Semester Thesis
Trapped Ion Quantum Information Group
(TIQI)

Implementation of a Python DDS communication protocol to drive AOMs

Danny Kun
14-994-896
dkun@student.ethz.ch
Supervisor: Matt Grau

Abstract

In this thesis, we developed a python class to control a Direct Digital Synthesiser (DDS) with a Raspberry Pi mini-computer (RPi). The DDS creates a digital signal, which is converted into an analogue signal and fed into an Acousto-Optic Modulator (AOM). The discussion will centre on the code behind this protocol and the hardware used to develop the communication.

ETH Zurich
October 29, 2017

Contents

1 Introduction

In the study of quantum systems, it is often impossible to compute the solutions of a certain model. By this, we don't mean that these models don't have an analytical solution, such as the three-body problem in classical mechanics for example. What we mean is that there are so many equations to solve in the model due to the quantum interaction of the particles, that it becomes impossible with the available technology to numerically compute the solutions to all equations. In fact, the (computational) complexity of quantum systems increases exponentially with the size of these systems, and so one needs a computational machine based on the same laws and techniques to effectively be able to simulate them [1]. To solve this problem we use so-called quantum simulators, which are set up in a way to exactly mimic the behaviour described in the model and then observe the behaviour of these simulators. This is why the study of quantum systems has motivated the development of quantum simulators.

The construction of such a quantum simulator is based on an elementary unit called the "quantum bit" or "qubit", in analogy to a classical computer. However, such a qubit differs dramatically to a classical bit in several ways. First and foremost, the quantum bit can be in a superposition of a '0' and a '1' state, whereas a classical bit can only be in either of the two states at any one time. Like with any quantum system, the qubit then has certain probabilities to be measured in one of the two states. This goes to show, that the elementary building block of a quantum computer would be a quantum system. A second big difference is the possibility to have entanglement between multiple qubits. This has no analogue in classical physics and is one of the features, which makes a quantum computer so much more powerful than a classical one.

In theory, qubits can be represented by any quantum two-state system. In the Trapped Ion Quantum Information (TIQI) Group at ETH Zürich, qubits are represented by the electronic states of ions. By choosing two distinct states of ions and using a laser, which is tuned to the transition between these two states, the qubit can then be controlled. Because of the nature of some of these transitions, the lasers involved in the control of these qubits have to be very sensitive, i.e. the setup must allow for the laser frequency to be calibrated on a very small scale.
Moreover, since the idea is to work with specific states of ions, these must first be under the complete control of an experiment. Specifically, the energy of those ions must be so low, that specific energy levels can be addressed. This means that the ions must first be trapped in some way, before they can be manipulated. One technique to do this is called Magneto-Optical Trapping (MOT), which is what is used by one team within the TIQI group. There again, the laser must be tuned to the appropriate energy levels.

To fine-tune the laser used in the MOT, a so-called Acousto-Optic-Modulator (AOM) is used. This AOM also needs a signal to be driven, which is what has been done in this paper. More precisely, we implemented a communication protocol (i.e. a python library) to drive AOMs with a specific type of digital to analogue converter (DAC), namely the AD9959.
In the next section (Section 2), we will give a very brief theoretical description of qubits, MOTs and AOMs, while in Section 3, we discuss the hardware and software used for the implementation of the protocol, as well as the developed protocol itself. In Section 4 we then show some specific examples of how the library is used in the lab.

2 Theoretical Background

2.1 Qubits

The smallest building block of a quantum computer, in analogy to a classical computer, is the qubit. Theoretically, any kind of quantum two-level system can be used as a qubit. However, since we want to perform measurements on these qubits, we want them to have a long coherence time, i.e. we want them to remain isolated for a time that is long enough, so we can make our measurements on them. When using the electronic states of ions, there are two ways of implementing a qubit: One can either use the hyperfine level splitting (called hyperfine qubits) or simply use an electronic ground state and a metastable excited electronic state of the ion (optical qubits). The advantage of the former is that it is very long-lived and has a very strong frequency stability. There are many other ways of representing a qubit, such as using electron spin states or photon polarizations, to name a few.

One of the main advantages of qubits over normal bits are their ability to be in a superposition state, which means that they can be both 0 and 1 at the same time, which would give the quantum computer as a whole much more calculation capability.

To induce the coupling between the states in an ion qubit, one must use an appropriate energy source such as lasers or microwaves, in the case of shorter transitions, which are tuned to the wavelength of the transition between those two states. Using the hyperfine structure of the ground state of an $^{25}Mg^{+}$ ion, for example, this transition frequency is 1.8 GHz [2]. We could thus drive this transition with a radio-frequency source, or with a virtual two-photon transition, using a laser of a certain frequency to move from one ground level state, $^{2}P_{1/2}$, to a virtual excited state $^{2}S_{1/2}$ and then, using a laser which is detuned by 1.8 GHz from the first laser, the ion would be set to the other hyperfine ground state.

