MONASH University

A software framework for control and automation of precisely timed experiments

Philip Thomas Starkey
BScAdv(Hons)

Supervisors:
Prof. Kristian Helmerson
Dr. Lincoln Turner
Dr. Russell Anderson

School of Physics & Astronomy
Monash University

Submitted in total fulfilment of the requirements of the degree of
Doctor of Philosophy
at Monash University, Australia

July 2019

Copyright Notices

Abstract

Modern experiments are increasingly reliant on computer control for coordination of an apparatus. Computer control is employed in many different contexts, from institution scale control systems for particle accelerators to lab-scale studies of quantum mechanics. These experiments are often characterised by the large number of devices that must be controlled and the time-scales of the process dynamics to be studied. In all cases, the need for a control system is motivated by the inability of humans to coordinate the entire experiment with the required timing precision.

While the last two decades has seen increasing development of control systems for precisely timed experiments, most are specialised and do not sufficiently address future requirements. Often this is the result of a disproportionate focus on the user interface of the software, at the expense of flexibility. However, the adaptability of the control system to future demands is just as important as the user interface. This requires a careful consideration of the underlying technologies used, which is often ignored in favour of the technologies familiar to the authors. Many existing control systems only control a portion of the experiment lifecycle, most often ignoring the analysis of acquired data and the feedback of results into future experiments. However, these features are playing an increasing role as more journals request the publication of raw data and analysis code and researchers experiment with automatic optimisation algorithms.

In this thesis I present *the labscript suite*, our next generation control system for precisely timed experiments. We focus specifically on controlling shot-based experiments; experiments with a distinct start and end time, which are typically repeated many times but often not identically. All data pertaining to a shot is stored in a single hierarchical data format version 5 (HDF5) file, allowing shots to be shared easily as part of a publication. While our control system is designed for ultracold atom research, which relies on such control systems to perform experiments, we show it can also be used for other shot-based experiments.

While many control systems present purely graphical interfaces for defining experiment logic, we balance it across both graphical and textual components. Experiment logic is best described by a textual programming language, so that a user can take advantage of existing control statements. We provide a consistent interface for controlling heterogenous hardware by providing Python methods based on the type of output (analog voltage, digital, radio-frequency (rf)) rather than the model of device. The interface is accessed via user-provided names for each channel, rather than a hardware identifier, making the experiment logic robust against hardware configuration changes. As experiment logic is defined using standard Python syntax, it can be spread across multiple files and Python modules. Parameter defi-

nitions are separated from the experiment logic and managed through a graphical interface. They are passed in to the experiment logic script as global variables when a shot is created. The exploration of parameter spaces is also handled by the graphical interface, invoking the experiment logic once for each point in a user defined parameter space.

Experiment shots can be executed autonomously by our control system. Created shots are placed in a queue, and executed in sequence on the experiment hardware without further user intervention. Acquired data is saved in the HDF5 file after a shot completes, and is forwarded on to our analysis framework. Our analysis framework runs a set of user-specified Python scripts, which are either run for each shot individually, or for a group of shots. These scripts can perform any task the user cares to write, including the generation of new shots (for example as part of an automatic optimisation routine).

The underlying technologies of our control system were chosen to maximise flexibility in order to control the widest range of experiments. We split the labscript suite into distinct software programs with well-defined communication protocols. This follows the Unix philosophy from the software engineering community, providing a mechanism to easily replace programs as the need arises. We use a high-level, object-oriented programming language (Python) to produce features that can be extended through subclasses. Python, a common programming language among data scientists, is also chosen as the language for defining experiment logic and analysis scripts. This allows us to utilise existing software engineering practices, such as version control, to better manage the experiment lifecycle.

Each program in the labscript suite is built for modularity. Our software is designed for a multi-user environment and provides quick ways to switch between experiment configurations. We do not prescribe the use of specific hardware devices; each lab is free to choose the set of devices most appropriate for their laboratory. Support for new hardware devices can be added through the addition of a new Python module, which can be built on top of our existing translations between high-level user commands and low-level device commands, significantly reducing the complexity of this task. Graphical interfaces for manual control of each hardware device are dynamically generated at runtime.

Our control system has proven very effective, and has been used to perform experiments across several laboratories running diverse experiments. We believe it marks a new standard for control of precisely timed experiments.

Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Philip Thomas Starkey
3rd July, 2019

Acknowledgements

It has been a long road to the completion of my PhD project, and many people have supported me along my journey. I'd like to take the opportunity to thank them all for their help, without which this thesis would not exist.

Firstly to my primary supervisor, Kris Helmerson. Thank you for your support and for your belief in me. It can be daunting starting a project in a new lab with no existing apparatus, but you gave me the freedom to take charge which allowed me to rapidly grow as an experimental physicist. I would not have the confidence and broad skill set I have today without this. Finally, I'd like to especially thank you for encouraging me to pursue this research project, even though it wasn't what I originally signed on for. When this research direction opened up, you saw the potential of this project and gave me confidence to pursue it, for which I will always be grateful.

I would also like to thank my co-supervisors Lincoln Turner and Russell Anderson. Firstly, for their advice while building our BEC apparatus, and encouraging us to share technology between the two BEC labs. More importantly, thank you for letting me push Python on the group and not minding that this meant throwing out some LabVIEW code we were working on at the time. Without your support, I doubt the labscript suite would be the successful open source project it is today.

I joined the Monash BEC group as part of a large number of new research students: Shaun Johnstone and Chris Billington who joined Kris's lab with me, and Martijn Jasperse, Alex Wood, Chris Watkins, and Lisa Starkey (née Bennie) who started working with Lincoln and Russ. This was a unique experience and it helped knowing we were all facing the same challenges together. I'd like to particularly thank Shaun and Chris. Shaun, thank you for working with me side-by-side in the lab over the (many) years as we built the apparatus up, and for never giving up hope that we would eventually make it work. Chris, thank you for pursuing the control system with me, for both the development you did and for the many hours of discussions we had as we honed the technology and philosophy behind this research. To Martijn, Alex and Lisa in the spinor lab, thank you for sharing your designs, experiences and (most importantly) optics components!

Thanks also to the wider Monash BEC research group, including our group's theorist Tapio Simula and his students Andrew Groszek and Thomas Mawson, who broadened my horizons and exposed me to their wonderful world of simulation. Mikhail Egorov, our post-doc, thank you for your help achieving our first BEC and handling the move to New Horizons. Your knowledge and experience was invaluable to Shaun and I, and your day-to-day help and supervision in the lab was much appreciated. Seb Tempone-Wiltshire, it was a pleasure working with you through your honours year and start of PhD. I enjoyed

transferring all my accumulated knowledge on how *not* to build a BEC machine! It was great to be a part of your new machine design, and to see it produce ultracold atoms successfully.

This would not be complete without giving my heartfelt thanks to the Science Faculty and the School of Physics and Astronomy. Firstly, I'd like to thank the faculty for funding my initial PhD scholarship, and the school and my supervisors Kris for continuing my scholarship when I ran overtime. Our head of school, Michael Morgan, also deserves many thanks; it is fair to say our research group would not exist without his support, and his dedication to the school has seen it grow tremendously during my time as a student. Jean Pettigrew, who handles the administrative load produced by our central postgraduate bureaucracy, deserves much more thanks than I could possibly give in this small space. Without her help and support, many students in the school (including me) would not have made it. Rob Seefeld, thank you for all your help with the move to the New Horizons building, and for not minding that we created a bug tracker for all of the initial problems with the building. Thank you for fighting to ensure the building infrastructure met our needs. David Paganin and Alexis Bishop, thank you for acting as post-graduate coordinators for our school and for your personal support, oversight and continual encouragement during my project. Finally, to all those in the school who I haven't mentioned by name, thank you for your friendship and making it such a wonderful community to be a part of.

I'd also like to thank the people who helped me find a passion outside of my research project, and who helped keep me sane by providing me with something other than my project to work on. Firstly, I'd like to thank Lincoln, Russ, Alexis, Manuel (Manny) Pumarol, and Theo Hughes for helping me to develop a passion for teaching. I'd also like to thank you all for providing me with paid opportunities to develop and introduce new laboratory activities, digital logbooks for students, and remote online laboratories. I enjoyed working on all of these projects and I hope I'll have opportunities to work on similar projects in the future. I was also lucky enough to be part of the team that organised the 9th annual Australia/New Zealand IONS KOALA[1] student conference in 2016. It was an amazing opportunity to serve as treasurer for the conference, and I'd like to thank all those involved including all committee members, sponsors and attendees for helping to make it a fantastic conference. Special mentions of course to Shaun, who helped drive the organisation of the conference with me, Rob Seefeld for his administrative support, Kris as our OSA student chapter advisor, and the other Monash committee members Seb, Tom, Andrew, and Sam Fischer.

To my parents, thank you for inspiring me to study science and believing I could finish. Similarly, to my in-laws, thank you for all of your support over the years.

Finally, thank you to my wife, Lisa, who not only completed her own PhD in our research group, but supported me in mine. Thank you for everything, forever and always.

1. Konference on Optics, Atoms and Laser Applications (KOALA), an International OSA Network of Students (IONS) event.

Contents

1 Introduction **1**
- 1.1 Types of control systems . . . 1
  - 1.1.1 Process control systems . . . 2
  - 1.1.2 Scientific control systems . . . 3
- 1.2 Controlling an ultracold atom experiment . . . 4
- 1.3 The labscript suite . . . 4
- 1.4 Thesis outline . . . 6

2 Ultracold atoms & control systems **9**
- 2.1 Ultracold atoms . . . 9
  - 2.1.1 Magneto-optical traps . . . 9
  - 2.1.2 Polarisation gradient cooling & optical pumping . . . 10
  - 2.1.3 Magnetic traps & evaporative cooling . . . 11
  - 2.1.4 Dipole traps . . . 11
  - 2.1.5 Novel physics . . . 12
- 2.2 Controlling an ultracold atom experiment . . . 13
  - 2.2.1 Timescales . . . 13
  - 2.2.2 Hardware requirements . . . 14
  - 2.2.3 Pseudoclocks . . . 15
- 2.3 A review of existing control systems . . . 17
  - 2.3.1 LabVIEW line-based systems . . . 18
  - 2.3.2 Cicero word generator . . . 20
  - 2.3.3 Strontium BEC control & vision . . . 25
  - 2.3.4 Others . . . 27
- 2.4 Summary . . . 29

3 Apparatus **31**
- 3.1 The laser table . . . 32
  - 3.1.1 Rubidium . . . 33
    - 3.1.1.1 Laser lock . . . 33
    - 3.1.1.2 Tapered amplifiers . . . 35
    - 3.1.1.3 Beamlines . . . 37
  - 3.1.2 Potassium . . . 38
    - 3.1.2.1 Laser lock . . . 38
    - 3.1.2.2 Tapered amplifier . . . 39

3.1.2.3 Beamlines . . . 39
3.2 The vacuum table . . . 40
3.2.1 Vacuum system . . . 40
3.2.2 MOT optics . . . 43
3.2.3 Optical pumping optics . . . 45
3.2.4 Imaging optics . . . 46
3.2.5 Dipole trap . . . 47
3.3 Lab process control . . . 47
3.3.1 Oven controller . . . 47
3.3.2 Coil interlock . . . 48
3.4 Electronics . . . 49
3.4.1 Radiofrequency amplification . . . 49
3.4.2 Shutter drivers . . . 50
3.4.3 Magnetic coil driver . . . 51
3.4.4 Microwave source . . . 52
3.5 Summary . . . 53

4 The labscript suite **55**
4.1 Terminology . . . 55
4.2 Components of the labscript suite . . . 56
4.2.1 Labscript API . . . 59
4.2.2 Runmanager . . . 60
4.2.3 Runviewer . . . 60
4.2.4 BLACS . . . 61
4.2.5 BIAS . . . 64
4.2.6 Lyse . . . 64
4.3 Design Themes . . . 66
4.3.1 Unix Philosophy . . . 66
4.3.2 Flexibility . . . 68
4.3.3 Graphical vs. textual interfaces . . . 69
4.3.4 Record keeping . . . 71
4.4 Technologies . . . 72
4.4.1 Programming language: Python . . . 72
4.4.2 HDF5 . . . 73
4.4.3 ZeroMQ . . . 75
4.4.4 Qt . . . 75
4.4.5 Multithreading and multiprocessing . . . 76
4.5 Summary . . . 77

5 Preparing experiments with the labscript suite **79**
5.1 Labscript API . . . 79
5.1.1 Connection table . . . 81
5.1.2 Experiment logic . . . 87
5.1.3 Global variables . . . 90
5.1.4 Pseudoclocks . . . 91

5.1.5 Gated clocks . . . 94
5.1.6 Controlling data acquisition . . . 94
5.1.6.1 Analog time series . . . 95
5.1.6.2 Images . . . 96
5.1.7 Waits . . . 96
5.1.8 Shot storage . . . 99
5.1.9 Unit conversions . . . 101
5.2 Runmanager . . . 103
5.2.1 The graphical interface . . . 104
5.2.2 Managing global variables . . . 107
5.2.3 Parameter space scans . . . 111
5.2.4 Evaluation of globals . . . 113
5.2.5 Shot creation . . . 114
5.3 Runviewer . . . 115
5.3.1 Generating output traces . . . 115
5.3.2 The graphical interface . . . 116
5.4 Summary . . . 119

6 Executing experiments with the labscript suite 121
6.1 BLACS . . . 121
6.1.1 Device tabs . . . 121
6.1.1.1 The graphical interface . . . 122
6.1.1.2 Worker processes . . . 123
6.1.1.3 State machine . . . 123
6.1.1.4 Handling waits . . . 131
6.1.2 Shot management . . . 131
6.1.3 Plugins . . . 134
6.1.3.1 The connection table plugin . . . 134
6.1.3.2 The labwatch plugin . . . 135
6.2 Secondary Control systems . . . 135
6.2.1 BIAS . . . 136
6.2.2 Others . . . 136
6.3 Lyse . . . 137
6.3.1 Pandas . . . 138
6.3.2 Single-shot analysis . . . 139
6.3.3 Multi-shot analysis . . . 140
6.3.4 Command-line usage . . . 140
6.4 Mise . . . 141
6.5 Summary . . . 141

7 Extending the labscript suite 143
7.1 Adding support for new hardware devices . . . 143
7.1.1 Labscript . . . 144
7.1.1.1 Implementing pseudoclock devices . . . 144
7.1.1.2 Implementing pseudoclocked devices . . . 146

7.1.1.3 Storing configuration attributes . . . 147
7.1.2 BLACS graphical user interface . . . 148
7.1.3 BLACS worker processes . . . 150
7.1.4 Runviewer . . . 152
7.2 Summary . . . 153

8 Case studies **155**
8.1 BEC apparatus control & optimisation . . . 155
8.1.1 Experiment control hardware . . . 155
8.1.2 K-Rb apparatus experiment logic & globals . . . 157
8.1.3 Apparatus optimisation . . . 158
8.2 Vortex dynamics . . . 163
8.2.1 Experiment design . . . 163
8.2.2 Single-shot analysis . . . 166
8.2.3 Multi-shot analysis . . . 168
8.2.4 Dataset publication . . . 169
8.3 Objective lens development . . . 169
8.4 Summary . . . 173

9 Conclusion **175**
9.1 Summary . . . 175
9.2 Future work . . . 177

Appendix A Glossary **179**

Appendix B Coil interlock design **185**

Appendix C Supernova design **193**

Appendix D K-Rb experiment details **199**
D.1 Globals . . . 199
D.2 Experiment logic . . . 214
D.3 Device hierarchy . . . 231

Appendix E y_vs_auto.py lyse analysis script **247**

References **251**

Chapter 1

Introduction

Reproducibility of a result is at the heart of the scientific method. For many experiments, particularly in fields such as particle physics and quantum physics (including cold quantum gases, which are the focus of this thesis), scientists now rely on control systems to provide precise timing of elements of the experimental apparatus. This is due to both the increasing complexity of apparatuses and the fast time-scales of the processes studied. As such, scientific research is increasingly relying on a heterogeneous mix of data acquisition hardware coupled with precisely timed analog and digital interfaces to the real world in order to make new discoveries. Accurate and detailed record keeping is also a key, although often tiring, part of performing a scientific experiment. In order to ensure the veracity of published research, many scientific journals and institutions are moving towards a model where supplementary materials, such as raw data and extensive details on the analysis techniques employed, are made available along with the published article.

Control systems provide a means to rapidly and consistently implement strict adherence to scientific principles while simplifying the precision control of real-world interfaces. In this thesis, we outline the principles we have developed for control systems for the scientific experiments in our laboratories, and then present the implementation of these principles in the *labscript suite*; an open source framework we have developed during this project for performing and analysing precisely timed scientific experiments. While this thesis will primarily focus on the development of the framework with respect to controlling an ultracold atom experiment, our control system has broad applicability to other experiments that must be precisely timed.

1.1 Types of control systems

It is important to define what we mean by a 'control system'. We consider control systems to be the software written to coordinate physical equipment via the analog and digital outputs of a hardware device. These outputs may be as simple as a single digital line or as complex as a radio-frequency (rf) generating direct digital synthesiser (DDS) device (with analog quantities for the frequency, amplitude, and phase, of the rf signal) or a spatial light modulator (SLM) containing an array of addressable pixels. The control system may also monitor analog and digital inputs (of similar or disparate types to the outputs) and record or even respond to the values obtained. This definition of a control system, and its

capabilities, is naturally very broad. However, for the purposes of this thesis, we can broadly categories control systems into two distinct groups: those that are designed to manage a fixed process and those that are designed to allow an operator to innovate with the control of a general purpose apparatus. Typically the former covers process control of industrial equipment while the latter covers control of scientific instrumentation. Many control system design philosophies and technologies were developed for process control, so we will initially introduce them in that context. However, we'll show (throughout this thesis) that many of them are equally useful for control systems of scientific experiments.

1.1.1 Process control systems

The most common use of a control system is in industries where machinery has been automated. Such industries include manufacturing, power generation and distribution, and resource extraction and refineries. Typically such control systems are referred to as 'process control systems' and are designed to ensure the safe and continued operation of the machinery they manage.

Basic process control systems utilise a programmable logic controller (PLC) which provides inputs and outputs for interacting with the machinery. Many PLCs are programmed using specialised languages, which typically represent the electrical structure of control system and equipment [1]. Often such languages are classified as either dataflow languages or ladder logic languages. These types of languages are often thought of as a natural choice for writing a control system. The hardware devices, after all, are simply handling the flow of data from the equipment, and so the translation between a electrical schematic and the programming of a specific device becomes simpler. Such languages also allow the programmer to better visualise the internal behaviour of each device. This ensures that the logic of the PLC is robust, which is the most important consideration when designing a process control system. These types of languages also bring several other benefits, not least a low barrier for entry (the language is often pictorial) and may come with automatically applied multi-threading capabilities.

An important descriptor of a process control system is whether it is open or closed loop. Open loop control systems by definition implement a static sequence of events with no feedback on changes in response to this. Closed loop control systems, in contrast, use acquired data to modify the control sequence in order to better reach the aims of the control system. Most modern process control systems are now closed loop, due to the availability of cheap data acquisition hardware built into PLCs.

Common features implemented in an individual PLC include finite state machines (FSMs) and proportional–integral–derivative (PID) controllers. FSMs ensure that the state of the machinery controlled is always well defined, and provides a formulaic approach to defining the set of rules for transitioning between these known states and the actions that should occur. This leads to increased robustness of the PLC logic. PID controllers are designed to automatically adjust available PLC outputs to ensure a measured quantity stays close to a provided set-point, and can be optimised for the response profile (transfer function) of a given feedback loop, ensuring stability of the machinery in spite of changes to external parameters (for example external temperature) which are not controlled by the PLC. Human interaction with a single PLC is typically done via a physical interface con-

nected to the PLC such as buttons connected to digital inputs or variable resistors connected to analog inputs, however some PLCs also support management via a connection to a PC.

More complex process control systems then often consist of multiple PLCs that are linked together via a common communications standard such as RS-232/485, Modbus, or Ethernet protocols. Such systems are known as a distributed control system (DCS). In some fields, where the system may be distributed across multiple physical sites, distributed control systems may also be referred to as a supervisory control and data acquisition (SCADA) system. Typically a DCS/SCADA system will utilise a heterogeneous mix of PLC hardware so as to correctly cater specifically to the set of requirements for the given part of the machinery that the PLC controls.

For distributed systems, a human-machine interface (HMI) is used to globally manage the set of PLCs. A HMI for a process control application is typically graphical in order to provide a clear visual status of the machinery being monitored, along with controls for modifying parameters of the control sequence, such as set-points. There are usually also controls for changing the state of a part of the machinery, such as starting or stopping a particular process.

1.1.2 Scientific control systems

While scientific control systems share many concepts with process control systems, we consider them a distinct category. This is largely due to the fact that scientific applications are often focussed on developing a new innovative process whereas process control systems are largely focussed on reproducing an identical outcome for an industrial process with a very tight tolerance. The major difference between process control and scientific control systems is that the latter are often more configurable, expect more user input, and are not expected to run continuously. Scientific experiments are also typically required to cover a large parameter space, while process control is typically designed to run repeatedly at a single optimal point. It is thus particularly important to note that scientific control systems are not meant to replace process control systems. For scientific applications, scientific control systems should form part of a broader control system that incorporates both standard process control for safety and maintenance of sections of the apparatus that you don't want to innovate with. Some aspects of scientific control systems will of course match those of dedicated process control systems, such as support for heterogeneous hardware that is distributed across multiple computing devices. Similarly, some aspects of scientific control systems (such as the one we present in this thesis) may be useful in engineering or manufacturing applications where the operator is expected to innovate with the process they are overseeing. Many existing scientific control systems are also written in a dataflow language such as LabVIEW, as this (again) provides a natural representation of the physical hardware. However, an increasing number are being written in high-level, general purpose programming languages, the benefits of which we'll outline later in this thesis.

1.2 Controlling an ultracold atom experiment

Ultracold atom experiments are one of the most technically challenging lab-based[1] physics experiments. The production of ultracold atoms relies on the application of several fields of physics, such as optics and atomic physics, scientific control systems, and complex electronics such as lasers, wideband servo control, cameras and photodetectors, high current switching, and input and output (I/O) devices. Ultracold atoms themselves are finding a niche in many evolving fields of physics such as quantum information [2, 3] and precision metrology [4]. They are also useful for quantum simulation of other complex physical systems, such as condensed matter systems [5], many-body quantum systems [6], and even astronomical phenomena [7], as they are easily controllable through a variety of optical and magnetic-field based techniques. Ultracold atom experiments are particularly dependent on control systems and much of the research in these areas could not have been performed prior to the advent of microprocessors and other modern computational technology. While not discussed here in detail, there are similar constraints on other quantum science experiments. For example, ion trapping experiments rely heavily on custom field-programmable gate array (FPGA) based hardware that updates faster than is needed for ultracold atom experiments.

Ultracold atom clouds are typically single use objects in that, despite being produced under ultra-high vacuum (UHV), interactions with the remaining background vapour limit their lifetime to minutes at best, and most measurement techniques are destructive. Most ultracold atom experiments are thus shot-based, where each shot has a distinct start and end point, with the production of one atom cloud per shot. The timing of I/O state changes between the start and end point of the shot is also precisely defined, where the level of precision is dictated by the science of the experiment. The science thus dictates the hardware devices required to produce an ultracold atom cloud, which in turn inform our decisions on the development of an appropriate control system. It is thus important that we have an in-depth understanding of the underlying science (of both the production of ultracold atoms clouds and the physical system to be studied using the ultracold atom cloud) in order to develop an appropriate control system.

