential, which must also be cooled to the ground state. Refer to [5] and [11] for further details.

allowing us to rotate towards any angles (θ, ϕ) on the Bloch sphere. We can now utilize the resonantly driven quadrupole interaction to perform single qubit gates. From the time independent Hamiltonian in Equation 3 we can obtain the unitary transformation [11]

$$U(t) = e^{-iHt/\hbar} = \exp\left(-i\frac{\delta t}{2}\sigma_z + i\frac{\Omega t}{2}(e^{-i\phi}\sigma_+ + e^{i\phi}\sigma_-)\right). \tag{5}$$

If we now drive this interaction on resonance ($\delta = 0$), we obtain the desired rotation as

$$U(t) = \exp\left(i\frac{\Omega t}{2}(e^{-i\phi}\sigma_+ + e^{i\phi}\sigma_-)\right). \tag{6}$$

with $\theta = \Omega t$, and the phase ϕ between laser and qubit. From this general rotation, we can obtain rotations around the x- and y-axis as

$$R_x(\theta) = R(\theta, \phi = \pi);\ R_y(\theta) = R(\theta, \phi = \pi/2) \tag{7}$$

where we can represent z-rotations in terms of phase-shifts for the following x- and y-rotations (*virtual Z gate*). [11, 12, 6] We can control these parameters by modulating the laser beam through an Acousto-Optic Modulator (AOM). The Radio Frequency (RF) phase, relative to the qubit, changes ϕ and the on-time and amplitude of the pulse modify $\theta \propto t\Omega \propto tE_0$. [6, 4]

1.4 AC Stark Shift

As we have seen above, driving gates requires precise control of the phase of any applied pulses. Errors in the phase lead to rotations around the wrong axis and therefore logical errors. So far we only considered the on-resonant case with the states of the bare Hamiltonian $|e\rangle$ and $|g\rangle$. However, due to the interaction with the laser, these states are no longer eigenstates of the Hamiltonian. We consider a two level system, as shown in Figure 2. [5, 11]

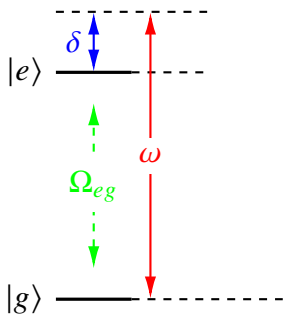


Figure 2: Off resonant drive ω in a two level system $\{|e\rangle, |g\rangle\}$ with detuning δ.

We again start from the Hamiltionian (3), where we set $\phi = 0$ and obtain

$$H = \frac{\hbar}{2}\begin{pmatrix}\delta & \Omega_{eg}\\ \Omega_{eg} & -\delta\end{pmatrix} = \frac{\hbar}{2}\begin{pmatrix}\delta & 0\\ 0 & -\delta\end{pmatrix} + \frac{\hbar}{2}\begin{pmatrix}0 & \Omega_{eg}\\ \Omega_{eg} & 0\end{pmatrix} = H_0 + H_1 \tag{8}$$

For this simple system, we can diagonalize H and obtain

$$\tilde{E}_g/\hbar = -\frac{1}{2}\sqrt{\Omega_{eg}^2 + \delta^2} \approx -\frac{\delta}{2} - \frac{\Omega_{eg}^2}{4\delta}, \quad \delta \gg \Omega_{eg} \tag{9}$$

where we expanded for $\delta \gg \Omega_{eg}$ and identify the generalized Rabi frequency $\tilde{\Omega}_{eg} = \sqrt{\Omega_{eg}^2 + \delta^2}$. In this expansion, we can see an additional shift of $\Delta\tilde{E}_g = -\frac{\Omega_{eg}^2}{4\delta}$ depending on the off-resonant drive. This means, if the qubit is driven off-resonantly, its frequency changes, resulting in a phase difference between the bare (undriven) qubit and the dressed (driven) qubit. We can also express the energy shift as a function of intensity, using [5]

$$I(r) = \frac{\epsilon_0 c |E(r)|^2}{2} \tag{10}$$

and obtain

$$\Delta E_g^{(2)} = \frac{|\langle e|\, \boldsymbol{d}\cdot\boldsymbol{\epsilon}\, |e\rangle|^2}{2\hbar\epsilon_0 c}\frac{I(r)}{\delta} \tag{11}$$

This means the energy shift we obtained in Equation 9 leads to a frequency change in the qubit transition depending on the intensity of the drive, given $\delta \gg \Omega_{eg}$.

We have now seen that an off-resonant drive produces a shift in frequency between two levels. If we drive any transition in an atom, due to the presence of other transitions (both dipole and quadrupole), we are always driving these *spectator transitions* off-resonantly. We will now consider the three-level system shown in Figure 3 to discuss the effects of the off-resonant coupling on the driven transition.[11]

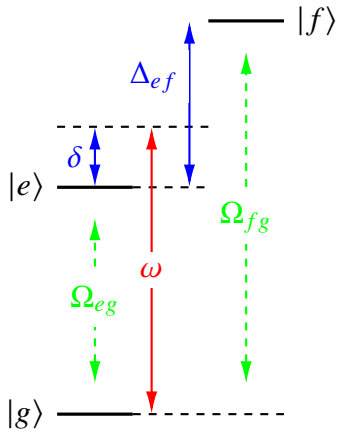


Figure 3: AC Stark shift with a spectator transition. To drive the transition $|g\rangle \leftrightarrow |e\rangle$ with Rabi frequency Ω_{eg} the drive frequency ω is on resonance with the transition (detuning $\delta = 0$). This drive, off resonant by Δ_{ef}, produces an AC Stark shift on the $|g\rangle \leftrightarrow |f\rangle$ transition. We assume no coupling between $|e\rangle$ and $|f\rangle$. Figure modified from [11]

If we want to drive Rabi oscillations between $|g\rangle \leftrightarrow |e\rangle$, we set the detuning $\delta = 0$. Then, the laser drive couples off-resonantly to the transition $|g\rangle \leftrightarrow |f\rangle$ with detuning Δ_{ef}. Furthermore, we assume that there is no coupling $|e\rangle \leftrightarrow |f\rangle$ in the system. The Hamilton describing the interaction between the three levels in the frame of the drive is then given as

$$H = \hbar \begin{pmatrix} 0 & \Omega_{eg}/2 & \Omega_{fg}/2 \\ \Omega_{eg}/2 & -\delta & 0 \\ \Omega_{fg}/2 & 0 & \Delta_{ef} - \delta \end{pmatrix} \tag{12}$$

$$= \hbar \begin{pmatrix} 0 & \Omega_{eg}/2 & 0 \\ \Omega_{eg}/2 & -\delta & 0 \\ 0 & 0 & \Delta_{ef} - \delta \end{pmatrix} + \hbar \underbrace{\begin{pmatrix} 0 & \Omega_{fg}/2 \\ \Omega_{fg}/2 & 0 \end{pmatrix}}_{\{|g\rangle,|f\rangle\}} = H_0 + H_1 \tag{13}$$

where we set $E_g = 0$. With this more complex system, we solve H_0 and treat H_1 as a perturbation and obtain for E_g

$$\Delta E_g^{(2)}/\hbar = \frac{|\langle f^{(0)}| H_1 |g^{(0)}\rangle|^2}{E_g^{(0)} - E_f^{(0)}} = \frac{\Omega_{fg}^2/4}{\delta - \Delta_{ef}}$$

With our assumption that $|e\rangle \leftrightarrow |f\rangle$ are not coupled, the $|f\rangle$ level will only produce a shift on E_g. If we assume the transition to the third level to be far off-resonant, we can approximate the energy shift $\Delta\tilde{E}_{eg}$ as

$$\Delta\tilde{E}_{eg}/\hbar \approx -\delta + \frac{\Omega_{fg}^2}{4\Delta_{ef}}, \quad \Delta_{ef} \gg \delta \tag{14}$$

Hence, for $\Omega_{fg} = 0$ we obtain a dressed state splitting of $\Delta E = \hbar\Omega_{eg}$ with the resonance at $\delta \approx 0$, similar to the two level system above. However, for $\Omega_{fg} > 0$ we see the resonance of the transition shifts to $\tilde{\delta} \approx \frac{\Omega_{fg}^2}{4\Delta_{ef}}$.[11]

This means, in the presence of spectator transitions, we need to adjust the drive frequency to resonantly drive Rabi oscillations. For the $^{40}Ca^+$ ion, with several transitions present, we need to consider all driven *dipole* and *quadrupole* transitions in the total shift. For an ion in a (harmonic) trap potential we need to additionally consider motional sideband transitions.[19]

The total Stark shift (often on the order of a few kHz) can be measured via a Ramsey experiment with an off-resonant pulse driving the desired transition. Figure 4 shows the AC Stark shift measured for the $S_{1/2} \leftrightarrow D_{5/2}$ transition.[7]

The major contribution stems from the dipole transitions, which can be compensated by a simultaneous laser pulse at another frequency [7] or positioning the ion in a standing wave [19]. Shifts from the quadrupole transitions are more difficult to compensate with the latter method, but could be minimized by increasing the Zeeman splitting with higher magnetic fields [19].