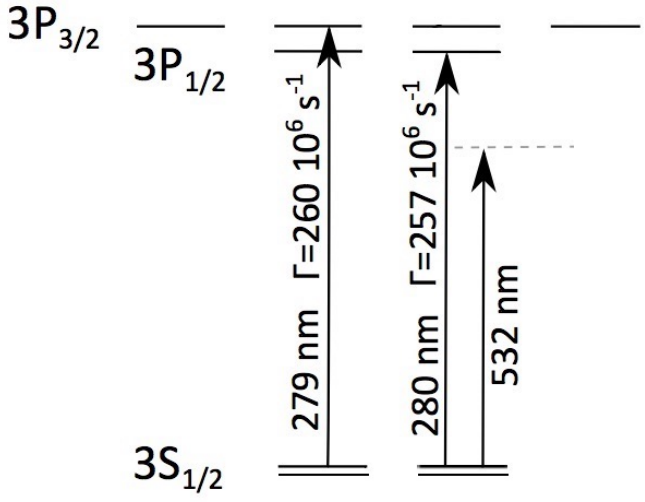


Figure 1: The fine structure of the $^{25}Mg^{+}$ ion at zero magnetic field, with an indicated hyperfine structure in the ground state: We can use a virtual excited state of the ion to cycle between the two hyperfine ground states using two lasers, which are detuned by 1.8 GHz with respect to each other. [3]

2.2 Magneto-Optical Trap (MOT)

A Magneto-Optical Trap (MOT) is used to produce very cold, trapped atoms. As the name suggests, a MOT consists of both optical and magnetic components and the whole

setup is built within a vacuum chamber. The optical component is made up of three circularly polarised laser beams - one for each coordinate axis. A mirror retro-reflects the beam at the other side of the chamber, which creates the opposite beam polarisation.
The laser beams provide the radiative (scattering) force, which push back the atoms towards the origin of the trap, i.e. the region where the three beams intersect. The asymmetry of this force - which makes it position-dependent - is produced by the magnetic field. The splitting of atom energy-levels within a magnetic field is due to the Zeeman effect. By tuning the lasers to a frequency slightly beneath the transition frequency of the atoms, they create a potential well. When an atom begins to move away from the centre of the trap, it experiences a restoring force due to the Doppler Shift: the atom scatters more photons because it receives photons closer to its transition frequency and is thus forced back towards the origin.[4] A schematic of a MOT setup can be seen in Figure 2.

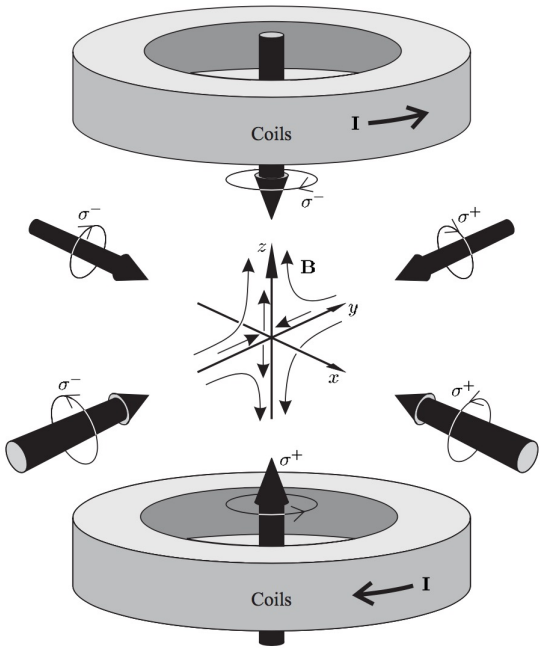


Figure 2: The MOT consists of a pair of Helmholtz coils with opposite currents and three orthogonal pairs of laser beams with respectively circular polarisation states. The small arrows indicate the direction of the quadrupole magnetic field produced by the coils. The MOT is able to keep atoms in its centre due to the selection rules for transitions between the Zeeman states, which lead to an imbalance in the radiative force from the laser beams that pushes the atom back towards the centre of the trap. [4]

Since the MOT lasers need to be tuned to the correct transition frequencies and these are often sensitive or close to one another (such as in the hyperfine state splitting example), the laser frequencies need to be stabilised to the specific frequency. This is done with the use of acousto-optic modulators (AOMs) and other stabilising devices. The present work focused on the setup of an AOM to do precisely that. In our case, the $^{25}Mg^+$ hyperfine levels are in a range of 1.8 GHz and the linewidth given is $\Gamma = 2\pi \cdot 80\,\text{MHz}$.

2.3 Acousto-Optic Modulators (AOMs)

As mentioned earlier, AOMs are useful devices to fine-tune laser frequencies. We will now give a brief overview of the theory behind AOMs.
The two main components of an AOM are a piezo-electric transducer, which generates a

sound wave of a certain frequency, Ω, (usually in radio-frequency range), and some material (e.g. glass or quartz), into which the wave is fed. Due to the periodic contraction and rarefaction of the material, the sound wave creates periodic changes in the refractive index of the material. Thus, a form of grating is created and acts much like a lattice for an orthogonally incoming light wave, as seen in Figure 3.