While the hardware device requirements vary from lab to lab, most experiments require computer control of multiple devices each with multiple analog and digital outputs with a timing resolution on the order of microseconds for an experiment that spans multiple 10s of seconds. A single scientific publication may also require thousands of shots to be run, spanning a complex multi-dimensional parameter space. The coordination and execution of such experiments is clearly beyond the direct ability of humans, and it is for this reason that we require a scientific control system.

1.3 The labscript suite

In this thesis we will present the labscript suite [8, 9], a software framework for the control and automation of precisely timed scientific experiments. The labscript suite consists of

1. By 'lab-based' we mean an apparatus directly operated by a small team of researchers in one of many laboratories at a research institution. Such labs typically do not have technical staff employed to maintain and/or run the apparatus. This is distinct from large scale shared facilities such as particle accelerators and synchrotrons which also rely heavily on control systems developed and maintained by dedicated technical staff.

several distinct programs, which operate together to provide a control system that covers everything from experiment preparation to the analysis of results. The labscript suite was designed with support for closed-loop operation, so that analysis results can be automatically fed back into the preparation of subsequent experiments.

Our development particularly focused on ensuring that our control system was applicable to the widest variety of experiments. We designed the labscript suite to be modular, and thus extensible, at every level of development. In particular, we ensured that limited prescription was placed on the hardware required to use the labscript suite, and where possible deferred the creation of experiment logic to Python scripts created by users for their specific apparatus. Emphasis was also placed on usability by choosing a high level programming language such as Python for these scripts, providing an easy entry point into programming (and is one of the common scientific languages in use today [10, 11]) By providing high-level (user friendly) application programming interfaces (APIs) for use in these Python scripts, we are able to minimise the entry barrier further for new users by automating complex tasks such as the creation of graphical user interfaces (GUIs) and the generation of complex device hardware instructions (including pseudoclocks) required for a precisely timed experiment.

While many control systems are either all textual or all graphical, we provide a balance between the two, which we believe evolves naturally from the tasks they are used for. The use of Python scripts for experiment preparation and analysis allow users to take advantage of common programming features such as control flow statements (for example 'if' statements, loops, and functions), code comments, and version control tools. Graphical interfaces on the other hand, provide a natural way to manage lists of parameters, the management of experiments to be run on an apparatus, the control of hardware (when precise timing is not required), and the display of analysis results.

Our system has a strong focus on hardware abstraction, allowing us to provide standard interfaces to heterogenous hardware. Hardware devices and their I/O channels are represented using a hierarchy of Python objects. I/O channels are represented by standardised objects of varying types (for example: analog outputs, digital outputs, etc.), provided by our control system for use with any device. The control interface for each I/O channel is provided by these standard objects, not the device object, ensuring a consistent interface for each I/O type, no matter the model of parent device. This allows I/O to be moved from one device to another without needing to change the experiment logic code beyond the device assignment. Our object-oriented approach also provides a simple way to extend the functionality provided by our control system at any level of the object hierarchy, should it be required by a specific device.

The labscript suite was also designed to simplify many of the standard scientific practices used in research. Parameters can be defined as complex equations, which can optionally reference any other defined parameter. We also provide mechanisms to define parameters in real-world units, and automate the translation of these to the units required by a hardware device using user defined calibrations. We provide simple interfaces for automating the traversal of complex parameter spaces, including the iteration of parameter lists in lock step as one of many axes of a parameter space. Such traversal of high-order parameter spaces is becoming increasingly common as experiments become more complex, and we believe is a fundamental requirement of any modern control system. We also ensured that detailed

metadata on each experiment run was recorded. For example, we automatically store a named layout of the I/O channels of the control hardware (and the connections between them), a list of all parameters, information regarding the preparation of the experiment (including both high level Python code and low level tables of raw hardware instructions), and any acquired data along with the results of analysis performed on this data. This ensures an accurate record of every experiment is recorded, reducing the burden on researchers to adequately document the mundane aspects of each experiment, which is particularly critical for laboratories that might run hundreds or even thousands of experiments in a single day.

We also have adopted several developments from process control systems and software engineering, in order to ensure our control system is a robust as possible. These include the use of state machines, multiprocessing in order to sandbox 3rd party libraries and user code from the main control system components, and the distributed nature of our programs and hardware. Our use of a textual interface for some user facing parts of our control system, using a common, high-level programming language, brings to these components the same benefits of encapsulation, debugging, and version control that would apply to any software project written using that language.

These features combine to make arguably one of the most comprehensive scientific control systems. This has resulted in the labscript suite being adopted by research groups at other institutions including Swinburne University of Technology in Australia, Technische Universität Darmstadt in Germany, the National Institute of Standards and Technology (NIST) in Gaithersburg, MD, USA, and the National Physics Laboratory (NPL) in the UK where an extensive hardware development effort targeting the labscript suite has recently been completed [12]. Such control systems are becoming a key requirement of ultracold atom experiments, particularly as they move into the regime of an automated self-contained piece of scientific equipment, either due to remote operation requirements (for example, in space [13]), or in their future use as commercial precision sensing machines. We also believe that such control systems will become more and more necessary for fundamental research areas as experiments continue to become even more complex.

1.4 Thesis outline

We start, in chapter 2, by reviewing the science behind the creation and study of ultracold atom clouds. This knowledge is then used to outline the hardware device requirements control systems must support to be effective. We then end chapter 2 with a review of existing ultracold atom control systems, where we take an in-depth look at what they do right, and what needs to be improved upon to support the increasingly complex research being performed.

The labscript suite was developed in parallel with the construction of a Bose–Einstein condensate (BEC) apparatus. We detail the construction of the apparatus in chapter 3 covering the optical and vacuum system designs. We also cover the custom electronics we built, including both the hardware to be controlled by the labscript suite and the custom process control systems that operate independently of the labscript suite.

In chapter 4, we introduce the components of the labscript suite and detail the design philosophy we followed during development. This includes both the software engineering

principles we followed as well as the reasons for choosing specific technologies. This chapter is particularly relevant to those who wish to design their own scientific control system.

In chapter 5 we delve into the design of the programs used for experiment preparation. This covers both the interface we have designed for people to use, as well as the underlying architecture of those interfaces, and the benefits this brings to the design and preparation of experiments.

We then detail, in chapter 6 the execution of a prepared experiment. This includes both how we control the experiment hardware, that in turn controls the apparatus, and the framework we have built for automating the analysis of acquired data. We then discuss how you can use the labscript suite to close the loop, and implement automated feedback of analysis results into the preparation of future shots.

In chapter 7 we discuss the extensibility of the control system where we demonstrate the ease of adding support for new hardware.

In chapter 8 we demonstrate the flexibility of the control system. This covers real-world examples of where we have used the labscript suite to perform scientific research, and ties together the apparatus presented in chapter 3 with the control system.

Finally, we'll conclude with details of where the labscript suite is heading in future development in chapter 9.

Chapter 2

Ultracold atoms & control systems

Our control system, the labscript suite, was designed for use with ultracold atom experiments. While we believe our control system has applications outside of this field, it's design was informed by the requirements of ultracold atom experiments. In this chapter, we first review the physics behind the preparation and study of ultracold atoms with a particular focus on the timescales and timing precision required. We'll then discuss the hardware requirements for controlling an ultracold atom apparatus as informed by an understanding of the physics, followed by a review of some of the other control systems employed in ultracold atom laboratories.

2.1 Ultracold atoms

In this section we'll introduce some basic theory on how to produce a cloud of ultracold atoms, and some examples of how you can use such a cloud to research novel physics. This theory will then inform the requirements of a general purpose control system required to run an ultracold atom experiment.

2.1.1 Magneto-optical traps

The production of an ultracold atom cloud begins with the collection of atoms from a background vapour. Most experiments use alkali metals [14, 15, 16, 17], although some groups are now also working with elements such as hydrogen [18], meta-stable helium [19, 20], chromium [21], strontium [22], ytterbium [23], and erbium [24]. The atoms are collected in a magneto-optical trap (MOT), which consists of three pairs of orthogonal[1] counter-propagating laser beams that are centred on a spatially varying magnetic field produced by coils in an anti-Helmholtz configuration (a spherical quadrupole field). The laser beams are slightly detuned from an atomic transition, known as the 'cooling transitions' due to its cyclic nature[2], so that the atoms in the centre of the trap are off resonant. The polarisation of the laser beams is set such that the Zeeman shift [25, 26] of the atomic energy levels, for

1. Strictly speaking, the pairs do not need to be exactly orthogonal. They must however have components in three orthogonal directions, which is best achieved by three orthogonal pairs.
2. Typically the cyclic transition is only effective for the collection of thermal atoms when present with an additional 'repumping' laser beam, due to other mechanisms that can cause atomic state changes. These laser beams ensure the atoms return to the correct state to continue cooling if they transition to an otherwise dark state.

atoms away from the centre of the trap, are brought on resonance with only the beam(s) that will impart momentum back towards the centre of the trap. The viscous damping and restoring force thus provide a collection mechanism for atoms that fall within the capture velocity of the MOT configuration [27, 28, 29].

The background vapour poses difficulties for the production of ultracold atoms as it provides a hot thermal bath, which would negate further cooling. As such, atoms captured in a MOT are typically transferred to a UHV chamber after collection. This is often achieved by pushing the atoms between two MOTs (separated by a differential pumping tube) with a laser beam [30, 31]. It can also be achieved using magnetic coils that create a moving magnetic trap for the atoms [32].

Alternatively, the atomic vapour source and the MOT can be separated, for example by a Zeeman slower. Zeeman slowers are also used in systems where the atomic velocity is too large to efficiently capture atoms directly from a background vapour. In these scenarios, it is typical to use a Zeeman slower [33] to produce an atomic beam with a sufficient flux at an appropriate velocity for capture with a MOT. Zeeman slowers slow atoms down via the transfer of momentum from photons to atoms, which typically requires a physically long path over which the atoms are slowed. The magnetic field profile along the Zeeman slower is designed to create a spatially varying magnetic field so that a fixed frequency laser beam stays on resonance with the atoms as they slow down. This can either be achieved with a stepped, multilayer, solenoid design (as in [33]) or a single layer solenoid with a varied pitch [34, 35].

2.1.2 Polarisation gradient cooling & optical pumping

While atoms in a MOT are reasonably cold, they are still several orders of magnitude above the temperature required for most experiments. The next stage in cooling is typically polarisation gradient cooling (PGC), which is similar to a MOT, but without the quadrupole magnetic field. In this situation, each pair of counter-propagating lasers produces a spatially varying polarisation (a polarisation gradient) [36]. The polarisation gradient results in a sinusoidal, atomic state dependent, potential for each atom. The lasers optically pump atoms between atomic states when an atom reaches the peak of the potential, and the atom is transitioned to a state with a lower potential energy. This extracts energy from the atom, and it continues at a (slightly) reduced speed, again through a sinusoidal potential, where the process repeats itself (usually returning to the original state). Repeated transitions ultimately cool the atoms to a temperature limited by the recoil energy of the photons used in cooling.

As PGC works by driving atoms between states, the atomic state is not uniform across atoms captured after PGC. PGC is thus usually followed by a stage of optical pumping, which drives the atoms into a common state. This state is typically a magnetically trappable state, such as $|F = 2, m_F = 2\rangle$ or $|F = 1, m_F = -1\rangle$ for rubidium-87. This is required for any further cooling done in a purely magnetic trap, and because most experiments performed with ultracold atoms require the initial atomic state of the sample to be known (even if it is then placed into a superposition of states afterwards). To optically pump the atoms, a bias field is used to establish a quantisation axis and atomic transitions are driven with at least two lasers until the atoms are in the desired state. For cases where the purity is critical

and the transfer is not 100% efficient, an additional purification stage will be performed to remove atoms not in the correct state.

2.1.3 Magnetic traps & evaporative cooling

Further cooling beyond the recoil limit is then done in a magnetic trap using a process called forced evaporation. The magnetic trap is formed by a quadrupole field, similar to the one used in the MOT stage but at a significantly higher strength. There are several types of magnetic trapping configurations, whose differences centre around how to prevent atomic spin flips in the centre of the trap where the field is zero. Common techniques are the time-orbiting potential (TOP) trap [37], the quadrupole and the Ioffe configuration (QUIC) trap [38], the use of an off-resonant repulsive optical field to plug the trap [15], or the use of an off-resonant attractive optical field (known as a 'hybrid trap') [39]. Evaporative cooling of the atoms in such traps is then forced by transferring the hottest atoms (which are located in regions of the trap with higher magnetic fields) to an anti-trappable state. This is typically achieved using either an rf or microwave field (dependent on the atomic structure of the atom and the transitions you wish to drive) which is resonant with only a small fraction of the trapped atoms (at any one time) due to the spatially varying Zeeman shift induced by the quadrupole field. As the remaining atoms will continually rethermalise after the hottest are removed, this procedure can be made to continuously cool the sample by sweeping the frequency of the rf or microwave field.

It is possible to perform scientific research on ultracold atoms in these traps (indeed some of the first BECs were made in a TOP trap [16] and a trap with a repulsive plug [15]). However, many people now perform a final evaporation stage in a purely optical trap.

2.1.4 Dipole traps

For experiments that wish to utilise an atomic state that is not magnetically trappable (or a mixture of trappable and anti-trappable states), or apply or measure additional magnetic fields (such as magnetometry experiments or those that utilise a Feshbach resonance), the atoms must ultimately end up in an optical trap. Such experiments (typically) initially use the hybrid magnetic trapping method mentioned previously, which already requires the use of an optical trap; most commonly the attractive crossed-optical dipole trap. Atoms can be transferred from a hybrid trap into a pure optical trap, by slowly decompressing the quadrupole magnetic field until it is removed. Further forced evaporation can then be performed in the purely optical trap by reducing the laser intensity (and thus the trap depth) of the dipole trap beams. This causes the hottest atoms to boil off, lowering the temperature of the remaining atoms as they rethermalise.

More complex optical traps are also possible, such as a repulsive laser in a TEM01 mode [40, 41] or box potentials created by many lasers [42] (both of which are often used to study 2D physics), or optical lattices [43, 44]. However, in many situations, the early preparation stages still use a magnetic trap and/or a standard optical dipole trap before transferring into these more complex traps.

2.1.5 Novel physics

There is a diverse range of novel ultracold atom research being conducted around the world. While there is far too much to detail in one place, we'll introduce some of the more widely researched topics here.

The study of quantum turbulence is one of these areas of research. While classical turbulence is notoriously difficult to study under controlled conditions, ultracold gases provide a controlled environment in which to study quantum turbulence where the circulation of vortices are quantised. Recent studies often focus on vortex dynamics [45, 46], the way that they cluster together [47] and the energy cascades between length scales [48]. This is an active research topic in our labs at Monash University and researchers hope that such studies will ultimately shed light on classical turbulence.

The use of ultracold atoms for precision metrology is one of several possible commercial applications for ultracold atom research, and is thus also an active area of investigation (including at Monash University). For example, ultracold atoms are used in the most precise atomic clocks [49] and can also be used as precision magnetometers [50] or gravimeters [51]. Techniques for improving the sensitivity of such sensors (for example by removing laser induced vector light shifts [52]) are also a significant area of research.

Other novel research areas include quantum chemistry where ultracold atoms are combined to form ultracold molecules [53] and the ionisation of cold atoms to produces cold-electron bunches [54, 55]. There is also much fundamental physics research done with ultracold atoms. For example, many researchers have been studying topological states [56, 57, 58], the BEC-BCS crossover [59, 60, 61], and the BKT transition [62, 63, 64].

Research groups also put significant work into developing new imaging and trapping techniques. Recently, new imaging techniques have focussed particularly on the ability to perform continuous imaging of the atomic cloud [65, 66] and/or the ability to image and extract more information about small scale structures such as vortex cores [66, 67], as well as bringing analogues of traditional solid state imaging techniques to the ultracold atom world [68]. Meanwhile, trapping techniques are being developed to hold atomic clouds in unique geometries such as ring traps [69, 70] and reconfigurable optical lattices [71].

Some groups also work with multiple atomic species at once. Typically only two species are used [72, 73, 74, 75], however in some instances a third species may be used to sympathetically cool one or both of the other species [76]. These are often useful for quantum simulation experiments or fundamental research into quantum matter. The combination of species is often chosen due to an appropriate inter-species Feshbach resonance, which allows the interactions between the two species to be controlled.

Many of these advances require new control methods, such as more complex control over output state, synchronisation of novel control devices, and/or challenging timing requirements. Working with multiple species also requires significantly more hardware, as the collection and cooling of each atomic species requires a distinct set of lasers for the cooling stages, and there may be additional transport stages required to effectively combine the species together, all of which need to be under computer control.

2.2 Controlling an ultracold atom experiment

As we can see from looking at the physics contributing to the creation and study of ultracold atoms, there are a broad range of hardware and timing requirements. We can also see a progression in the complexity of the cooling stages. This matches closely with the control techniques available at the time they were developed. For example, initial laser cooling experiments were controlled using simple digital electronics and delay lines. As the complexity grew, programmable function generators were included for simple analog ramps, which could be triggered by a digital signal. Ultimately with the increased availability of low cost PCs and general purpose I/O hardware, these simple systems were replaced with computer control systems. As PC control systems naturally evolved out of reasonably 'fixed' electronics based systems, the early attempts followed a design pattern that was also rather static, leading to control systems that were designed for a singular purpose and were difficult to adapt to changing experiment requirements (this will be discussed further in §2.3). In order to design a better, general purpose control system, we must first understand the range of requirements that encompasses the many possible ultracold atom apparatuses.

2.2.1 Timescales

The processes used to create a cloud of ultracold atoms operate over a range of timescales. The MOT load (other than switch on and switch off), including designs using a Zeeman slower, is completely static in most experiments[3]. MOT load times are typically on the order of seconds, although this is dependent on many parameters such as available laser power, beam diameter and number of atoms available. MOT transfer systems (using a push beam or moving magnetic trap) typically require millisecond control of either the laser beams or magnetic fields. Similarly, PGC typically operates on the order of milliseconds, depending on the intensity and detuning of the lasers used, and the velocity distribution of the atoms to be cooled. Optical pumping, on the other hand, typically takes between 10 µs and 1 ms, depending on the intensity and detuning of the lasers used, and often requires precision timing at the microsecond level in order to stagger the switch-off of the lasers (which ensures the atoms are all pumped into a common state).

While forced evaporative cooling typically takes on the order of a few seconds (which is similar to the length of the MOT load), this is not the only relevant time scale for computer control. The atoms, prior to the load into a pure magnetic trap, must be in a magnetically trappable state. As such, it is critical that the atoms be able to adiabatically follow the quadrupole field when it is switched on (this similarly applies when it is switched off). This requires that the magnetic field be switched on gradually, and that field is sampled sufficiently between the off and full strength states. Further more, the frequency of the rf or microwave field that drives the evaporative process must be swept, typically overs tens of MHz, but with sufficient granularity to ensure the hottest atoms are those being addressed. As such, the forced evaporative cooling stage typically requires a sub-millisecond update rate, over a period of several seconds, for multiple outputs. Similar requirements exist for the transfer into a purely optical trap, and any further evaporation that occurs there.

3. It is important to stress here that while there are dynamics occurring during the collection of atoms, the laser beams and magnetic fields are set at constant values.

Once an ultracold atom cloud has been produced, the physical processes that are being studied typically occur on timescales that range from microseconds to more than one second. Data acquisition may then need sub-microsecond resolution (in order to capture the dynamics of the process) and will likely need to be synchronised with the change in output states to the same precision. For example, camera acquisitions are usually triggered by a change in state of a digital output.

We can see from this that the timing requirements go beyond those that a general purpose PC can provide by itself. Most operating systems (such as Microsoft Windows) can typically only guarantee timing to the millisecond level. Real-time operating systems do exist and can get to the microsecond level, but require overly complex programming to take advantage of this, and are still only borderline acceptable for modern day experiments. As such, most ultracold atoms experiment use dedicated devices to interface with the experiment, that are not bound by PC operating system limitations.

2.2.2 Hardware requirements

Ultracold atom experiments typically require control of dozens of I/Os. For example, digital outputs may be used to turn on or off the various laser beams used in the cooling stages of the experiment (using a digitally triggered shutter), or to trigger a camera to take one or more pictures at specific times. Analog outputs can be used to control magnetic fields smoothly (using a magnetic coil driver) and rf or DDS outputs can be used to control laser frequencies (via acousto-optic modulators (AOMs)) or the rf used during forced evaporation. The experiments run using ultracold atoms might also make use of devices such as SLMs or digital micromirror devices (DMDs), and may need to acquire analog input traces as well as camera images. Producing this set of I/O typically involves on the order of a dozen different devices that must be controlled in parallel while remaining synchronised over the length of the experiment. The devices used vary depending on the requirements of a particular experiment, but are often drawn from a mix of commercial off-the-shelf hardware and custom hardware designed to push the limits of what is commercially possible in order to study novel physics. Given that such hardware is so dependent on the specific physics to be studied, it is important that a general purpose control system be able to support a wide variety of hardware, and for it to be easy to swap the hardware in use when the requirements of the experiment inevitably change as new physics is investigated.

In order to maintain precise timing of updates to hardware outputs, I/O devices typically prefill an onboard buffer with hardware instructions and step through them according to a provided clock tick. The most common approach is then to step through these instructions (after receiving an appropriate trigger to start) with a clock that ticks at a constant rate (see figure 2.1), since such clocks are easy to generate and can be trivially included in the design of the hardware. However, this becomes unwieldy for experiments where the ratio between the experiment length and the required timing resolution is large. For ultracold atom experiments, this ratio is at least 10^6 (10 microsecond timing resolution during experiments of 10 seconds) and may even hit 10^9 (100 nanosecond timing resolution during experiments of 100 seconds). Each output channel driven by a constant rate clock would then require somewhere between 10^6 and 10^9 hardware instructions in order to achieve the desired timing resolution and duration. This could be as much as 2 gigabytes of data per

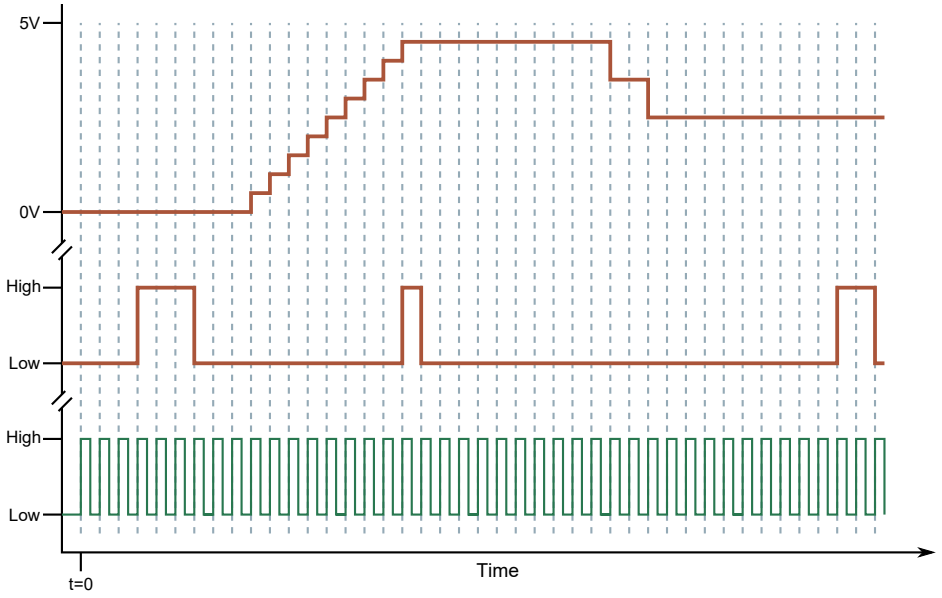


Figure 2.1: An example of a standalone I/O device output (orange), internal clocking signals (green) and timepoints at which hardware instructions are required (dashed blue). The upper trace shows an analog output which starts at 0 V, ramps to 4.5 V over 9 timesteps, waits for 11 timesteps and then ramps down to 2.5V over 3 timesteps. The middle trace shows a digital output which produces 3 pulses of various lengths and the lower trace shows the internal clock that steps the output through the hardware instructions. The rate of the fastest ramp effectively sets the frequency at which the clock must tick. As the internal clock is a fixed frequency clock, and does not make use of a pseudoclock, hardware instructions (holding the output state of all channels) must be stored for every clock tick (indicated by the blue, dashed, vertical lines). There are thus several periods where sequential instructions store the identical output state, wasting space in the onboard buffer.

channel (for a 16-bit analog output). However, most output devices do not have the onboard memory to support such large instruction sets. While some manufacturers instead stream the hardware instruction set to the device during the experiment execution, thus avoiding onboard memory limitations, this scheme is difficult to implement well (and so only large manufacturers such as National Instruments (NI) attempt it). It comes with a risk of failing to stream data fast enough if resources or bus bandwidth on the PC are consumed by other processes, and the number of hardware instructions is still limited by the memory of the control PC.