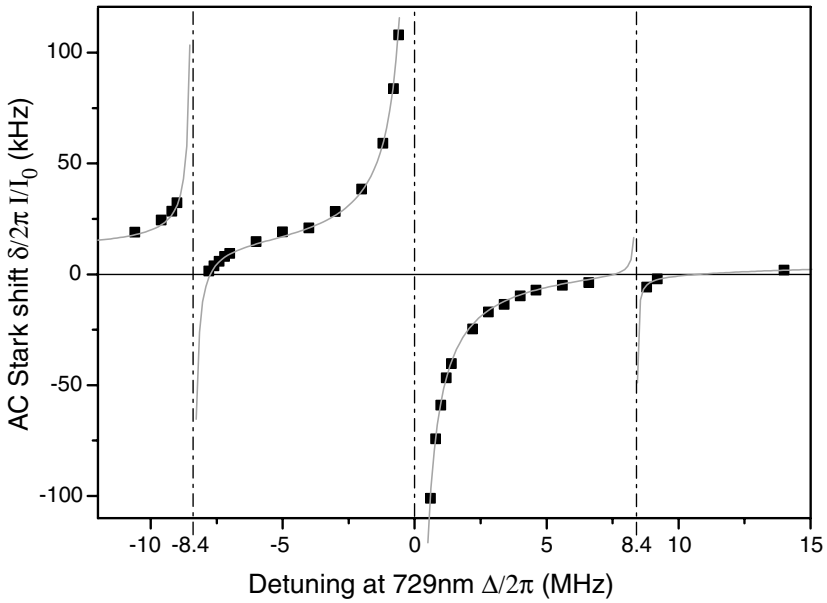


Figure 4: Measured AC-Stark shift data (squares) for a $^{40}Ca^{+}$ ion normalized according to the measured laser power I/I_0. The calculated Stark shift δ (line) is plotted vs. the detuning Δ from the $S_{1/2} \leftrightarrow D_{5/2}$ resonance, with the divergences due to the $(m = 1/2) \rightarrow (m' = -5/2, -1/2, +3/2)$ resonances (from left to right). Figure from Häffner et al. [7].

2 Trapped Ion Control System

QIP with trapped ions requires coordinated Direct Current (DC), RF and digital signals for trapping, coherent control and readout. The control system used in the Trapped Ion Quantum Information (TIQI) group employs a hybrid Central Processing Unit (CPU) / Field Programmable Gate Array (FPGA) system, where a sequencer on the FPGA is used for real time processing while readout processing and decision making is done on the CPU. This section will give an overview over the components of the control system. Then, we will see how pulse sequences are structured and compressed on the low-level systems. Finally, we will see how coherent pulses are currently achieved. [14, p26ff][17]

2.1 Overview

An overview of the systems components is given in Figure 5. Ions in the ion trap (lower left) are trapped in a dynamical electrical field. RF electrodes are driven by an RF source with frequencies on the order of 10s of MHz (not pictured). DC electrode voltages are controlled by the Direct Ethernet-Adjustable Transport Hardware (DEATH) Arbitrary Waveform Generators

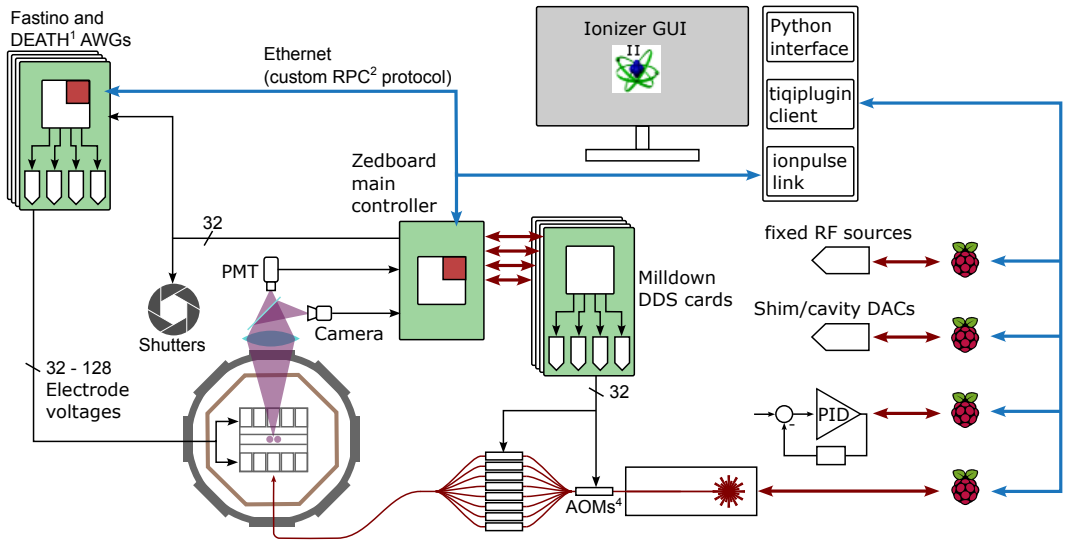


Figure 5: General overview and connections between the components of the control system. See text for details. Figure adapted from Martin Stadler following [14, p32]

(AWGs) and vary on longer timescales in order to perform tasks such as transport of ions through an array of traps. Coherent control and cooling of the ions is achieved with lasers modulated (pulsed) by Acousto-Optic Modulators (AOMs) , where the AOMs are controlled with RF signals from the Direct Digital Synthesis (DDS) cards. Readout with PhotoMultiplier Tubes (PMTs) or camera[7] is controlled directly via the Zedboard, which also acts as the main controller and provides the interface to the control Personal Computer (PC) hosting the Ionizer Graphical User Interface (GUI). Asynchronous devices, such as Rasperry Pis (RPis) interacting with other parts of the experimental setup are driven with a custom Python client (tiqi-plugin) directly from the control PC.

With this CPU centered design, experiments are orchestrated from the Zync System on Chip (SoC)[22] on the Zedboard [21] acting as a main controller. Experiments are written in C++, compiled, linked to a single executable and then programmed (together with the synthesized FPGA design) onto the Zedboard (or in many cases run with the Xilinx Vitis debugger). The user then interacts with the IonizerGUI to set parameters, start experiments, and collect and save data. The IonizerGUI on the control PC can also interact with several asynchronous devices in other parts of the experimental setup. When an experiment is started from the PC, the Zync-SoC creates and communicates a series of instructions to the sequencer on the Znyc-FPGA part (Pulseway) and to the sequencers on DDS cards (one sequencer (called idecoder) for each channel) via the Back Plane (BP). Each DDS card contains four RF channels with one AD9910 [1] each and a Spartan 6 [16] FPGA programmed with the synthesized FPGA

7 Pre-Processing of the camera is done on another FPGA, where only the binned counts are transmitted to the Zedboard.

design (Minotaur). To start the sequence, Pulseway sends the trigger to start all sequencers simultaneously. The FPGA design for both Zedboard and DDS cards is written in the Hardware Description Language (HDL) Verilog. For a further in depth description of the components of the control system, developed by Negnevitsky [14] and Stadler [17] and others, refer to [14, 17].

2.2 Experimental Sequence Structure

An experiment consists of n shots. Each shot executes the same sequence (here called "main sequence"). Creating a sequence requires a set of instructions for the above described sequencers to operate. These instructions have to be stored on each FPGA in the "working memory" of each sequencer. However, memory on the FPGAs is limited which leads to the need of compressing an experimental sequence. The control system achieves this compression through re-use of events, sequences and loops. Events (Edges) are the smallest building blocks - a collection of parameters and a wait time. After the wait time, an action is triggered, e.g. the RF channel is switched on with certain frequency, phase and amplitude (FPA) parameters. Wait times can also be added by adding wait blocks. These will not change the output parameters of the system and can be used to synchronize pulses of different lengths between channels.

To demonstrate how compression helps conserve memory, we analyse the example sequence shown in Figure 6. If we were to simply execute the sequence in a linear fashion, storing all

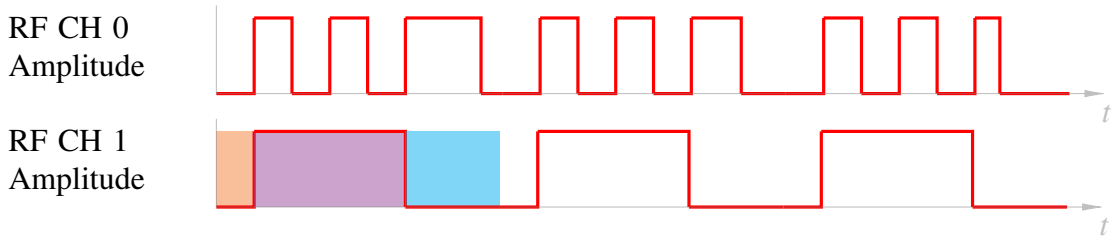


Figure 6: Simple control sequence on two RF Channels with uniform maximum amplitude. Two 'RF Edges' and one 'RF Wait' are marked, turning the signal on ■, off ■ and leaving it in the current state ■. (see text)

edges, we would need 9 elements for CH 1 and 21 elements for CH 0. In this simple example, we are only looking at the amplitude and time. To fully parametrize a pulse, FPA and time (duration or wait time until the change) must be set, which we would have to individually store for each element, leading to 4 parameters per edge. However, we can identify repeating elements within this sequence. All three marked elements in Figure 6 are just repeated 3 times, referencing the same parameters. This means our linear sequence for CH 1 still needs to store 9 references to the pulses, but not 36 FPA parameters (for all 9 pulses) but only 12 parameters. Furthermore, we can collect these three pulses into one loop sequence, which we repeat 3

times. Then, our main sequence for CH 1 only contains one looped sequence with 3 iterations referencing the three pulses. For CH 0 we can collect the first two identical pulses (4 Edges) and the first edge of the next pulse into one linear sequence of 5 edges, as shown in Figure 7.

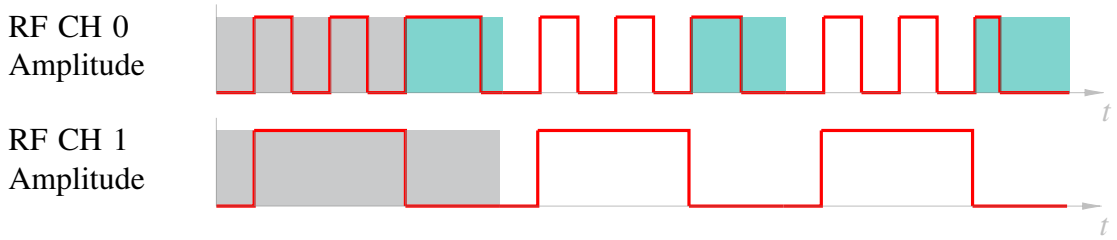


Figure 7: Simple control sequence on two RF Channels with uniform maximum amplitude. Repeating sequences are marked in ■, two parametrized edges are marked in ■. (see text)

However, we can not collect them in a loop, as one of the pulses is changing over the three repeated cycles. For this case, our control system features parametrized loops. Parameters can be updated during a loop ("looped parameter"), resulting in a changing behaviour for each loop cycle. This means, with an 'intelligent' sequencer, we can compress this sequence on CH 0 from a collection of 84 parameters down to

- 1 loop with 6 parameter (time) updates[8],
- 2 'ON' RF Edges (fixed),
- 2 'OFF' RF Edges (one fixed, one with looped parameters) and
- 1 RF Wait (with looped parameters[9]).

