Computer Control System for Cold Atom Experiments

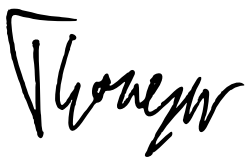
Tore Homeyer

Bachelorarbeit in Physik
angefertigt im Institut für Angewandte Physik

vorgelegt der
Mathematisch-Naturwissenschaftlichen Fakultät
der
Rheinischen Friedrich-Wilhelms-Universität
Bonn

September 2022

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die Zitate kenntlich gemacht habe.



Bonn, 30.09.2022
Datum

Unterschrift

1. Gutachter: Prof. Dr. Sebastian Hofferberth
2. Gutachter: Prof. Dr. Dieter Meschede

Acknowledgements

My thanks go to everyone inside and outside the group who helped with this thesis. First of course to Sebastian, for allowing me to write my thesis in the NQO group and for the quantum optics lecture that brought me here. Professor Meschede, for being the second corrector and his excellent optics and QO lectures. Hannes, Cedric and Julia for their company in HQO and ready help whenever needed. Everyone else in the group, for interesting conversations over lunch and the amazing atmosphere in general. My parents Anja and Sascha for always supporting me and being the best parents I could wish for. Finally, Rhea for being my partner and all around awesome person (and for finding all comma-related mistakes I made).

Contents

CHAPTER 1

Introduction

Nonlinear quantum optics concerns itself with investigating strong interactions of photons with each other by creating nonlinearities in optical materials. One way to do so is to use ultra cold atoms with an electron excited to a very high state, so called Rydberg atoms [1]. Rydberg atoms exhibit a number of interesting properties: they are highly polarisable with strong dipole-dipole interactions between each other, have long lifetimes and suppress additional Rydberg excitations nearby through a blockade mechanism [2]. These strong interactions can be mapped to single photons, which leads to strong non-linearities [3], meaning strong photon-photon interactions can be mediated using Rydberg atoms. This allows the creation of quantum devices such as single-photon transistors [4] and multi-photon absorbers [5], which have applications in quantum computing and quantum information technology.

The Hybrid Quantum Optics experiment (HQO) [6] plans to create hybrid quantum systems of photons, Rydberg atoms and superconducting circuits by coupling ensembles of rubidium Rydberg atoms to integrated microwave circuits, photonic chips or electromechanical oscillators [7, 8]. The rubidium atoms are first cooled and captured in a magneto-optical trap (MOT) [9]. Then, they are moved to a cryostat chamber using a magnetic transport system [10], where they are are trapped over an atom chip [11] and excited to Rydberg states. Here the Rydberg atoms shall for example interact with a microwave oscillator to cool one of its vibrational modes to the quantum-mechanical ground state [7, 8]. The experimental process requires precise timings of various devices (function generators, lasers, frequency references, etc.) with diverse time scale requirements. Some steps like the MOT loading need devices operating on the milli- to microsecond time scale, while the latter experiment parts requires laser pulse lengths of a few nanoseconds, meaning a reliable and fast computer control system with capabilities over the complete time scale range is needed.

The aim of this thesis is to adapt the experiment control system used in the Rubidium Quantum Optics (RQO) [12] and Ytterbium Quantum Optics (YQO) [13] experiments to the new setup in HQO. The heart of the system is an ADwin-Pro II [14] (from here on shortened to 'ADwin'), providing analogue voltage sequences (e.g. used for power supplies for the magnetic field coil currents) and digital triggers to most other devices with microsecond resolution, while a fast pulse generator triggered by the ADwin is used to reach nanosecond time scales. First, the design principles and workings of the existing software controlling the ADwin, as well as the communication protocols used for the different experiment devices, have to be understood. After setting up the lab computers and network, the experiment control devices are successively installed. The Python scripts controlling them are adapted to the HQO setup and requirements, all devices are tested. Finally, the whole system is tested with a realistic workload.

CHAPTER 2

Control Structure

2.1 Design Philosophy

The control system has to fulfil the following requirements:

- Centralisation; control of all devices in the experiment from a single computer
- Precision; timings in the experiment sequence have to be as accurate as possible, while providing resolutions from ms to ns
- High stability; every sequence should be the same
- High reliability; running the experiment for long times should cause no issues and not impact the stability
- Data protection; all experiment data, including metadata, is saved/stored and backed up periodically
- Flexibility; new devices can be integrated easily

To this end a graphical user interface (GUI) first developed in Stuttgart for the SuperAtoms experiment is used [15].

2.2 Experiment Control GUI

The experiment control GUI is used to create arbitrary voltage sequences (both analogue and digital) with a time resolution of 20 µs, which are output using the Jäger Messtechnik GmbH ADwin-Pro II [14]. During normal operation these experiment cycles are repeated for as long as the experiment is running. Additionally the GUI executes a Python script at the beginning of each cycle to prepare all other devices, like function generators, cameras and digital-to-analogue converters (DAC). Every cycle has a predefined wait time before the hardware output starts, to make sure that all instruments are ready. If the time-critical script does not finish in this time, an error is raised. Other Python scripts may be executed in a non-time-critical manner, meaning the output starts without waiting for them to finish. All experiment data, including time stamp, global counter as cycle identifier and variables defined in the GUI are saved to a database (see Section 2.4.2), where they are available to the other devices. An

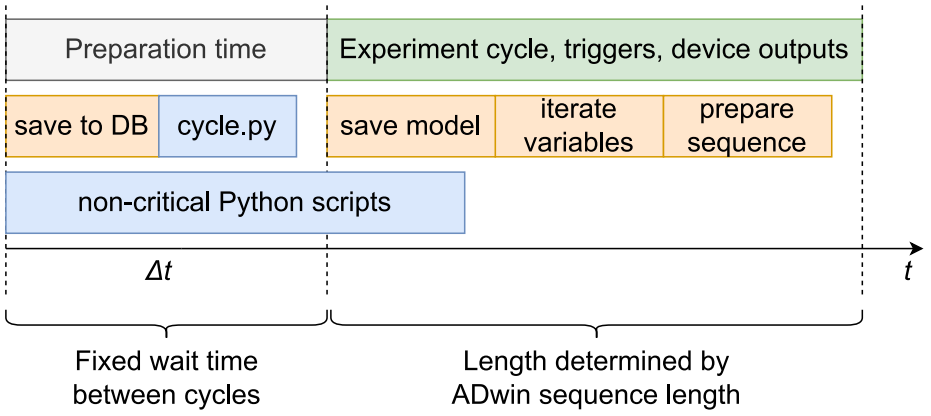


Figure 2.1: Schematic of an experiment cycle. Δt is the fixed wait time during which the variables are saved to the database and the time critical Python script has to finish. The next sequence is prepared during the ADwin sequence output.

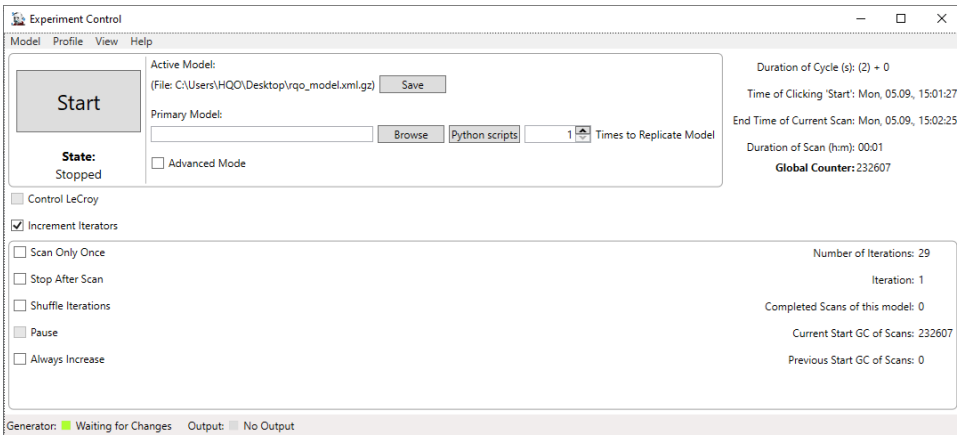


Figure 2.2: Main window of experiment control GUI.

illustration of this process is shown in Fig. 2.1. The data transfer speeds from the control computer to the ADwin was measured using a program provided by Jäger Messtechnik GmbH. With the current network setup they are 17.63 MB s^{-1} to the 'first in, first out' storage and 18.21 MB s^{-1} for other data transfers.

In the main window (Fig. 2.2) the cycling can be started. It also gives information on the currently loaded sequence (here described as 'Active Model'), some metadata, and access to settings like the Python script used. The available outputs, wait time between cycles and the database access credentials can be configured and saved as a profile.

2.2.1 Analogue and Digital Outputs

Fig. 2.3 shows a cut-out of the GUI analogue output tab. Here the voltage sequences can be configured for each channel. In every step the value in V and the duration in ms can be set. The standard setting is to output a constant voltage for the time specified, but linear ramps are implemented as well. By combining these methods arbitrary patterns can be built. Another option to create an arbitrary sequence is to load a

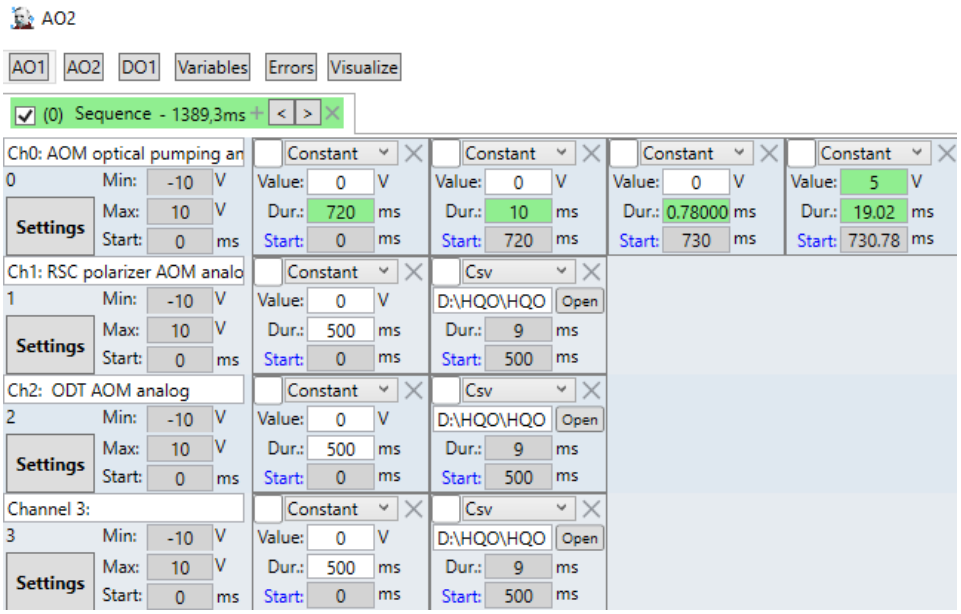


Figure 2.3: The experiment control analogue output, each channel can be programmed to output voltage patterns. Standard options are constant values and linear ramps, arbitrary sequences can also be given as CSV files.