We use the concept of a pseudoclock to work around these memory limitations, by providing a mechanism to vary the time between instructions. This allows us to maintain high timing resolution only during the segments of the experiment where it is necessary, reducing the required number of instructions to within the capabilities of most devices with small onboard memory buffers.

2.2.3 Pseudoclocks

Despite requiring precise timing resolution during some parts of an experiment, it is unlikely that a long experiment requires a change in the output state at every one of those time

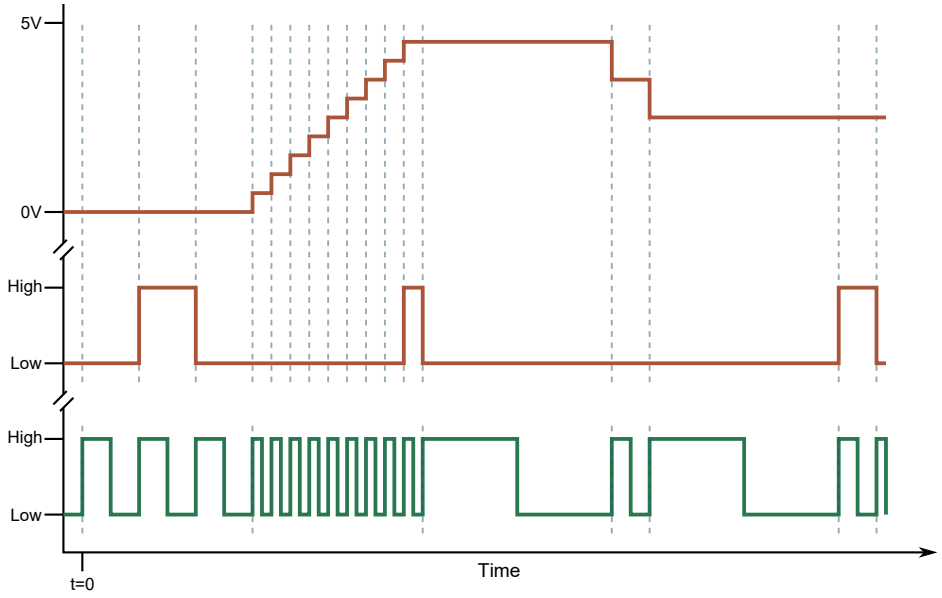


Figure 2.2: Here we show an I/O device with equivalent device output (orange) to figure 2.1, but this time using an external pseudoclock (green). We see that the number of hardware instructions (for the I/O device) is reduced to 17 (upper dashed blue lines). The pseudoclock itself may require as few a 7 instructions (lower dashed blue lines) if it supports looping instructions and requires only a single instruction per change in clock rate. This would increase to 14 if it supported looping instructions but required two instructions per change in clock rate: one instruction for transitioning from low to high and another for transitioning from high to low, with additional data in those instructions to indicate they should be looped over the requisite number of times.

intervals. A device that steps through instructions based on a clock that ticks only when an output needs to change state then only needs to store instructions when a change in output state is required. We term such a clock a 'pseudoclock' (otherwise known as a 'variable frequency clock' [77]) and usually consists of a device programmed to produce a non-uniform clocking signal (an arbitrary train of digital pulses). The pseudoclock scheme thus results in a significant reduction in the onboard memory requirements of I/O devices.

Offloading the timing to a pseudoclock still requires the pseudoclock device to have a large onboard memory buffer if the pseudoclock instructions are being stepped through at a constant rate, as the pseudoclock will need an instruction every x seconds where x is defined by the shortest time interval during the experiment. While this scheme results in an overall reduction in memory use (you only need to store instruction data for a single clock output at the $\frac{1}{x}$ rate, rather than for every output driven by this clocking signal), it is still not the optimal solution. The memory requirement for the pseudoclock can be reduced further if pseudoclocks are implemented on devices that support complex CPU-like instructions such as 'delay', 'branch', 'period' and/or 'loop'. For example, a pseudoclock with 'period' and 'loop' CPU instructions could be used to create a single hardware instruction that produces multiple pseudoclock output state changes, with the 'period' relating to the rate of the pseudoclock during an instruction and 'loop' referring to the number of times the pseudoclock should tick before processing the next instruction. Pseudoclock instructions

can thus (on some devices) be reduced to one instruction per output clock rate change (see figure 2.2).

Of course if every output device supported such complex instructions, we would not need separate pseudoclocks. However, processing these complex instructions requires a much more advanced processing core for the device, and so almost all manufacturers avoid implementing such features in order to reduce development time and production costs. As such, we believe this pseudoclock architecture provides the best option until such time as all hardware devices have much larger onboard memory or support for more complex instruction sets, and is a key component of the architecture of our own control system[4].

2.3 A review of existing control systems

Control systems for ultracold atom experiments are particularly interesting to study because they (a) were only able to be experimentally realised in the PC age and (b) are not yet used in any commercial or large scale project. This means that current control systems have been built by scientific researchers, and often are made available as open source projects. It has also led to a variety of approaches rather than being dominated by a single commercial offering.

The most well known designs are *line-based* systems (colloquially known as a 'green dots' system, named for its common implementation with LabVIEW's archetypal green button design) which covers the many home-grown control systems that never propagated beyond the original developers, despite being recreated in various forms by many laboratories. In line-based systems, the description of the experiment logic is done via a graphical representation of the hardware instructions, which commonly takes the form of a 2D array of buttons indicating the state of a digital channel for a given time. However, this design makes it hard to modify the experiment logic, which we'll touch on during this review and then more formally cover in §4.3.3. For line-based systems written in LabVIEW, the limitations LabVIEW places on dynamical GUI generation mean a modification of the set of hardware controlling the apparatus also requires 'rewiring' the underlying design of the control software from within LabVIEW. More modern systems move away from the LabVIEW dataflow language to avoid this, but often replicate an identical interface for defining experiment logic, thus maintaining many other limitations. Only a small number of control systems brake away from this design pattern entirely, providing a more generic API to describe the experiment logic.

We find that ultracold atom experiment control systems fall into two broad categories. The first encompasses basic home-built line-based control systems (typically written in LabVIEW) and more advanced variants such as the Cicero word generator. Such systems are categorised by their user interface design, which is built around sequential time steps (or sequence of 'words' in the case of Cicero) containing an array of values for each channel. While such systems may contain some advanced features, the user interface design forces the user to work in terms that are very close to native hardware instructions. These systems also

4. The use of a pseudoclock is not a novel concept, as evidenced by the use of a similar scheme in some of the control systems we will review in the next section. However, we contend (as we hope is born out by this thesis) that our implementation is the most general to date by removing restrictions requiring a specific model of device and supporting the use of an arbitrary number of pseudoclocks.

typically have limited hardware support (or a large barrier to adding hardware support), and may be geared towards supporting older hardware devices such as those that do not support pre-programming with a table of instructions but are instead updated in software (PC) time by the control system. They also (typically) have more limited automation of parameter space scans. We thus feel this group should be termed 'first generation control systems'.

The second group comprises those control systems that use a higher level representation for defining experiment logic, and more advanced parameter space management features. These are the 'second generation control systems' of which, we believe, the control system we present in this thesis is the most comprehensive offering so far. Support for a wider range (or dynamic configuration) of hardware is also a common theme of such systems.

We'll review some of the most well-known control systems for ultracold atoms, looking particularly at:

- their interface for defining experiment logic,
- their ability to link experiment logic to a separately defined list of parameters (otherwise known as global variables) and automate the collection of data over a parameter space scan,
- shot management, including the level of automatic record keeping, how data is stored, and the way they prepare, stage, execute, and analyse shots,
- the range of supported hardware devices and the ease of adding support for new hardware, and
- their ability to perform analysis on data acquired during an experiment.

2.3.1 LabVIEW line-based systems

The line-based control system is the most familiar system for ultracold atom experimentalists despite being the least well standardised. These systems are characterised by a grid of buttons (often the standard LabVIEW 'green dot' button) that allow a bank of digital channels to have their state graphically controlled at a variety of time points (see figure 2.3) and are written using the LabVIEW graphical programming language[5,6]. Such systems are prevalent due to the ease of creating such a system in LabVIEW, as well as the excellent interoperability with the widely used hardware devices from the same manufacturer (NI). This has led to many laboratories creating their own custom control system with a variety of feature sets specific to each laboratory and with varying degrees of completeness and robustness. Despite the diverse nature of this type of control system, we will attempt to review what we consider to be common features and design decisions.

5. It is important to note that by "graphical programming language" we don't refer to a programming language that can make graphical interfaces (although LabVIEW does that as well) but rather we refer specifically to a programming language where the logic of the program is defined using graphical symbols rather than the more conventional text-based code.
6. We focus solely on control systems written in LabVIEW here. We separately cover variations on the 'green dots' line-based theme, written in other programming languages, in the following sections.

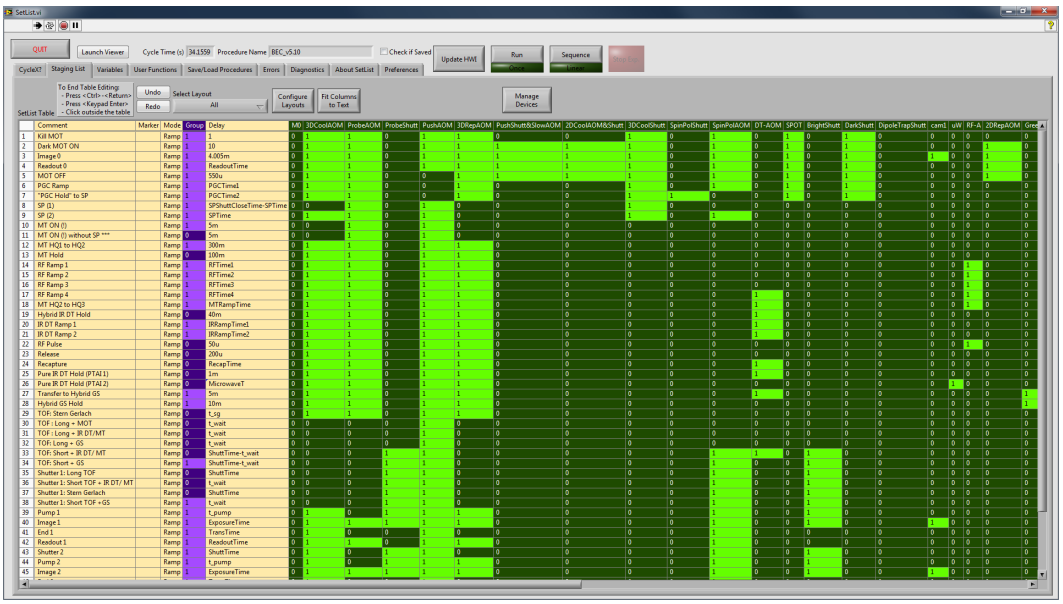


Figure 2.3: The LabVIEW line-based control system 'SetList' from JQI. Image provided by Chris Billington via private communication (1st Feb. 2019), reused with permission.

Experiment logic interface

As previously discussed, the main interface for this class of control system is graphical. Experiment logic is defined using a 2-dimensional array of graphical widgets, with axes of output channels and timepoints. The value of a widget thus defines the associated channel state at the associated time. While originally many systems only controlled digital outputs, with the advent of modern hardware many systems also incorporate analog widgets as well (for example, to define a voltage). While static analog values map well to the 2D grid structure, the ability to specify analog ramps requires additional complexity in the interface as ramps are not practical for a user to implement using an array of manually specified timepoints. Time is also typically defined using an analog widget, which leaves open the possibility of non-uniform time steps either through the use of a pseudoclock or the duplication of instructions (to create a fixed frequency update rate) at programming time.

Parameter management

Parameterisation of experiment logic may exist in some advanced LabVIEW line-based systems, but many do not provide any such management. If parameter management is implemented, it is usually limited due to the difficulty of implementing such a feature in a graphical description of experiment logic.

Shot management & storage

Due to the 'simple' nature of this type of control system, shot management and storage is often non-existent. There is typically no system for managing the execution of a sequence of shots, and while saving of data acquisitions are supported, there is typically limited meta-

data (such as experiment run time, experiment configuration, or a record of the experiment logic) saved. Cataloguing and storage of useful data (along with relevant parameters) is typically done by hand, rather than automated into consistent formats and hierarchies.

Hardware support

There is no particular limitation to the hardware that can be supported by such systems. However, the grid interface (and the way that LabVIEW handles data internally) makes it difficult to abstract away any of the hardware implementation details. As such, the graphical interface is often made up of sets of controls specific to a given device. New devices are thus more difficult to implement because everything from the interface down to the programming of the device must be written again. There are of course ways to make it simpler to add new devices, but these require significant development effort and developers willing to put in that much time typically end up eschewing LabVIEW for another programming language. As such, LabVIEW line-based systems are typically limited to the control of a few pieces of hardware that feature in a given lab, which may explain why these systems are not typically shared between research groups.

Analysis

Line-based systems typically do not include analysis systems, although simple fitting to images of atomic clouds may be included if the system incorporates the acquisition of images through the LabVIEW interface.

2.3.2 Cicero word generator

The Cicero Word Generator [78] is an open source control system developed by the Wolfgang Ketterle group at MIT. While ultimately a variation on the LabVIEW line-based theme, it is perhaps the most comprehensive of such systems. This is likely due to the use of C# as the programming language of choice which, as the authors note in [77], provides more flexibility than LabVIEW. As one of the few open source control systems, Cicero also has a wide user base across multiple institutions.

Experiment logic interface

The primary interface for defining experiment logic consists of a sequence of 'words' which define the state of each output during the specified time interval (see figure 2.4). While this may sound similar to the standard line-based approach previously discussed, care was taken to overcome many of the limitations of that design [77]. For example, the user interface provides the ability to enable/disable, reorder or duplicate specific words. The set of words that form the experiment logic for a given experiment can be saved (into a proprietary file format) and loaded at a later time, allowing lab users to easily switch between different 'experiments'. The state of a specific channel in a word can also be bound to a global variable or be configured to 'continue' (use) the value of the previous word. Analog ramps can be defined using predefined functional forms (with parameters bound to global variables) or via the entry of a custom equation. Digital outputs can be programmed to trigger prior to or after the boundary between two words, providing a mechanism for managing devices

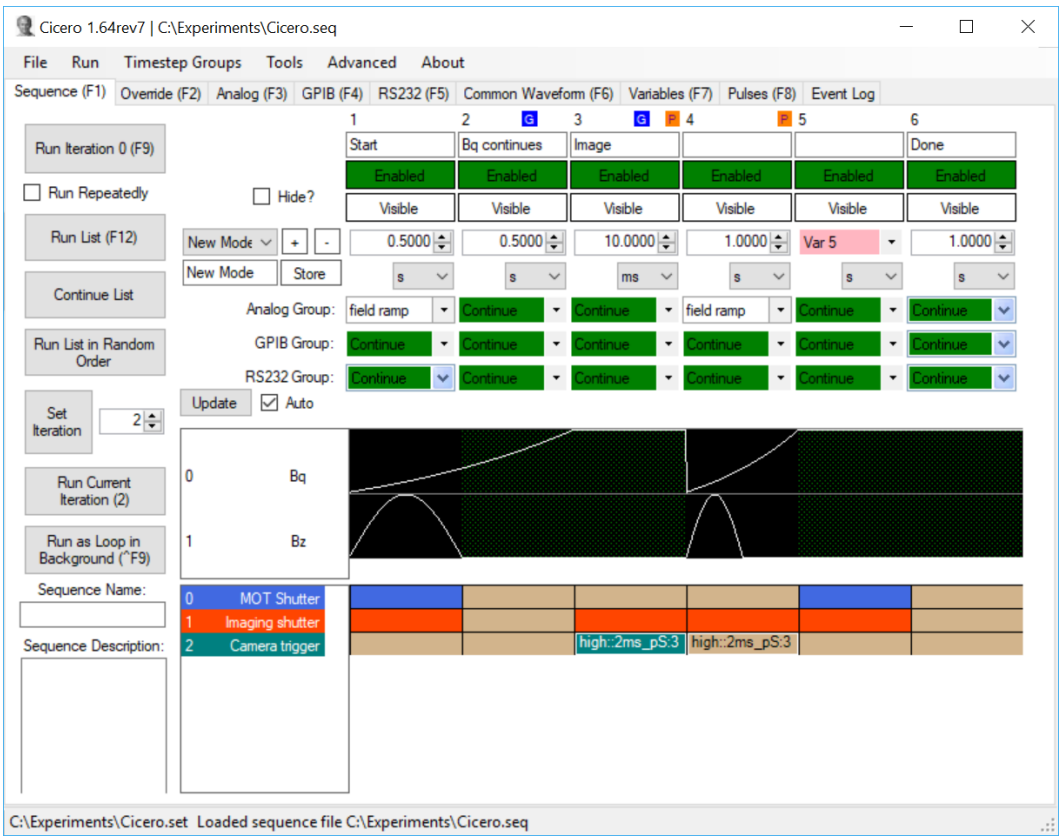


Figure 2.4: An example of the Cicero Word Generator interface. Here, each 'line' is a column of the main interface (labelled 1, 2, 3...), defining a time point and the state of each channel at that point. Analog channels can follow more complex equations during a time step, can span more than one time step, and the shape of the trace is pictured. Values, such as instruction durations, can be bound to global variables. Digital lines can be delayed relative to the boundary of a time step as shown for the camera trigger in time steps 3 and 4.

with known delays (such as shutters). Cicero also supports programming arbitrary devices over serial (RS232) or GPIB.

Despite this effort to overcome some of the inherent limitations of the line-based design, it is clear that this style of graphical interface has its limits. For example, despite RS232 and GPIB being similarly generic communication protocols, only GPIB devices can be programmed to follow an analog ramp from within Cicero. Also, while it is easy to bind a global variable to a parameter of a waveform or the length of a word, if two parameters are related by a common parameter (for instance a parameter that should be half of another) you need to explicitly define additional global variables, even for just a simple scaling. Comparatively, this would be considerably simple to implement in a text based language where equations form a natural part of the programming language and can thus be used at any point in the code (for example an argument to a function call could take a parameter set as `2*parameter`).

Defining delays on digital channels is also non-trivial. For example, the Cicero manual [79] defines the procedure as follows:

> *To create a pulse that will make a digital output true, will start 2 ms after the start of the timestep it is in, and end 3 ms before the end of the timestep it is in, set:*
>
> - ***Start Condition*** *to* ***TimestepStart****,*
> - ***Start Pretrig/Delay Enabled*** *to true,*
> - ***Start Pretrig/Delay*** *time to 2ms,*
> - ***Start Delay*** *to true,*
> - ***End Condition*** *to* ***TimestepEnd****,*
> - ***End Pretrig/Delay Enabled*** *to true,*
> - ***End Pretrig/Delay*** *time to 3ms,*
> - ***End Delay*** *to false, and*
> - ***Pulse Value*** *to true.*
>
> *To insert a pulse into the sequence, right click on the digital box you wish to apply it to (in the sequence tab digital grid, corresponding to the channel and timestep you want the pulse to apply to), to see a drop down list of available pulses.*

While it is commendable that such a feature exists (as it does not in most other line-based systems), it exposes the inherent difficulties imposed by defining experiment logic graphically as a sequence of discrete time units. In contrast, this feature becomes almost trivial to implement with a high level text based API for defining experiment logic which might look like[7]:

```
# word starts at time=t

# set output high 2ms after start of word
digital_channel.go_high(t+2*ms)

# other events go here at time=t
# ...

# increment time counter
t += word_length

# now set the output to low 3ms before end of word
digital_channel.go_low(t-3*ms)
```

7. This is a somewhat contrived example. In a text based language you don't need to be defined by 'words' at all, nor do commands necessarily need to be defined in time sequential order. As such, the text based version presented in the above example is actually far more verbose that it need be (it's two thirds comments!), necessitated by the desire to provide an example that could be directly compared to the Cicero example.

Parameter management

Cicero provides a simple GUI for managing parameters (global variables). While there is no limit to the number of global variables, they can not be grouped or easily reordered. This suggests using a large number of them would be prohibitive. Global variables can be defined as floating point numbers, bound to a list of numbers or defined as an equation (which can make use of standard mathematical functions, operators and other global variables). Lists are used to produce a sequence of shots where each shot uses a particular value for the list. For a single list, the number of shots produced is equal to the number of items in the list, where the associated global variable takes on a different value from the list in each shot. When multiple lists are in use, the behaviour of Cicero can be set to either iterate over the lists in lock-step (in which case the lists should be of equal length and the number of shots produced is equal to that length), or to take the outer-product of the lists in order to span a large parameter space (in which case the number of shots produced is equal to the product of the list lengths)[8]. Most complex combinations of iterating over some lists in both lock-step and taking the outer product of others are also possible, although the interface is not always consistent in this regard. For example, while simple combinations can be controlled via buttons next to the lists (that change the behaviour for that list between lock-step and outer-product), defining two separate groups of lists that internally iterate in lock-step, while the groups themselves are also combined in an outer-product, is only possible by setting the lists to be combined via an outer-product and using multiple global variables to define the groups of values that will iterate in lock-step. This imposes the condition that values in each independent group are actually related via a common list. It is also, unfortunately, not possible to define lists programmatically (for example using a function that returns a range of values, such as the MATLAB [80] or Python NumPy [81] `linspace` function).