Therefore we only need to store the 4 parameter updates, as well as parameters for 5 edges, a total of 24 parameters, giving us a reduction to below 30% even for such a short sequence. However, this makes the sequencer more complex. We need to store the references to the elements (Edges, ...), be able to count iterations and be able to keep track of which 'subsequence' we are currently executing.

Another important element for recent QIP is measurement based feedback. For this purpose, the sequencers include 'forks' to perform actions conditioned on a readout. A simple example with a measurement dependent wait time on the second pulse is given in the Figure 8.

Here, depending on a measurement result, a different subsequence 'path' is chosen. The fork therefore includes a conditional (e.g. a certain threshold reached) and a measurement channel.

[8] Note that all other parameters except the time can already be passed with the first iteration. For all following iterations, only the updated parameters need to be overwritten. After the last iteration, the updated parameters have to be reset to the initial value.

[9] This wait time is variable to keep the overall time of the pulse (RF OFF + RF Wait) constant.

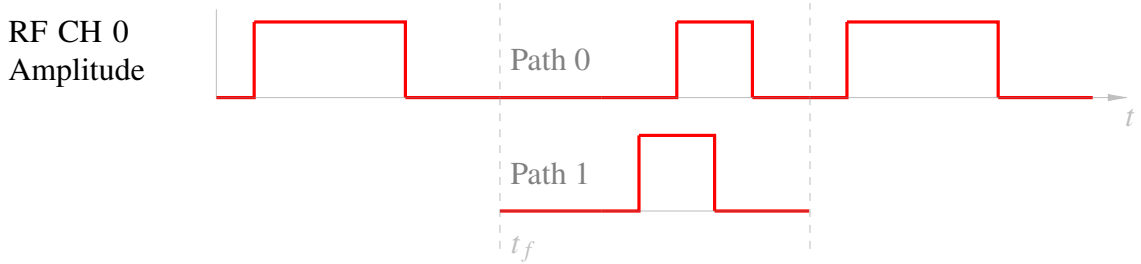


Figure 8: Simple control sequence with a fork depending on a readout result at time t_f (see text).

For this, the Zedboard SoC processes the raw readout counts from the PMT or camera and makes a decision based on the outcome, i.e. choosing a 'path' (sequence) to execute. This decision is then communicated to the sequencers. While this approach also has limits, it is a good compromise between latency (due to re-sending parts of the sequence) and required memory.[14, 17]

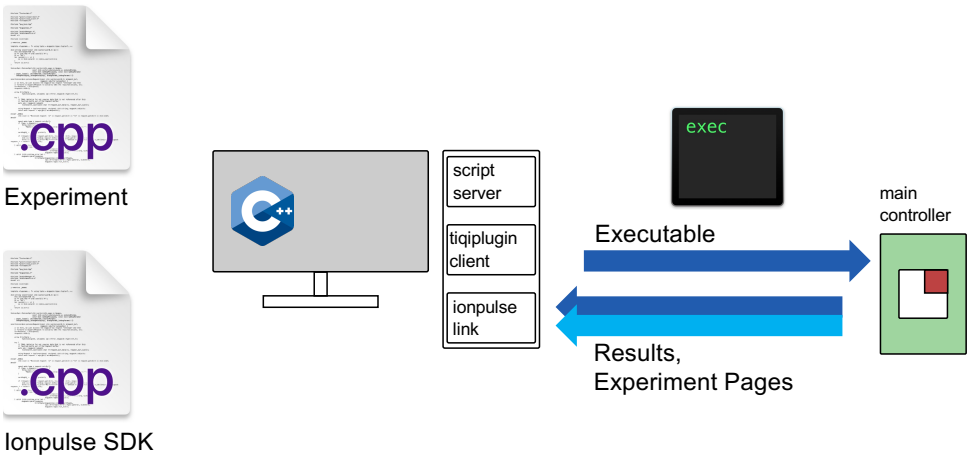
2.3 Experimental Sequence Generation

From the sequence elements discussed in the last section, we now have to create an experimental sequence. While it is possible to create a sequence from just the basic subblocks, it is far more convenient to have a higher level interface for the user.

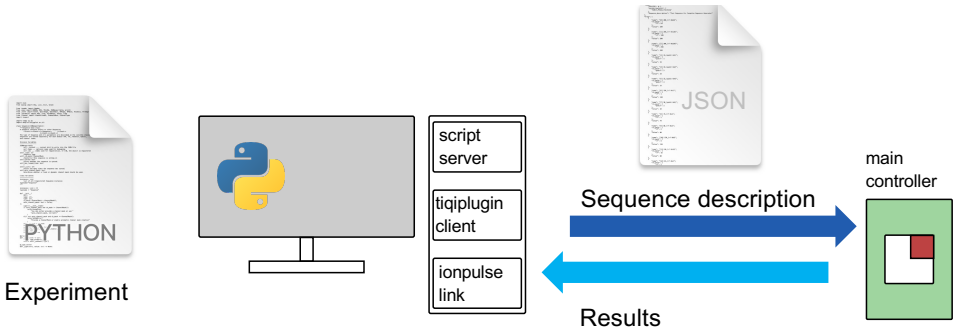
This interface is provided in the C++ framework of Ionpulse SDK. With this *compilation based* approach, the experiments are written in C++ and then compiled and linked with the Ionpulse SDK and programmed onto the Zedboard SoC.

However, writing experiments in C++ and programming the Zedboard for any (non parametrized) changes to the sequence is inconvenient. Quick trial and error is not possible due to the compile time, and when uploading a new sequence, the Zedboard is reset. As an alternative interface to the Zedboard, a different, *description based* approach with a high level Python library has been developed. The user writes the sequence using Pycrystal [13] (high level library) which in turn uses the Ionpuse Sequence Generator to provide sequence elements understood by the control system. The Ionpulse Sequence Generator[10] has been initially developed by Stucki [18] as an interface between Qiskit and the control system. Figure 9 illustrates both approaches.

[10] The Ionpulse Sequence Generator has been developed earlier by Martin Stadler and Marco Stucki and was then modified during this thesis while keeping the original interface to the high level intact. Additional modifications to the generator were then made by Mose Müller, introducing features for pycrystal and improving the speed of the library.



(a) An experiment, compiled and linked with Ionpulse SDK library, is programmed onto the Zedboard (main controller). Experiments (pages) are communicated to the control PC, which triggers an experiment. When the sequence is finished, the Zedboard sends the results to the control PC.



(b) The Ionpulse Sequence Generator (Python) on the control PC generates a sequence description which is sent to the Zedboard (main controller). This sequence is triggered from the main PC, executed and results are communicated back to the control PC.

Figure 9: Paradigm change from compilation-based (a) experiments to description-based experiments (b). C++ Logo [9]. Python Logo [20]. File images from macOS 12.6.

With the Ionpulse Sequence Generator, the experimental sequence is written in Python and can be exported in an intermediate format. For this interface format, we use JavaScript Object Notation (JSON) (for human readability), but other (binary) formats are also possible. The resulting description is then sent to the Zedboard, where it is interpreted and translated to driver instructions for the various parts of the control system. The software on the Zedboard SoC (Ionpulse) only needs to be programmed once and just provides an interface which accepts this interface format.

2.4 Coherent Pulses

As we have seen, coherent pulses and good control of the phase are important to drive gates on trapped ions. As discussed in 1.4, the qubit frequency ω_{eg} and the drive frequency ω of the laser are not always the same. This difference in frequency translates to accumulation of a difference in phase $\Delta\phi$ which affects subsequent pulses. Therefore, if we implement a gate, which requires a fixed phase ϕ_{pulse} relative to the qubit phase ϕ_q, we need to adjust the phase of the pulse by $\Delta\phi$ prior to the gate in order to ensure that the phase is correct. Figure 10 shows an example sequence with the qubit frequency evolution.

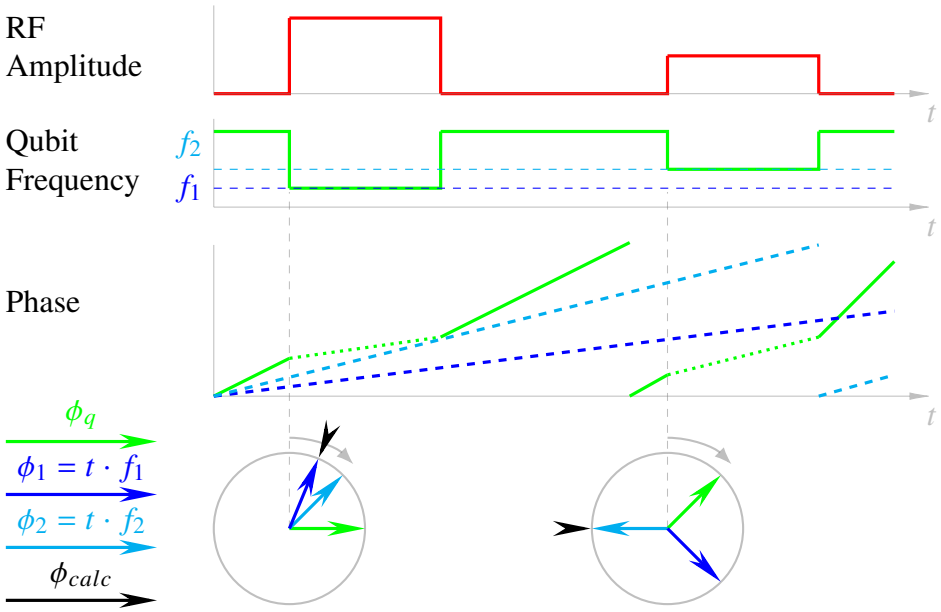


Figure 10: Control Sequence consisting of two RF Pulses with different amplitude shifting the frequency of a qubit. Phases are indicated as an integrated frequency over time and a phase diagram at the start of each pulse. The phase of the qubit ϕ_q differs from both drive phases ϕ_1 and ϕ_2. ϕ_{calc} indicates the phase calculated by the control system (see [14, p42]), multiplying elapsed time t with the RF frequency $f_N^{(RF)}$ of the current pulse N.