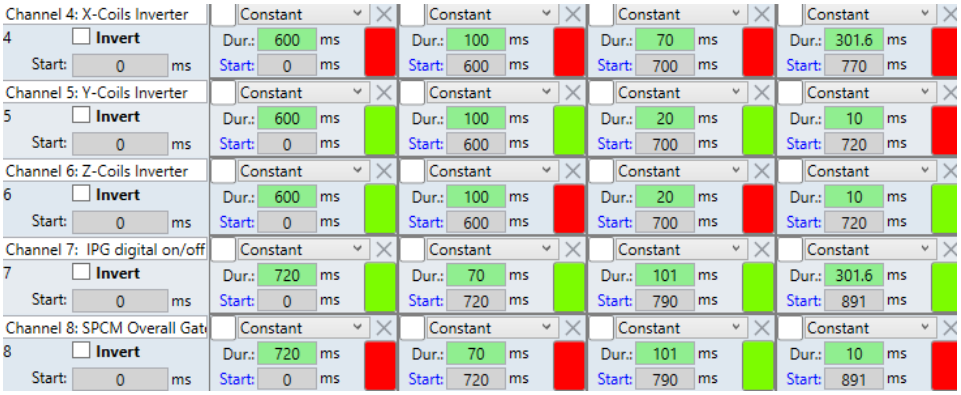


Figure 2.4: The experiment control digital output, each of 32 channel can be set individually to HIGH (green) or LOW (red) for arbitrary time sequences.

CSV file, where the voltage values are defined for each time step of 20 µs.

The digital output (Fig. 2.4) works in the same way as the analogue one, except that the output can only be set to HIGH (green) or LOW (red). The digital output is used to trigger all other experiment devices, directly or indirectly via a faster pulse generator.

2.2.2 Variables and Iterators

Three types of variables are definable in the GUI (Fig. 2.5):

- Static: Static variables keep their value until the user changes it.

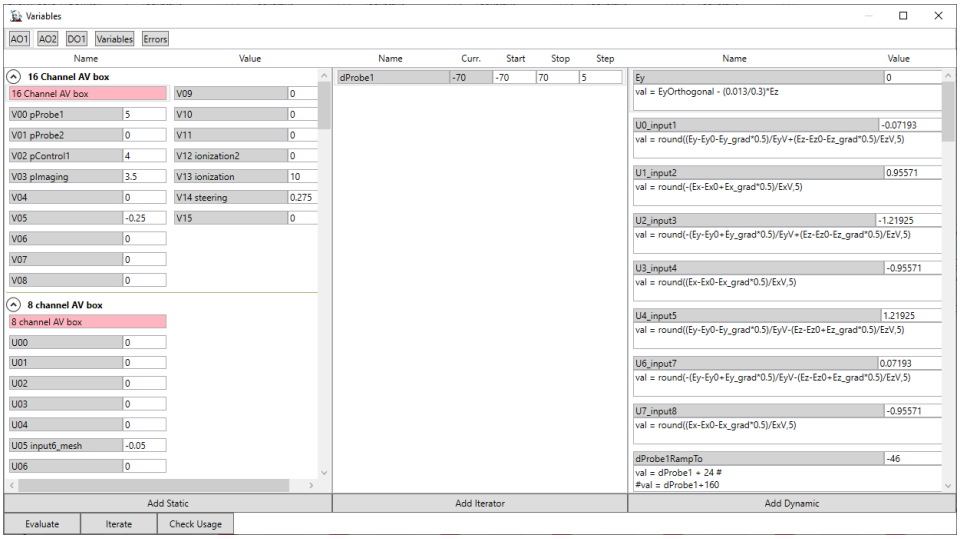


Figure 2.5: Variables tab of the control GUI. Iterators, dynamic and static variables defined here can be used as values/durations in the output tabs or provided to other devices via the database.

- Iterator: Iterators go from specified 'Start' to 'Stop' values with step size 'Step', with one step at the beginning of each cycle. Should the new value exceed 'Stop', the iterator is set to 'Start' again. This allows scanning over a range of experimental parameters.
- Dynamic: The value of dynamic variables is the result of Python code, which has access to all static variables and iterators. The calculation uses the current value of the iterator, so dynamic variables can change during a scan.

As these variables can be used in 'Duration' and 'Value' fields in the output tabs, the resulting voltage sequences can be changed from one cycle to the next.

2.3 Device Communication

Only the ADwin is programmed directly by the experiment control GUI, all other devices are controlled with Python scripts. Every cycle (in the waiting time) the GUI executes `cycle.py` as a time-critical python script, meaning the GUI waits with the hardware output until the script is finished. Should the waiting time be too short, an error is raised. `cycle.py` uses network sockets to send starting signals to instrument server scripts running on other lab computers, where the experiment devices are connected. These servers then handle database communication and control their devices.

2.3.1 `cycle.py` and Sockets

The network communication in `cycle.py` uses the `socket` package, which is an implementation of the BSD socket interface [16]. Two different network protocols are used in the script: User Datagram Protocol (UDP) and Transmission Control Protocol (TCP). UDP is connectionless, the data sent is addressed to the intended recipient, but there is no guarantee it will be received, as there is no acknowledgment sent back [17]. With TCP a connection has to be established before data is sent and all received packages are

```
######========== MCC DAC boxes ==========
try: # MCC Analog boxes.
        HOST = IP_HQODEVICECNTRL
        PORT = PORT_USBANAL
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((HOST, PORT))
        s.sendall(DATA_USBANAL)
        s.close()
except:
        message = "Error with USB analogue boxes"
        print(message)
        raise
```

Listing 1: Code example of how `cycle.py` communicates with the instrument servers.

acknowledged, it is a connection based protocol [18]. This means additional overhead, but also makes sure the recipient is actually receiving the messages and an error can be raised if communication was not possible.

The bulk of `cycle.py` consists of code blocks to communicate with instrument servers over TCP. An example is shown in Listing 1. First, the IP address `IP_HQODEVICECNTRL` and open port `PORT_USBANAL` of the computer where the instrument is connected are read from a configuration file. Then a socket of the `AF_INET` address family (meaning it is using IPv4) with type `SOCK_STREAM` (TCP uses stream-type sockets) is opened. It connects to the previously read `(HOST, PORT)` pair and sends a data package `DATA_USBANAL`. The data is a unique string for each device. If the package was received correctly, the socket is closed and the next device can be addressed. Should any errors occur (in opening the socket, connecting or sending data) an exception with a message identifying the problematic component is raised and the script is terminated, the experiment cycle does not start.

```
# Broadcast internal network
# Call pulse generator sequence
# Create a UDP socket
try:
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        s.bind((IP_HQOCONTROL, 0))
        s.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
        server_address = ('<broadcast>', PORT_BROADCAST)
        print ('sending data') # Send data
        sent = s.sendto(DATA_START, server_address)
except Exception as e:
        print (e)
finally:
        print('closing socket')
        s.close()
```

Listing 2: UDP socket used for broadcast at the beginning of `cycle.py`.

Only one UDP socket is used at the beginning of `cycle.py` to broadcast a message to the whole network. The code is shown in Listing 2. This time `SOCK_DGRAM` is used to specify the socket as UDP. It is bound to the local IP address `IP_HQOCONTROL` and set to enable broadcasts. The server address is `'<broadcast>'`, which is the special form for the broadcasting address *255.255.255.255*. Finally, `DATA_START` is broadcast to the network and the socket is closed.

2.3.2 Instrument Server

The instrument server (Listing 3) is a Python class used by most devices to listen for their starting signals. On instantiation it takes three arguments:

- `instrumentIdentity`: A string describing the device using the class, e.g. `'Arbitrary Function Generator'`
- `instrumentThread`: A thread class handling the specific instrument in its `run()` method
- `comms`: A dictionary containing the communications port, expected data and address of the host computer, for example:
  `comms = {'PORT': PORT_FUNCGEN,'DATA': DATA_FUNCGEN,'ADDR': IP_HQOCONTROL}`

Its function is very straightforward, on running the server thread it opens a TCP socket listening for incoming connections on the specified port. When `cycle.py` opens the corresponding socket the connection is accepted. The received data is checked, only when it matches the expected content and comes from the correct IP address `IP_HQOCONTROL`, the instrument thread is started.

```
1
2  # Comms is a dictionary: {'PORT': int, 'DATA': str, 'ADDR': str}
3  # instrumentThread is an instantiated class containing a method run()
4  # instrumentIdentity is a human readable string that identifies what the service is doing
5  class InstrumentServer(threading.Thread):
6          def __init__(self, instrumentIdentity, instrumentThread, comms):
7                  threading.Thread.__init__(self)
8                  self.instrID = instrumentIdentity
9                  self.instrumentThread = instrumentThread
10                 self.comms = comms
11
12         def run(self):
13                 print("Ready for sequence: Waiting for connection")
14                 host = ""
15                 port = self.comms['PORT']
16                 buf = 1024
17                 # Open socket and listen to network ping
18                 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
19                 s.bind((host, port))
20                 s.listen(1)
21                 while True:
22                         try: #Receive data
23                                 conn, addr = s.accept()
24                                 wholedata = b""
25                                 while True:
26                                         data = conn.recv(buf)
27                                         wholedata += data
28                                         # If data comes from the correct source and contains
29                                         # the start signal, start the instrument thread
30                                         if (data == self.comms['DATA'] and
31                                             addr[0] == self.comms['ADDR']):
32                                                 t = time()
33                                                 self.instrumentThread.run(t)
34                                                 break
35                                         if not data: break
36                                 conn.close()
37                         except:
38                                 print("%s communication error......" % str(self.instrID))
39                                 raise
```

Listing 3: Instrument Server class used in most devices to listen for network pings.