Shot management & storage

Shot management is quite advanced, when compared to most control systems, providing the ability to loop a single experiment in the background if necessary (typically used to 'keep warm' an apparatus when it is not being used for data collection), run a sequence of shots as defined by the parameter lists (and to do so in a random order or not), and to run a single shot repeatedly. An independent shot can also be set to run every N shots, when iterating over the parameter space, in order to provide regular base-line measurements of the apparatus for calibration. Running shots (except for looping the background shot) blocks the user interface, which does prohibit queuing up multiple sequences of shots to run.

When shots are run, Cicero creates a 'log' file for each shot. This file contains a record of the experiment sequence, which can be re-opened within an instance of Cicero for later viewing. There is however, no unified method of storing any data acquired during the shot,

8. Confusingly, the Cicero manual[79] terms taking the outer-product as scanning over the list "in parallel" and scanning in lock-step as "in series". From a programming perspective, 'in parallel' implies something happens at the same time which seems to better correspond to updating the current value used from each list at the same time. Similarly, 'in series' could equally apply to nested loops (which is ultimately what taking the outer-product corresponds to) as 'in series' implies a dependency between the two items. It is for this reason that we use 'outer-product' and 'in lock-step' (or 'zip group' which is Python terminology for 'in lock-step') to describe the two different behaviours in our own control system.

limiting the log to a record of what was done to the apparatus rather than a record of what actually happened.

Hardware support

The Cicero Word Generator implements interfaces to hardware via a separate application called Atticus. Cicero can communicate with multiple Atticus servers at once, allowing for control of distributed hardware. The Atticus server is designed to be primarily used with National Instruments hardware that utilises the NI DAQmx library as an interface [77]. The creators of Cicero also provide custom firmware for an Opal Kelly FPGA development board for use as a pseudoclock (not unlike the PineBlaster pseudoclock developed for our own control system [8, 82, 83]).

Support for older hardware that can only be updated in software time (cannot be synchronised via the pseudoclock) is also provided through GPIB and RS232 interfaces. GPIB devices can be programmed to execute ramps of analog outputs that have been defined via a parameterised waveform in the Cicero interface. RS232 devices can only be sent a single command per word.

Support for additional hardware can be added by producing a custom version of Atticus. An example server template is provided as part of the installation. However, this is primarily designed to support additional hardware devices which are similar in function to the NI cards. Adding support for devices with additional output types (such as images for display on a SLM) would also require modifying Cicero itself which is unlikely to be a trivial task.

There is no documented support for managing image acquisition from cameras or analog acquisitions from NI cards. However, there is evidence of some undocumented support in the source code. Unfortunately, I was unable to confirm whether this feature worked, as the NI DAQmx drivers we had installed on our laboratory PC were not compatible with the currently compiled version of Atticus available for download. This outlines another downside common to control systems implemented in programming languages that require compilation of source code into executables. Implementing interfaces to 3rd party hardware libraries is difficult to maintain, due to the static compilation process which links to a specific version of libraries at compile time. This means users must have the identical version of all 3rd party libraries installed if they cannot be distributed along with the software, or users must be walked through the recompilation of the software, which usually requires the installation of complicated toolchains for both the main software and possibly the 3rd party libraries. Such problems are less likely to occur in higher level languages like Python, where interfaces to 3rd party libraries can be dynamically generated at runtime from a range of compatible versions that are already present on the users system (as is the case for the PyDAQmx library [84] for controlling NI hardware from Python).

Analysis

The Cicero Word Generator does not provide any analysis tools.

2.3.3 Strontium BEC control & vision

The Strontium BEC Control & Vision system is an open source control system developed in a BEC group at the University of Texas at Austin [85]. It consists of two main programs, 'Control' and 'Vision', and a set of ancillary programs for interfacing with certain camera drivers. The software is written in a combination of Visual C++ (for 'Control' and the ancillary programs) and Borland C++ (for 'Vision').

Experiment logic interface

Unlike the control systems reviewed in the previous sections, this control system is a managed through a combination of both textual (code) and graphical interfaces. The textual interface is provided by the Visual C++ integrated development environment (IDE) which provides tools for directly modifying the source code of an application. In the case of this control system, the user is directly editing the source code that produces the 'Control' application and associated graphical interfaces. The textual interface is used to define both a list of parameters that can be later controlled graphically, and the logic of the experiment sequence. By implementing the experiment sequence in code, users are able to utilise standard programming control flow tools and text editor tools (such as copy/paste) to manage the experiment logic. There are many additional benefits to a textual interface for defining experiment logic, which we discuss further in §4.3.3 and §5.1.2. While, in general, this provides a powerful and feature rich interface, the specific implementation in the Strontium BEC control software brings added complexity that may turn off many users.

The primary issue with modifying the source code of your control system directly is that the operation of your *entire* control system is dependent on the correct modification of the source code at each step. Incorrect modification can result in a variety of issues that prevent compilation (and thus execution) of your control system application. These range from simple syntax errors, which must be tracked down and fixed, to more subtle issues such as poor memory management that result in memory leaks, which may ultimately slow and crash the entire operating system, or memory access violations, which could cause the control software to exit unexpectedly. These issues can be minimised with careful use of version control and appropriate debugging tools, but the use of such tools requires someone with significant software engineering knowledge in each laboratory, which may not be practical. Recompilation of graphical software is also typically slow, which leads to added delays between designing experiments and executing them.

Parameter management

Global variables are defined in the source code of the application (much like the experiment logic) and these definitions are used to automatically generate the graphical interface of the software. This allows the user to modify key parameters of the experiment logic without needing to recompile the Control application. The 'Control' software thus supports global variables much like Cicero, but with significantly more management features. For example, global variables can be grouped into different 'menus' (which effectively act as different dialog windows for the application) and additional descriptive text can be placed on the window. The order of global variables can also be controlled by the order they are defined in

the source code, something that Cicero does not yet support (due to the increased difficulty of adding complex graphical controls).

Due to the choice of C++ as the implementation language, global variables are typed. Users can choose between `long` (an integer), `double` (similar to a float, or floating point number, but with twice the precision), a `bool`, or a string (either freeform, or with selectable options from a drop down menu). Equations are not supported via the graphical interface (and thus globals cannot depend on other globals). However, due to the textual nature of the experiment logic, equations can be defined there instead (although these are then not as easily visible to the user).

Global variables can also be varied over a range of values in order to make measurements of a quantity as a function of another, which is also similar to Cicero.

Shot management & storage

The Control software contains a queue of experiments (or 'measurements') to run. Again, this is quite similar to Cicero except that it appears the queue does not block the graphical interface while shots are executing.

There is very little in the way of shot storage in Control and Vision. The list of global variables is passed to the Vision software, which saves them as part of the analysis process. The absorption images of a BEC, atom cloud fit results and single point analog measurements are also saved. There is no record kept of the experiment logic or other metadata unless the user implements their own systems (such as source code version control).

Hardware support

Control is shipped with support for a wide variety of hardware including NI devices. As users are already expected to work with the source code of the application to modify the experiment logic, adding support for new hardware is easily accessible to end users. There is no native support for distributing hardware across multiple PCs. Control was also designed around a set of custom hardware that runs and communicates over a digital bus. As such, it does not appear designed to support a general purpose pseudoclock system[9].

Unfortunately the support for analog acquisition is minimal, with no obvious way to acquire analog time series measurements. It appears the primary aim of this control system was to investigate ultracold atoms via images and that analog time series were not critical. This was perhaps true in the early BEC days, however our experience is that analog time series acquisitions are becoming more critical to the analysis of more complex BEC experiments.

Vision was also designed for modular hardware support of different cameras. Vision communicates over a TCP network connection with small 'camera server' applications that act as an intermediate between the camera driver and the Vision software. These intermediates can be written in any language (provided a TCP or socket networking library is available), which ultimately means there are very few limitations on the models of camera

9. There is evidence that the custom hardware has been pseudoclocked [86], however it doesn't seem to have been integrated with Control, and instead was developed to work with an 'unfinished' control system called Z.759 [87] (part of the ZOINKS project). This unfinished control system was eventually abandoned in favour of a Python alternative [88], but that system was never made publicly available.

that can be integrated with Vision. Vision ships with support for many Andor and Apogee cameras, which are popular in many laboratories.

Analysis

The 'Vision' application provides basic analysis for BEC experiments. This is primarily through 2D fits to an absorption image of a BEC, from which various results such as trap geometry, temperature or atom number can be determined depending on the experiment performed. Vision also provides a scatter plot widget, which can be configured to display results obtained over several shots. A history of previous shots is also displayed, allowing the user to quickly revisit the analysis of the last 6 shots. Vision was initially designed for use with an apparatus that contained both lithium-6 and lithium-7, and thus supports switching between the analysis of multiple isotopes. Supported elements and isotopes can be changed by adjusting relevant values in the source code of Vision, and the isotope parameter can also be repurposed to represent other quantities (such as whether the image was from a camera imaging from the side, or a camera imaging from the top). This is quite advanced when compared to other control systems, but is not easily extensible for use with experiments that do not align with the original design.

2.3.4 Others

There are additional ultracold atom control systems that have been developed but not released publicly. These include one from the group of Tilman Esslinger [89] and one from the group of David Hall [90]. Due to the inability to access manuals or source code for these, it is difficult to evaluate them. However we will discuss what is known about them from available publications.

The system used in the Tilman Esslinger group, created by Thilo Stöferle [91], appears to be very similar to Cicero. Experiment logic appears to be defined graphically, using an array of instructions. Like Cicero, there is support for global variables (supporting mathematical functions in their definitions) and arbitrary waveform generation for analog outputs. Unlike Cicero, the generation of hardware instructions from the experiment logic is handled by a separate program and these hardware instructions are generated independently of shot execution ensuring shot execution does not block graphical interfaces. The control system also includes an application for managing cameras and providing routine analysis of acquired data. While there is no description of the features of this system available, screenshots indicates it is quite similar to the Vision software from the Strontium BEC group detailed in the previous section. Theses published as late as 2016 [92] indicate this control system is still in use in the lab, although it is unclear how many upgrades have been performed since it was originally detailed in 2005.

The control system from David Hall's group uses a combination of LabVIEW to create a graphical interface and a C++ program to generate hardware instructions[10]. This allows

10. We should point out here that the title of the paper describing this system calls it a "line-based" system. In this case, it seems they are referring to the line-based nature of the high level programming language, and not the line-based nature of the hardware instructions. In this thesis, line-based refers to experiment logic that is close to the low-level hardware instructions. Programming languages that are used to abstract away the generation of such instructions are (in this thesis) typically referred to as 'high-level' interfaces instead.

users to benefit both from the rapid ability to create easy to use graphical interfaces and the benefits of defining experiment logic in a textual interface. This system will suffer from similar issues to the Strontium BEC control system, however they will be mitigated somewhat by the fact that the C++ component does not generate the graphical interface, making it more robust and faster to compile. Experiment logic is defined as C++ code using a simple API and compiled into an executable file. This executable file is launched by LabVIEW when a shot is to be run, and the executable reads in globals values from a text file. The executable then generates the required hardware instructions and writes this to a file. LabVIEW then reads this file and programs the hardware, before starting the experiment. LabVIEW can also be used to modify the experiment logic (and initiate recompilation of the C++ executable), modify the globals parameters and respond to data acquisitions. Additionally, experiment logic can be broken up across multiple C++ executables, which LabVIEW can be instructed to run in sequence. This allows for LabVIEW to insert software timed logic in-between segments of hardware timed logic, which can be quite powerful. This architecture does not lend itself to queuing up a sequence of shots to run though, requiring the LabVIEW interface to be specifically designed to automate the adjustment of parameters. Finally, while not explicitly stated, the examples in [93] imply that the software supports automatic pseudoclock generation.

Outside of ultracold atom research, there are also some control systems worthy of mention here: ARTIQ [94], QCoDeS [95], Qudi [96], and EPICS [97, 98]. ARTIQ and QCoDeS are primarily designed for quantum information experiments. ARTIQ is designed to work with specific FPGA devices running custom firmware. The firmware allows the FPGA to run Python code (to define the experiment logic) which in turn communicates with a separate real-time core running on the FPGA to ensure precise timing. However, it is not shot-based and is not a comprehensive control system for parameterising, executing and analysing experiments. This makes it unsuitable for ultracold atom research by itself, however it is well suited to ion-trapping experiments (the target audience) and could be of use as one of several devices in an ultracold atom experiment (provided there was an additional scientific control system to manage them all).

QCoDeS on the other hand, is a framework for controlling many different hardware devices. However, the focus of QCoDeS is on usability through Python Jupyter notebooks[11]. This design philosophy seems to have lead to a design that does not focus on hardware timed control and acquisition, at least to the extent of ultracold atom control systems. It does, however, lead to a friendly interface for performing real time analysis on acquired data.

Qudi is similar to QCoDeS, but primarily designed for quantum optics experiments. While Qudi also doesn't focus on precision timing of experiments, it does focus on distributed graphical interfaces as the primary means of interaction. Finally, EPICS was designed for use in particle accelerators (such as particle colliders and synchrotrons). While technically a 'scientific control system', it has a significant focus on process control of the entire particle accelerator as well as data acquisition for scientific experiments. Again, the mixing of process control into the scientific control system has lead to a design that does not focus on hardware timed control as much as ultracold atom control systems. So while it does purport to be a 'real-time' control system, it is not precisely timed.

11. These have similar functionality to Mathematica notebooks.

2.4 Summary

In this chapter we reviewed the physics behind ultracold atom experiments, and used this to inform ourselves of the hardware device requirements for such an apparatus. We then reviewed some of the existing control systems, which covered a broad range of approaches to the control of ultracold atom experiments. Some of these, like Cicero and LabVIEW line-based systems, are the product of historical control systems. More recent ones like the Strontium BEC control system and David Hall's control system take a more modern approach, but have not been designed as general purpose control systems. However, despite many 'good' features existing across the range of control systems, none comprehensively implement all of them. In all cases, analysis and acquiring data over complex parameter spaces are second class citizens, despite the fact that these elements are a key requirement for many of the current ultracold atom research directions. With all ultracold atom experiments using a control system, and much PhD, post-doc, and supervisor time being put towards developing single use control systems for particular labs, there is a strong need for a comprehensive, general purpose control system that is capable of meeting the demands of modern ultracold experiments.

Chapter 3

Apparatus

BEC research is one of the common fields that require a control system. This project was conducted during the development of the first two BEC apparatuses at Monash University. During this project I was primarily involved with the development and construction of the dual-species (K-Rb) BEC apparatus in my supervisor, Kris Helmerson's, lab. The other BEC apparatus (a spinor BEC machine) was constructed by other PhD students in my co-supervisors, Lincoln Turner and Russell Anderson's, lab. Many common designs and ideas were shared between the students and supervisors of the two labs. A large section of this project was building the dual-species (K-Rb) BEC apparatus. We developed the control system for use on both BEC apparatuses. This allowed us to test our control system on more than one real experiment apparatus, which provided useful feedback for generalising the control system to multiple scientific experiments and improving the capabilities of the control system.

At the start of my project, in 2011, our lab was almost empty. The vacuum system was partially constructed and no lasers or optics had been setup. The construction of the apparatus was, of course, developed by several students simultaneously; our major contributions were:

- The vacuum system and magnetic coils were designed and built by Brad Murnane (a fellow PhD candidate, 2010-2012).
- Chris Billington (a fellow PhD candidate, 2011-2018), assisted with construction and performed the bake-out of the vacuum system.
- The construction of the apparatus post-bake-out, including the design and construction of the optics on the laser and vacuum tables (excluding dipole trap optics) and the determination of the desired laser frequencies, were performed equally by Shaun Johnstone (a fellow PhD candidate, 2011-2018) and I, with later contributions from Dr. Mikhail Egorov (post-doctoral fellow, Dec. 2011 to Apr. 2014).
- The majority of the dipole optics designed and constructed by Shaun Johnstone, with contributions from Dr Mikhail Egorov.
- Shaun Johnstone, Mikhail Egorov, and I designed and optimised the experiment logic to produce BEC. This will be discussed further in §8.1.

- I designed and constructed the current oven controller and v1.0 coil interlock. I later assisted Sebastien Tempone-Wiltshire (a fellow PhD candidate, 2015-2019 (expected completion)) in designing the latest circuit board for the v1.1 coil interlock.
- I constructed the PulseBlaster boxes, based directly on a design by the Monash University Electronics and Imaging services.
- I modified the coil driver circuit design and constructed the coil drivers. Original driver circuit design based on work by Dr. Russell Anderson (Monash University) who based his design on work by Carl Sauer and Adam Kaufman at JILA. I later assisted Sebastien Tempone-Wiltshire in designing the latest circuit board for his second generation experiment.
- Shaun Johnstone and I designed and built NovatechDDS9m (rf Amplification) boxes based on an original design provided by the Monash University Electronics and Imaging services.
- Microwave laser offset lock box constructed solely by Shaun Johnstone, based on a design by Martijn Jasperse (a fellow PhD candidate, 2010-2015) - original design by J. Appel *et al.* [99].
- Tuneable microwave IQ upconverter constructed solely by Shaun Johnstone, based on a design from Alex Wood (a fellow PhD candidate, 2011-2015).

In this chapter, we outline the design and intended capabilities of the dual-species BEC apparatus and some of the work on the construction of the apparatus (the remaining information on the construction is contained in the thesis of Shaun Johnstone [100]). We will also cover the work I completed on the process control systems (see §1.1.1 for an introduction to process control systems) for the dual-species BEC apparatus, which are separate to the scientific control system we discuss in the later chapters of this thesis. These process control systems ensure safe and consistent operation of our apparatus, and are a complementary system to the labscript suite. We will thus cover four main sections in this chapter: the laser table (which produces the light required for interacting with the atoms in order to create and interact with a BEC), the vacuum table (which contains the vacuum system we use to produce and hold a BEC), the electronics that control the equipment on these tables (that operate between the scientific control system and the apparatus), and the process control systems. The use of this apparatus, in conjunction with the labscript suite, will be discussed in chapter 8.

3.1 The laser table

Production of a BEC requires precise control over the frequency and intensity of multiple laser beams. Our apparatus is designed to produce clouds of ultracold potassium-41 and rubidium-87 atoms. Both these atomic species have a complex hyperfine structure, requiring several different frequencies of light to successfully cool and image the atoms.

We use external cavity diode lasers (ECDLs) to create the necessary laser light as they produce a high quality Gaussian TEM_{00} beam profile with a linewidth narrower than the

natural linewidth of the atomic transitions [101, 102], which is ideal for manipulation by down-stream optics, and addressing the hyperfine structure of our atomic species. The lasers used on this table are based on a MOGLabs ECDL design and are controlled with MOGLabs 'MOGbox' DLC202 laser diode controllers (which include locking electronics). The lasers are frequency locked, relative to an atomic transition, via active stabilisation of the laser cavity [103]. To produce a sufficient quantity of light we also split off some of the laser light and amplify it using a tapered amplifiers (TAs). Finally, we split the amplified light into several 'beamlines' and use acousto-optic modulators (AOMs) under computer control to adjust the frequency and intensity of each laser beam as required. These beamlines terminate at fibre couples, which both spatially filters the light and transforms any pointing instability (introduced by components on the laser table) to an intensity fluctuation. The fibre optic cables remove optical aberrations introduced by the TA (and down stream optics on the laser table) and allows the light to be transported a significant distance. The layout of the laser table is shown in figure 3.1.

3.1.1 Rubidium

As the hyperfine splitting (see figure 3.2 and [104]) of the ground state of rubidium-87 is larger than the typical frequency shift achievable using an AOM, we use two lasers for trapping and cooling these atoms. We call these lasers the master (or trap) laser and the repump laser. They are used to address the $5^2S_{1/2}, F = 2 \rightarrow 5^2P_{3/2}, F'$ manifold transitions and the $5^2S_{1/2}, F = 1 \rightarrow 5^2P_{3/2}, F'$ manifold transitions respectively. This results in two sets of TAs and beamlines, which are shown in the right half of figure 3.1(a) for the rubidium trap laser and figure 3.1(b) for rubidium repump.

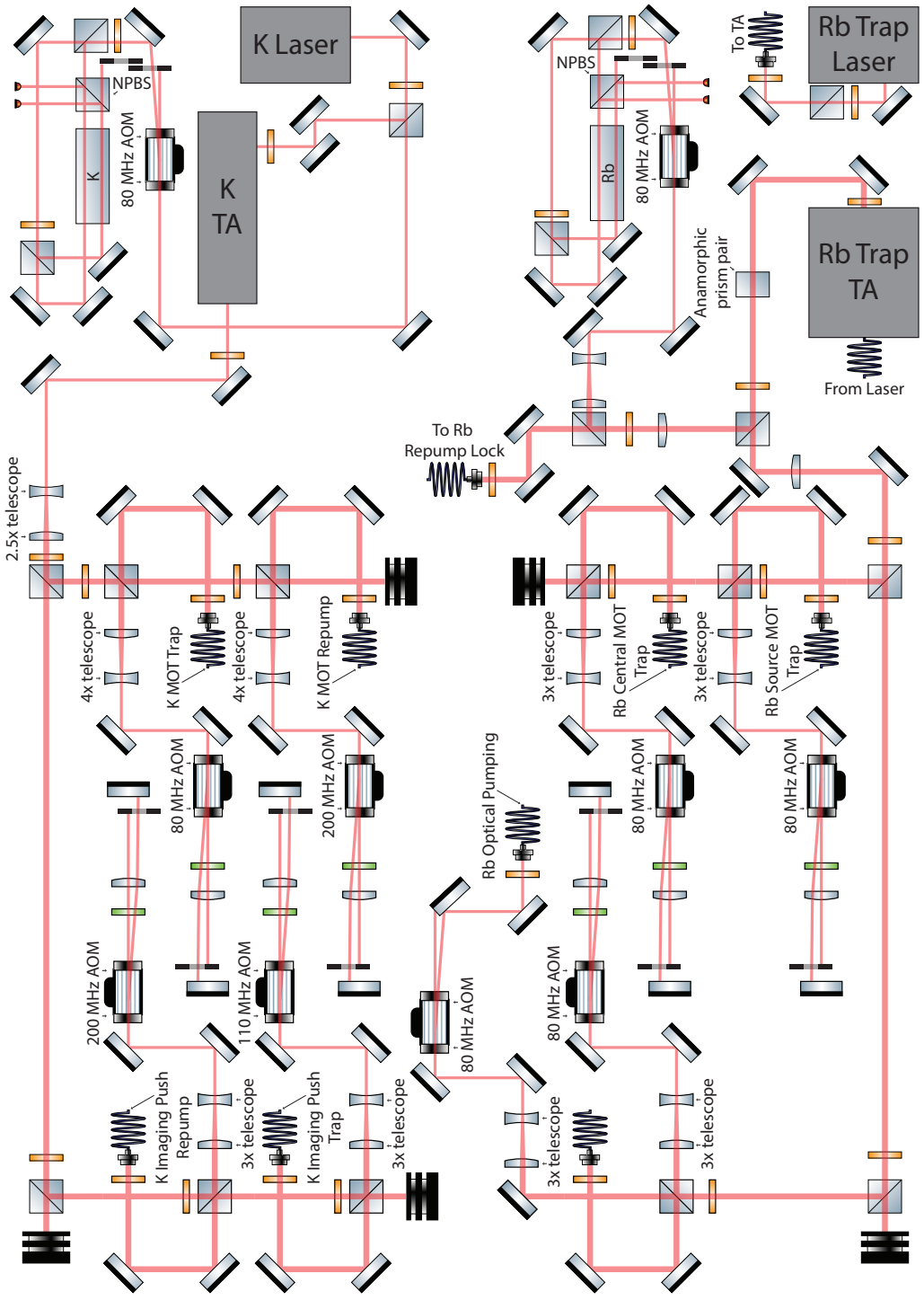
3.1.1.1 Laser lock

The master laser is locked, via saturated absorption spectroscopy [105], 47 MHz red detuned from the crossover peak produced between the $F = 2 \rightarrow F' = 2$ and $F = 2 \rightarrow F' = 3$ transitions. This detuning is created by a single-pass AOM, which blue shifts the light used in the saturated absorption lock. The master laser is thus locked 180 MHz red detuned from the cooling transition (the $F = 2 \rightarrow F' = 3$ transition). The optical layout for the rubidium master lock is shown in figure 3.1(a). We use a differential photodetector to remove the Doppler broadened background for the signal, leaving only peaks corresponding to direct transitions or to crossover transitions. An error signal (for locking) is generated by the MOGbox by modulating the current of the laser diode, and is configured to lock to the top of a selected peak.