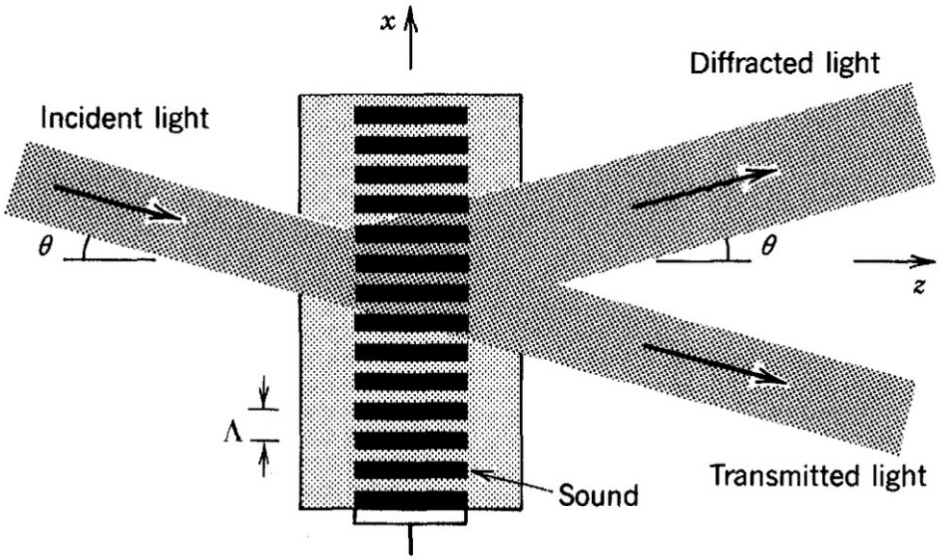


Figure 3: Theory of an AOM [5]. A sound wave is introduced into a medium (e.g. glass or crystal) and acts as a grating for an incoming light beam. The light is diffracted into several different orders and some part of it is simply transmitted in its 0th order.

When the light beam enters at an angle θ, satisfying the Bragg condition,

$$\sin\ \theta = \frac{\lambda}{2\Lambda}, \tag{1}$$

where λ is the wave length of the light and Λ represents the distance between two wave fronts, the light gets partially reflected by the medium.
As the light passes through the medium, the reflected light receives a frequency shift through a form of Doppler shift. The shift in frequency Ω, equals the frequency of the sound wave [5]. Thus, the reflected light frequency ω_r, will be given by

$$\omega_r = \omega + \Omega, \tag{2}$$

where ω is the initial frequency of the light. This is how the frequency of the light can then be tuned very precisely. The precision is due to difference of orders of magnitude between the light frequency ($\sim 10^{15}$) and the sound frequency ($\sim 10^{8}$).

We can achieve a downshift of the frequency in the same way as we've achieved the upshift, simply by changing the sign of the incoming angle, i.e. by having the beam come in from the other side of the normal axis, but with the same angle. We would then get a new reflected light frequency of

$$\omega_r = \omega - \Omega. \tag{3}$$

The next section will explain, how we designed the control of the signal, which was to be fed into the AOM as an acoustic wave.

3 pyDDS Implementation

3.1 Hardware

3.1.1 AD9959 DAC converter

Theory of Operation For the implementation of this library, the AD9959 digital to analogue (DAC) converter was used. The AD9959 is a set of four direct digital synthesisers (DDS) cores, which allow independent modulation of frequency, phase and amplitude on each of the four channels. Each of these DDSs is connected to an integrated, high-speed 10-Bit DAC, which in turn creates the synthesised signals. These are the signals we then use to drive the AOMs.
To create a signal for the DAC, the DDS needs a reference signal, which is fed into its reference clock (REFCLK) input. To create a more stable output signal (i.e. a smooth signal, without sudden increases of the amplitude), it is better to have a high frequency source. In case of a low frequency source, we can increase the frequency with the internal DDS reference clock multiplier (PLL), which we can do by addressing the corresponding DDS register (as explained in Section 3.2). To create a signal, we then need to send the appropriate commands (or signals) to the appropriate DDS registers (using our micro-controller presented below). This signal is then output on the channel(s) we have indicated. Several DDS functions require specific pins to be toggled. These are, for example, passing the programmed information to the DDS chip (toggle ioupdate pin), initiating the linear sweep (use the channel profile pins - Pins 0, 1, 2, 3 - set high for rising sweep, set low for falling sweep) or resetting the DDS (toggle reset pin). The connections of these pins are explained below and the pyDDS functions use these pins to implement the desired effects.