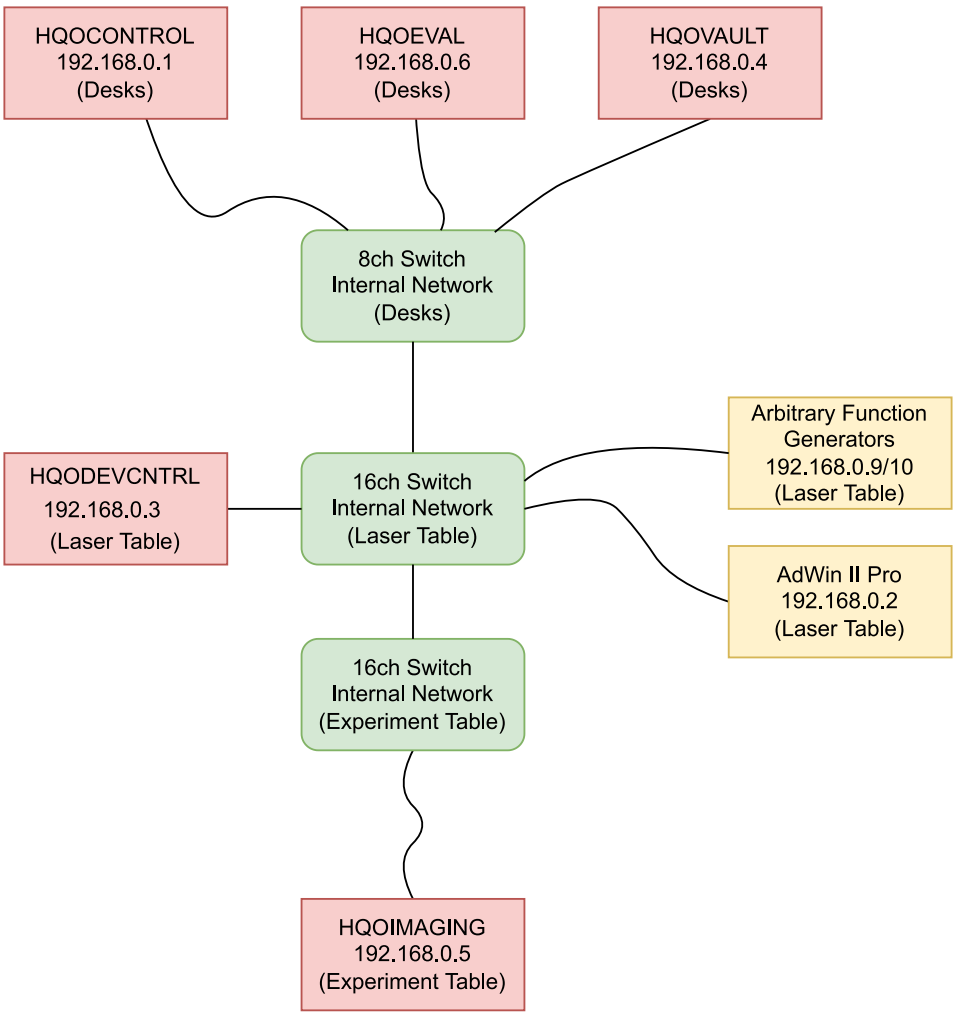


Figure 2.6: Network schematic of the HQO internal lab network.

2.4 Data Management

2.4.1 Network Communication

All computers are connected to the internet as well as an internal lab network, which is not exposed to the outside. Other instrument devices requiring an Ethernet connection are also connected to the lab network, a schematic is shown in Fig. 2.6. The complete computer setup with all devices and Python scripts is shown in Appendix A.1.

2.4.2 Database

All experiment data is saved to a MySQL database running on HQOVAULT at the beginning of each cycle. This includes the current global counter, all variable and iterator values, and metadata such as time stamps, operating mode and the number of completed iteration scans. Table 2.1 shows one row of

Cell name	Value
globalCounter	26185
startTime	2022-07-15 23:50:15
startCounterOfScans	6883
iterationOfScan	25
numberOfIterations	51
completedScans	378
variables	V00 4.8, V01 4.8, (...), iterator 4.8, cntGlobal 26185, CycleDuration 300
iterators	iterator 4.8 0 10 0.2
operatingMode	ITERATION
startCounterOfRoutine	0
modelNumber	0
routineArray	0

Table 2.1: Database entries saved during an experiment control GUI test.

database entries. Instrument threads can then access the database to obtain values with which to program the devices using the MySQL Connector Python package [19]. Database access time ranges from 8 to 20 ms for each device control script.

2.4.3 Backups

A backup of the database is performed every night on HQOVAULT, these local backups are kept for 30 days. The network drive on HQOVAULT is synchronised to a backup on nqovault using FreeFileSync [20]. Finally nqovault is backed up to Institute of Applied Physics (IAP) hardware daily.

CHAPTER 3

Experiment Devices

This chapter provides an overview over all experiment control devices. It presents important information on their function, how to communicate with them and characterises their behaviour with test measurements where appropriate. All measurements were carried out after the whole system was set up and use a realistic workload comprising of an experiment sequence from RQO. The communication timings were measured using the inbuilt Python function `time.time()` [21].

3.1 ADwin-Pro II

The ADwin-Pro II is a real-time system, capable of precisely timing analogue and digital voltage output sequences [14]. It is directly programmed from the experiment control GUI, which transfers the sequence data over the internal lab network. The HQO ADwin shown in Fig. 3.1 provides 16 analogue and 32 digital output channels. It is the heart of the computer control system, sending trigger pulses to all other devices, either directly or indirectly by starting more precise pulse generators. If the number of devices in the experiment increases, the ADwin chassis provides easy expandability with room for additional in-/output cards.

Figure 3.1: ADwin-Pro II mounted on top of the laser table. Next to the ADwin module are the digital (high density connector) and analogue (BNC sockets) output cards.

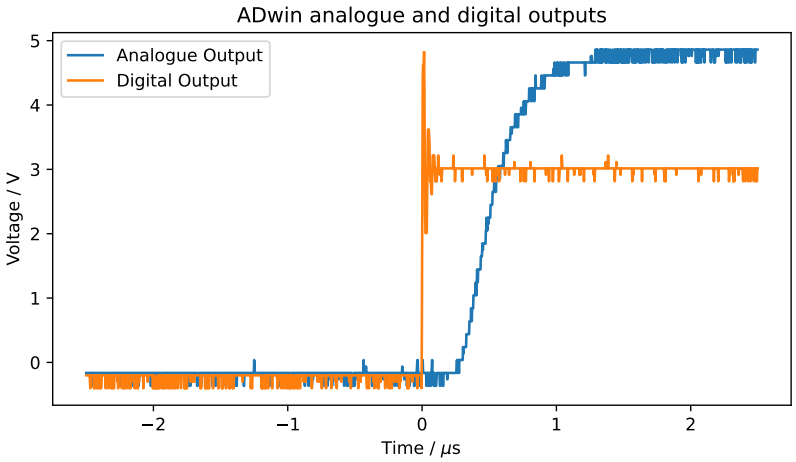


Figure 3.2: ADwin digital and analogue outputs started at the same time, measured with an oscilloscope. Rise time of the analogue output is much slower, it settles after $(1.3 \pm 0.1)\,\mu s$, while the settling time of the digital signal is $(0.10 \pm 0.02)\,\mu s$.

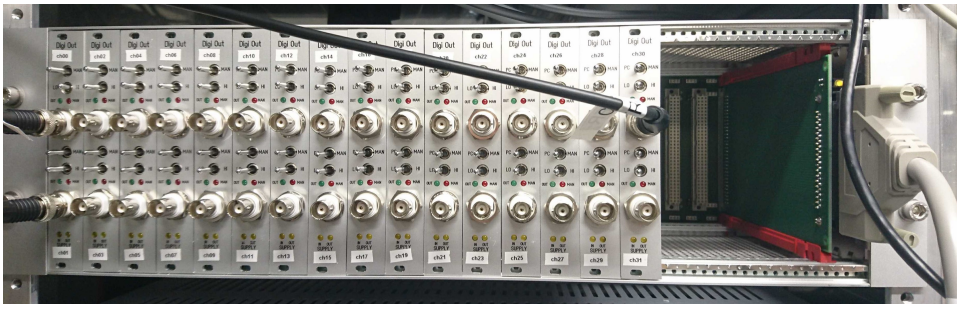


Figure 3.3: ADwin digital output box, used to break out the high density connector (right) to individual BNC sockets (left). The channels can also be manually controlled using the toggle switches.

3.1.1 Analogue Output

Two 'Pro II AOut-8/16' output modules are installed directly to the ADwin chassis, providing 16 channels of analogue voltage output with a time resolution of 20 µs. Each channel has a range of −10 to 10 V with a settling time of 3 µs according to the manufacturer [22, p. 11]. They will primarily be used to control the magnetic field coil currents and acousto-optic modulators (AOM). An analogue output (jump from 0 to 5 V) is shown in Fig. 3.2, its settling time is $(1.3 \pm 0.1)\,\mu s$, which is faster than the manufacturer claims.