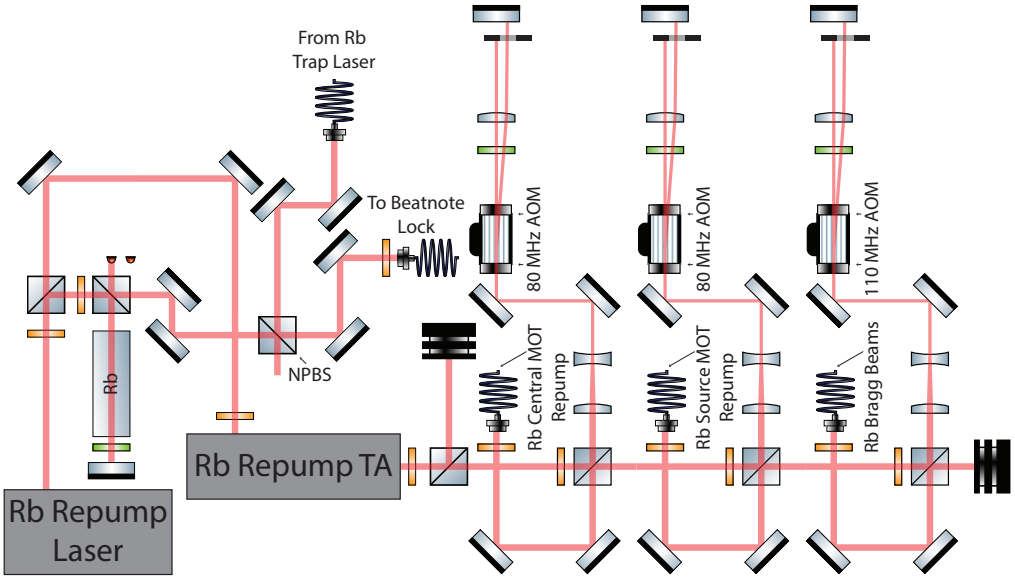
The repump laser is offset locked 6.590 GHz above the master laser via a microwave offset (beat-note) lock. Further details of this microwave offset lock can be found in the theses of Martijn Jasperse [106] and Shaun Johnstone [100]. This locks the repump laser light 160 MHz red detuned from the repump transition (the $F = 1 \rightarrow F' = 2$ transition).

Both the trap and repump lasers are thus approximately 160 MHz detuned from the required MOT frequencies[1]. These were chosen as it allows us to use 80 MHz AOMs in

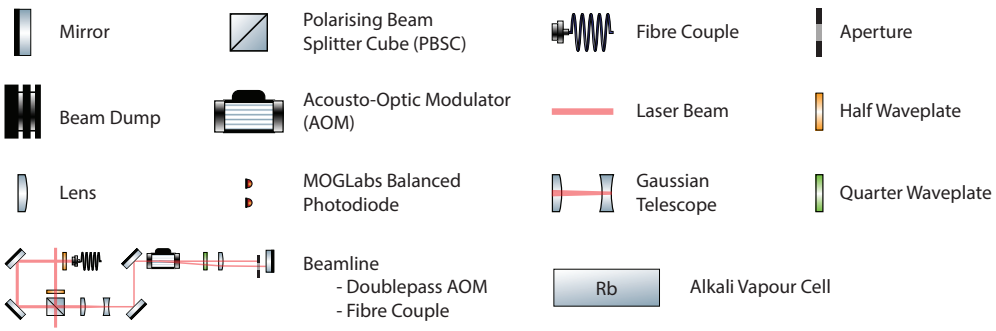
1. While the rubidium trap laser is detuned 180 MHz from the cooling transition, the MOT trap light should be detuned approximately 20 MHz from the cooling transition (see figure 3.2), thus making both repump and trap approximately 160 MHz detuned from the optimal MOT frequencies.



(a) The layout of the potassium and rubidium trap optics on the laser table. The laser systems each consist of a master laser, a vapour cell locking system, a TA and several fibre coupled beamlines. The beamlines typically consist of an AOM in a cat-eye double pass configuration and a fibre couple. Note that the cat-eye lenses have a focal length of 150 mm. For further details, see the main body text of §3.1. Note: The layout in this diagram matches the physical layout on the optics table, and while individual components are not to scale, the layout is approximately to scale.



(b) The layout of the rubidium repump laser and optics. For further details, see the main body text of §3.1.



(c) Optical components in parts 3.1(a) and 3.1(b).

Figure 3.1

a double-pass configuration to create the required laser frequencies, while also maintaining peak diffraction efficiency of the AOMs during the stage of the experiment that requires the most laser power.

3.1.1.2 Tapered amplifiers

ECDLs do not provide sufficient laser power to cool enough atoms to make a BEC. We use TAs to amplify the laser light from the ECDL by approximately a factor of 100. The rubidium master laser is amplified by a Thorlabs TPA780P20 [107], producing 1.7 W after an optical isolator. The beam from the Thorlabs TA is elliptical and divergent, so we use an anamorphic prism pair (with a ratio of 2:1) to reshape the beam followed by an additional lens to collimate it. The resultant beam is approximately a Gaussian TEM_{00} mode, however

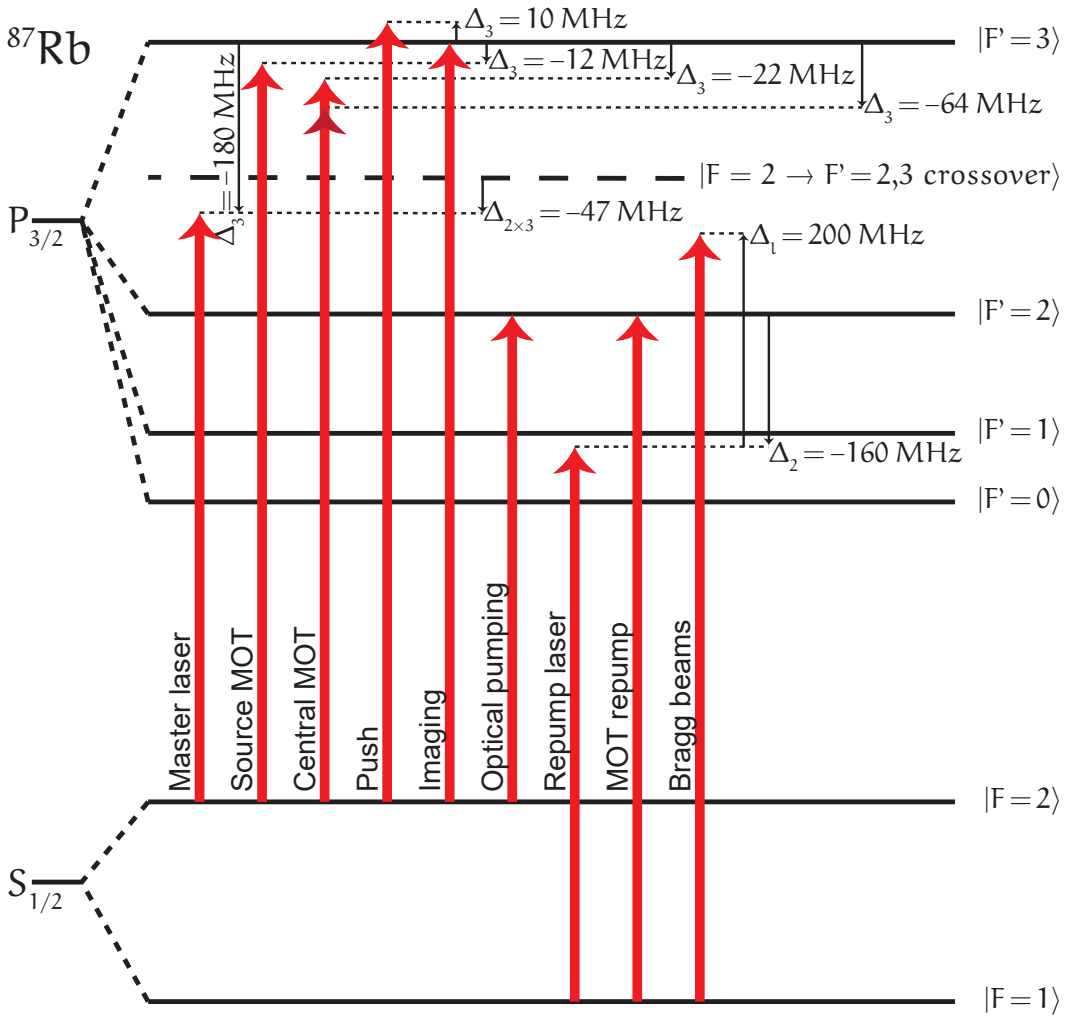


Figure 3.2: The rubidium laser frequencies used in various stages of the experiment, relative to the hyperfine structure of rubidium-87. Dark red arrow heads represent the change in laser frequency between the MOT and compressed MOT stages of the experiment. Figure adapted from Shaun Johnstone's thesis [100] with permission.

there is a long tail in the vertical dimension. We use an aperture to filter this out, which results in 1 W of available power in a good quality beam[2].

Typically, the amount of repump light required is less than the output of an ECDL. However, after passing the light through AOMs and fibres, the available power can be borderline, especially if the diode in the laser is getting old. We thus amplify our repump light using a spare Sacher TA, which was partially damaged by humidity and temperature fluctuations from repeated air-conditioning failures[3]. Despite the damage, the output power from the TA is higher than we could ever need for repumping.

2. The optical components used were chosen based on their availability (we had them in the lab) rather than their suitability to the task. We expect that with more carefully chosen parts, a higher power in a TEM_{00} mode could be achieved if it was required.
3. The air-conditioning failure were the result of moving our apparatus during 2013 to a new facility, prior to the new facility being sealed to the outside and internal services commissioned.

3.1.1.3 Beamlines

We typically split the output of the TAs into beamlines that consist of a double pass AOM and a fibre couple (see figure 3.1). The AOM provides independent control over the frequency and intensity of the light incident on the fibre. We double pass the light through the AOM, following the cat-eye configuration detailed in [108, 109, 110]. This configuration keeps the fibre couple aligned when the frequency shift of the AOM is changed (which would normally result in beam displacement at the fibre face).

We have such beamlines for:

- source MOT trap light,
- central MOT trap light,
- source MOT repump light,
- central MOT repump light, and
- imaging/push beam light (the output of this fibre is split on the vacuum table, see §3.2.4).

The source and central MOT trap beamlines are blue shifted by 168 MHz and 158 MHz respectively, placing them 12 MHz and 22 MHz red detuned from the cooling transition (the $F = 2 \rightarrow F' = 3$ transition). This frequency shift was experimentally chosen by optimising for maximum atom number in the MOTs and corresponds to approximately a 2–4 Γ detuning (where Γ is the natural linewidth of the transition). The source and central MOT repump beamlines are blue shifted by 160 MHz, which is resonant with the repump transition.

While we have separate laser beams on the vacuum table (see §3.2) for imaging light and the push beam used to load the central MOT from the source MOT, these two beams require similar laser frequencies and laser powers, and are used at distinctly separate times in the experimental sequence. We are thus able to derive these two beams from a single beamline. The imaging/push beamline is thus initially set to be 10 MHz blue detuned from the cooling transition, by using a 80 MHz AOM (run at 95 MHz) in a double-pass configuration. As with previous laser frequencies, this was experimentally determined to be the optimal frequency, but it also falls within the expected range observed from simulations that Chris Billington had previously run. During an experiment, the frequency is then adjusted for imaging, to be on resonance with the cooling transition by adjusting the AOM frequency to 90 MHz.

We also have a single-pass AOM coupled into a fibre for optical pumping. This AOM is run at 86 MHz and is thus on resonance with the $F = 2 \rightarrow F' = 2$ transition, and is used to ensure that our atoms are in the $|F = 2, m_F = 2\rangle$ state prior to the magnetic trapping stage.

Our AOM double-pass efficiencies fall in the range of 50-75% and fibre couple efficiencies fall in the 40-60% range (both subject to alignment and beam quality). This gives an overall beamline efficiency in the range of 20-45%. The fibres are all routed to the vacuum table (see §3.2).

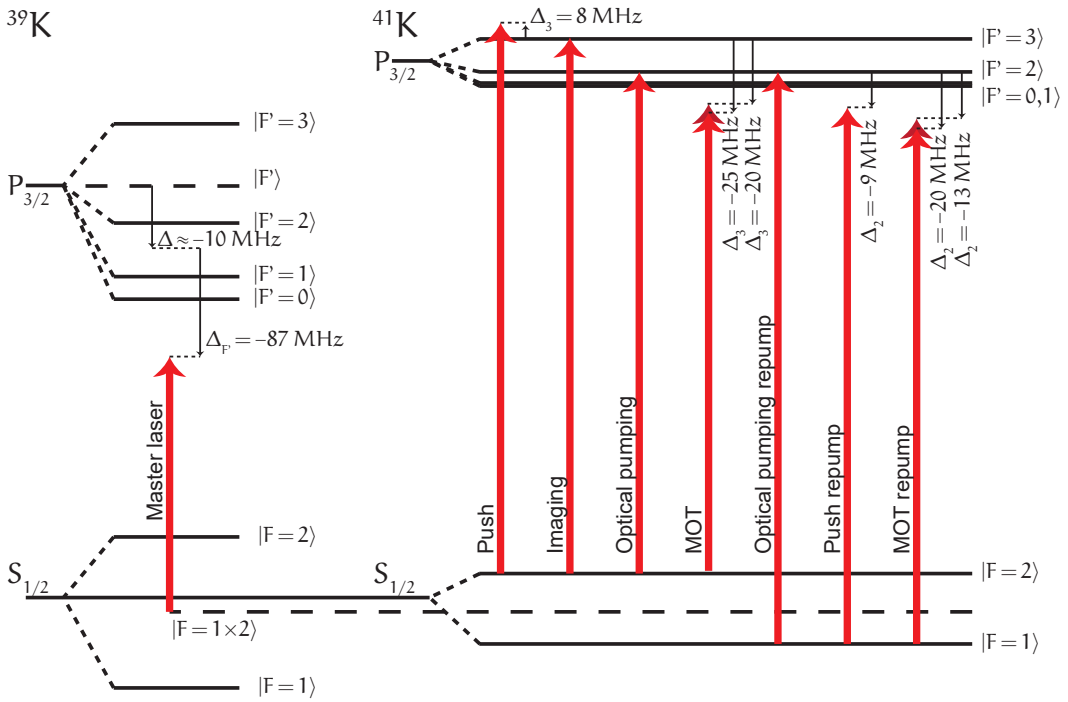


Figure 3.3: The potassium laser frequencies used in various stages of the experiment, relative to the hyperfine structures of potassium-39 and potassium-41. Dark red arrow heads represent the change in laser frequency between the MOT and compressed MOT stages of the experiment. Reproduced from Shaun Johnstone's thesis [100] with permission.

3.1.2 Potassium

The hyperfine splitting of the ground state of potassium-41 is small (254 MHz, see figure 3.3 and [111]) compared to other alkalis like rubidium-87 (6.8 GHz [104]). We can thus address the $4^2S_{1/2}, F = 2 \rightarrow 4^2P_{3/2}$ manifold transitions and the $4^2S_{1/2}, F = 1 \rightarrow 4^2P_{3/2}$ manifold transitions from a single laser source, using only AOMs to control the frequency and amplitude.

3.1.2.1 Laser lock

Unlike rubidium-87, we are unable to resolve the hyperfine splitting of the excited state of the potassium-41 D2 line via saturated absorption spectroscopy. This is primarily due to the low natural abundance of potassium-41 in our vapour cell, but also partially due to the smaller excited state splitting that is on the order of the natural linewidth of the transitions. The only well defined feature typically visible is the crossover between the two hyperfine ground states of the potassium-39 D2 line. However, as the potassium-39 D2 line is only 236.2 MHz red detuned from the potassium-41 D2 line, this provides an acceptable locking point from which potassium-41 frequencies can be reached using AOMs.

The expected frequency of this saturated absorption feature is difficult to determine, due to the small excited state splitting, which ultimately causes many saturated absorption spectroscopy features to blur into a single feature. We experimentally determined (by finding the resonant imaging frequency of the potassium-41 atoms that we later trapped and cooled)

that the peak of this feature corresponds only approximately to the crossover between the two hyperfine ground states of potassium-39 and the $4^2P_{3/2}$ manifold. There is a 10 MHz shift of unknown origin present in our laser frequency calculations (see figure 3.3) which may be simply due to an electronic offset in the laser lock error signal that we did not remove correctly. Alternatively, the shift could be due to the shape of the saturated absorption feature, which is affected by the relevant transitions strengths of the allowed transitions, the velocity distribution of the atoms (which affects the strength of the crossover transition features that contribute to the overall feature we see)[4], or a combination of these along with the electronic offset in the error signal previously described. However, the difficulty in identifying the lock point of the laser does not pose a problem to the operation of the apparatus. It did however make optimising the apparatus more complicated as we could not be certain of our laser frequencies until we had successfully cooled our atoms to the point where we could perform absorption imaging. Other groups pursuing potassium experiments should thus be aware that laser frequencies may be 10s of MHz away from the expected the lock point.

As with the rubidium lock (see §3.1.1.1), we use a single-pass AOM to frequency shift the light in the laser lock. The single-pass AOM is configured to provide a 87 MHz blue shift for the locking light, which results in a red shift of 87 MHz of the light entering the TA.

3.1.2.2 Tapered amplifier

As we only have a single ECDL for interacting with potassium-41, we only need one TA. We use a Sacher TA, which produces a maximum of 1.4 W of light after the optical isolator Unfortunately, there appears to be dust (or another defect) on the TA, which results in a poor spatial mode from the TA. We deliberately misalign the input beam to the TA to produce a slightly better mode, at the expense of output power (1.3 W). However, the loss in power is mitigated by higher fibre coupling efficiencies in the beamlines from the better spatial mode.

3.1.2.3 Beamlines

We use a similar configuration of beamlines as for rubidium (see §3.1.1.3). The same double-pass AOM coupled into a fibre configuration is used in four places, for the following beamlines:

- MOT trap light,
- MOT repump light,
- imaging/push beam trap light, and
- imaging/push beam repump light.

4. We use a Thorlabs vapour cell heater [112] to increase the vapour pressure inside the cell in order to increase the signal-to-noise ratio of the saturate absorption signal. Given that this changes the velocity distribution of the atoms, it is possible that this may affect the shape of the feature and thus the lock point. It may also contribute to instability in the lock point if the temperature is not well regulated.

Potassium poses a more difficult challenge to cool successfully, due to the narrow spacing between hyperfine transitions in the excited state of the D2 manifold compared to the natural linewidth of the transition. This makes it impossible to address a single hyperfine transition, complicating the repumping system and making it difficult to perform sub-Doppler cooling. We follow the path travelled by previous groups, by red detuning our MOT beams with respect to the entire excited state manifold [113]. This necessitates a more even ratio of trap and repump light (when compared to rubidium). We found the optimum ratio was 3:2, which agrees with M. Prevedelli *et al.* [114]. Sub-Doppler cooling of potassium was not achieved until 2011 by M. Landini *et al.* [115], which was during the construction of our apparatus. We attempted to reproduce their results, but were unsuccessful. We suspect this is more to do with our inability to produce a simultaneously well-aligned central MOT for both rubidium and potassium, rather than any issue with the technique.

As well as the MOT beams, we produce two beams (containing repump and trap frequencies respectively) for pushing atoms between the source and central MOTs. These beams are also later used for absorption imaging, which again requires both trap and repump frequencies due to the inability to interact with a singular hyperfine transition.

As with the rubidium beamlines, all the fibre outputs are located on the vacuum table (see §3.2).

3.2 The vacuum table

The vacuum table hosts the vacuum system, which holds the sources of rubidium and potassium atoms used in the experiments (under high and ultra-high vacuum (UHV)) and is used in the process of cooling and experimenting on the atoms. The vacuum table also holds the optical components required to process the laser light required during the experiments (mainly from the fibres fed to the vacuum table from the laser table).

3.2.1 Vacuum system

The vacuum system was custom designed for our lab by fellow PhD candidate Brad Murnane, and is shown in figure 3.4. It consists of two source segments on the outer edges, which feed the respective alkali metals into a glass cell where they are initially cooled in the source MOTs. These atoms are then pushed (by a laser beam) from their respective source chambers, into the central chamber of the apparatus where they are captured in a dual-species MOT and then cooled further using standard techniques such as PGC and forced evaporative cooling in both a hybrid magnetic-optical trap and a pure cross-dipole trap. Attached to the central chamber are two high quality glass cells (a cylindrical cell and a square prism cell), which were chosen in combination with a custom objective lens to provide high resolution imaging of our cold atom clouds during scientific experiments. However, we were unable to successfully transport the cold atoms from the central chamber to the square cell reliably, likely due to stray magnetic fields caused by vacuum system bolts that became magnetised after construction. Due to this, we instead study the ultracold atoms in the central chamber, using a lower resolution imaging system. Further details of the work we performed on this transport, and the details of our imaging systems, can be found in Shaun Johnstone's thesis [100].