Implementation We connect to the AD9959 using the general purpose input/output (GPIO) pins of the Raspberry Pi (RPi) and use Serial Peripheral Interface protocol (SPI) to communicate with it. In our implementation we set the communication mode to 3-wire mode, which configures the Secure Digital Input Output (SDIO) pins unidirectionally, using pin SDIO_0 as the input and SDIO_2 as the output pin [6].
To clock the DAC, we use the internal clock of the RPi. Using the "Minimal Clock Access" script provided with the pigpio library [7], we use the oscillator of the RPi as our REFCLK source (in our experiment, we set it to 50 MHz). We then toggle the reference clock multiplier on the AD9959 to generate a high frequency clock signal on the DAC, which allows us to output a stable signal in the desired frequency range (approx. 80 MHz). The subsequent communication with the DAC will be implemented through simple read and write commands to the AD9959 chip. The chip holds a total of 25 registers to program all of its functionality. Depending on the register size (in bytes), the correct amount of information is sent to the according registers to program a specific functionality. The exact information pertaining to the chip and the registers can be found in [6].

3.1.2 Raspberry Pi

The Raspberry Pi (RPi) is a small single-board computer, which is very useful for small scale control and DIY-project applications. It is powered by a 5 V Micro-USB connection [8]. Using the GPIO connections on the RPi, we connect the RPi to the DAC and obtain direct access to the Chip. The connections can be seen in Table 1.

The library we build to access the DAC's functionalities is written in Python, which is one of the primary supported languages of the RPi. We use the *spidev* and *RPi.GPIO* python libraries as the building blocks of the library and *time* for potential timing of repeated initialisations and function calls (e.g. linear sweeping of signal frequency). The *spidev*

PIN Function	AD9959/PCBZ connection	RPi connection
Pin 0	15	15
Pin 1	16	13
Pin 2	17	11
Pin 3	18	12
IO_UPDATE	19	16
CSB	20	24
SCLK	21	23
RESET	22	18

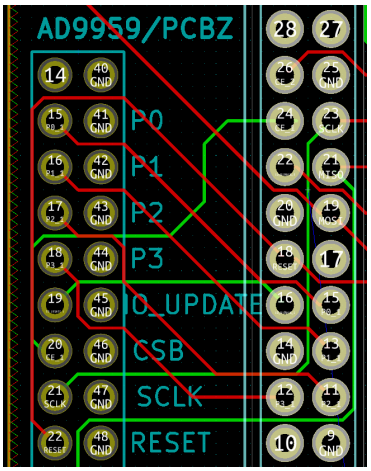


Table 1: AD9959 GPIO connections to the RPi.

library is the implementation of the SPI protocol in Python, allowing for easy access to the AD9959 register. It forms the basis for reading and writing commands to the chip. The precise reading and writing procedure will be discussed below.
The *RPi.GPIO* library allows the manual toggling of pin states on the RPi, allowing indirect manual control of the DAC chip. This is especially useful, as the DAC requires a supplementary I/O update command for programmed settings to take effect.

3.2 pyDDS library (Software)

In this section, we describe the functionality and implementation of the pyDDS library. For a quick overview of the functionality of the library, please consult Table 3 in the appendix (pyDDS functions and functionality).

At the beginning of the script, we first define all the necessary pin numbers, which can be read out from Table 1. We also define several dictionaries, attributing the correct pin numbers as well as register names and lengths for latter use. We also select the **BOARD** mode for the GPIO library so that the pin numbering corresponds to the board numbers and finally we set all the relevant pins as output pins.

The registers (names and lengths) we define in the dictionaries, correspond to the registers on the DDS, which are carefully explained in the AD9959 data sheet [6]. Here we give a brief explanation of their functionality. The basic idea is to write all settings we want the DDS to execute to its respective registers. Each register is made up of a certain number of bytes, where every byte consists of 8 bits, which often have to be addressed separately. In Table 2, we list all registers and give a very brief idea of their use. For a more detailed explanation of the full register functionality, please consult pages 36-43 of the data sheet [6].

Within the class, we first set up the DDS device and the class. We define system state variables, e.g the REFCLK, the PPL and the clock frequency and channel state variables, e.g. the currents, amplitudes, phases and frequencies that have been set. In this experiment, we decided to automatically initialise the REFCLK with 50 MHz. Finally, we reset the DDS, activate channel 0 and set all currents to maximum output.

***Ioupdate*, *reset* and *init* functions** These functions all use the *GPIO* library to set pin states. The *ioupdate* function toggles the RPi IOUPDATE pin twice to set it low -

high - low. This way we ensure that the commands sent to DDS since the last update are passed into the registers. Every *set_* function (which will be discussed later) has an ioupdate option, which allows the user to automatically push the values into the DDS register after having passed them to the function.
The *reset* function simply toggles the RESET pin to reset all registers on the DDS to their default values.
The *init* function has the same effect as the *reset* function on the DDS but additionally resets all class variables to their default values too, since they serve as the class memory. It also resets the PPL to 10 and deactivates all channels except for channel 0 as a default. The latter two settings can be customised.