Due to AC Stark shifts, the qubit frequency f_q is shifted proportional to the intensity of the laser light, which is dependent on the RF amplitude, via the modulation of the AOM. In order to compensate for it, this shift must at first be calibrated. This can be performed by applying a pulse with low intensity and measuring the offset for the same pulse with higher intensity.

In the current control system, the DDS card calculates ϕ_{calc} (see [14, p42]), by multiplying the

elapsed time t with the current RF frequency $f_N^{(RF)}$.

$$\phi_{calc} = f_N^{(RF)} t = f_N^{(RF)} \sum_{n=0}^{n=N-1} t_n \tag{15}$$

The DDS card then calculates the phase of the pulse as

$$\phi''_{pulse} = \phi_{pulse} + \phi_{calc} \tag{16}$$

and outputs ϕ''_{pulse} to the DDS chip. In this simple picture, the calculated phase ϕ_{calc} has an error of

$$\Delta\phi_{err} = \sum_{n=0}^{n=N-1} (f_{qn} - f_N^{(RF)}) t_n + \Delta\phi_n, \tag{17}$$

where we multiply the frequency differences to the current frequency $f_N^{(RF)}$ from each pulse $(f_{qn} - f_N^{(RF)})$ with the respective pulse time t_n to obtain the phase difference for each pulse. $\Delta\phi_n$ is added for e.g. shaped pulses with varying amplitude and therefore non-constant f_{qn}, where $\Delta\phi_n = \int_{t_n}^{t_{n+1}} f_q(t)dt$ and $f_{qn} = 0$. Experimentally, $\Delta\phi_n$ of a particular shaped pulse is usually determined directly. The phase shift of the qubit is described by

$$\Delta\phi = \sum_{n=0}^{n=N-1} f_{qn} t_n + \Delta\phi_n = \phi_{calc} + \Delta\phi_{err}. \tag{18}$$

To compensate for the error in ϕ_{calc}, we can adjust the phase of the pulse transmitted to the DDS card to

$$\phi_{pulse} \rightarrow \phi'_{pulse} = \phi_{pulse} + \Delta\phi_{err} \tag{19}$$

resulting in a phase output from the DDS card of

$$\phi''_{pulse} = (\phi_{pulse} + \Delta\phi_{err}) + \phi_{calc} \tag{20}$$

However, as we have seen in subsection 2.2, this makes almost all coherent pulses unique and therefore does not allow for compression of pulses. This becomes even worse for forks, where each path has a different sequence of pulses. Here, we need to pre-calculate the phase for each path, having to store all phases for all paths in memory. At the end of the fork paths (as shown in Figure 8), each path can result in a different phase difference $\Delta\phi$. Of course, we need to take $\Delta\phi$ into account for the following pulses, effectively extending the paths until the end of the whole sequence. This scheme is either very memory (and computationally) intensive for forks or leads to higher latency when selecting a path, as phases have to be recalculated and updated.

3 Phase Tracker Implementation

3.1 Design Considerations

For coherent operations, qubit gates, and therefore RF events, need to take the accumulated frequency and phase shifts of the qubit during the sequence into account. In the following, I will discuss four possible approaches:

Pre-calculating the qubit phase for the whole sequence

Incremental pre-calculating updates of the qubit phase for each gate

Calculation with a separate sequencer of the qubit phase in real-time

Calculation with an RF sequencer of the qubit phase in real time

Currently, the *pre-calculating* approach is implemented. It adapts the phase for each gate, as discussed in 2.4. This approach has two main disadvantages. First, events with different phase shifts can not be compressed efficiently, making long sequences unfeasible (the sequencers on the DDS cards are currently limited to approx. 160 unique pulses)[11]. Second, pre-calculating phases with forks is computationally expensive as it requires each path to be computed. Furthermore, all following sequences must either be split into paths or all phases must be updated at the beginning of the fork increasing the latency (as described in subsection 2.4).

The *incremental pre-calculating* approach requires an additional phase accumulator for each qubit on the FPGA. Then, from the sequence of phase and frequency shifts the incremental shifts for each event can be pre-calculated. Each event updates the phase accumulator and retrieves the current qubit phase from it. This overcomes both disadvantages of the first approach. Events can be compressed with the same parameters where updates can, for example, be stored in a First In First Out (FIFO) accessed by each event. Forks can be treated by applying an update before the beginning and at the end of each fork path, with the accumulator carrying the phase from the chosen path. The major disadvantage of this system is that it limits access to the accumulator to one channel during a time-slot. For example, we want to apply two events starting at the same time t on two different channels to the same qubit (e.g. two AOMs switching on). Then, these events may have different wait times $t_0 < t_2$. Phase updates on the accumulator can then be ordered with t_0 and t_2, resulting in the correct behaviour. However, a third event on the first channel starting at t_1 with $t_0 < t_1 < t_2$ leads to two issues. This sequence is depicted in Figure 11.

The first issue is parallel updates of the accumulator. Updating the accumulator from two different channels will result in an incorrect phase for the third event in the example above. This can be overcome by limiting the updates of the accumulator to one channel, limiting the overall capabilities of the system. An alternative is splitting the updates into two parts, an

[11] Note that this can be, and is in a recent implementation by Martin Stadler, overcome by updating parameters during the sequence. This however produces additional communication frames on the backplane.

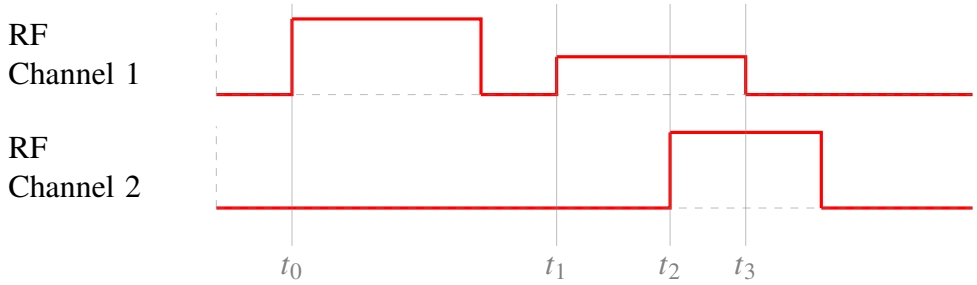


Figure 11: Example sequence for *incremental pre-calculation* with $t_0 < t_1 < t_2$. Updates and reads from the phase accumulator at the start of each RFEdge will cause an incorrect phase on channel 2. See text for further details.

update of the accumulator ($\Delta\phi([t_0, t_1])$) and an update before the event ($\Delta\phi([t_1, t_2])$), to be added to the phase of the accumulator to get the phase offset for the event.

The second issue stems from the mitigation of the first. We need to pre-calculate the phase offset for the event on channel two for the time interval $[t_1, t_2]$. The phase offset on channel two needs to take all offsets from channel one into account. This creates dependencies between sequences on separate channels and increases the complexity of the pre-calculation. Otherwise, if a channel is reading the accumulator value at an intermediate time, the accumulator value will not represent the correct qubit phase. In other words, with this scheme, the accumulator only represents the correct qubit phase only directly after an update. This limits the overall capabilities of the system, as this system would either need to handle multiple sources for the qubit phase or limit itself to one source.

Calculation with a separate sequencer requires a sequencer and a phase accumulator for each qubit. The sequencer will then feed the accumulator with the sequence of phase and frequency shifts for the qubit. An RF event can then read the phase from the accumulator of its target qubit. This approach requires more hardware than the previous approach but but allows us to run an arbitrary phase and frequency sequence on the qubits, completely decoupled from the rf sequences. This also enables us to use the same compression and architecture already used for RF sequences.

Similarly, *calculation with an RF sequencer* uses the already existing sequencers, which orchestrate the RF sequence, on the DDS cards to generate the frequency sequence for the qubit. This requires adding additional instructions for these sequencers to address changes in qubit frequency on top of setting the RF parameters. This results in potentially less hardware usage but has the disadvantage that qubit frequency changes will put additional load on the sequencers and must be synchronous to RF events.

We chose to perform the phase *calculation with a new sequencer*, as it allows us to model the full phase evolution of the qubit. Additionally, if implemented as a separate module, this design remains easily portable to future systems.

3.2 Design Overview

Implementing the phase tracker in the existing design, as described in Figure 5, requires changes to the different logical layers of the control system. Figure 12 provides an overview over the layers modified and added. The following sections will then treat the changes in each layer in depth.

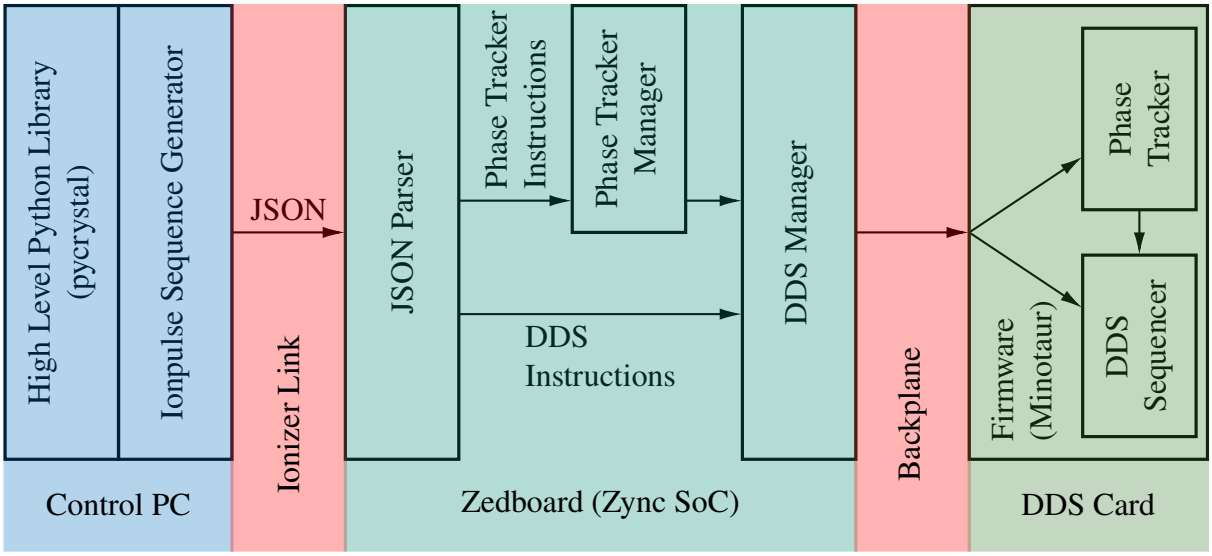


Figure 12: Modified layers of the control system. Sequences written by the user in Python are translated to JSON, parsed by the control system and then split into two paths for phase tracker instructions and RF instructions. These are then passed on to the DDS card via the DDS manager and executed on the phase tracker and DDS sequencer to produce an RF sequence.