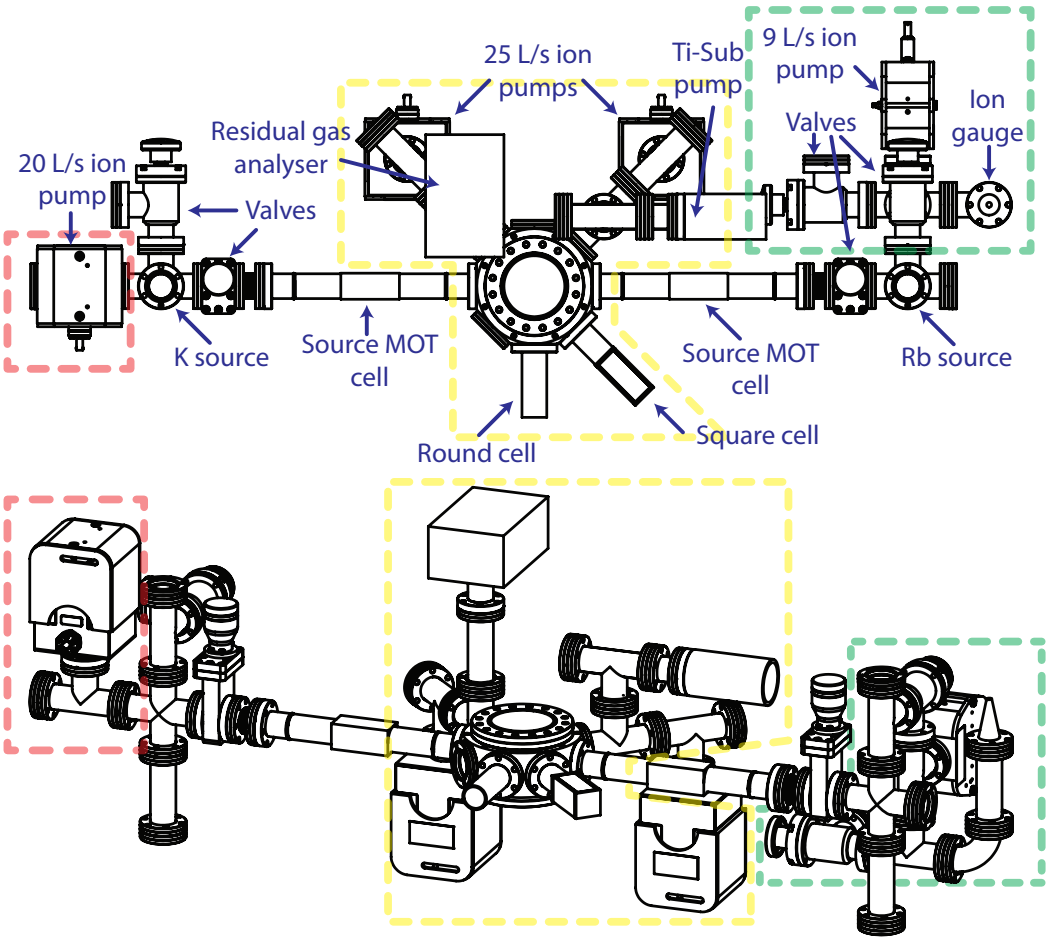


Figure 3.4: Schematic of our dual-species BEC apparatus. We have a source chamber at each end of the vacuum system that contains an ampule of alkali metal. One contains rubidium while the other contains potassium. Both are heated to produce a background vapour in the respective chambers. We load a source MOT for each species from this background vapour, and then push these atoms into a combined central chamber where they are captured in a dual-species MOT before further cooling. Ideally, the ultra-cold atomic cloud would be transported to one of two science cells (with a square or round profile), however this has yet to be realised reliably and we instead perform experiments in the central chamber. The source chambers are separated from the central chamber (yellow section) by differential pumping tubes, which maintain a pressure difference of at least 2 orders of magnitude, ensuring the central chamber is maintained at UHV. The vacuum is primarily maintained by two ion pumps connected to the central chamber, however we also added extra pumps to the source chambers to improve the background pressure (see main body text) which are highlighted by the red and green outlines. This figure was reproduced from Shaun Johnstone's thesis [100] with permission.

The vacuum inside the system is maintained via several ion pumps. The source ovens and source MOT cells are separated from the central chamber by differential pumping tubes which maintain a pressure difference between the central chamber are the source cells of 2–3 orders of magnitude. The entire system was also baked at approximately 180° C to reduce the outgassing rate of the metal components, and remove any residue that was not removed by our manual cleaning process. The central chamber is pumped by two 25 L/s ion

pumps [116] and a layer of titanium deposited via a titanium sublimation pump [117], which was estimated to provide sufficient pumping. Due to the (last minute) addition of an angle valve with a Viton seal, and a vacuum failure in 2016, we have had to add additional ion pumps to the source sections of the apparatus that were otherwise only pumped through the differential pumping tube. Unfortunately, the design of the vacuum system did not include sufficient ion gauges to adequately measure the pressure in the various sections of the vacuum system. We did attempt to add an ion gauge at the time we added an ion pump to the rubidium source chamber, however this ultimately failed due to an electrical fault of the gauge (possibly shorting to the vacuum system). As such, we are unsure of the exact pressure inside the vacuum system. Ultimately, the pressure determines the lifetime of the atoms within a trap, and we experimentally found this to be 6.1 ± 0.5 s with no ion pumps on the source chambers, 51 ± 4 s in 2012 (after adding the additional ion pump to the rubidium source chamber) and 24 s after the 2016 vacuum failure [100] (where we broke vacuum due to a cracked window, added an additional ion pump to the potassium source chamber, and did not rebake the system). This was acceptable for our proposed experiments, although not ideal.

We have three sets of magnetic coils around the vacuum system. The potassium source MOT has four rectangular coils, one parallel with each rectangular face of the glass cell, forming the field for a 2D MOT. The rubidium source MOT has two round coils, parallel to the top and bottom faces of the glass cell, forming the quadrupole field for a 3D MOT. The central MOT, again, has two round coils forming the quadrupole field, which are inset into the recessed top and bottom windows of the central octagon. We also have a set of bias coils attached to the central quadrupole coils in order to shift the vertical location of the magnetic trap. While not part of the original design, we also added 2 further sets of coils made from ribbon cable to produce bias fields in the remaining 2 dimensions.

After completing, and using this apparatus over the past 7 years, we discovered a significant number of difficulties with the original vacuum system design, including:

- the small surface area that could be covered by the titanium sublimation pump in the central chamber (the inside of a single 2.75 inch 'T' piece), which limited the effectiveness of this pump,
- the use of large amounts of metal in the construction of the central chamber (and not coated by the titanium sublimation pump) which significantly contributes to the background pressure of the system,
- the use of large amounts of metal in the construction of the source ovens, without the addition of any dedicated pumping, which significantly contributes to the background pressure of the system,
- the use of valves with Viton seals (including two gate valves between the ovens and source MOT cells) which outgas considerably,
- the lack of accurate pressure monitoring,
- the medium sized ion pumps, which do not seem to produce sufficient pumping for the design but produce annoying magnetic field gradients,

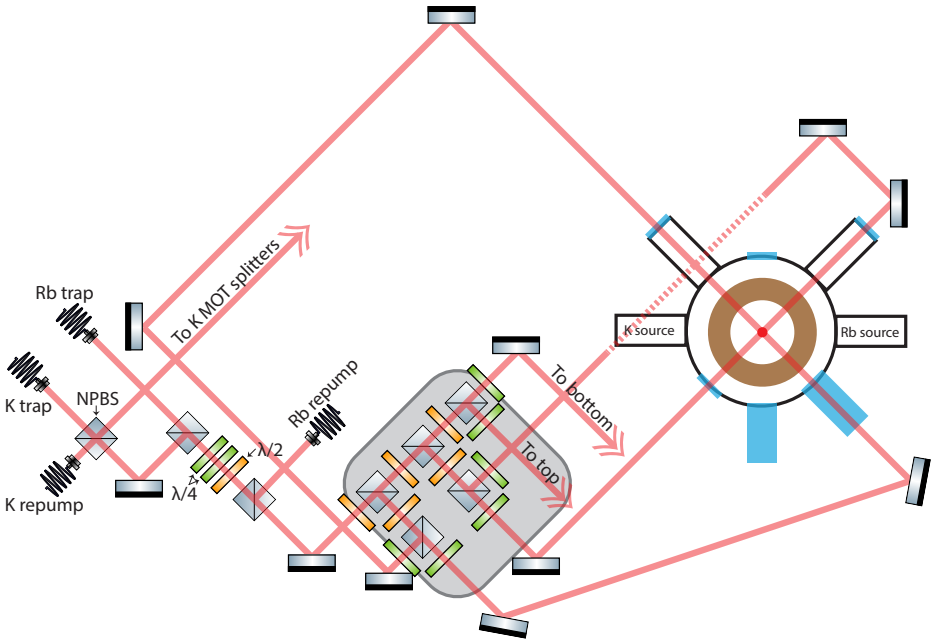


Figure 3.5: The optical layout for combining all rubidium and potassium MOT lasers for the central MOT, followed by the optics used to split the combined beam into the required number of MOT beams. See the main body text (§3.2.2) for further details. Reproduced from Shaun Johnstone's thesis [100] with permission.

- the lack of well designed, 3-axis bias coils, capable of producing a significant bias field, for the central chamber,
- the lack of optical access to the science cells and central chamber, and,
- the use of bolts that were not specifically nonmagnetic when constructing the vacuum system.

The lessons Shaun and I learnt from this apparatus were passed onto fellow PhD student Sebastien Tempone-Wiltshire, who has built a second-generation apparatus, which will be described in his upcoming thesis.

3.2.2 MOT optics

As discussed in §3.1, we have coupled light into several fibre optic cables, each with independent frequency and intensity control via an AOM, which terminate on the vacuum table. We then combine and split these beams to produce the required MOT beams (which are always a combination of at least two separate frequencies of light).

The rubidium source MOT optics are the simplest of the three MOTs. Here we combine the 'rubidium source MOT trap' and 'rubidium source MOT repump' fibres on a polarising beam splitter cube (PBSC) into a single beam (which requires that the two components have opposite polarisations). This single beam is then split into three beams of equal power through the use of two PBSCs, each preceded by a half waveplate that is used to control the ratio of light split by the cube. Each of the three beams is then split again using PBSCs (one per beam) to produce three pairs of beams containing the MOT light. The

half waveplate prior to these last three cubes is used to balance the power within each beam pair such that the MOT successfully catches atoms[5]. Each of the (now six) beams passes through a quarter waveplate (used to set the appropriate beam polarisation) and telescope that magnifies the beam diameter by a factor of 10 [118]. The six beams are then routed to the source MOT cell such that each pair is counter-propagating along one of the three orthogonal spatial axes, and such that they all intersect in the centre of the cell. Combined with a anti-Helmholtz coil pair also centred on the cell, this forms a 3D MOT capable of catching atoms from the background vapour. It is worth noting that the repump light will split in the opposite ratio to the trap light at the first of the five splitting cubes due to the opposite polarisations of the two beams resulting from the initial combination optics. We set the splitting ratios correctly for the trap light only as this forms the majority of the power in the beams and is the major component in the cooling process. This is not a significant issue as we still obtain sufficient repump light in each pair of beams and the repump light matches the polarisation of the trap light after the first cube, ensuring the repump light is also balanced within each pair of counter-propagating beams.

While we were successful in creating a 3D rubidium source MOT, we had significant trouble with the potassium source MOT due to the inability to distinguish potassium 3D MOT fluorescence from the background vapour. As such, we opted to switch to a 2D MOT for potassium (which is usually considered the better option anyway) so that we could image along the long axis of the MOT in order to observe a stronger integrated MOT fluorescence on a camera. The potassium 2D source MOT optics are configured in an almost identical way to the rubidium source MOT, but simply produce four beams instead of six. Ideally these beams would be elliptical in order to fill a greater volume of the source chamber, but unfortunately we did not have the optical components to do that at the time. The main difference in MOT optics comes from the way we initially combine the potassium trap and repump light. Due to the requirement of needing close to equal power between the repump and trap light for a potassium MOT (see §3.1.2.3) we combine the repump and trap beams on a non-polarising beam splitter (NPBS) cube that splits each incident beam approximately 50:50. We thus produce two beams that contain both potassium trap and repump light for the MOT. One of these is used for the 2D source MOT and the other is fed to the central MOT (see figure 3.5). This unfortunately means that we don't maintain independent frequency and amplitude control of the MOT light between the potassium source and central MOTs.

The 3D central MOT is designed to hold both species of atoms simultaneously. This requires that all four frequencies of light are present in the MOT beams. The optical system that achieves this is shown in figure 3.5 and largely follows the rubidium source MOT layout. We combine the dual frequency potassium MOT beam (introduced in the previous paragraph) with the rubidium light on a PBSC, which results in orthogonal polarisations for the potassium and rubidium light. As this will split unevenly on subsequent

5. While it is expected that you would have equal powers in each beam, reflection losses, perturbations in the beam overlap and mismatches in beam diameter (either from the miscollimation of the independent telescopes or from uneven propagation distances between the beams in the counter-propagating pair) ultimately mean that the power balance in the pairs of beams is slightly unequal and must be determined experimentally.

PBSC, we construct a dichroic half waveplate[6] from 2 multi-order quarter waveplates and a multi-order half waveplate giving a retardance at 780 nm of 0.92λ and a retardance of 1.54λ at 767 nm[7]. This allows us to independently rotate the polarisation of the potassium beam to match that of the rubidium beam. We then inject repump light into this beam using another PBSC, which again results in uneven splitting of the rubidium repump light amongst the MOT beams. However, as with the rubidium source MOT, this does not pose a problem. The cube used to mix in rubidium repump light also cleans up the polarisation of the combined rubidium–potassium frequencies, which may not be purely linear due to the imperfect dichroic waveplate used.

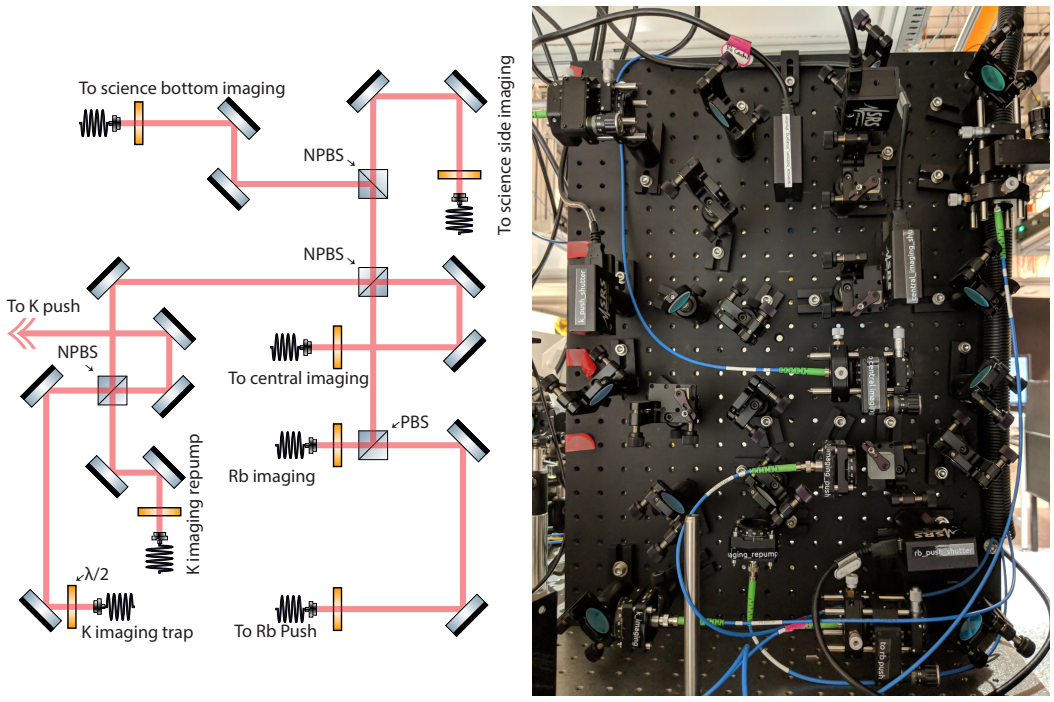
While the central MOT is capable of trapping both potassium and rubidium atoms pushed from the respective source MOTs, the performance of the potassium MOT is typically poor. It is unclear whether this is due to the difficulty in overlapping the potassium and rubidium beams or the fact that the zero-order waveplates produce slightly different polarisations between potassium and rubidium (even though they perform much more uniformly than multi-order waveplates). However, it is likely that the issue could be compensated by providing independent control over the final polarisation of each MOT beam. As such, we recommend future experiments utilise only dichroic waveplates in the dual-species MOT optics, specifically manufactured for the wavelengths in use, so that the experiment operators have fine control over the MOTs.

The lack of independent control over potassium MOT frequencies was another of the caveats we attempted to correct when helping with the design of the second generation experiment currently being developed by Sebastien Tempone-Wiltshire. In his experiment, the use of a retro-reflected 2D source MOT as part of the initial design allows him to use a NPBS cube to produce the two required source MOT beams. The 3D central MOT beams can also be produced in a similar way, although he is ultimately required to throw out a quarter of the power used in the central MOT (or have one pair of beams have twice the power of the others) and the process of combining rubidium and potassium light for the central MOT requires the use of dichroic waveplates. This is likely a small price to pay for independent control of the potassium MOTs. Further information on this design will be given in the upcoming thesis of Sebastien Tempone-Wiltshire.

3.2.3 Optical pumping optics

The optical pumping light is fed into the central MOT through the round science cell on the vacuum system. As this is also the axes along which we image the atoms (due to a lack of optical access along other axes of the system) we use a 'D'-shaped mirror to place the rubidium optical pumping light at a small angle to the imaging light. The rubidium optical pumping light contains both the optical pumping light from the single pass AOM beamline previously discussed, and a small amount of rubidium MOT repump light that we split off from the rubidium central MOT repump fibre output and overlap with the rubidium optical pumping light using a NPBS cube. As the potassium optical pumping frequencies can be

6. A dichroic waveplate is a waveplate that acts as a $\frac{\lambda}{x}$ waveplate for one wavelength of light (where x is typically 2 or 4) and a λ waveplate for another wavelength of light. In this case, we have constructed an approximate half waveplate for potassium that does not cause a net change of the rubidium polarisation.
7. These are calculated based on the specifications provided by Thorlabs.



(a) (b)

Figure 3.6: (a) The optical layout of the optics used to combine and split the imaging and push lasers for use as push beams and imaging probe beams. We produce three separate imaging probe beams, plus the push beams for rubidium and potassium from three input fibres (generated by the potassium imaging trap and repump, and rubidium imaging beamlines detailed in §3.1). All but one of the combined beams is coupled into fibres for transportation to other sections of the apparatus. See the main body text for further details (§3.2.4). (b) A picture of the optical layout in the lab, which matches the layout shown in (a).

reached using the imaging/push beamlines, we use the potassium central MOT imaging light for optically pumping potassium, which we discuss in the next section.

3.2.4 Imaging optics

We previously introduced the beamlines for the imaging/push light for rubidium and potassium. This consisted of two potassium beamlines and one rubidium beamline. As with the dual-species central MOT, imaging a dual species cloud of atoms[8] requires that we have multiple frequencies of light in a single beam. We use an array of NPBS cubes (see figure 3.6) to produce three imaging beams that contain all the imaging frequencies (except for rubidium repump, which is typically unnecessary due to the atomic state we work with[9]). These imaging beams are coupled into fibres, which transport the light to the relevant section of the vacuum system. Currently we only use two of the imaging beams (central

8. It is worth mentioning here that we cannot simultaneously image both species of atoms. We instead image across two (otherwise identical) shots where we image one species in the first shot and the second species in the second shot. This still requires that the imaging beam contain dual species frequencies in order to ensure we image the two species in an identical way across shots.
9. When it is necessary to repump rubidium atoms prior to imaging, we use the MOT repump light, which is sufficient.

imaging and science bottom imaging) as we are not using either of the dedicated science chambers. We also split off light for the push beams during the combination stage.

For information on the optics used to image the atoms onto a camera, see Shaun Johnstone's thesis [100].

3.2.5 Dipole trap

We use an crossed-beam optical dipole trap during the last stages of the cooling process. This trap is formed using a 20 W, 1064 nm fibre laser (Keopsys CYFL-MEGA-20-LP-1064-AM0-ST0-OM1-B208-C4). The laser beam is split into two and each beam is single passed through an AOM, providing amplitude control of the beams[10]. The two beams are then focussed to a $(1/e^2)$ waist of approximately 70 µm and overlapped with the MOT beams in the horizontal plane using dichroic mirrors. For further details of the dipole trap, and the other configurations we tried during the development of this apparatus, see Shaun Johnstone's thesis [100].

3.3 Lab process control

As alluded to in chapter 1, our control system, the labscript suite, is not meant to replace an entire process control system. As a result, we also employ several separate systems that manage the critical components of the apparatus that run continuously. These systems are what one might consider traditional process control systems as described in §1.1, in that they regulate the apparatus regardless of whether scientific experiments are being performed. Most of the process control systems are implemented as interlocks, such that they shut down malfunctioning systems and place the apparatus in a safe state. As these interlocks are critical safety systems, they are kept separate from the labscript suite framework. The interlocks do not directly interface with the control and acquisition hardware that we use to perform scientific experiment and while programs in the labscript suite may have read access to the state of the interlocks (and data about the equipment the interlocks are monitoring), they never issue commands to the interlocks.

3.3.1 Oven controller

The oven controller regulates the temperature of the vacuum system around each source of alkali metal, and shuts down the ovens if the temperature of the oven or pressure of the vacuum system goes outside set bounds. This ensures no damage to our apparatus occurs if a component of the oven fails. We believe such control is vital, as we have witnessed colleagues at other institutions suffer critical failures of power supplies that resulted in their ovens being run at excessively high temperatures for hours before anyone noticed. Such failures typically require the vacuum system to be partially disassembled to repair the damage, costing months of work. Our own vacuum system developed a spontaneous leak in 2016, which tripped the interlock and shut down the ovens. We were able to replace the damaged window under a nitrogen environment and recover the system without needing

10. One beam is blue shifted while the other is red shifted. This ensures we do not accidentally create an optical lattice in the region where they cross.

to rebake the vacuum system, which was partially due to the expeditious shutdown of the ovens.

The heart of our oven controller is a Galil RIO-47100 PLC [119], providing 8 analog and 16 digital I/O, 2 independent PID loops, an Ethernet interface and multi-threaded programming (up to 4 threads). As we have two ovens (one per atomic species) we dedicate a thread of the Galil to monitor each oven and to start or stop a PID loop. Each PID loop monitors the temperature of the oven via an analog input. The PID then controls the current through heater tape via an analog output connected to the external control interface of a Manson HCS-3402 power supply. The heater tape was made from nichrome resistance wire (13.77 Ω/m) contained within a fibreglass sheath. Independently of the PID loop, each oven thread is programmed to periodically record the temperature of the oven and shut down both PID loops if either oven is out of bounds. One of the two central chamber ion pumps outputs the current pressure as an analog voltage, which is connected to an analog input of the Galil so that the Galil can shut down the PID loops if the pressure becomes too high.

Messages containing temperature and pressure measurements are logged approximately every 30 s over the Ethernet interface. These messages are parsed by a Python program (written by Martijn Jasperse, see [106]), running on a Linux logging server, and sent to the syslog server on the same machine. The syslog server is configured to send emails and/or SMSs to lab members if the syslog message has an severity of `WARNING` or higher. The data is also stored in a HDF5 file, which is processed by a script on the server that generates plots visible via a web interface.

3.3.2 Coil interlock

The coil interlock handles the cooling of the quadrupole coils around the central chamber of our vacuum system. The coil wire has a square cross-section, and the coil is wound such that there are multiple turns of the wire that have no faces exposed to the air. Without extra cooling, the coils would heat up significantly during the magnetic trapping stage where we run 140A at around 8V through them for several seconds. The coils were designed with this problem in mind, and as such, the wire is hollow allowing for chilled water to be pumped through each turn of wire. The coil interlock is responsible for monitoring the temperature of the coils and the flow rate of the water through them, and shutting down the high current power supply if the cooling is inadequate.

The coil interlock uses the same Galil PLC as the oven controller. The Galil is programmed as a simple state machine, transitioning between the `OFF`, `RUNNING`, `COOLING` and `SHUTDOWN` states. The `COOLING` and `SHUTDOWN` states are transitory. The `COOLING` state disables the PSU but continues to run water through the coils for 30 seconds before transitioning to the `SHUTDOWN` state. The `SHUTDOWN` state disables both the PSU and water cooling, and waits three seconds to ensure they have correctly disengaged (and sends a message with syslog severity 'ALERT' if the system has failed to disengage) before transitioning to the `OFF` state. The state machine can be transitioned through these states by buttons on the front panel of the box that trigger digital inputs of the Galil. The Galil code for the coil interlock is presented in appendix B.