3.1.2 Digital Output

The digital channels are output using the 'Pro II-DIO-32' module and an additional board to break out the 32 channels to individual BNC connectors (Fig. 3.3). They will provide trigger signals to start all other devices. Every channel can be controlled with the ADwin or manually set to HIGH/LOW. The digital outputs provide trigger signals for other instruments like function generators and cameras. Fig. 3.2

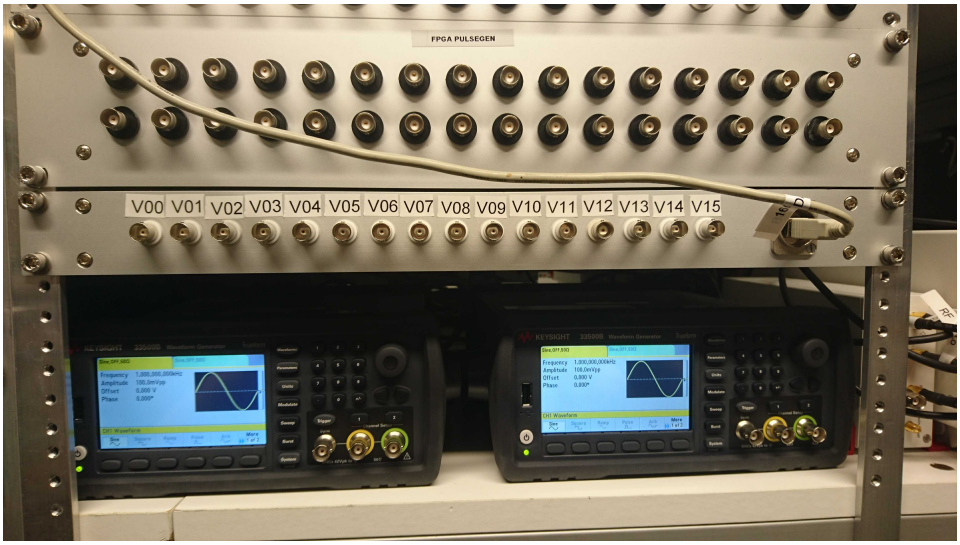


Figure 3.4: FPGA pulse generator, 16 channel USB analogue output and two Keysight arbitrary function generators are mounted in a rack on top of the laser table.

also shows a digital output (orange) started at the same time as the analogue one (blue). The former rises much quicker and settles after (0.10 ± 0.02) µs.

3.2 Arbitrary Function Generators (AFG)

Two Keysight 33522B arbitrary function generators [23] with two channels each (at the bottom in Fig. 3.4) are used to shape the control and probe laser output power profiles. Both AFGs are controlled with the script `funcgen_server.py`, which was adapted from the YQO implementation. Currently implemented model functions are 0 V constant, Gaussian function, sum of two Gaussian functions, Tukey window function, sum of two Tukey and two Gaussian functions and a linear ramp from 0 to 10 V. The last two models are also available in a calibrated form, dependent on the output laser power.

An instrument server is started to wait for the appropriate network ping, then the script reads the current variables from the database. All variables used to program the selected function need to be of the form 'fgen{1/2}ch{1/2}p{00–16}', e.g. 'fgen1ch2p05' for function generator 1, channel 2, parameter 5. In general, parameter 00 defines the length of time for which the function is calculated in ns, parameter 01 is an integer choosing the model to be used, 02 to 15 are the function parameters (amplitude, mean, standard deviation, etc.) and parameter 16 is the sampling rate determining the time step size in the calculations. All parameters have to be defined in the experiment control GUI, even if not all of them are needed for every model. An example from the GUI with all variables for one channel is shown in Fig. 3.5.

The current variables are then compared to the values of the previous cycle, the channels are reprogrammed only if the values differ to reduce unnecessary calculations. Next, the voltage values for the selected model are calculated with the given parameters for every time step. As the AFGs are programmed using the 'Standard Commands for Programmable Instruments' (SCPI) [24], the list of voltages is converted to a comma separated string, which is sent to the AFGs with other commands to

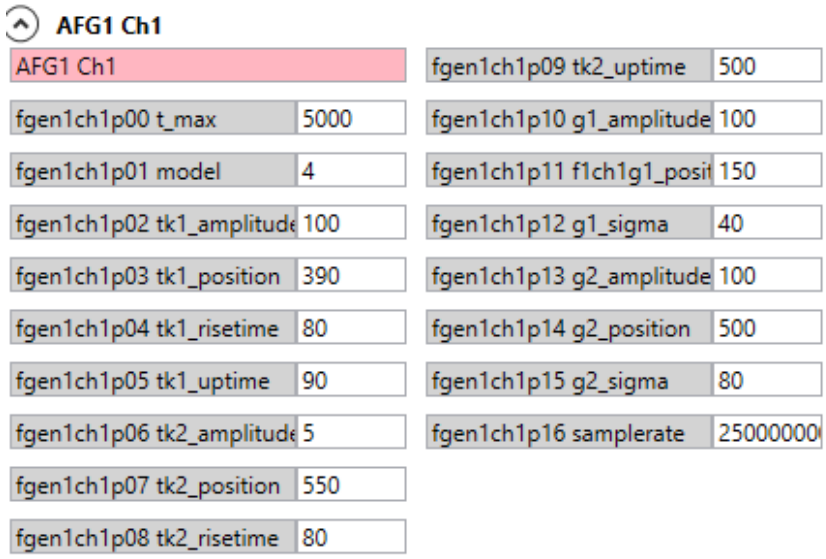


Figure 3.5: Set of variables for one AFG channel.

configure the device using a TCP socket. The function output starts once an external trigger is received, for example from the ADwin digital output. The time needed from network ping until the AFGs are ready to output is (45 ± 10) ms.

Fig. 3.6 shows the output of one channel when programmed with a Gaussian function (10 V amplitude, $\mu = 1\,000$ ns, $\sigma = 400$ ns) measured with an oscilloscope, the trigger pulse is supplied by the ADwin digital out. The expected bell curve is clearly visible, although it is shifted slightly by the inevitable delay between receiving the trigger and actually starting to output. The latency is (190 ± 10) ns, which is slightly higher than the < 135 ns specified by the manufacturer [23, p. 25].

3.3 Direct Digital Synthesizer

A direct digital synthesizer (DDS) is used to provide reference frequencies (10 to 210 MHz) for laser locks in the HQO experiment. It is based on the Analog Devices AD9959 DDS board [25], combined with an Arduino microcontroller capable of programming the DDS board output over a serial interface and a breakout board to provide BNC outputs (breakout board and Arduino code were created by Michael Schlagmüller for his dissertation [26]). The Arduino itself is controlled from HQODEVICECONTROL using a USB-to-Serial converter and a Python script `DDSsocket.py`. The HQO box containing all hardware as seen in Fig. 3.7 was built by Julia Gamper during her bachelor's thesis [27].

First, `DDSsocket.py` finds the virtual COM port the DDS box is connected to, by looping through all active COM ports and checking if the manufacturer matches the USB-to-Serial converter. In previous software versions the COM port number had to be hard coded. The script uses the instrument server class to listen to the experiment control. The variables used to set the output frequency are `dProbe1`, `dProbe2` for channels 1 and 2 respectively. In addition to constant frequencies, a linear ramp can be used if `dProbe{1/2}RampTo` and `dProbe{1/2}RampSteps>1` are set. Channel 3 is used for a possible Raman lattice, while channel 4 is currently not in use. The Arduino code also supports changing the

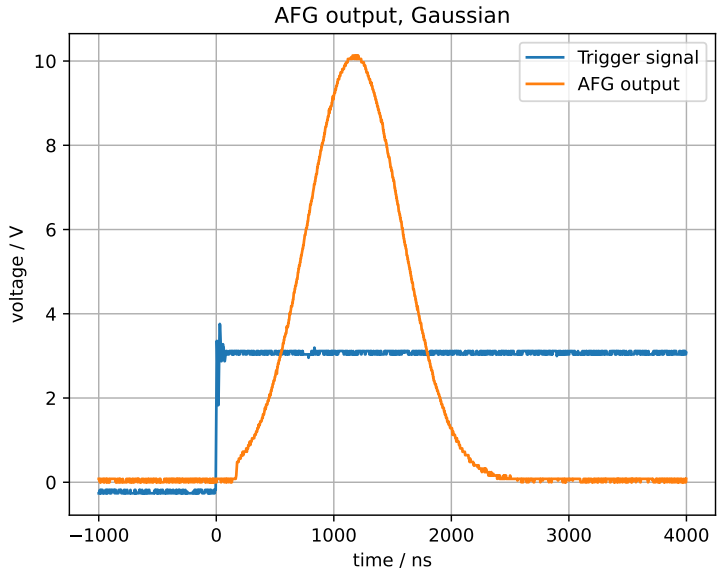


Figure 3.6: Trigger pulse and AFG output when programmed with a Gaussian function, measured with an oscilloscope. The latency from trigger to output of (190 ± 10) ns is visible at the beginning.