***Read* and *Write* Functions** The basic functions to input data to the DDS chip are the *read* and *write* functions. These are implemented using the *spidev* library functions *readbytes* and *writebytes*.
The *read* function takes the register name as an input (defined in a dictionary at the beginning of the script) and sends the **read** command as well as the register number in one byte using *writebytes*. Then, using *readbytes* and inputting the length of the selected register (in number of bytes), we read out the information stored in the selected DDS register.
The *write* function works in a similar fashion, only instead of the **read** we use the **write** byte and the register name. The function takes the data to be written into the register as an input in the form of a list of bytes. Here it is important to ensure that the correct number of bytes is sent. Otherwise, the next byte will be interpreted as the next command, which leads to undefined behaviour. This is what we do in the assertion.

Fundamental *set_* functions The most fundamental things that need to be set in the class as well as the DDS itself are REFCLK frequency, the PPL, as well as the channel, which one wants to address any changes to. These three features are implemented through *set_refclock*, *set_freqmult* and *set_channels* respectively.
The *set_refclock* function simply sets the class variable for the REFCLK frequency to the given value and updates the clock frequency of the class taking into account the current PPL value set. The function also warns the user if the clock frequency lies out of the functionality range of the DDS, which is between 100 MHz and 500 MHz.
The *set_freqmult* sets the PPL value and also updates the clock frequency variable. It also warns the user if the clock frequency lies within the range of 160 MHz and 255 MHz, because the AD9959 data sheet doesn't guarantee operation in this range. Note that, since it is necessary to always give the correct number of bytes to every DDS register, even if we only want to change parts of the contained information, we have to copy its initial state and specify the unchanged information again within the "new" state. This is implemented in the same fashion for all registers that contain bits that shouldn't be changed by a specific *set_* function.

Simple *set_* functions The DDS is able to independently set frequency, phase and amplitude of the signal it outputs for ever channel. It can also set one of four predefined values for the current it outputs on every channel. These set functions work by passing a list of channels (or a single channel number), which one wants to set these values to and the respective value that should be set. The channels should be selected from [0, 1, 2, 3]. Each of these functions first activates the desired channels.
The frequency, phase and amplitude functions then proceed to first assert, whether the given values lie within the allowed ranges and then to encode the value into their respective tuning words, i.e. frequency tuning word (FTW), which consists of 32 bits, phase offset

word (PTW) consisting of 14 bits, and the amplitude scale factor (ASF) consisting of 10 bits. Finally they encoded data is written to the respective DDS register.
Additionally, the *set_frequency* and s*et_amplitude* functions turn off the linear sweep mode in case it was turned on previously, since otherwise the channel won't output the given value.
The *set_current* functions works slightly different to the others. Since only four settings can be chosen, namely 1/1, 1/2, 1/4 or 1/8 of the full output current, the functions takes the divider as an input i.e. a value from [1, 2, 4, 8] and makes sure that one of these has been chosen. It then encodes it into the correct bits and writes the new information to the register.
Finally, every one of these *set_* functions writes the given value into the appropriate class variable, saving the value for every selected channel. This can be used for later retrieval and read-out of the current state of the channel.

***get_* functions** Every *set_* function also has a respective *get_* function, (except for *set_refclock*, since this value can be read out directly from the refclock variable). In the case of the fundamental functions, the *get_* function reads out the value stored in the respective register, e.g. for the PPL value or the active channels and returns the values it read. The *get_* functions for the *set_* functions in the previous paragraph do two things: On the one hand they simply return the values stored in the state variable during the latest use of the respective *set_* function. On the other hand, they also print out the value currently stored in the respective DDS register, which only contains the value but not the information pertaining to the channels. This was implemented as a testing feature but can also be used to check for consistency, i.e. at least one value from the state variable should be equal to that value.
Finally, there is also a *get_state* function, which simply reads out all values stored in the DDS registers. By default, this function prints out the data in hexadecimal representation, but it also has the option to print in binary representation.