Two flow sensors monitor the flow rate of water exiting each coil, to ensure water is actu-

ally flowing at an acceptable rate through them. The flow sensors are RS Pro Radial Flow Turbine Flow Meters (RS Stock No.: 257-149) which output a pulsed signal corresponding to the flow rate. This signal is converted to an analog voltage via a circuit designed by Martijn Jasperse. Two solenoid valves are connected to the feed and return line of the buildings processed cooled water supply, and are controlled by the Galil via an digital output and a relay. The temperature of the coils is also monitored using a thermistor. The Galil can transition itself from `RUNNING` to `COOLING` if the temperature exceeds specified bounds or from `RUNNING` to `SHUTDOWN` if the flow rate is out of bounds (indicating a malfunction of the supply or a leak in the lab). Two override switches are also present on the front panel, allowing the solenoids to be manually opened and the high current power supply to be engaged, bypassing the state-machine[11]. Bi-colour red-green LEDs on the front panel provide a real-time indication of the state of each flow and temperature sensor. These features provide useful real-time debugging information when diagnosing a failure of the water cooling or interlock system. The status of each sensor and override switch is also logged in the same manner as the oven controller (see §3.3.1).

As the Galil is a general purpose PLC, we have designed a circuit board to act as an intermediary between the Galil and the LEDs, relays, switches, and sensors. The schematic and layout for this circuit board is also included in appendix B.

3.4 Electronics

There is often a need for a significant amount of electronics between the I/O devices that the control system manages, and the apparatus itself. In this section we detail those electronics, which in many cases, are custom designs that we have developed.

3.4.1 Radiofrequency amplification

As detailed in §3.1 and §3.2.5, our apparatus relies on many AOMs to control the frequency and amplitude of our laser beams. Each AOM requires a computer controllable rf source of sufficient power to drive the AOM (for most AOMs, this is approximately 2 W). Our primary rf sources are generated by a custom designed piece of hardware that is based on off-the-shelf components. We call this device the 'Supernova'.

The design is based on an initial version developed by the Monash Electrical and Imaging services workshop (who were previously part of our department). The rf is produced by a Novatech DDS9m board [120], which is programmed by our control system. This board can produce four independent rf sources (although only two support updating during a hardware timed experiment). Each rf output is fed into a Mini-Circuits ZX80-DR230+ rf switch, which allows us to turn the rf source on/off in 2 µs. This is faster than the update rate of the 2 hardware timed channels on the Novatech board and is especially useful for the 2 software timed channels that can't be updated at all (which allows us to make use of them during a hardware timed experiment). Each rf switch has two outputs, one is fed to the 'test' BNC output on the front panel for debugging and the other is fed to a 2 W rf

11. In this situation, a 'WARNING" message is logged approximately every 15 minutes, triggering email and SMS alerts to lab users, to remind all lab users that the interlock has been bypassed.

Figure 3.7: The front panel design of two Supernova boxes.

amplifier (Delta RF Technology LA2-1-525-30). Each amplifier output is then routed to a BNC connector on the front panel. An example of the front panel is shown in figure 3.7.

We developed a custom circuit (see appendix C) for controlling the rf switches and amplifiers. The power for the amplifiers can be turned on/off via push buttons on the front panel. The power is also shut off automatically if a thermal limit of the amplifiers heatsink is breached, via the use of normally-closed thermal circuit breaker switches connected in series with the off button. Each rf switch can be configured (via a toggle switch on the front panel) to either be on, off or controlled via a transistor-transistor logic (TTL) or low-voltage transistor-transistor logic (LVTTL) signal. A second toggle switch (for each channel) sets whether the rf output of the switch should be directed to the 'test' output or the amplified output, and LEDs light up next to each BNC output to indicate if rf is currently coming out of that port. Our circuit also contains a tri-state buffer needed for connecting a clocking signal to the Novatech DDS9m board in order to step through the instructions for the hardware timed channels. This portion of the circuit was designed by Shaun Johnstone and is detailed in [121].

We also generate some of our rf using PulseBlaster DDS-II-300-AWG devices [122] and an in-house device called the RFBlaster. These both provide unamplified rf on the order of 0 dBm, but support a much faster update rate of their output state than the Novatech DDS9m. As such, we use a modified version of the Supernova containing just the rf amplifiers when connecting these devices to AOMs.

3.4.2 Shutter drivers

While the AOMs in our beamlines provide fast control over the amplitude of the laser light in use on the vacuum table, there is always a small leakage. This leakage can destroy ultracold atom samples, either by heating the atoms or by producing unwanted atomic state transitions. Like most laboratories, we use a series of shutters to block out the light from

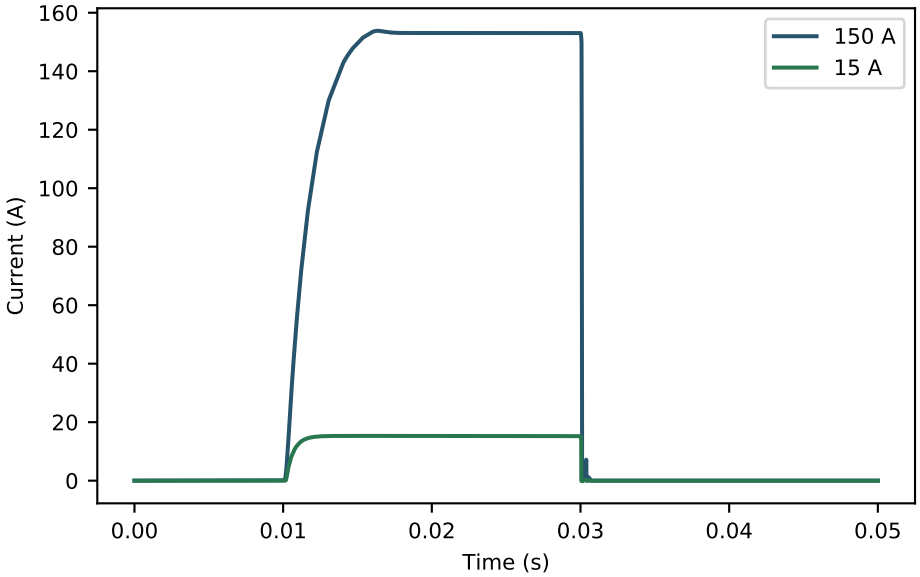


Figure 3.8: The simulated coil driver response to a 20 ms square wave pulse beginning at 10 ms. Blue: With the coil power supply set to 12.5 V, a coil driver control signal of 10 V (not shown) should drive a 150 A current. The simulation shows the current smoothly reaching a stable level of 153 A within 10 ms of the change in control signal. Green: With the coil power supply set to 8 V, a coil driver control signal of 0.96 V (not shown) should drive a 15 A current. The simulation shows the current smoothly reaching a stable level of 15.3 A within 10 ms of the change in control signal. Both modes of operation have a very fast switch-off time on the order of 40 µs. There is a small amount of ringing during the simulated switch-off which is clamped by protection diodes that are not included in the simulation.

each beamline entirely. We also use them in instances where we have split beamlines after they have exited the fibre on the vacuum table, in order to provide some independent control over the beams.

We use the SRS SR475 model shutter, with most being driven by a digital output routed through the SRS SR474 4-channel shutter driver. We later learnt that this driver was unnecessary, and that the shutter heads themselves could be driven directly by a digital signal provided an appropriate power supply was obtained to power the logic controller built into the shutter head.

3.4.3 Magnetic coil driver

We use custom coil drivers in order to drive currents through our various magnetic coils. For our low current bias coils (and source MOT coils), we use a design developed by colleague Dr. Russell Anderson at Monash University (based on a design by Carl Sauer and Adam Kaufman at JILA), which he called the 'mag-neat-o'. The circuit takes an analogue voltage (commanded by our control system) and uses an op-amp in a PI configuration to drive an insulated-gate bipolar transistor (IGBT) that controls the flow of current through one or more coils. Feedback is provided to the op-amp via a current transducer (Hall sensor, LEM

LA 100-P) which measures the current passing through the IGBT (and thus coil(s)). This design was able to drive up to 20 A in a single direction only. We augmented this design with relays (commanded, via a digital signal, by our control system) to invert the direction of current through the coil, in order to create a bi-polar design (albeit with a relatively slow switching time between current directions of 4 ms).

Using the mag-neat-o as a base, I modified the circuit in order to create a coil driver capable of driving up to 150 A, which was necessary for our central quadrupole coils that are used for magnetic trapping. Unfortunately I was unable to create a design that was stable at both low and high currents through the coils at a common power supply voltage. In addition to this, the power dissipated by the IGBT (a function of the current demanded through the coils, and the voltage of the power supply used to drive the coils) became dangerously close to the maximum allowed at certain currents. To solve these problems, I adopted a method of switching the power supply voltage (using a digital signal) between a low (8 V) and high (12.5 V) setting. We switch between these voltages at an intermediate current of 80 A using our control system, in order to ensure the driven current does not oscillate and the IGBT does not catch fire[12]. Simulations of our coil driver circuit driving a typical MOT current and a typical magnetic trap current are show in figure 3.8.

3.4.4 Microwave source

The Zeeman splitting within the $F = 2$ ground state of rubidium-87 and potassium-41 are equivalent when captured by our magnetic trap[13]. This means that any forced evaporation using rf to transition atoms between trapped and anti-trapped Zeeman sub-states will interact equally with both species. However, this is undesirable as it is more efficient to apply forced evaporation to rubidium-87 only, and let it sympathetically cool the potassium atoms (thus reducing the losses of potassium atoms). To achieve this, we use a microwave source during the forced evaporation stage of our experiment to transition the rubidium atoms between the trapped $|F = 2, m_F = 2\rangle$ state and the anti-trapped $|F = 1, m_F = 1\rangle$ states. This transition has a different resonance frequency than the same states in potassium due to the different hyperfine ground state splitting, leaving the potassium atoms in the trapped state. Rather than buy a commercial microwave source, with amplitude and frequency control via a preprogrammed table, we use a quadrature modulator board to combine a fixed frequency and amplitude microwave local oscillator with two rf outputs from a RFBlaster. This generates two frequency side-bands either side of the microwave carrier, the frequency and amplitude of which can be controlled via the frequency and amplitude of the rf signals. The carrier and one of the sidebands can be suppressed by adjusting the relative phase between the two rf signals, resulting in a tuneable microwave source with precision timing (as limited by the rf source). Further details of this tuneable microwave source are available in the theses of Alex Wood [123] and Shaun Johnstone [100].

12. This might seem like an exaggeration, but our IGBT did actually catch fire once!
13. Similarly, the Zeeman splitting within the $F = 1$ ground state is equivalent between the two species.

3.5 Summary

In this chapter we introduced the dual-species BEC apparatus constructed primarily by Shaun Johnstone and myself. We detailed both the laser and vacuum systems, and outlined some of the complexities that come with developing a dual species apparatus. Following this we detailed the electronics that we have developed to help control the BEC apparatus. This consisted of two process control systems, to safe-guard critical areas of the apparatus, and a set of (mostly custom) electronic systems that interface between the apparatus and the scientific control system, such as coil drivers and AOM drivers. In the following chapters, we'll introduce and detail our scientific control system. Later, in §8.1, we'll discuss the use of our scientific control system with the apparatus described in this chapter.

Chapter 4

The labscript suite

In this chapter, we will discuss the philosophy behind the development of the labscript suite and the technologies we used to achieve our goals. However, understanding the labscript philosophy requires some prior knowledge of the labscript suite and the individual components (computer programs) within. As such, we'll briefly introduce each component here first, and describe how they link together[1]. We'll then focus on the guiding principles of our development process, which builds on the previous work of G. Varoquaux [124].

The labscript suite is a large project, and many people have contributed to the development. Chris Billington [125] and I are the primary architects. Shaun Johnstone [100] and Martijn Jasperse [106] also made significant contributions. There are also several people at other institutions (who are using the labscript suite) who have contributed new features and bug fixes. A full record of code contributions to this project is available on Bitbucket [9][2], however this does not include development discussions, which cannot be easily quantified.

4.1 Terminology

In the process of developing the labscript suite, we have developed a set of common terminology. While some of this will be familiar to experimental physicists already, others are specific to the labscript suite. In this section we'll outline the key terms that are frequently used throughout the discussion of the labscript suite. Other terminology specific to a single component will be introduced in the relevant section, and all definitions are in Appendix A.

We introduce the concept of an 'experiment' as the scientific aim of the research. Examples range from a very broad definition such as "the procedure to study an ultracold atom cloud using one of 10 techniques we have available" to very specific definitions such as "the procedure to produce an ultracold atom cloud, stir it with a repulsive laser beam, and then study the evolution of the vortex dynamics using absorption imaging in the plane orthogonal to the direction of gravity". It is important to stress that the aim of an experiment is the same regardless of the particular parameters used in any particular performance of an experiment. A scientific publication is then typically the result of the execution of a 'sequence of shots' (or 'experiment sequence'), where each shot performs the experiment with a set of parameters that typically vary between shots. The procedure of an individual

1. We will revisit each of the components in more detail in the following chapters (see chapters 5 and 6).
2. A copy of the labscript suite, as it was on the date of submission for this thesis, is also available at [126].

shot is then defined by what we term 'experiment logic', which determines how individual hardware I/O should respond as a function of the provided parameters.

We also introduce terminology to describe the people who interact with the labscript suite. We categorise people into three groups: users, developers, and architects.

Users comprises of people who define, run, and analyse experiments using the labscript suite. Users write basic Python scripts, such as those that define experiment logic or an analysis routine. They also are the primary users of the graphical programs.

Developers (who are often users too) are those who perform more complex programming, and expose that programming to users via friendly interfaces. Primarily, this group encompasses people who write code to support new hardware devices. However it also includes those who write helper functions and/or libraries for use in Python code, or plugins for the graphical applications. We expect code written by developers to be modular, so that new features can be turned on and off as required by different laboratories.

Architects are those who understand the inner workings of the entire software suite. They are responsible for maintaining the functionality of the labscript suite, and integrating new features into the core code-base of the labscript suite. Changes made by architects affect everyone using the labscript suite, which distinguishes them from developers.

4.2 Components of the labscript suite

The labscript suite is made up of six programs: labscript, runmanager, runviewer, BLACS[3], BIAS[4], and lyse. With the exception of labscript, they are all GUI applications[5]. The programs can be split into two categories: those that are used for experiment preparation and those used for experiment execution (which we expand upon further in chapters 5 and 6 respectively).

Figure 4.1 shows how the components in the labscript suite are linked together, and how data flows between them. In most cases, communication is via a network socket where a path to a shot file is provided along with an associated action to run at the receiver. This follows the actor model, a well defined paradigm in software engineering. The technologies shown (such as HDF5 and ZeroMQ) are discussed in detail in §4.4.

Experiments are primarily prepared using a combination of labscript and runmanager. Labscript is an API; a text-based interface for accessing the code we have written to generate hardware instructions for hardware devices. The interface is very high-level, allowing the logic of an experiment to be defined with code that is easily understood by humans. Runmanager is a graphical application for managing experiment parameters, and producing shots to be executed on the hardware (at a later time). Experiment parameters can be access from the experiment logic script, and can take the form of standard Python data types like integers, floats, strings, Booleans, or tuples. For example, a parameter called `MOT_load_time` could be used to control the length of a MOT load sequence, or a Boolean parameter called `enable_dipole_trap` could be used with an `if` statement to enable or

3. Originally B.L.A.C.S. was an acronym standing for the 'BEC lab apparatus control system'. This was later updated to the 'better lab apparatus control system', when it became clear our software was generalisable to other experiments, before being dropped in favour of just using 'BLACS' as a name.
4. Like BLACS, B.I.A.S. originally stood for the 'BEC image acquisition system', was later updated to the 'better image acquisition system', and then dropped in favour of just using 'BIAS' as a name.
5. Although, lyse also relies on user prepared analysis scripts written in Python (much like labscript).

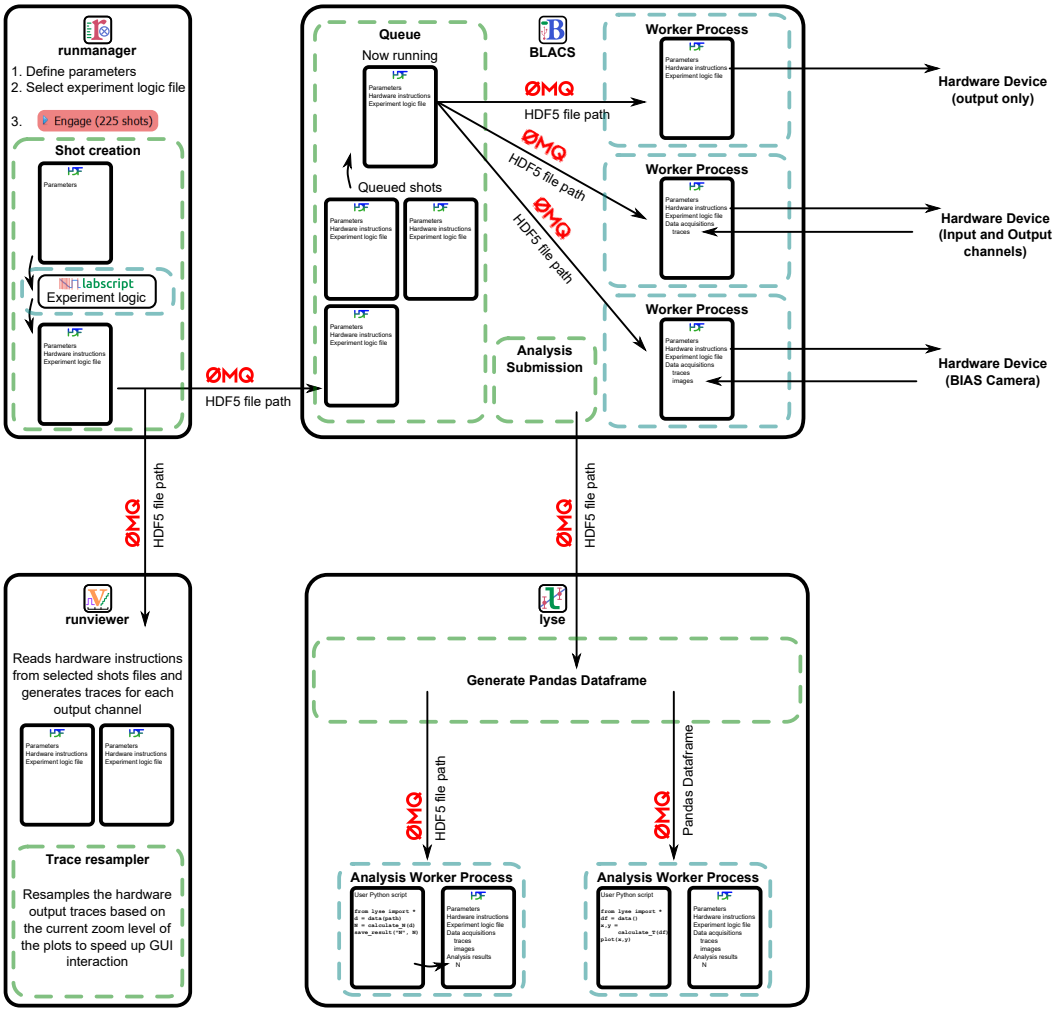


Figure 4.1: The flow of data between components of the labscript suite is shown. Green dashed lines denote threads within a program while blue dashed lines denote a worker process. Shots are born in runmanager, where the HDF5 shot file is created containing the set of experiment parameters. Runmanager then executes the experiment logic script in a worker process, which uses the labscript application programming interface (API) to generate hardware instructions. Shots are passed to BLACS and/or runviewer. BLACS executes shot files from a queue and any acquired data is saved in the shot file. After execution, shot files are then passed to lyse, where they are analysed by a series of user specified analysis routines (Python scripts), which may save analysis results in the shot file. References to a shot file are passed via ZeroMQ, both between components of the labscript suite and between a component and its worker processes.

```
# Pseudoclock definitions
PulseBlaster(name='pulseblaster_0', board_number=0)
# Clocklines
ClockLine(name='pulseblaster_0_clockline_fast',
          pseudoclock=pulseblaster_0.pseudoclock,
          connection='flag 0')
ClockLine(name='pulseblaster_0_clockline_slow',
          pseudoclock=pulseblaster_0.pseudoclock,
          connection='flag 1')

# Output device definitions
NI_PCIe_6363(name='ni_card_0',
             parent_device=pulseblaster_0_clockline_fast,
             clock_terminal='ni_pcie_6363_0/PFI0',
             MAX_name='ni_pcie_6363_0',
             acquisition_rate=100e3)
NovaTechDDS9M(name='novatechdds9m_0',
              parent_device=pulseblaster_0_clockline_slow,
              com_port="com10")

# Channels on the hardware devices
Shutter  (name='MOT_shutter',
          parent_device=ni_card_0,
          connection='port0/line1')
AnalogOut(name='B_quad',
          parent_device=ni_card_0,
          connection='ao2')
DDS      (name='MOT_aom',
          parent_device=novatechdds9m_0,
          connection='channel 0')
```

(a) The connection table definition, which defines how the hardware devices are connected together.

disable a block of experiment logic via the graphical interface of runmanager. Users thus choreograph experiments through the creation of a labscript (Python) file that defines the experiment logic (using functions and classes provided by the labscript API) and a set of parameters defined through the graphical interface of runmanager. Runmanager then uses the labscript file and set of parameters to create shots files (one file per shot in the experiment sequence). Runviewer can then be used to visualise the expected output of the hardware devices, as determined by the hardware instructions stored in a shot file, prior to actually executing shots on the real apparatus.

BLACS is designed to execute shot files on an apparatus. After shots are generated by runmanager, they can be sent to BLACS where they are queued and executed in sequence by BLACS. BLACS coordinates the programming of hardware devices, including via secondary control systems such as our imaging system BIAS (which may be running on a separate PC), and the saving of results to the shot file. Once a shot has been executed on the apparatus, the shot file is forwarded to lyse for analysis. Lyse runs a series of user-defined analysis routines (also written in Python) that access one or more shot files and perform the requested analysis.

```
# Begin the experiment
start(); t = 0

# Turn on MOT AOM to 0.9 amplitude and 80 MHz
MOT_aom.setamp(t, 0.9)
MOT_aom.setfreq(t, 80*MHz)
# Open MOT shutter
MOT_shutter.open(t)
# Set quadrupole field analog output to the value of the MOT_field
# global variable (that is defined in runmanager)
B_quad.constant(t, MOT_field)

t+= 2 # Wait 2 seconds

# Turn off MOT and ramp quadrupole field
MOT_aom.setamp(t, 0)
MOT_shutter.close(t)
# Quadrupole field analog output ramp from 3V to 10V over 4
              seconds,
# at a sample rate of 10 kHz. The ramp function returns the
              duration,
# which is used to update the t variable
t += B_quad.ramp(t, duration=4, initial=3, final=10, samplerate=
              10e3)

stop(t + 1e-3) # stop the experiment 1 ms after the ramp finished
```

(b) The experiment logic definition, which defines what the hardware devices should do during a shot.