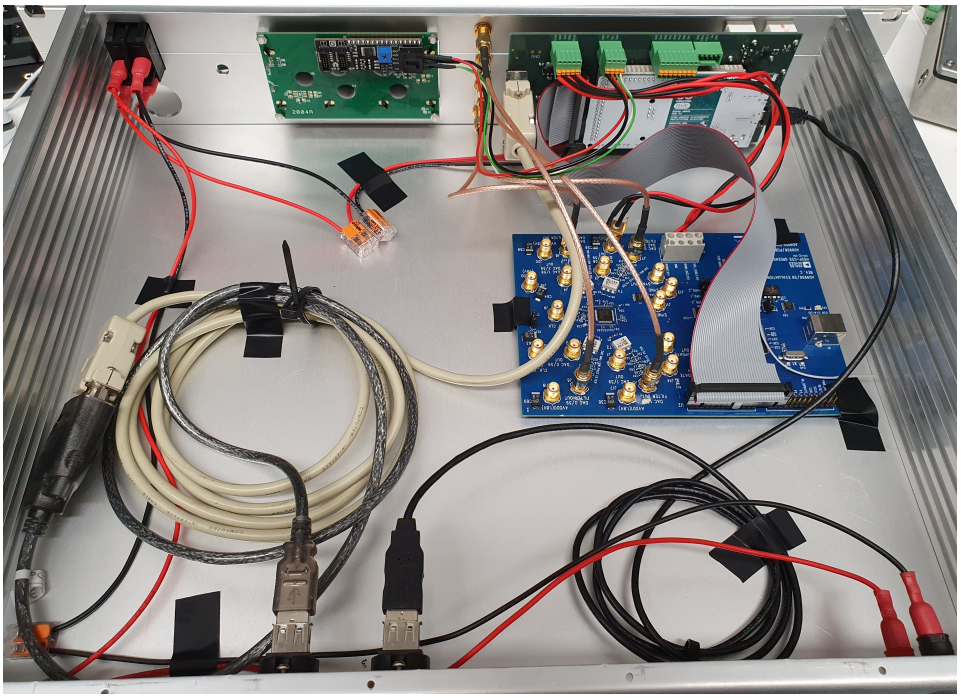


Figure 3.7: Reference frequency box with Arduino (top right), breakout board (top left) and AD9959 DDS board (middle right). Picture by Julia Gamper.

Figure 3.8: The FPGA pulse generator breakout board was designed by Helmut Fedder and Felix Engel [28] and provides 24 digital channels in addition to a trigger input.

output amplitude, triangular frequency scans and lists of frequencies to iterate over once a trigger is received. The commands are sent to the Arduino over the USB-to-Serial converter using the `pyserial` package, which works very similar to the `socket` package used in network communications. The strings sent are of the format '<command, channel, parameters>', e.g. '<f, 0, 20000000>' to set channel 1 to 20 MHz (note that the channels in the commands start at 0). On average (960 ± 30) ms are needed to reprogram all three channels.

3.4 FPGA Pulse Generator

For applications where a higher time resolution than what is provided by the ADwin is needed, a field-programmable gate array (FPGA) based pulse generator was developed by Helmut Fedder and Felix Engel in Stuttgart [28], who also built the device in use in the HQO lab (shown in Fig. 3.8). The 24 channel pulse generator is based on an OpalKelly XEM3005 FPGA [29], combined with a breakout board for power supply, output connectors and a trigger interface. All channels can be programmed individually with a time resolution of 2 ns.

The patterns are defined in a text file containing Python code as a list of tuples `(channels, time)`, setting the given channels to HIGH for the duration in ns. A short example is given in Listing 4, this pattern would set channel 1 to HIGH for 1 µs, all channels to LOW for `time1` and finally channels 2 and 3 to HIGH for 1 µs. Channel 3 is then left on HIGH, while all other channels are set to LOW. The output is only started when an external trigger is provided, and the pattern is run once per trigger. More complex Python code can also be used to create patterns, as long as the end result contains a list of tuples called `pattern`, and the continuos channels, loop and trigger mode are defined. A set of Notepad++ scripts was created by Asaf Paris-Mandoki to help with this process, they allow testing the code and visualisation of the resulting sequence (see Fig. 3.9).

The Python script controlling the pulse generator is called `pg_hqo.py`, it is executed on HQODEVICE-CONTROL. It listens for the UDP broadcast sent by the experiment control to start programming the pulse generator. After the variables are read from the database the `pulsepattern.txt` containing the

```
pattern = [(['ch1'],1e3),([],variables['time1']),(['ch2','ch3'],1e3)]
continuousChannels = ['ch3']
loop=False
triggered=True
```

Listing 4: Pulse patterns are defined as a list of tuples (channels, time) in a text file; continuos channels and other behaviour can be set as well.

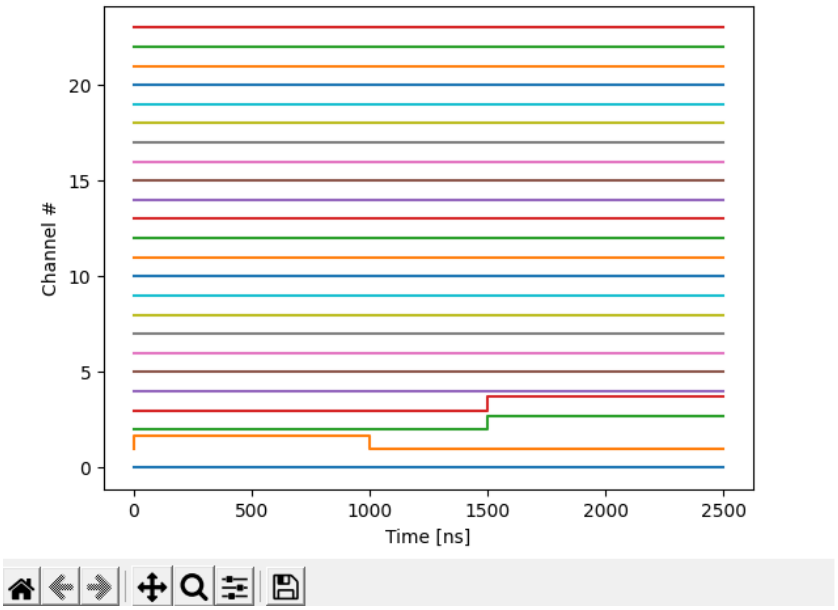


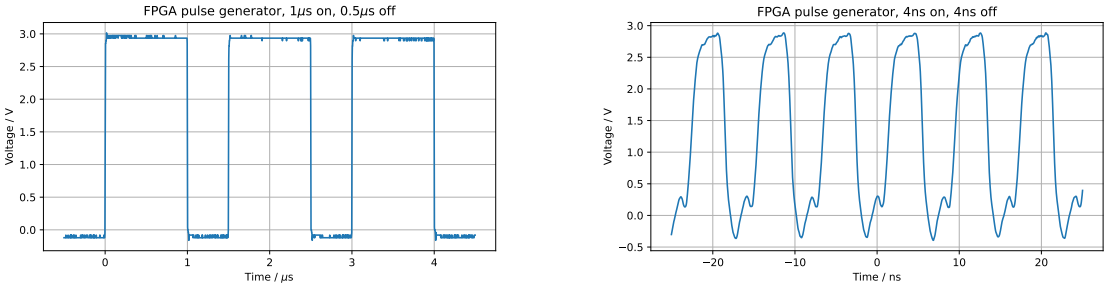
Figure 3.9: Visualisation of the sample pulse pattern in Listing 4 with Notepad++ script, here `time1`=500 ns.

code to assemble the sequence is executed. Then, the pulse sequence is sent to the FPGA using code provided by Helmut Fedder, which turns the list of tuples into a binary format before programming the FPGA. Total communication time highly depends on the complexity of the pulse pattern, it ranges from (130 ± 10) ms for very simple sequences to (600 ± 30) ms for a complex pattern used in the RQO experiment.

Fig. 3.10 shows two pulse patterns of the FPGA output measured with an oscilloscope to test the time resolution. While the pulse generator has no issues with time scales of 1 µs (a), the pulses with lengths of 4 ns (b) show clear deformations.

3.5 Cameras

Two PCO Pixelfly [30] cameras will be used in the HQO experiment for absorption imaging [31] to measure the optical density and number of atoms in the magneto-optical trap, as well as extract 3D information on the shape of the atom cloud. Two Python GUIs are used, one to acquire the images, the other to display and analyse them. The programs used in HQO were adapted from the RQO



(a) Longer pulse lengths on µs time scales are rectangular.

(b) Short, 4 ns long pulses show deviations from rectangle pulses.

Figure 3.10: FPGA pulse generator outputs at different time scales.

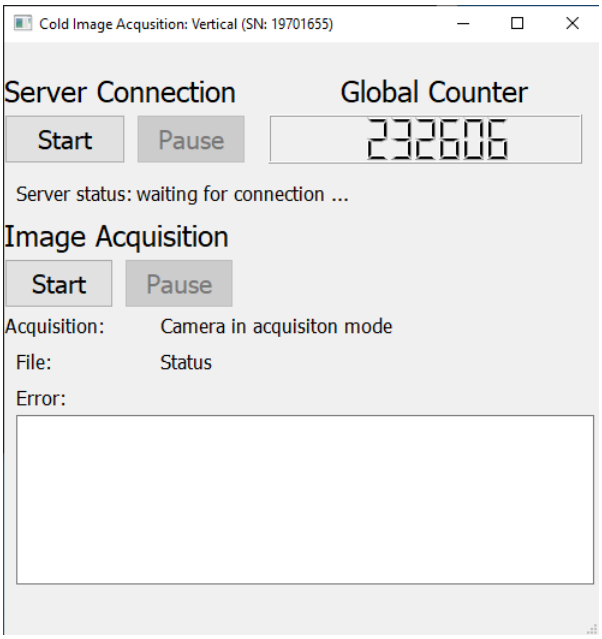


Figure 3.11: Image Acquisition GUI, with imaging direction, camera serial number and current global counter.

implementation, which was created by Rafael Rothganger de Paiva and Mogens From. The image acquisition GUI running on the computer connecting to the cameras is shown in Fig. 3.11. It is started using a batch file, where the imaging direction and serial number of the camera to be used are specified. The image analysis GUI however can run on any computer, as long as it is connected to the internal lab network.

3.5.1 Image Acquisition

The image acquisition thread uses a class called `cameraControl` with methods to open communication with the camera, set the exposure time and read single images. It opens a camera object `PcoPixelflyUSB(identifier)`, which implements all low-level functions from the manufacturer's program library needed to operate the camera. On initialisation a method to loop over all available cameras is called, it only opens the camera where the serial number matches the identifier. If no identifier is given, the first camera is opened. Previous versions only allowed the usage of a single camera per computer and did not allow discerning them by serial number. This functionality was was added with the help of Cedric Wind by adapting some functions from the `pco` Python package [30].