The user writes the sequence in a high level python library (pycrystal, refer to Müller [13] for a description of this library), which includes setup information and uses the Ionpulse Sequence Generator[18] to translate the sequence information into objects which can be transferred to and interpreted by the control system. These objects are then exported to JSON and parsed[12] by the main controller on the Zedboard, generating the driver instructions and configuration. Phase events are passed on to the Phase Tracker Manager and translated to low-level instructions. They are forwarded to the DDS Manager, which sends the instructions to Minotaur where they are passed on to the phase trackers themselves. RF events are directly passed to the DDS Manager, translated to low-level instructions and sent to the DDS boards where they are forwarded to the control unit of the respective RF channel. These modifications are based on,

[12] The JSON Parser was originally written by Martin Stadler and has been largely rewritten in the scope of this thesis, keeping the driver interface intact.

include and modify existing parts of the control system mainly designed by Negnevitsky [14] and Stadler [17].

3.3 Python Sequence Generation - Ionpulse Sequence Generator

The Ionpulse Sequence Generator has been initially developed by Stucki [18] as an interface between Qiskit and the control system. The generator is part of a paradigm change for writing experiments within the control system, as discussed in 2.3.

The JSON generation and different aspects of the generator have been adapted while keeping (mostly) the same python interface functions.

The identification of sequence elements has been changed from names (strings) to array indices for parameters and sequence elements (events, sequences), in order to represent the structure on the Ionpulse SDK side and improve performance. This allows us to directly use those indices to e.g. find events in a sequence without searching in a map.

In addition to the already mentioned changes to sequence element identification, changes necessary for controlling the phase tracker have been made. The `QubitEdge` (qubit event) has been adapted. It takes frequency, phase, time, qubit address and an optional phase tracker reset, which are passed through the control system to control the phase tracker of the specified qubit.

3.4 Interface Format - JSON Structure

As mentioned above JSON is used as an interface format. JSON uses `objects` with `key:value` pairs to store information. Sequence objects discussed in 2.2 (e.g. RF event, linear sequence, loop, fork, ...) and parameters for these objects are stored in this JSON format in a way that allows a parser to reconstruct the main sequence.

This JSON structure has been modified from the previous version (see [18, p57ff]). The previous JSON structure used the parameter and sequence element names (string) as keys. This approach had the considerable downside that the key did not specify the type of the element. Therefore, one had to keep track of the context in which the key is parsed to identify the correct type of object.

The new format replaces name strings with array indices (names can be added for debugging purposes). Sequence objects and events are now stored in separate arrays. For easier indexing, object types are only partially specified by array membership (e.g. Event, Sequence). An additional type field is used for the full specification (e.g. RF Event, Linear Sequence). Figure 13 shows the structure of this new JSON structure.

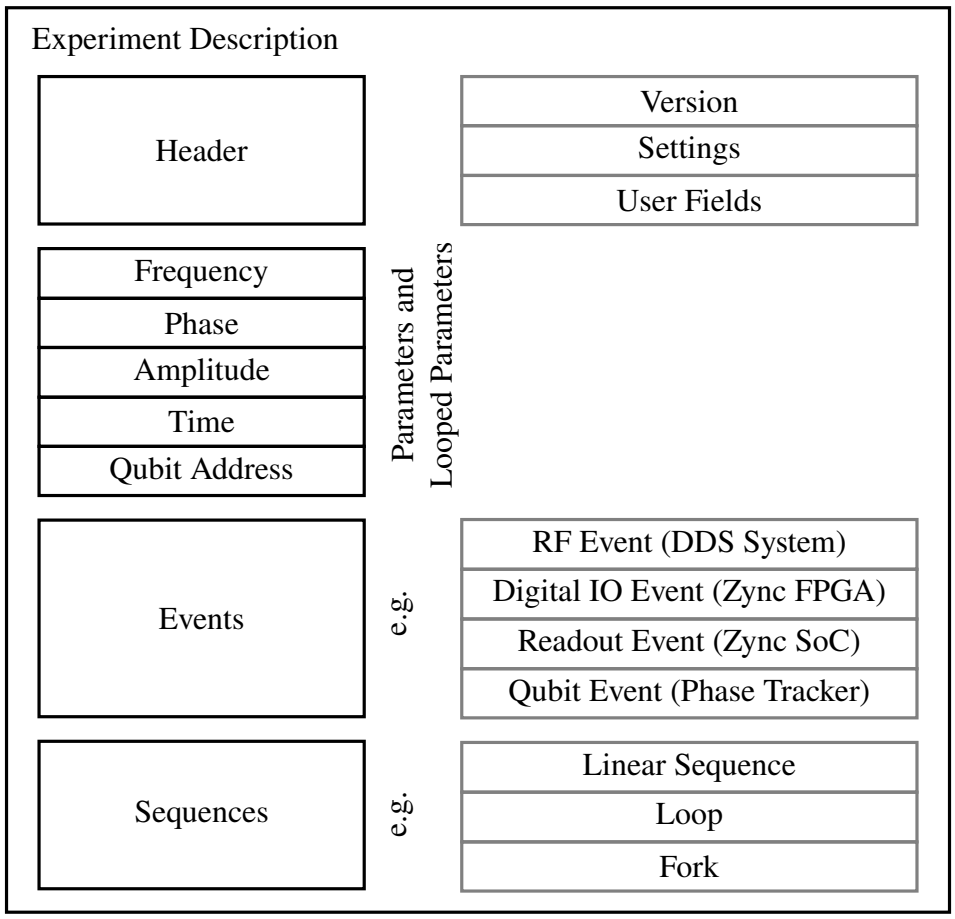


Figure 13: Structure of the JSON file generated by the Ionpulse Sequence Generator. Objects are shown with thick lines, subobjects of the experiment description (except the header) contain an array of objects. The header stores the generator version, specific settings and user definable fields (not parsed). Parameters, Events and Sequences are stored in separate arrays, where the last element of the sequence array is interpreted as the main sequence.

3.5 Sequence Interpretation - JSON Parser

With the changes to the JSON structure, the parser has also been largely rewritten. The parser is still based on the RapidJSON C++ library and uses the Simple API for XML (SAX) interface. Using a SAX-based parser compared to a Document Object Model (DOM)-based parser allows the application to sequentially parse the document instead of parsing the whole document and then filtering for objects of interest.

The new parser uses a state design pattern. This approach yields a more modular parser, where each (sub-)module implements its own parsing methods. For this approach, in addition to

the `JSON Handler` which handles the JSON stream, an interface, `JSON SequenceObject`, is defined. This interface is implemented by all classes parsing JSON objects.

Parsing the new JSON format works as follows: The JSON stream is passed to the `jsonhandler` class. It expects the keys Header, Parameters (see Figure 13) or Event or Sequence. If a key is found, it creates a new object from the respective class and forwards all parsing events (e.g. key, value, end object, ...) to that object until this object finished parsing. Upon receiving an end object event, the object will construct the corresponding driver-object and store it in a vector. Events and sequences will then use these vectors to retrieve referenced sub-objects.

3.6 Sequencer Driver - Pulseway

For processing sequence instructions a sequencer (domain specific processor, as outlined in 2.2) is used. We chose `pidecoder` (from the already existing Pulseway FPGA design, written by Vlad Negnevitsky and Martin Stadler) as the sequencer for the phase tracker (see subsection 3.8). `pidecoder` reads instructions from a memory and executes these instructions. These instructions include more complicated jump and conditional jump instructions for loops and forks as well as basic input-output (IO) operations with 24 bit payload. To translate the sequence elements into instruction for `pidecoder`, a driver (manager) is used. From the Pulseway driver, `pwy::Manager` and `pwy::Edge` are adapted, handling the `QubitEdge` objects created by the parser. The `Manager` requires a different method to send objects to the sequencer, as we need to pass them to the DDS cards through Hiway (see below). The new `Edge` class needs to use different functions for frequency and phase conversion from Hiway (DDS cards have a 125 MHz clock) as well as add new instructions for `pidecoder`.

The frequency and phase conversion of Hiway targets the AD9910 DDS chips on the DDS boards, which have a 16 bit phase and 32 bit frequency resolution and receive a $f_{dds} = 1$ GHz input clock. In order to keep the same resolution on the phase tracker at a $f_{clk} = 125$ MHz clock, the resulting frequency conversion is given by

$$f_{pt} = \frac{\Delta f_q \cdot 2^{32}}{f_{clk}} = \frac{8 \cdot \Delta f_q \cdot 2^{32}}{f_{dds}} \tag{21}$$

where Δf_q is the qubit frequency offset and f_{pt} the accumulation value per clock cycle on the DDS card.

Since the `pwy::Manager` does not feature channels (pidecoder is the only sequencer on the Zedboard), a way of addressing the independent phase tracker channels has to be added. For the first version, this is done by a separate manager for each channel which includes the qubit index when passing information to Hiway.

3.7 Communication - Hiway

Sending information to the DDS cards is handled by Hiway, which also includes the driver with the appropriate generation of message frames for the DDS cards. The `dds::Manager` has been modified with a method to accept the new instructions from the phase tracker `Manager`. Instructions from the `dds::Manager` are encoded in message frames and sent over the BP to the DDS cards. The BP communication layer implements all low-level communication and offers 56 bit payload for message frames. To address the phase tracker channels, these message frame have been adapted, as shown in Figure 14.