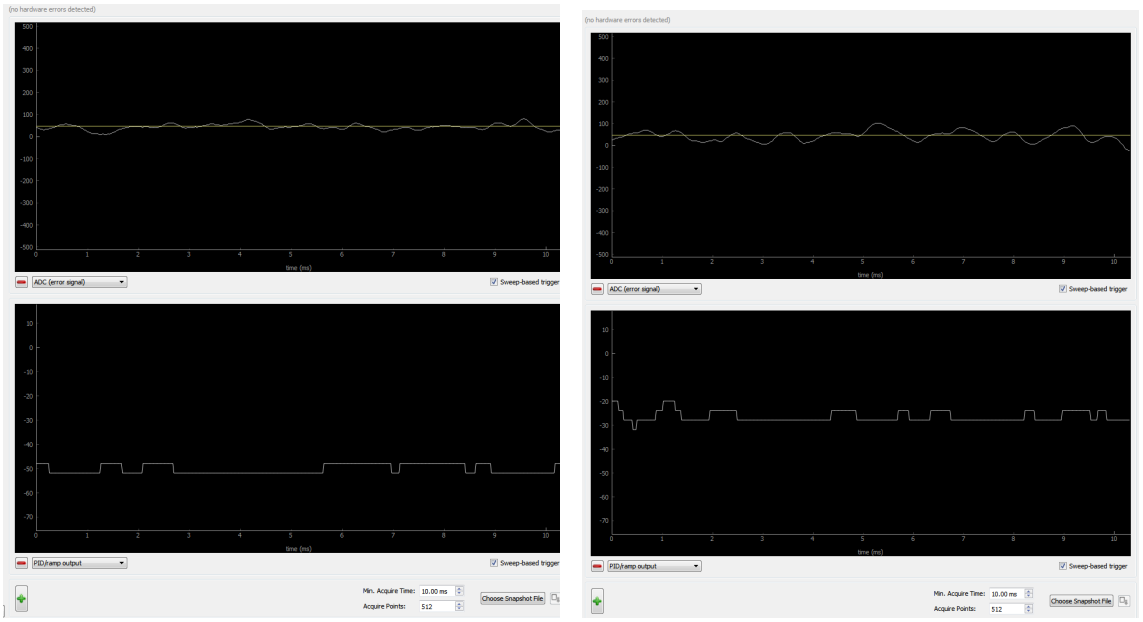
Sweep functions These functions program the DDS for linear sweeps. In our pyDDS script, we have focused on sweeps of frequency (*set_freqsweep*) and amplitude (*set_ampsweep*). For a linear sweep, the DDS requires information about the start and end value of the sweep as well as a rising and falling delta tuning word (RDW, FDW) and rising and falling interval step size (RSI, FSI). The latter determine the rising and falling slope of the linear sweep. As usual, the function needs a list of channels to which to write the commands and also has the option of directly triggering an ioupdate. Additionally, the DDS supports a 'no-dwell' mode, which resets the swept value to its starting point after the rising sweep finishes. Finally, there is also a 'trigger' option, which, if set to 'True', automatically pushes the information to the DDS and triggers the first sweep.
There are several constraints imposed on the functionality of the sweep functions by the DDS characteristics. Since the frequency and amplitude values are encoded in 32 and 10 bits respectively, this sets a minimum step size. Same holds for the time interval, which is limited to 8 bits. Thus, at a clock frequency of 500 MHz, the time interval can be chosen to be between 8 ns and 2.048 µs, which sets a limit to the total duration of a linear sweep as a function of the RDW/FDW.
Specifically, we may want to sweep through a specific region in an experiment, (we will see such an example below), which could be limited to, say, 5 MHz. In this case, choosing the smallest possible RDW and largest possible RSI would still produce a very fast sweep. This is even more the case for the amplitude, which offers significantly less space than the frequency, in terms of number of bits, for encoding its RDW.
For ease of use, we included associated sweep timer functions, *set_ampsweeptime* and *set_freqsweeptime*, which allow to select an arbitrary **sweeptime** i.e. the length of the

sweep in seconds, instead of manually inputting the RDW and RSI. This is implemented by manually selecting an RSI in the function (this should be changed in future versions of the script) and then calculating the number of steps by dividing the time by this RSI. Finally, the step size, i.e. the RDW, is calculated by simply dividing the end-to-start range by the computed number of steps. However, this function did not allow us to extend the maximum possible sweeptime. Future versions should try to find a solution for this.

***sweep_loop* and *select_CHPINS* functions** The final two additions made in this first version of the script were the *sweep_loop* and the *select_CHPINS* functions. The latter function is used in all sweep functions and serves as a way for the functions to know, which GPIO pins on the RPi to trigger, i.e. what DDS channels to trigger the sweeps on. These functions serve as a way of executing several sweeps in regular intervals. Short of a more sophisticated implementation at this point, we used the python *time* package for the interval timing. This has the clear drawback, that the communication speed of the RPi using this package is not fast enough compared to the speed of the DDS. For real use, a better solution for timing will need to be found, such as connecting directly to an FPGA.

4 "Library in Action"

The idea for this particular AOM in the TIQI lab is to detune the MOT, which is used to capture magnesium ions. Due to the fact, that lasers can naturally drift, laser-locking methods need to be employed to stabilise the laser frequency. In short, locking requires a reference frequency, which in our setup is a specific transition of iodine, and an error signal, which is given by the difference between the iodine and laser signal when performing spectroscopy of iodine with the laser. By changing the AOM frequency, we modulate the frequency of said laser, thereby changing the error signal. Below in Figure 4 some pictures of a laser-locking program in operation. Specifically, we can see the change of the output signal in the lower part of the two pictures, i.e. by comparing Figures 4(a) and 4(b). This corresponds to two different frequencies set on the DDS, namely once to $\Omega = 75\,\mathrm{MHz}$ and then $\Omega = 80\,\mathrm{MHz}$.