Afterwards the camera is prepared with the correct settings and trigger mode, so it is ready to record images. Once an external trigger is received, a picture is taken, immediately read out and buffered in a Python dictionary with the current global counter, timestamp, camera type and number of images taken. The dictionary is then added to a list `imgs` containing all images and metadata acquired this cycle. It takes 50 ms for this process to complete, after which the next image can be recorded.

Meanwhile, the server connection thread is waiting for the UDP broadcast from the experiment control GUI. As soon as the ping is received the exposure time `tExposure`, number of images `numImgs_cam1` and global counter for the next cycle are read from the database, the new settings are then applied to the camera. Finally, the saving thread is invoked, which compares the number of images in `imgs` to the specified number (usually three for absorption imaging) and saves them in a single MATLAB file in hqovault/HQO Results if the numbers match. Otherwise, an error is raised and no images are saved.

3.5.2 Image Analysis

The image analysis GUI (Fig. 3.12) implements the methods used in absorption imaging to obtain the atom number, cloud dimensions and optical density in the MOT. It features a broadcast mode, where the pictures shown are updated in real time as soon as they are saved by the acquisition GUI. Additionally, older image sets can be loaded by date or global counter. The second tab offers more in depth analysis tools of multiple picture sets.

3.6 USB Multichannel Analogue Output

For devices that require a constant but accurately controllable voltage supply during an experiment cycle, two USB digital-to-analogue converters (DAC) are used. Both are housed in rack mounted boxes, containing a USB-3100 series DAC by Measurement Computing [32], the models used are the 3112 and 3114. They only differ in the amount of output channels (8 and 16), both offer ±10 V or 0 to 10 V output ranges with 16 bit resolution. The DACs are housed in a 19 inch box, and are connected to BNC outputs. They are powered by a 5 V power supply. Communication is handled via a USB connection

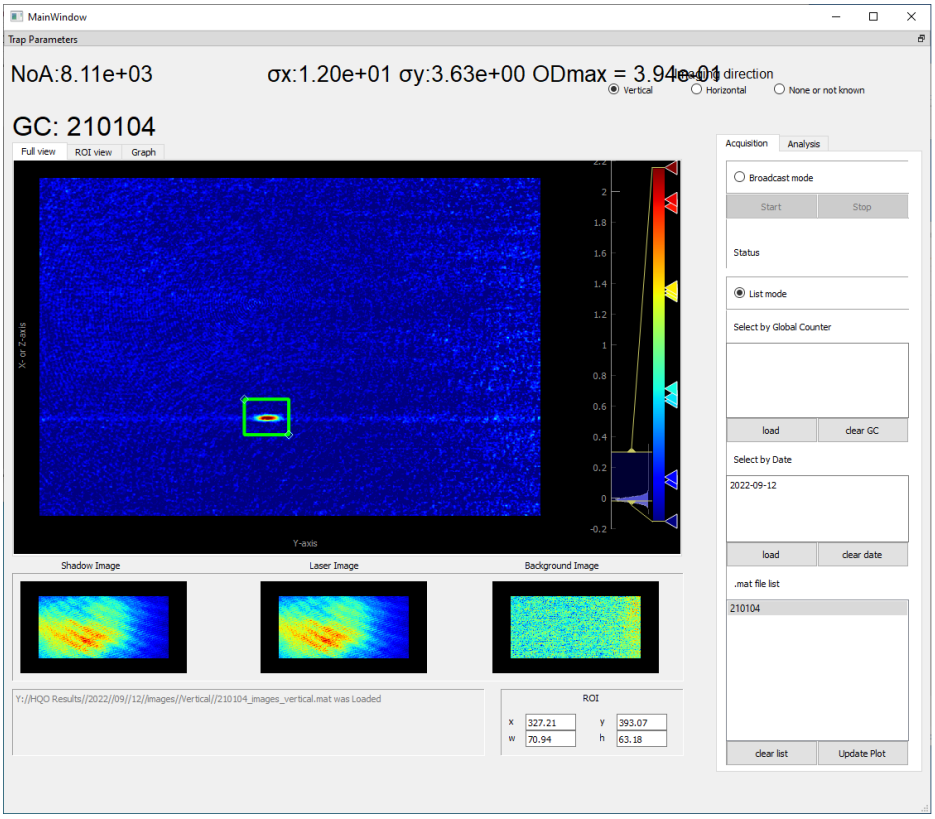


Figure 3.12: Image analysis GUI with previous data provided by RQO. The atom cloud is clearly visible, number of atoms and optical density are automatically calculated.

to HQODEVICECONTROL. Both HQO devices shown in Fig. 3.13 were assembled by Benjamin Scheeben in April 2022.

Operation of the DACs requires the `mcculw` Python package [33] in addition to the MCC DAQ Software from the manufacturer, which contains drivers, program libraries and a GUI to test the devices with. The script controlling both boxes is called `mcc_dac_server.py`, here the connected devices have to be specified by serial number; if they are not connected on startup an error will be raised. The DACs can also be set to unipolar (0 to 10 V) or bipolar (±10 V). As usual the script uses the instrument server to wait for the experiment control GUI ping and then reads the variables from the database. Currently the variable names are U00–U07 and V00–V15 for the 8 and 16 channel DAC respectively, the script will terminate with an error if they are not defined. USB communication is handled by the `mcculw` package, it includes functions to set the individual channels to the values given by the variables. Updating both DACs takes on average (30 ± 10) ms.

3.7 USB RF Signal Generator

Several RF signal generators are used to provide the main sideband for the laser locks using the Pound-Drever-Hall technique [34]. The solution in HQO was built by Julia Gamper and includes up to four Windfreak SynthUSBII [35] devices, which are connected to HQODEVICECONTROL via a USB

(a) 8 channel DAC

(b) 16 channel DAC

Figure 3.13: DAC boxes used in HQO.

hub. Their outputs are amplified and mixed with an input signal to obtain the modulation signal used to create the sidebands. More information on the specific technique used in HQO can be found in the theses of Julia Gamper [27] or Florian Pausewang [36].

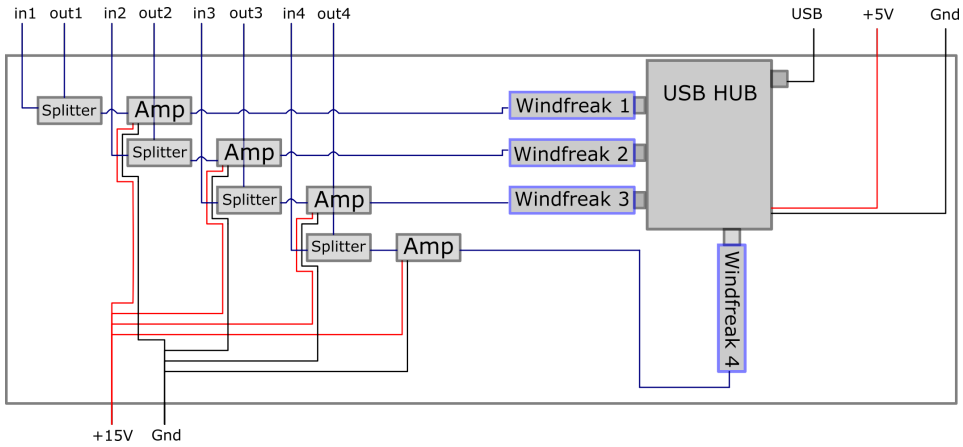


Figure 3.14: RF signal generator box schematic. Windfreak RF outputs are amplified and combined with an input signal [27].

The Python script `windfreak_server.py` on HOQDEVICECONTROL is used to control the RF outputs. Communication with the Windfreak SynthUSBII devices is handled using a wrapper class `windfreak.py`, which implements all necessary serial commands. First, the script checks how many RF generators are connected, by looping through all open COM ports, opening a `windfreakusb2()` device and trying to obtain its serial number using the Windfreak specific command. If successful, the device number, current frequency, RF output power and device Python object are saved to a dictionary for later use. An instrument server is invoked, waiting for the experiment control GUI's start signal. The variables controlling the output frequencies are defined as `wf{device number}_freq`, their values are given in MHz. To change the frequency the appropriate serial command is sent to the RF generator. A

complete list of commands can be found in the manual [37]. The communication time is (25 ± 10) µs per device reprogrammed.

3.8 Long-term Reliability and Stability Measurement

To test the assembled system's reliability and get a measure of its stability, a long run of continuous experiment cycles was executed. The sequence was previously used in the RQO experiment, thus it is a realistic workload for the connected devices. Additionally, seven ADwin analogue output channels were programmed with CSV files containing the voltage sequences for the magnetic transport coils. These files were provided by Cedric Wind, who designed the transport system and simulated the current flow through the coils. Logging functionality for the current global counter and time stamps at each network ping from the experiment control GUI was added to the instrument scripts for this test.

In total, 18138 cycles were completed during the run. The length of one cycle is 3.5 s, including the fixed preparation time, so it can be expected that the time difference between network pings is 3.5 s. Histograms of the time differences measured for each device can be found in Appendix A.2. Gaussian functions are fitted to the distributions, to obtain the centres μ and standard deviations σ. Averaging these values gives $\bar{\mu} = (3.50006 \pm 0.00002)$ s and $\bar{\sigma} = (6.23 \pm 0.02)$ ms. The distribution means are very close to the set sequence length, and no cycle took significantly more time. The standard deviation $\bar{\sigma}$ can be explained with the fact that the Python scripts exact timing is dependant on the scheduling jitter. As Windows 10 is not a real-time system other processes may delay the execution of the device scripts by a few milliseconds, which leads to the distributions measured.

Logging the global counters enables a simple check to confirm that not GC was missed or doubled over the duration of the test: The difference between subsequent GCs has to be equal to 1 for all GCs, and the number of logged counters has to match the total cycles completed. For all devices used in this measurement these condition are fulfilled, as can be seen in Appendix A.2. The same holds true for the images saved by the cameras.

All in all, the experiment control system performs as expected. All network pings reach their designated recipient and the device scripts work as intended. The timing between cycles is very close to the expected value. However, no measurement of the sequence internal jitter was performed due to time constraints, and the accuracy of the cycle length measurement could be improved by measuring it with a time tagger or similar device instead of Python logging functionality.