55:52	51	50:46		45:37							36:0						
word_page	req	dds_ch		cfg							bram_inst						
	51	50	49:46	45	44	43	42	41:39	38	37	36	35	34:32	31	30:28	27:18	17:0
0000	req_odec_status	dds_ch_mask_valid	dds_ch_mask	cfg_valid	cfg_trig	cfg_idec_rst	cfg_mtim_rst	empty	cfg_dds_ext_pwr	cfg_dds_mstr_rst	Bram_inst_valid	bram_sel_high_valid	bram_sel_high	bram_fifo_inst	bram_sel_low	bram_addr	bram_data

55:52	51	50:46		45:37							36:0					
word_page	req	dds_ch		cfg							bram_inst					
	51	50	49:46	45	44	43	42	41:39	38	37	36	35	34:30	29	28	27:0
0000	req_odec_status	dds_ch_mask_valid	dds_ch_mask	cfg_valid	cfg_trig	cfg_idec_rst	cfg_mtim_rst	empty	cfg_dds_ext_pwr	cfg_dds_mstr_rst	Bram_inst_valid	pt_sel	qubit_addr	pt_fifo_sel	pt_buffer_sel	pt_payload

Figure 14: Minotaur Message Frames, (a) original message frame [17] and (b) phase tracker frame with the changed fields highlighted in green. Two payloads are concatenated to obtain a 24 bit address and 32 bit data for pidecoder. Numbers on top indicate the bit-position in the frame. See text for further details.

The phase tracker message frame uses the unused `bram_sel_high_valid` bit in the original message frame as a select bit (`pt_sel`). If this bit is set, the following bits are interpreted qubit address (`qubit_addr`, 5 bit, max. 32 channels), FIFO select (`pt_fifo_sel`) for selecting the FIFO as write target, buffer select (`pt_buffer_sel`) and payload (`pt_payload`). In order to get 56 bit width (32 bit data, 24 bit address) for pidecoder, two consecutive payloads are concatenated. If the buffer select is high, the payload is interpreted as 24 bit address and upper 4 bit of data. Buffer select low writes the instruction to BRAM or FIFO, where the payload is interpreted as the lower 28 bit of data. In addition to these changes, the `dds::Edge` class has

also been adapted to accept a qubit address. This allows the DDS sequencer to read the qubit phase from a certain phase tracker.

3.8 DDS-Card FPGA Design - Minotaur

The DDS card FPGA design (Minotaur) needs to incorporate the logic blocks for the phase tracker. Figure 15 shows a schematic of Minotaur (refer to [17] for further details).

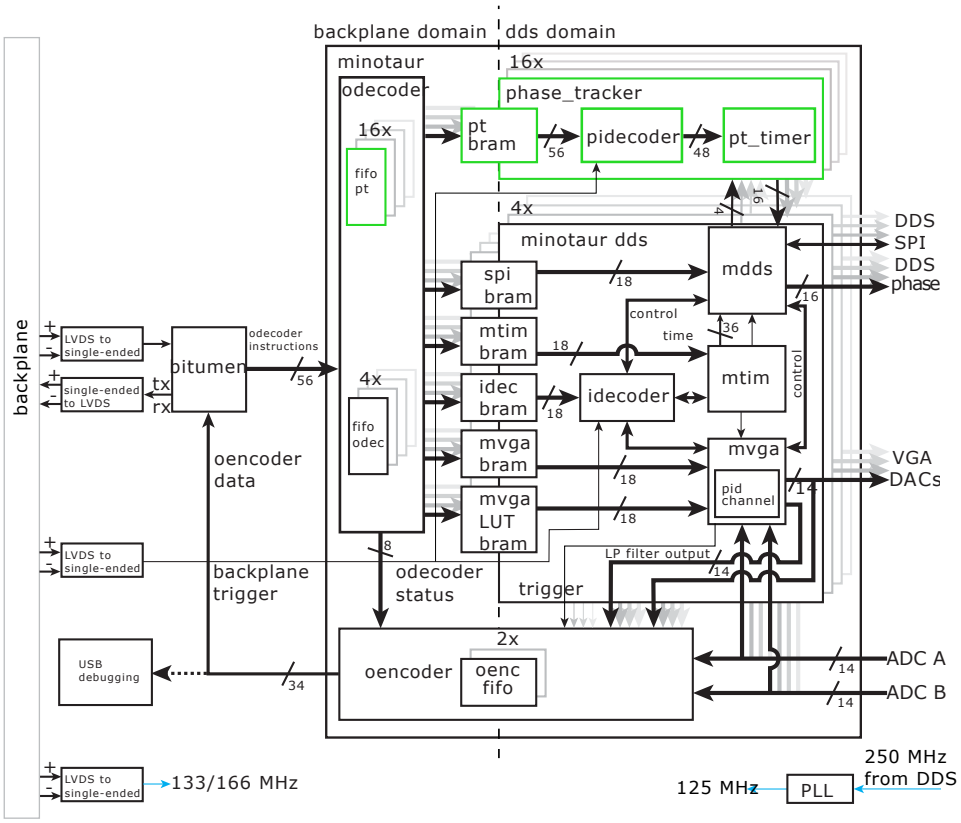


Figure 15: Schematic of the Minotaur FPGA design. Additional modules for the phase tracker module are highlighted in green. `odecoder` passes instructions to the `pt bram`. These are executed by the `pidecoder` sequencer, passing information to the timer and trigger unit `pt_timer`. The DDS interface `mdds` then selects which phase register information to pass to the DDS chip depending on the qubit address setting of the pulse. This figure is adapted from [17]

.

Minotaur accepts 56 bit message frames (see Figure 14) from the BP (the FPGA design for the de/serializer interface is called Bitumen). These control words are interpreted by `odecoder` and written to the respective submodule Block Random Access Memorys (BRAMs) or FIFOs

for `idecoder` or `phase tracker`. The DDS channel design (Minotaur DDS) includes a sequencer (`idecoder`), timer unit (`mtim`), DDS chip interface (`mdds`) and a Variable Gain Amplifier (VGA) interface. `idecoder` executes the sequence instructions from the `idec bram` , controlling the other modules and starting the timer. The timer then triggers actions on these modules, previously programmed by `idecoder`. The DDS interface `mdds` communicates with the DDS chips via Serial Peripheral Interface (SPI). SPI frames are constructed using FPA information from the `spi bram`. Loops and forks are executed by `idecoder` with jump instructions to different addresses in the `idec bram`, storing the next adress as the return address, equivalent to functions in C. Updates during a loop are handled with FIFO pops, where the `odecoder fifo` updates the BRAMs. Communication back to the Zedboard via the backplane are done via the `oencoder` module. [17, 14]

For the phase tracker, `odecoder` has been modified to accept additional instructions (see Figure 14) filling the phase tracker BRAM and FIFO. The FIFOs are managed by the Zedboard, which needs to monitor FIFO fill levels to prevent overflows. For this purpose, a status bit has been added to encode the status of the phase tracker FIFOs.

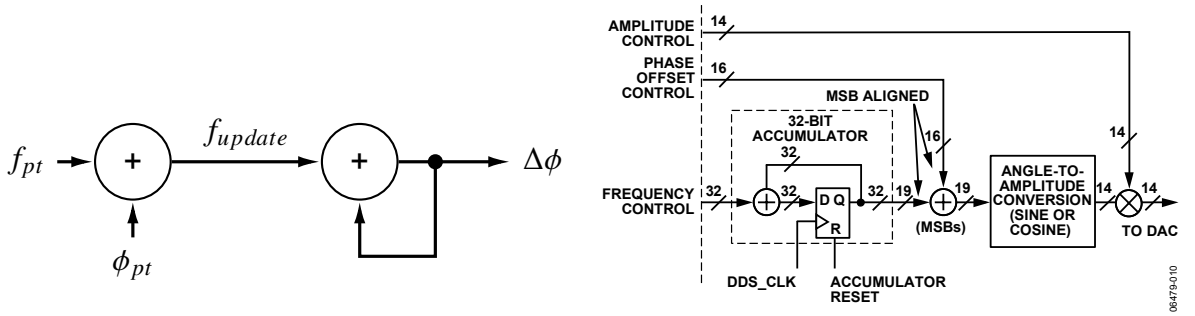
16 phase tracker modules have been added, each containing the BRAM, `pidecoder` and the `pt_timer` module. Due to the high utilization and lower speed grade (-2) of the Spartan 6 FPGA [16], the original `pidecoder` design does not meet the timing constraints in the 125 MHz domain on the DDS card. Therefore, `pidecoder` jump instructions had to be modified to include an additional clock cycle of latency. Further modifications have been made by Martin Stadler to improve the calculation pipeline and instruction execution. Meeting the timing constraints on the Spartan 6 FPGA with the added 16 phase tracker modules also requires additional buffering of signals, as the mapping to the FPGA often produces long connection paths between logic elements.

The timer module interacts with `pidecoder` to execute the sequence of programmed frequencies and phases synchronously. Analogous to Equation 2.4, we need to accumulate phase differences, defined by phase and frequency deltas to get $\Delta\phi$ of the pulse. For the N-th pulse this is given by

$$\Delta\phi = \sum_{n=0}^{n=N-1} f_{qn} t_n + \Delta\phi_n. \tag{22}$$

However, since the accumulator works on discrete timesteps defined by a clock cycle, we have to convert frequency f_{qn} to a 32-bit frequency f_{pt} according to Equation 3.6. $\Delta\phi_n$ is converted to a 16 bit phase $\phi_{pt} = \Delta\phi_n \cdot 2^{16}/(2\pi)$. These are added in a first stage adder. Then a second stage adder accumulates this result and therefore integrates the frequency, yielding the tracked phase for this channel, as shown in Figure 16a. $\Delta\phi_n$ is non-zero only for a single clock cycle at a time to avoid adding the phase delta multiple times.