(a) $\Omega = 75\,\mathrm{MHz}$ (b) $\Omega = 80\,\mathrm{MHz}$

Figure 4: Laser lock and Output signal at different AOM frequencies: In the lower parts of the two figures, we can clearly see the shift of the error signal with respect to the reference frequency, induced by the change in AOM frequency. When we increase the frequency, Ω, of the AOM, the laser frequency is shifted and has thus a wider gap to the iodine transition frequency. This is the difference seen in the lower parts of the figures. The units on the vertical axis are arbitrary.

The ability to program frequency ramps on the new pyDDS allows a smooth transition from one signal to another, which pre-empts the need to manually relock the laser. If the frequency were to jump suddenly, the lock would need to be recalibrated to the new frequency, whereas the smooth transition allows the locking mechanism to continuously account for the drifting frequency. This is one of the main advantages of the new DDS system. By gradually changing the AOM frequency and therefore also the laser frequency, the lock program has time to calculate the new output signal and therefore keeps the laser locked. This is important for keeping the MOT operational and thus keeping the ions trapped.

5 Conclusion

This thesis discussed the implementation of a Python DDS communication protocol to drive AOMs. Utilising the SPI protocol to communicate with the DDS, we are able to input precise settings to the DDS and generate customised signals. The possibility of outputting frequency ramps in our signal is an advantage specific to this new DDS system, as it allows for a smooth signal transition. This is important for the locking of the laser the AOM is modulating, as the laser in question in the present setup was a MOT laser, i.e. for trapping ions. It is thus paramount, that the laser remains at a very precise frequency.

What has not been covered during this work, was the development of a remote access interface to the RPi, which would allow easy configuration of the DDS settings, without the need for command-line coding. This should be considered the immediate next steps.

On the other hand, there is clear scope for improvement in the pyDDS code, particularly with respect to pulsing ramps i.e. using a faster time function than the python time library or extending the ramp profiles from purely linear to more arbitrary ramps.

Appendices

A DDS Registers

Table 2: "DDS Registers"

B pyDDS Functions

Table 3: pyDDs Functions And Functionality"

Register Name	pyDDS Register Code (string)	DDS Address (hex)	Length (in bytes)	Short (Partial) Description
Channel Select Register	**CSR**	0x00	1	Activates Channels sets communication mode.
Function Register 1	**FR1**	0x01	3	Enables and sets reference clock multiplier value.
Function Register 2	**FR2**	0x02	2	Clears all sweep accumulators
Channel Function Register	**CFR**	0x03	3	Linear sweep mode and output current settings.
Channel Frequency Tuning Word 0	**CFTW0**	0x04	4	Frequency value (start value for sweep)
Channel Phase Offset Word 0	**CPOW0**	0x05	2	Phase value (start value for sweep)
Amplitude Control Register	**ACR**	0x06	3	Amplitude value (start value for sweep)
Linear Sweep Ramp Rate	**LSR**	0x07	2	Rising and Falling step (time) intervals for sweep
LSR Rising Delta Word	**RDW**	0x08	4	Rising step size (freq, phase, amp) for sweep
LSR Falling Delta Word	**FDW**	0x09	4	Falling step size (freq, phase, amp) for sweep
Channel Word 1	**CTW1**	0x0A	4	End value (freq, phase, amp) for sweep
Channel Word 2	**CTW2**	0x0B	4	Not used in pyDDS (for arbitrary intermediate steps)
Channel Word 3	**CTW3**	0x0C	4	”
Channel Word 4	**CTW4**	0x0D	4	”
Channel Word 5	**CTW5**	0x0E	4	”
Channel Word 6	**CTW6**	0x0F	4	”
Channel Word 7	**CTW7**	0x10	4	”
Channel Word 8	**CTW8**	0x11	4	”
Channel Word 9	**CTW9**	0x12	4	”
Channel Word 10	**CTW10**	0x13	4	”
Channel Word 11	**CTW11**	0x14	4	”
Channel Word 12	**CTW12**	0x15	4	”
Channel Word 13	**CTW13**	0x16	4	”
Channel Word 14	**CTW14**	0x17	4	”
Channel Word 15	**CTW15**	0x18	4	”

Table 2: DDS Registers

Table 3: pyDDS Functions And Functionality. All set_ functions (except set_refclock) have an optional ioupdate flag, which by default is set to False. If the flag is set to True, the command executes immediately. All optional variables are marked with an asterisk (*) and their default values are indicated.