CHAPTER 4

Conclusion

Over the duration of this thesis the computer control system for the HQO experiment, based on the existing solutions for the RQO and YQO experiments was set up. The connected devices were also tested and characterised where appropriate.

The network infrastructure including the database necessary to communicate with all devices was installed and the correct functionality of the experiment control GUI including executing the time-critical Python script `cycle.py` and control of the ADwin output sequences was confirmed. All other devices were set up and controlled with the `cycle.py` script and trigger signals from the ADwin. The Python scripts needed to communicate with the instruments were adapted to the HQO setup and tested. Some larger changes include alterations to the camera GUI, which enables the use of two cameras simultaneously to extract 3D information of atom clouds in a trap. Improvements were also made to the device detection in the USB RF signal generator and DDS frequency generator scripts to remove hard coded COM ports, which makes these scripts more robust to hardware changes and reduces possible errors, that are difficult to find.

A long-term test run of the complete system with a reasonable experiment sequence that stresses all devices was completed to verify its reliability. Over the 18138 cycles no database entries and network pings were missed, no cycle took significantly longer than the expected 3.5 s and the cameras saved the correct amount of images. A measurement that was not taken due to time constraints is the internal jitter, which should be measured to ensure that the timings in a sequence are sufficiently accurate.

Some ideas for future development of the control system include bundling the device control Python scripts into one program and further standardising the communication methods. This could reduce the necessary network pings and database queries and improve flexibility in the type and number of connected devices. Usability could also improve, if a graphical interface for the device scripts is created.

The laser and vacuum systems were set up in parallel to this work. All three systems together enable the construction of magneto-optical trap as the next step in the HQO experiment. Afterwards, a room temperature experiment chamber in place of the proposed cryostat will be installed, where first experiments using electromechanical oscillators and atom chips can be carried out. The magnetic transport connecting MOT and experiment chamber will be driven by the ADwin analogue output, which was already tested with simulated voltage sequences.

APPENDIX A

Appendix

A.1 Computer Setup

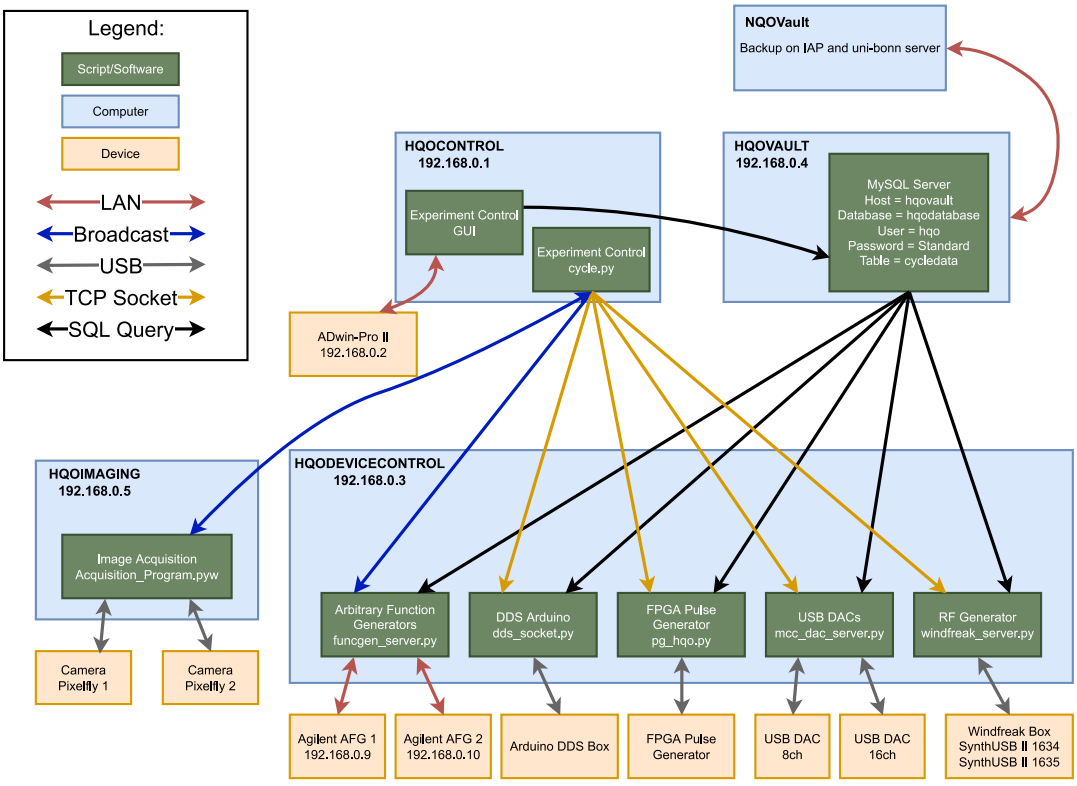
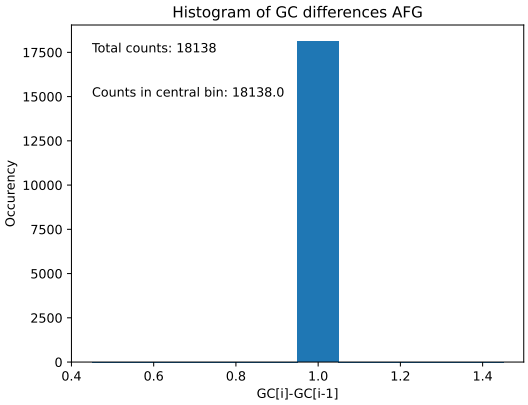
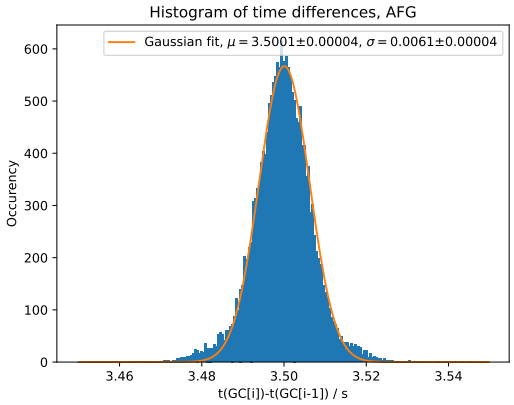


Figure A.1: Computer setup with connected devices and Python control scripts

A.2 Test Measurements

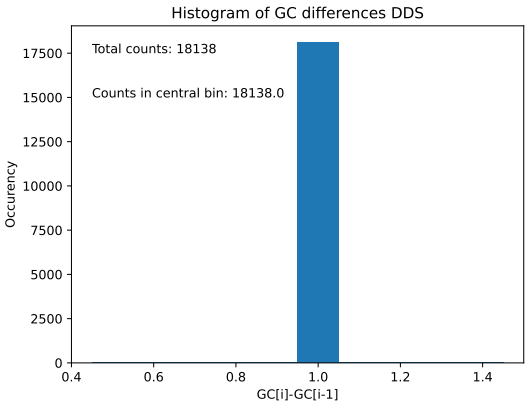


(a) Difference between concurrent global counters logged

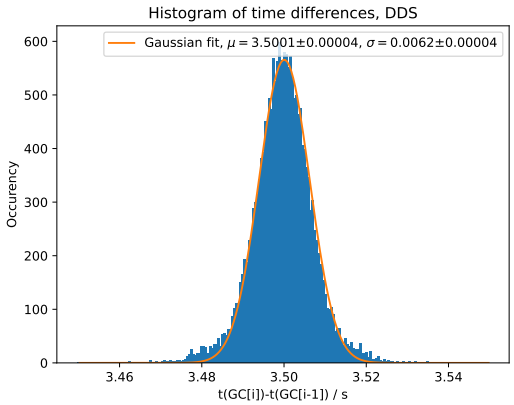


(b) Time between concurrent network pings received

Figure A.2: Measurements to verify correct function of AFGs over a day of experiment cycles.

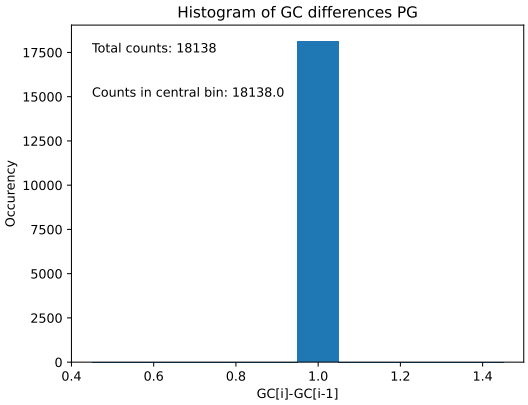


(a) Difference between concurrent global counters logged

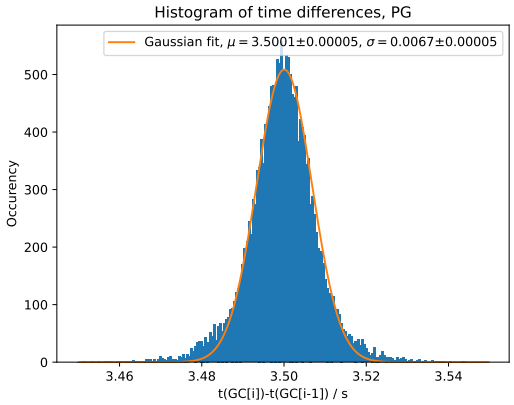


(b) Time between concurrent network pings received

Figure A.3: Measurements to verify correct function of DDS box over a day of experiment cycles.