The accumulator works very similarly to a DDS core, for example from the AD9910 [1], shown in Figure 16b. The DDS core accumulates the frequency control word and adds a constant offset. This phase output is then converted to a digital sine, using a sine table or other methods,



(a) (b)

Figure 16: (a) Phase accumulator for the phase tracker module with input frequency f_{pt} and phase ϕ_{pt} accumulates f_{update} with frequency $f_{dds}/8$ to calculate $\Delta\phi$. (b) DDS core from AD9910. The frequency control input is accumulated with frequency f_{dds} ('DDS_CLK'), offset by the phase offset control and converted to a digital sine-amplitude. This is then multiplied by the amplitude control and sent to the Digital-to-Analog Converter (DAC) to produce the analogue sine signal. Figure from the AD9910 datasheet [1].

multiplied with the amplitude control and converted to an analog signal with a DAC.

Apart from not converting the phase to a sine wave, the main difference to the phase tracker is, that it has to sum phase deltas into the accumulator as these offsets are relevant for the future evolution of the phase.

With this implementation of the phase tracker, we obtain frequency and phase resolutions of

$$\delta f_q^{(pt)} = \frac{f_{dds}}{8 \cdot 2^{31}} \approx 0.06 \text{ Hz},$$

$$\delta \phi_q^{(pt)} = \frac{360^o}{2^{16}} \approx 0.005^o.$$

The phase resolution in the HDL design can be chosen arbitrarily. The usable resulution is limited by the DDS chip to 16 bit, but it can be improved if other systems are used. Note that this is only resolution from quantization and does not represent an error estimate.

Finally, the $\Delta\phi$ output of the phase tracker has to be sent to the DDS chip. For this, the `mdds` module has been adapted to accept an additional setting for the qubit address, which will be read after the FPA entries in the `spi bram`. This allows the `mdds` module to switch to the correct phase tracker register and add the phase of the pulse, similar to the previous design. The phase output to the DDS chip is then given by

$$\phi''_{pulse} = \phi_{pulse} + \Delta\phi. \tag{23}$$

4 Results

In order to test the implemented phase tracker, we use the Ionpulse Sequence Generator to produce a test sequence plotted in Figure 17. In a real experiment, we would drive the transition on resonance, shifting the frequency of the channel to match the frequency of the transition. For testing, this would make extracting the phase shift more difficult, hence the frequency of the RF channel is kept constant, while the phase tracker receives a sequence with changing frequencies. In addition, another channel is set at the same frequency and amplitude as a reference. With these considerations a showcase sequence to test the phase tracker is implemented, as shown in Figure 17.

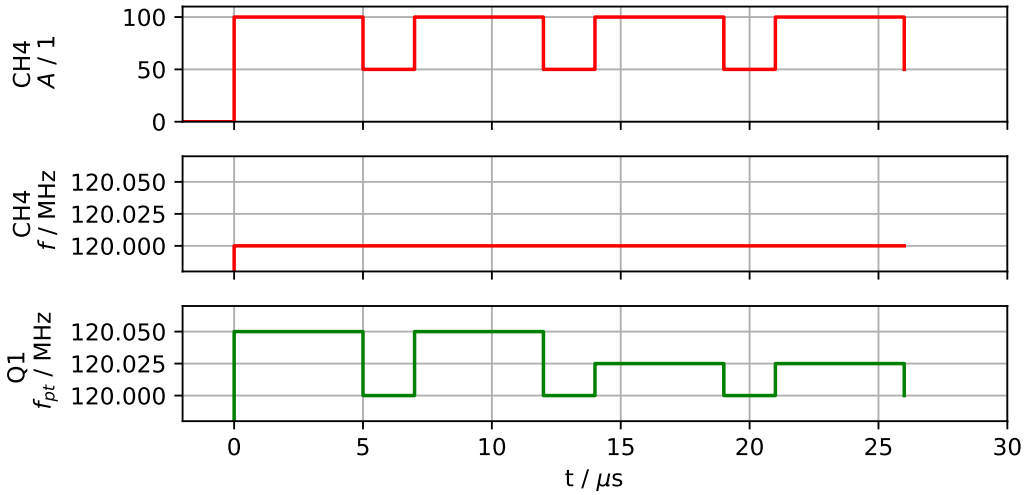


Figure 17: Test sequence plotted from Ionpulse Sequence Generator data. RF CH4 (and CH5, not shown) pulses 4 times between 50% and 100% amplitude, while the frequency is set to 120 MHz. Phase tracker Q1 is set to a frequency of 120.05 MHz for the first and 120.025 MHz for the last two pulses. See text for further details.

The phase shift for each pulse is given by $\Delta\phi = 360 \cdot f_{pt} \cdot t$, resulting in 90^o for the first two pulses and 45^o for the second two pulses, where we reset the phase tracker before the first and before the second two pulses to test the reset and separate the two groups of pulses.

The first step of full system testing happens in the Ionpulse emulator. Components of the system have been individually tested using pre-existing or adapted testbenches and test cases to catch major bugs.

For the emulator, the FPGA design is compiled with Verilator in order to emulate both the Zedboard and DDS system. Then, the resulting C++ library is linked against the compiled driver code to obtain an executable, which emulates the time-coherent part of the control system. The advantage of the emulator compared to testing directly on the rack is improved debugging, as all signals in the FPGA design are recorded. Bugs in the design are easier to

find with access to signals in multiple levels of the design. On the DDS card itself, one has to resort to including a logic probe in the design to record signals. This logic probe is limited in memory depth and input signal count by the timing constraints and resources in the FPGA. Furthermore, triggering on the events of interest is often difficult and might require additional logic. With the emulator recording all signals, triggering is not a problem. However, running the emulator is computationally intensive and requires more memory the more signals (trace depth) are recorded and the longer it is run. Therefore, runtime is usually limited to 50 ms which equals a few shots of the experiment.

When running the test sequence shown in Figure 17 in the emulator, we obtain the signals shown in Figure 18.

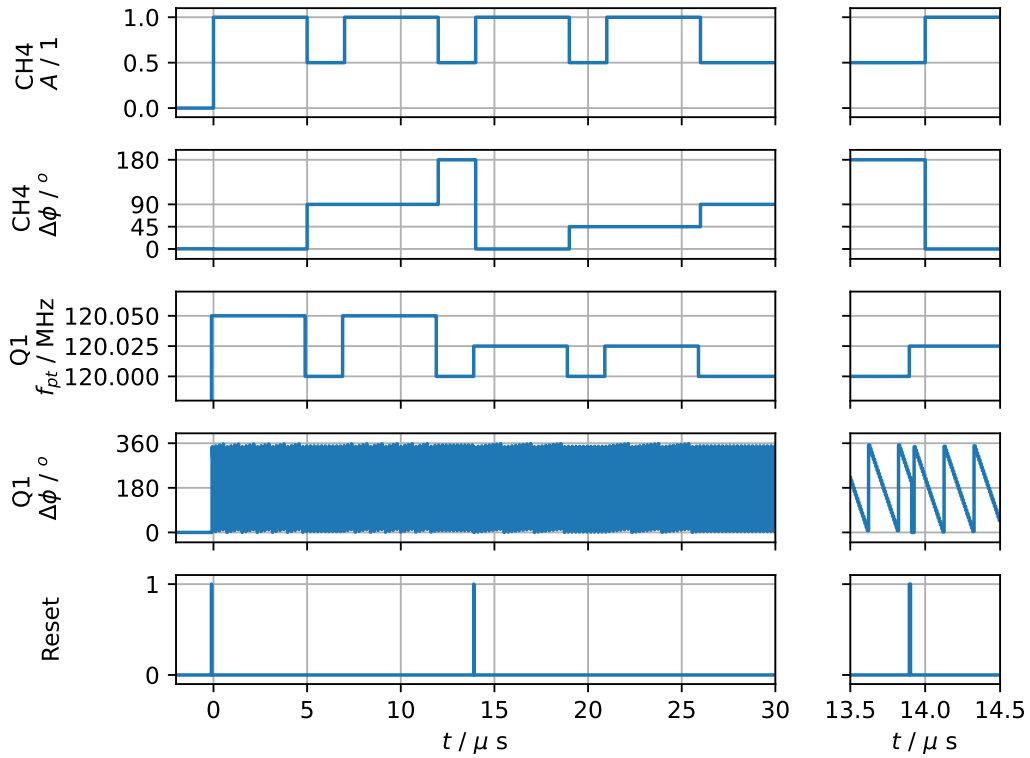


Figure 18: Test sequence from Figure 17 created by emulating Zedboard and DDS system. (left column) Amplitude (A) of CH4 (and CH5, not shown) pulsing between 100% and 50%, frequency (not shown) is set to 120 MHz. Phase tracker Q1 is set to a frequency of 120.05 MHz for the first and 120.025 MHz for the last two pulses. The phase accumulator output of Q1 ($\Delta\phi$) changes on a timescale of $f_{dds}/8$ (see right column) and is registered at each edge of CH4, resulting in the offset $\Delta\phi$ at CH4. The reset signal resets the phase tracker before the first and after the second pulse. Note that the phase tracker signals are ahead of the pulses. See text for further details.

In the left column of Figure 18, the phase shift on RF CH4 from the phase tracker is shown. We see the expected shifts of 90^o and 180^o for the first two pulses. Then, the reset sets the phase accumulator to 0^o (magnified in the right column). For the last two pulses we obtain again the expected phase shifts of 45^o and 90^o, from the pulse time and the applied frequency shift. In the right column the time frame around the reset is magnified and the steps of the phase accumulator are clearly visible. The phase accumulator sums f_{pt} with a frequency of $f_{dds}/8 = 125$ MHz and overflows from 2π back to 0. In this case, with $f_{pt} > f_{dds}/16$ the accumulator overflows every clock cycle, resulting in the visible 'down slope'. The reset to the phase accumulator is set at $t \approx 13.9$ µs and results in the phase accumulator being forced to 0. This reset and other signals from the phase tracker occur shifted by ≈ 0.1 µs to the RF edge because of the internal delays of the emulated model of the DDS chip.