Function	Arguments	Returns	Effect
ioupdate	-	-	Toggles IO_UPDATE pin on the RPi to load all commands to the DDS sent since last ioupdate
reset	-	-	Resets the status of the DDS, sets all register entries to default
init	*freqmult=10, *channels = 0	-	Resets the status of the DDS, then sets all register entries to default. Also resets the stored values from set_ functions to default. By default, sets frequency multiplier to 10 and activates channel 0 (and sets communication mode to 3-wire).
write	register, data	-	Takes a register name (string) to write to and data (list) of appropriate number of bytes to write to the register.
read	register	data (list)	Takes a register name (string) and returns the data stored in the register.
get_state	-	-	Prints all values in DDS registers in either hex or bin format. By default in hexadecimal.
set_channels	channels, *ioupdate		Takes in channel number from 0, 1, 2, 3 or list of any number of channels.
get_activechannels	-	channels (list)	Returns list of all channels which are currently active.
set_refclock	frequency	-	Sets the class variable refclock_freq and automatically updates class variable clock_freq with new refclock value. Prints out current values for refclock_freq, clock_freq and freqmult, which together form the clock frequency.
set_freqmult	freqmult	-	Sets the frequency multiplier on the DDS. Automatically updates clock frequency.
get_freqmult	-	freqmult	Returns current value of frequency multiplier set on the DDS.
set_frequency	channels, frequency	-	Takes in a frequency to assign and a (list of) channel number(s) to which to set it.
get_frequency	-	frequencies (list)	Returns a list (with four entries) of the frequencies assigned to every channel.

Table 3: pyDDS Functions And Functionality. All set_ functions (except set_refclock) have an optional ioupdate flag, which by default is set to False. If the flag is set to True, the command executes immediately. All optional variables are marked with an asterisk (*) and their default values are indicated.

Function	Arguments	Returns	Effect
set_current	channels, divider	-	Takes in divider and (list of) channel(s). Sets the channel(s) output current to one of four possible settings: 1/1, 1/2, 1/4 or 1/8 of the full output current.
get_current	-	currents (list)	Returns the current values set in all channels as a list.
set_amplitude	channels, scale factor	-	Sets amplitude scale factor to given channels, i.e. scales the voltage amplitude between 0 and 1 of full output.
get_amplitude	-	amplitudes (list)	Returns the voltage amplitude scale factors set in all channels as a list.
set_phase	channels, phase	-	Sets phase offset (as number between 0 and 359.987 deg) for selected channel(s).
get_phase	-	phases (list)	Returns the phase offsets currently assigned to all four channels in a list.
set_ampsweep	channels, start_scale, end_scale, RSS, RSI, *FSS=False, *FSI=False, *no_dwell=False, *ioupdate= False, *trigger=False	-	Turns on amplitude linear sweep mode and programs the settings: start/end amplitude, rising/falling step size and rising/falling time interval. If FSS and FSI are not passed, they are set to equal RSS and RSI If no_dwell is activated, amplitude returns to start value after the sweep. If trigger is activated (must also activate ioupdate), sweep executes immediately.
set_ampsweeptime	channels, start scale, end scale, sweeptime, *no dwell=False, *ioupdate= False, *trigger=False	-	Activates linear amplitude sweep mode. Sweeping from start_scale to end_scale with a duration of sweeptime. Options are the same as set_ampsweep
set_freqsweep	channels, start_freq, end_freq, RSS, RSI, *FSS='same', *FSI='same', *no_dwell=False, *ioupdate=False, *trigger=False	-	Turns on frequency linear sweep mode and programs the settings as passed: For a description of the settings, see the set_ampsweep entry.
set_freqsweeptime	channels, start_freq, end_freq, sweeptime, *no dwell=False, *ioupdate= False, *trigger=False	-	Activates linear frequency sweep mode. Sweeping from start to end frequency with duration sweeptime. Options the same as set_freqsweep.
sweep_loop	channels, reps, interval	-	Initiates a loop of reps PIN toggles with a specific interval between toggles.
select_CHPINS	channels	PINS (list)	Returns a list of PIN numbers corresponding to selected channel profile PINS.

References

[1] R.P. Feynman, *Simulating Physics with Computers*, Int. J. Theoret. Phys. **21** (6-7), 467-488 (1982)

[2] W.M. Itano, D.J. Wineland *Precision measurement of the ground-state hyperfine constant of* $^{25}Mg^{+}$, Physical Review A, **24** No.3 (1981)

[3] M. Marinelli, *High finesse cavity for optical trapping of ions*, Master Thesis, ETH Zürich (2014)

[4] C.J. Foot, *Atomic Physics*, Oxford University Press, (2005)

[5] B.E.A. Saleh, M.C. Teich *Fundamentals of Photonics*, Wiley & Sons, (1991)

[6] Analogue Devices *AD9959 DataSheet* [Online] August 2017 `http://http://www.analog.com/media/en/technical-documentation/data-sheets/AD9959.pdf`

[7] Minimal Clock Access script [Online] August 2017 `http://abyz.co.uk/rpi/pigpio/examples.html#Misc_code`

[8] Raspberry Pi Wiki, Hardware [Online] September 2017 `https://elinux.org/RPi_Hub#Hardware_.26_Peripherals`