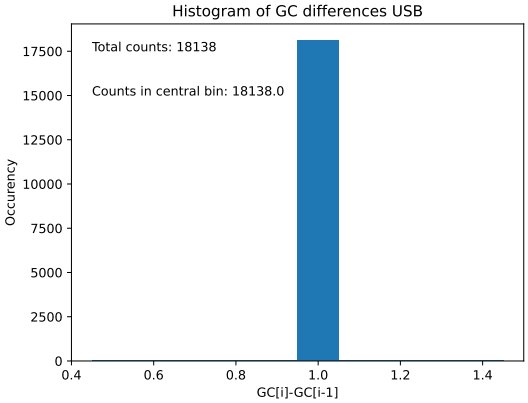


(a) Difference between concurrent global counters logged

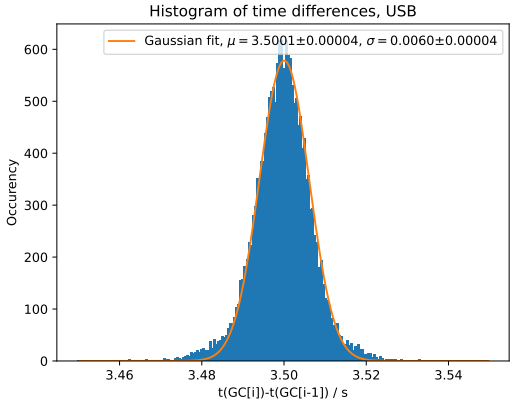


(b) Time between concurrent network pings received

Figure A.4: Measurements to verify correct function of the FPGA pulse generator over a day of experiment cycles.

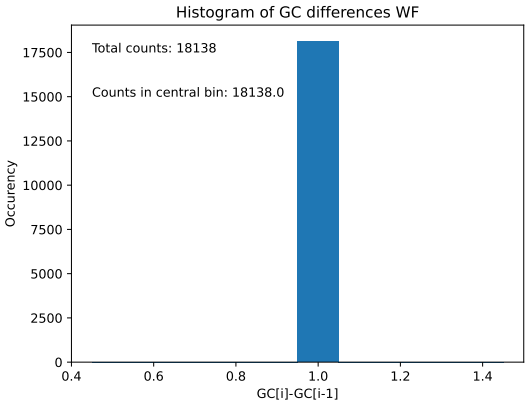


(a) Difference between concurrent global counters logged

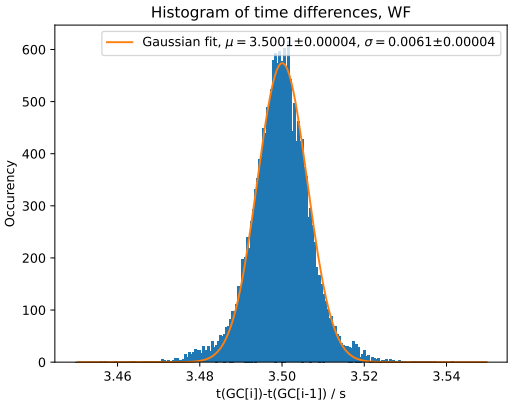


(b) Time between concurrent network pings received

Figure A.5: Measurements to verify correct function of DACs over a day of experiment cycles.



(a) Difference between concurrent global counters logged



(b) Time between concurrent network pings received

Figure A.6: Measurements to verify correct function of RF generators over a day of experiment cycles.

Bibliography

[1] T. F. Gallagher, *Rydberg Atoms*,
Cambridge Monographs on Atomic, Molecular and Chemical Physics,
Cambridge University Press, 1994.

[2] M. D. Lukin et al.,
Dipole Blockade and Quantum Information Processing in Mesoscopic Atomic Ensembles,
Phys. Rev. Lett. **87** (3 2001) 037901,
URL: https://link.aps.org/doi/10.1103/PhysRevLett.87.037901.

[3] O. Firstenberg, C. Adams and S. Hofferberth,
Nonlinear quantum optics mediated by Rydberg interactions,
Journal of Physics B: Atomic, Molecular and Optical Physics **49** (2016).

[4] H. Gorniaczyk, C. Tresp, J. Schmidt, H. Fedder and S. Hofferberth,
Single-Photon Transistor Mediated by Interstate Rydberg Interactions,
Phys. Rev. Lett. **113** (5 2014) 053601,
URL: https://link.aps.org/doi/10.1103/PhysRevLett.113.053601.

[5] N. Stiesdal et al., *Controlled multi-photon subtraction with cascaded Rydberg superatoms as single-photon absorbers*, Nature Communications **12** (2021).

[6] Nonlinear Quantum Optics Group, *Hybrid Quantum Optics*, 2022,
URL: https://www.nqo.uni-bonn.de/research/hybrid-quantum-optics.

[7] R. Stevenson, J. ř. Miná ř, S. Hofferberth and I. Lesanovsky,
Prospects of charged-oscillator quantum-state generation with Rydberg atoms,
Phys. Rev. A **94** (4 2016) 043813,
URL: https://link.aps.org/doi/10.1103/PhysRevA.94.043813.

[8] M. Gao, Y.-x. Liu and X.-B. Wang,
Coupling Rydberg atoms to superconducting qubits via nanomechanical resonator,
Phys. Rev. A **83** (2 2011) 022309,
URL: https://link.aps.org/doi/10.1103/PhysRevA.83.022309.

[9] E. L. Raab, M. Prentiss, A. Cable, S. Chu and D. E. Pritchard,
Trapping of Neutral Sodium Atoms with Radiation Pressure, Phys. Rev. Lett. **59** (23 1987) 2631,
URL: https://link.aps.org/doi/10.1103/PhysRevLett.59.2631.

[10] M. Greiner, I. Bloch, T. W. Hänsch and T. Esslinger,
Magnetic transport of trapped cold atoms over a large distance, Phys. Rev. A **63** (3 2001) 031401,
URL: https://link.aps.org/doi/10.1103/PhysRevA.63.031401.

[11] J. Fortágh and C. Zimmermann, *Magnetic microtraps for ultracold atoms*, Rev. Mod. Phys. **79** (1 2007) 235, URL: https://link.aps.org/doi/10.1103/RevModPhys.79.235.

[12] Nonlinear Quantum Optics Group, *Rubidium Quantum Optics*, 2022, URL: https://www.nqo.uni-bonn.de/research/rubidium-rydberg-nqo.

[13] Nonlinear Quantum Optics Group, *Ytterbium Quantum Optics*, 2022, URL: https://www.nqo.uni-bonn.de/research/ytterbium-rydberg-nqo.

[14] Jäger Computergesteuerte Messtechnik GmbH, *Adwin-Pro II - flexible and modular*, URL: https://www.adwin.de/us/produkte/proII.html.

[15] PI5 - University of Stuttgart, *Cold Physics Experiments Control Software (CPECS)*, 2020, URL: https://github.com/coldphysics/experiment-control.

[16] Python Core Team, *socket - Low-level networking interface*, Python Software Foundation, 2022, URL: https://docs.python.org/3/library/socket.html.

[17] J. Postel, *User Datagram Protocol*, RFC 768, 1980, URL: https://www.rfc-editor.org/info/rfc768.

[18] W. Eddy, *Transmission Control Protocol (TCP)*, RFC 9293, 2022, URL: https://www.rfc-editor.org/info/rfc9293.

[19] Oracle, *MySQL Connector/Python*, 2022, URL: https://dev.mysql.com/doc/connector-python/en/.

[20] FreeFileSync, *FreeFileSync*, 2022, URL: https://freefilesync.org/.

[21] Python Core Team, *time - Time access and conversions*, Python Software Foundation, 2022, URL: https://docs.python.org/3/library/time.html#time.time.

[22] Jäger Computergesteuerte Messtechnik GmbH, *ADwin Product list*, 2021, URL: https://www.adwin.de/us/produkte/files/productlist0419.pdf.

[23] Keysight Technologies, *33500B and 33600A Series Trueform Waveform Generators*, 2022, URL: https://www.keysight.com/au/en/assets/7018-05928/data-sheets/5992-2572.pdf.

[24] SCPI Consortium, *SCPI-1999 Specification*, 1999, URL: https://www.ivifoundation.org/docs/scpi-99.pdf.

[25] Analog Devices, *AD9959*, 2016, URL: https://www.analog.com/media/en/technical-documentation/data-sheets/ad9959.pdf.

[26] Michael Thomas Schlagmüller, *A single Rydberg Atom interacting with a Dense and Ultracold Gas*, PhD thesis: Universität Stuttgart, 2016.

[27] Julia Gamper, *Frequenzstabilisierung zur Laserkühlung von Rubidium*, Universität Bonn, 2022.

[28] Felix Engel, *Entwicklung eines FPGA basierten Pulsgenerators mit Nanosekunden-Auflösung für schnelle Rydberg-Experimente*, Universität Stuttgart, 2013.

[29] Opal Kelly, *XEM3005*, 2022, URL: https://opalkelly.com/products/xem3005/.

[30] Excelitas PCO GmbH, *pco.pixelfly USB*, 2022,
URL: https://www.pco.de/scientific-cameras/pcopixelfly-usb/.

[31] G. Reinaudi, T. Lahaye, Z. Wang and D. Guéry-Odelin,
Strong saturation absorption imaging of dense clouds of ultracold atoms,
Opt. Lett. **32** (2007) 3143,
URL: http://opg.optica.org/ol/abstract.cfm?URI=ol-32-21-3143.

[32] Measurement Computing, *USB-3100 Series*, 2018,
URL: https://www.mccdaq.com/PDFs/specs/USB-3100-series-data.pdf.

[33] Measurement Computing, *mcculw Python package*, 2020,
URL: https://github.com/mccdaq/mcculw.

[34] R. W. Drever et al., *Laser phase and frequency stabilization using an optical resonator*,
Applied Physics B Photophysics and Laser Chemistry **31** (1983) 97.

[35] Windfreak Technologies, *SynthUSBII: 34MHz – 4.4GHz USB RF Signal Generator*, 2022,
URL: https://windfreaktech.com/product/usb-rf-signal-generator/.

[36] Florian Pausewang,
Aufbau, Optimierung und Charakterisierung von frequenzstabilisierten Lasersystemen,
Universität Bonn, 2022.

[37] Windfreak Technologies, *Windfreak SynthUSBii Serial Communications*, 2013, URL: https://windfreaktech.com/wp-content/uploads/docs/synthusbii/synthusbiicom.pdf.