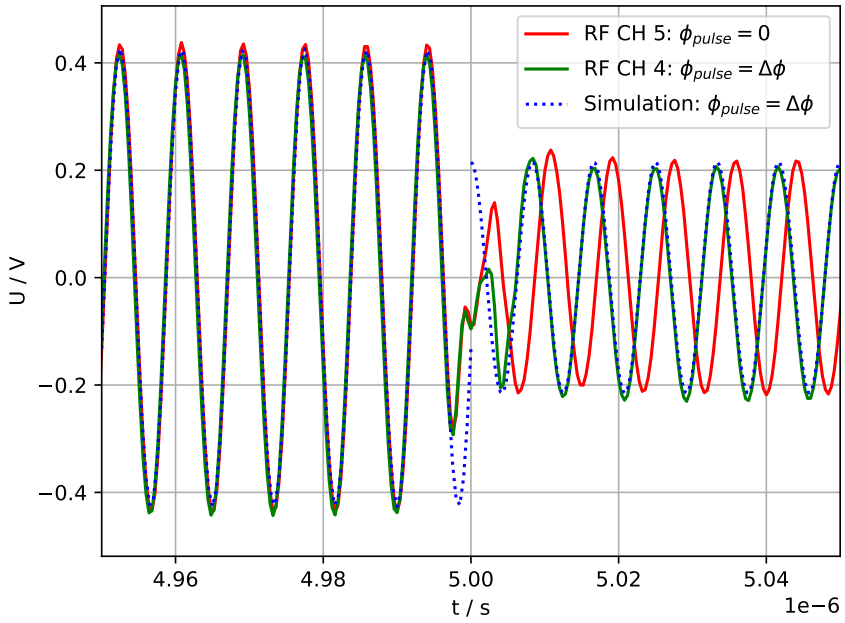


Figure 19: End of the first pulse of the test sequence, measured output from the DDS card. Phase reference CH 5 with phase offset $\Delta\phi = 0$ and shifted CH 4 with phase offset $\Delta\phi$ from the phase tracker. Both channels output sine waves at $f = 120$ MHz. The simulation adds a phase shift of $\Delta\phi = 90^o$ at the end of the first pulse compared to the reference channel CH5. See text for more details.

After the successful tests in the ionpulse emulator, the phase tracker has been tested on hardware with the same test sequence. The control system is set up with Zedboard, backplane and a single DDS card. FPGA gateware and software are programmed onto the hardware. The test sequence JSON file is sent to the Zedboard to run the sequence. Outputs from the DDS card

are connected to an oscilloscope to record the RF signals. The outputs as well as simulated data with the expected phase offset are shown in Figure 19. The output from the DDS system shows the expected offset of 90^o after the first pulse to the reference signal. The simulated signal (taking FPA of CH5 as reference) with the expected phase offset fits the measured signal. Deviations from the simulation around $t = 5$ µs are a result from a 4ns shift between the amplitude and phase change together with bandwidth limitations and filtering.

5 Conclusion and Outlook

In conclusion, we have designed and tested a phase modelling system fitting the requirements from both the qubit evolution and the control system.

The qubit phase evolution requires a system to replicate the frequency changes from Stark shifts and phase offsets induced from transport (changes of the trap potential) and shaped pulses. This becomes particularly important for longer sequences where phase errors as a result frequency deviations on the order of a few kHz can no longer be ignored.

The control system requires as few as possible unique pulses to stay within memory limitations of the DDS cards. Hence, offsetting the phase for each pulse is not an option, as a sequence of multiple π-pulses needs to store each pulse individually due to its unique phase offset.

We have discussed four options for implementation to fulfill these requirements, where we chose to implement the phase *calculation with a new sequencer*. This implementation offers the most flexibility when modelling the phase and allows the different RF channels to offset the pulses by the accumulated phase of the targeted qubit.

The design was integrated in the control system, which involved significant changes to many layers from the high-level sequence description to the low-level HDL code. The final design was tested in the emulator as well as on the hardware. Here, we found the measured offsets in good agreement with the programmed offsets.

With the drivers and FPGA designs tested and functioning, some tasks still remain. For better usability of the system, Pycrystal needs to be adapted with high-level functions including phase tracker instructions. The original plan to test the phase tracker with randomized benchmarking on an actual experiment had to be abandoned due to a lack of time. Before it can be used productively in the lab, the phase tracker still has to be tested in an actual experiment.

As a further outlook, the current system supports only 16 phase trackers per rack. With increasing system size in quantum computation, a modular approach (with 16 phase trackers per DDS card) is more appropriate. As an intermediate step, decoupling of the phase trackers per card is possible, by addressing each card's phase trackers individually. This allows us to use the full 16 phase trackers per card for all RF channels of this card. However, for full flexibility, exchanging phases between phase trackers must be possible. This scheme is compatible with the current implementation, but requires some additional modifications. The additional communication delays have to be compensated when communicating phases from one DDS card to another via the Zedboard.

6 References

[1] *AD9910 Datasheet and Product Info | Analog Devices*. URL: https://www.analog.com/en/products/ad9910.html (visited on 05/04/2022).

[2] S. M. Barnett and P. M. Radmore. *Methods in Theoretical Quantum Optics*. Oxford Series in Optical and Imaging Sciences 15. Oxford : New York: Clarendon Press ; Oxford University Press, 1997. 284 pp. ISBN: 978-0-19-856362-4.

[3] Claude Cohen-Tannoudji et al. *Photons and Atoms: Introduction to Quantum Electrodynamics*. Physics Textbook. Weinheim: Wiley-VCH, 2004. 468 pp. ISBN: 978-0-471-18433-1.

[4] E. A. Donley et al. "Double-Pass Acousto-Optic Modulator System". In: *Review of Scientific Instruments* 76.6 (June 2005), p. 063112. ISSN: 0034-6748, 1089-7623. DOI: 10.1063/1.1930095. URL: http://aip.scitation.org/doi/10.1063/1.1930095 (visited on 10/28/2022).

[5] Christoph Fischer. "Quantum Non-Demolition Readout for Optically Trapped Alkaline-Earth Rydberg Atoms". Doctoral Thesis. ETH Zurich, 2022. DOI: 10.3929/ethz-b-000546638. URL: https://www.research-collection.ethz.ch/handle/20.500.11850/546638 (visited on 10/08/2022).

[6] H. Häffner, C. F. Roos, and R. Blatt. "Quantum Computing with Trapped Ions". In: *Physics Reports* 469.4 (Dec. 1, 2008), pp. 155–203. ISSN: 0370-1573. DOI: 10.1016/j.physrep.2008.09.003. URL: https://www.sciencedirect.com/science/article/pii/S0370157308003463 (visited on 10/27/2022).

[7] H. Häffner et al. "Precision Measurement and Compensation of Optical Stark Shifts for an Ion-Trap Quantum Processor". In: *Physical Review Letters* 90.14 (Apr. 9, 2003), p. 143602. ISSN: 0031-9007, 1079-7114. DOI: 10.1103/PhysRevLett.90.143602. URL: https://link.aps.org/doi/10.1103/PhysRevLett.90.143602 (visited on 10/22/2022).

[8] D.F.V. James. "Quantum Dynamics of Cold Trapped Ions with Application to Quantum Computation". In: *Applied Physics B: Lasers and Optics* 66.2 (Feb. 1, 1998), pp. 181–190. ISSN: 0946-2171, 1432-0649. DOI: 10.1007/s003400050373. URL: http://link.springer.com/10.1007/s003400050373 (visited on 10/30/2022).

[9] Jeremy Kratz. *English: The Officially Endorsed Logo of Isocpp.Org*. Jan. 30, 2017. URL: https://commons.wikimedia.org/wiki/File:ISO_C%2B%2B_Logo.svg (visited on 11/15/2022).

[10] Rodney Loudon. *The Quantum Theory of Light*. 3rd ed. Oxford Science Publications. Oxford ; New York: Oxford University Press, 2000. 438 pp. ISBN: 978-0-19-850177-0 978-0-19-850176-3.

[11] Maciej Malinowski. "Unitary and Dissipative Trapped-Ion Entanglement Using Integrated Optics". ETH Zurich, 2021, 396 p. DOI: 10.3929/ETHZ-B-000516613. URL: http://hdl.handle.net/20.500.11850/516613 (visited on 04/04/2022).

[12] David C. McKay et al. "Efficient Z Gates for Quantum Computing". In: *Physical Review A* 96.2 (Aug. 31, 2017), p. 022330. DOI: 10.1103/PhysRevA.96.022330. URL: https://link.aps.org/doi/10.1103/PhysRevA.96.022330 (visited on 10/27/2022).

[13] Mose Müller. *A Flexible Python Framework for Generating Pulse Sequences for Multi Zone Operations in Ion Traps. In Preparation.* Oct. 2022.

[14] Vlad Negnevitsky. "Feedback-Stabilised Quantum States in a Mixed-Species Ion System". ETH Zurich, 2018, 165 p. DOI: 10.3929/ETHZ-B-000295923. URL: http://hdl.handle.net/20.500.11850/295923 (visited on 04/04/2022).

[15] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. 10th anniversary ed. Cambridge ; New York: Cambridge University Press, 2010. 676 pp. ISBN: 978-1-107-00217-3.

[16] *Spartan-6 FPGA Family*. Xilinx. URL: https://www.xilinx.com/products/silicon-devices/fpga/spartan-6.html (visited on 05/04/2022).

[17] Martin Stadler. *Integrated Laser Amplitude Stabilization for Mixed Species Trapped-Ion Experiments*. Sept. 2018.

[18] Marco Erwin Stucki. "Enabling a Quantum-Gate-Level Interface with a Trapped Ion Control System". In: (Dec. 2021), p. 102.

[19] Alfredo Ricci Vasquez et al. *Control of an Atomic Quadrupole Transition in a Phase-Stable Standing Wave*. Oct. 5, 2022. arXiv: 2210.02597 [physics, physics:quant-ph]. URL: http://arxiv.org/abs/2210.02597 (visited on 10/30/2022).

[20] www.python.org. *English: Python Logo*. Aug. 6, 2008. URL: https://commons.wikimedia.org/wiki/File:Python-logo-notext.svg (visited on 11/15/2022).

[21] *ZedBoard | Avnet Boards*. URL: https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/zedboard/ (visited on 10/08/2022).

[22] *Zynq-7000 SoC*. Xilinx. URL: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html (visited on 10/08/2022).