Figure 4.2: An example Python script written using the labscript API. (a) The connection table. Here we define a pseudoclock that has an NI PCIe-6363 card and a Novatech DDS9m device attached. We then define channels on these devices, which are referred to in (b) using the names specified. (b) The experiment logic. At the start of the experiment, we turn on the MOT and load for 2 seconds. We then turn off the MOT light, and ramp the magnetic field linearly over 4 seconds. Note the use of the global variable `MOT_field`. The value for this is specified in runmanager as a parameter, and the variable itself will be created by the internals of labscript when runmanager executes the labscript file as part of the shot creation process. Note that for this simple example, both (a) and (b) reside sequentially within the same Python file. Further details on the labscript API can be found in §5.1.

4.2.1 Labscript API

The labscript API is a text based interface for defining the hardware devices to be controlled during an experiment and the experiment logic those devices should perform. It is used by 'users' to choreograph an experiment. Inside a Python file, the API is first used to define the hardware devices in use, how they are connected to each other and the controlling PC(s), and the input and output channels in use. We call this section of user written code the connection table. The connection table code creates Python objects that can then be used in the remainder of the script to define the experiment logic. The experiment logic is where the user defines the state of each output, at various times throughout the experiment. Each object has a set of methods provided by the labscript API for achieving this. For example, digital output objects have methods such as `go_high(t)` and `go_low(t)` to command the

output to change state at the specified time.

An example script containing both the connection table and experiment logic is shown in figure 4.2. Outputs are controlled through a unified interface, which provides common function names for outputs of a particular type. For instance, digital outputs always have the previously mentioned methods regardless of the actual model of device in use. Internally, the labscript API generates the required hardware instructions in a format specific to each device, and generates the required instructions for the pseudoclock(s) that manage the timing of each hardware device. This abstraction allows the user to focus on just writing the experiment logic they wish, without having to worry about the particular implementation details of a device, or the generation of the clocking signal.

4.2.2 Runmanager

Runmanager (see figure 4.3) is the primary interface for adjusting the behaviour of the experiment and generating shot files. Runmanager allows you to define parameters via a graphical interface and select the experiment logic file to use. The parameters are inserted into the experiment logic (as global variables) when shot generation is started via the 'Engage' button.

The parameters can take the form of any Python data type that can be stored in a HDF5 file. When a parameter is specified as a list or array, runmanager automatically detects that you wish to explore a parameter space. Runmanager will generate a separate shot for each point in the array. If multiple arrays are specified across two or more parameters, runmanager automatically takes the Cartesian product of those parameters in order to generate shots that span the entire parameter space. For more complex experiments, parameters can be linked together so they iterate in lock step by defining one or more zip groups. If multiple zip groups are defined, the parameter space explored will be the Cartesian product of each zip group and any other parameters containing an array or list.

4.2.3 Runviewer

Runviewer is used for viewing, graphically, the expected changes in each output across one or more shots, and is shown in figure 4.4. Its use is optional, but can be extremely useful for debugging the behaviour of experiment logic. The output traces are generated directly from the set of hardware instructions stored in a given HDF5 file. This provides a faithful representation of what the hardware will actually do. In effect, runviewer provides a low level representation of the experiment, which complements the high level representation provided by the experiment logic written using the labscript API. As such, runviewer traces provide a way to view the quantisation of outputs[6], which can be seen in the 'central_Bq' and 'central_bias_z_coil' channels in figure 4.4. You can also view the pseudoclock outputs. The `pulseblaster_0_ni_clock` and `pulseblaster_0_novatech_clock` channels demonstrate the independent clocking of devices from a single PulseBlaster pseudoclock. Similarly, `pulseblaster_1_clock` shows an entirely independent secondary pseudoclock.

6. While this is always true in time, the output values may not be correctly quantised if the labscript device implementation does not quantise the output values correctly and instead relies on BLACS, the device programming API or the device firmware, to correctly quantise the output values.

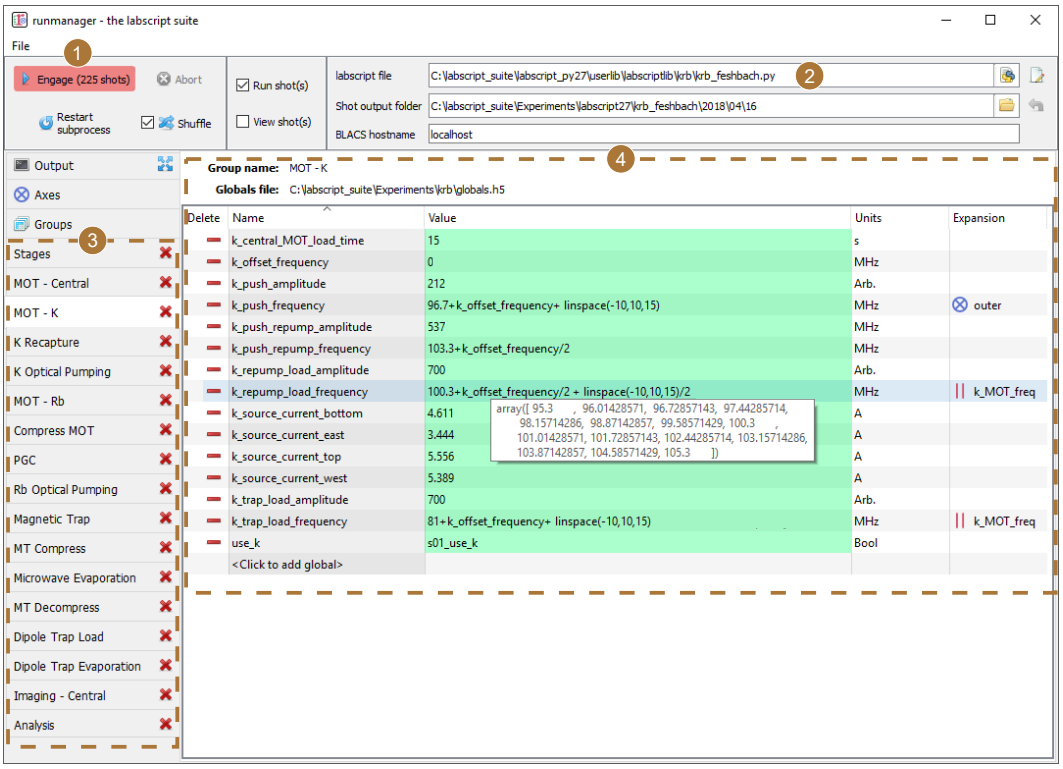


Figure 4.3: The runmanager interface, used to generate experiment shot files (1) using the experiment logic defined using the labscript API in a Python file (2) and groups of parameters (3), for example those shown for the 'MOT - K' group (4). Further details on the runmanager interface can be found in §5.2.1.

4.2.4 BLACS

BLACS interfaces with the hardware devices controlling an experiment, and manages the queue of shots to execute on the apparatus. BLACS has two modes of operation: the execution of shots under hardware timing, and the manual control of hardware devices (by the user) via the BLACS GUI. The interface is shown in figure 4.5 and is split into two sections that align with the two modes of operation: the queue of shots to execute, and a GUI interface for manually controlling the output state of the hardware devices when not running shots (which can be useful for manual debugging of an apparatus).

The shot queue contains standard controls for adding deleting and reordering shots. The queue can also be paused or put into one of several modes that repeat the shots in the queue. When a shot finishes, and the results have been saved to the HDF5 file, the shot may be optionally sent to the lyse server specified in the GUI.

The GUI for each hardware device is dynamically generated at runtime, based on a connection table written using the labscript API. A device tab is created for each device, and communicates with a unique worker process which, in turn, handles the communication with the hardware device. The device tab GUI is populated with controls for each input/output (IO) channel present. To aid in the identification of a relevant I/O channel, controls are labelled in BLACS using both the hardware I/O port name and a user specified name from

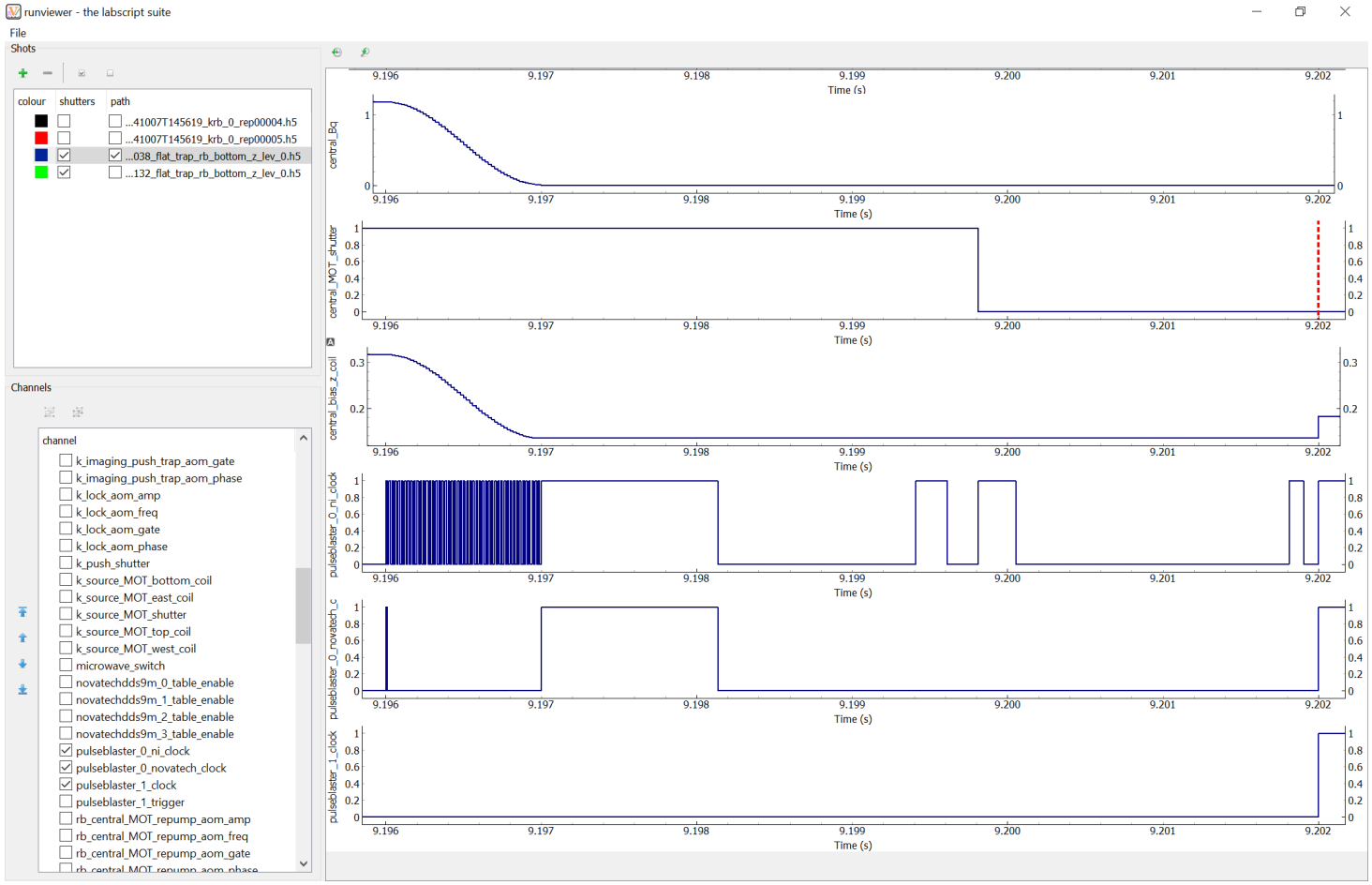


Figure 4.4: The runviewer interface shows output traces for selected channels in selected shot files. Further details on the runviewer interface can be found in §5.3.2.

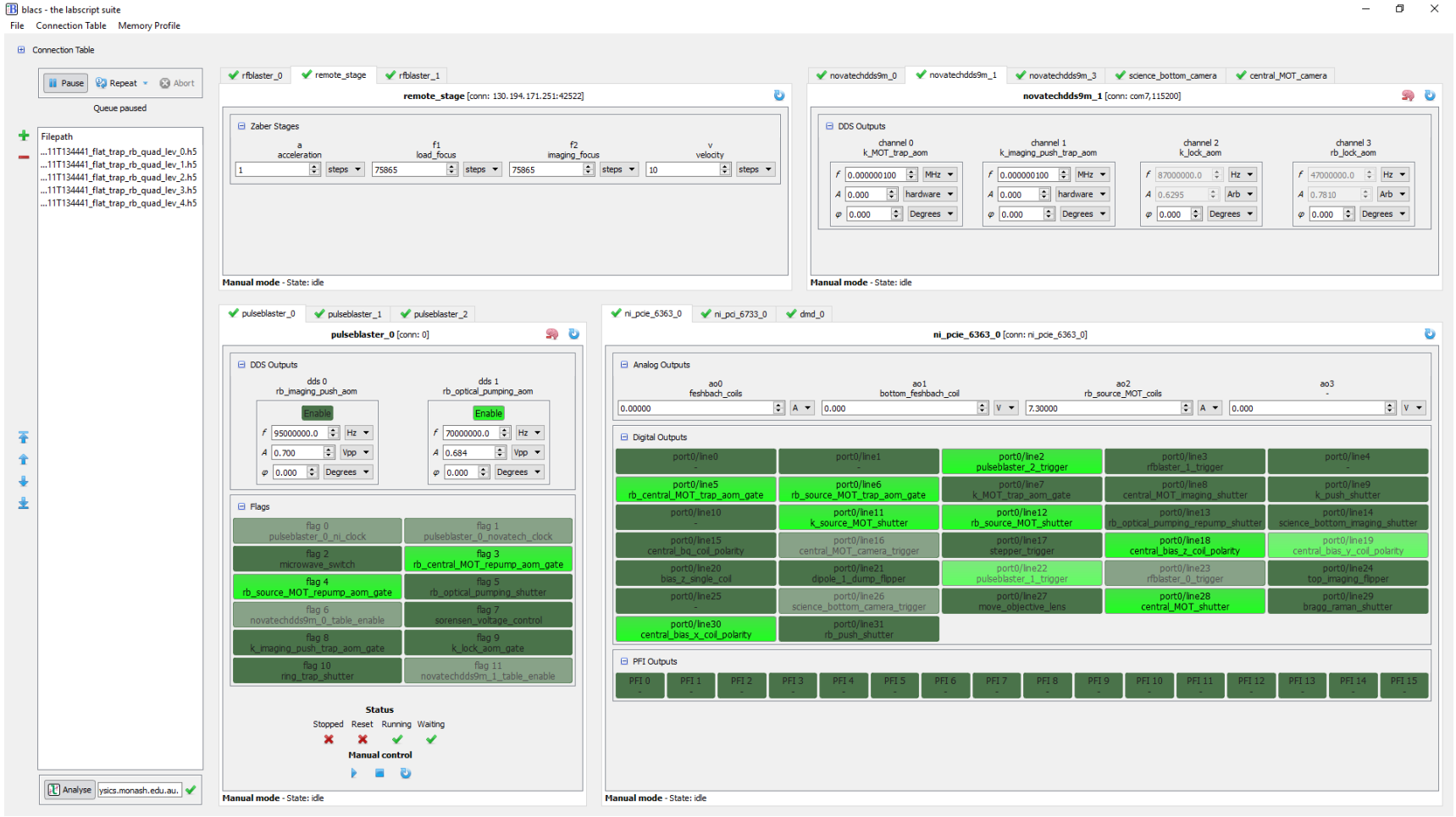


Figure 4.5: The BLACS interface. Left: The shot queue. Main: A set of tabs (one for each hardware device) that provide a manual control interface for each device. Further details on the BLACS interface can be found in §6.1.

the connection table of a labscript file. Analog channels (or more complex output types like a DDS that are represented by several analog numbers) also integrate with the unit conversions specified in the connection table, allowing both control and the display of units other than the (device specific) default hardware unit. Channels connected to sensitive equipment can have the output values limited or the control locked entirely to prevent accidental changes. Output values are stored on exit, and restored on start-up, to avoid unexpected output transients.

4.2.5 BIAS

BIAS is the imaging system we presented in our paper on the labscript suite [8], and was developed by Martijn Jasperse. The aim of BIAS was to provide an interface that performed rapid analysis of images immediately after a shot, without the use of lyse. For BEC experiments, this consists of fitting to an absorption image of an ultracold atom cloud and returning the optical density and atom number.

While this component is currently unmaintained[7], it demonstrates the modular nature of the labscript suite. BIAS was originally developed independently from the labscript suite. However it is now fully integrated, and is managed by BLACS as a secondary control system. For all intents and purposes, BLACS considers BIAS to be a hardware device, despite the fact that is actually a software interface that, in turn, controls a hardware device. BIAS is written in LabVIEW, which is often the language of choice in scientific research laboratories, and provides a clear demonstration of how labscript suite can be used with existing control software (even that written in another language). We discuss the integration of the labscript suite with existing control systems further in §6.2, and the integration with other programming languages in §4.4.

4.2.6 Lyse

Lyse is a framework for performing arbitrary analysis on the data stored in a shot file, and the graphical interface is shown in figure 4.6. We break analysis into two distinct groups: single shot analysis that only accesses a single HDF5 file, and multi-shot analysis that performs higher order analysis on data from many shots. A set of Python scripts can be loaded for each analysis category, which will run (when appropriate) on new shot files as they are sent to lyse from BLACS. Acquired data, global variables from runmanager and analysis results generated in lyse, for all loaded shots, are stored in a table (a Pandas DataFrame, see §6.3.1). The DataFrame is displayed in the GUI and can also be accessed remotely from another PC.

7. BIAS is unmaintained largely due to a lack of LabVIEW developers and the difficulty in new developers understanding any pre-existing LabVIEW code. While it is disappointing that we have not found a LabVIEW developer in the wider community to continue maintaining BIAS, it is worth mentioning that this has not been a problem for the Python labscript suite components. Many of the groups using the labscript suite have been modifying and contributing back to the labscript code-base (an important part in ensuring the longevity of the project) and we expect that ultimately this will apply to a future Python-based imaging component.

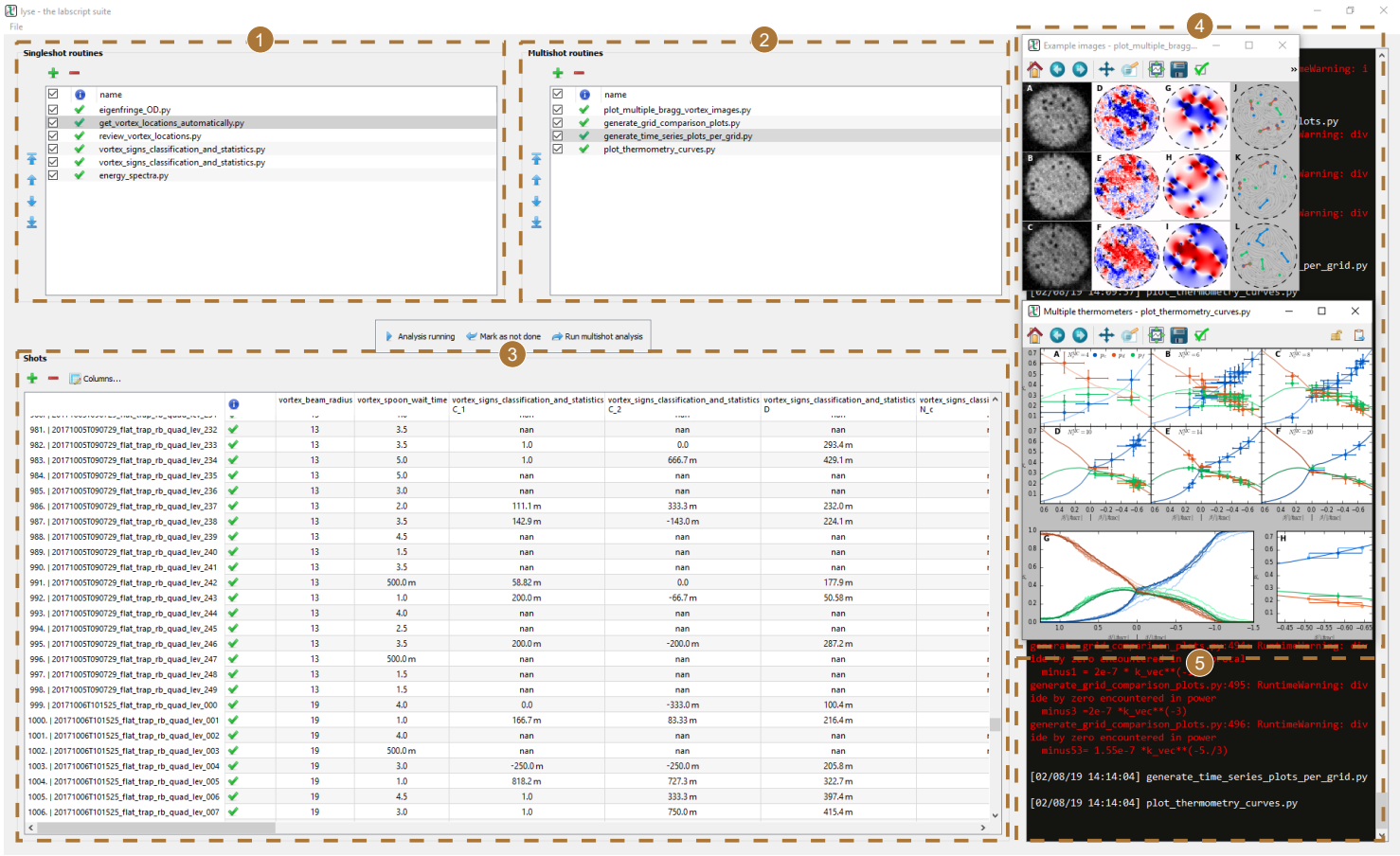


Figure 4.6: The lyse interface. (1) single shot analysis scripts. (2) multi shot analysis scripts. (3) A graphical representation of the Pandas DataFrame. (4) Figures generated by the analysis scripts. In this case, these are publication-quality figures generated for one of our recent papers published in *Science* [127]. (5) The output log from lyse and the analysis scripts.

4.3 Design Themes

In this section we'll detail and discuss several of the key design themes that form the basis of the labscript suite. While these are key to understanding the labscript suite, we also believe they have wider applicability and should be considered by those beginning the process of writing their own control system.

4.3.1 Unix Philosophy

In order to facilitate the creation of a flexible, future proof set of software, we followed the Unix philosophy when developing our control system. There are several versions of the Unix philosophy, but they are all based on the words of Doug McIlroy [128], quoted below:

1. *Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features".*
2. *Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.*
3. *Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.*
4. *Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.*

Eric Raymond [129] summarised and extended these into 17 rules based on the software developed by those espousing the Unix Philosophy. The most relevant to our software development process are quoted below:

- *Rule of Modularity: Write simple parts connected by clean interfaces.*
- *Rule of Composition: Design programs to be connected with other programs.*
- *Rule of Generation: Avoid hand-hacking; write programs to write programs when you can.*
- *Rule of Optimization: Prototype before polishing. Get it working before you optimize it.*
- *Rule of Extensibility: Design for the future, because it will be here sooner than you think.*

Our most obvious implementation of the Unix philosophy is our avoidance of creating monolithic programs. As described previously, the labscript suite is composed of several programs (see §4.2). Each program is designed to do one thing, and do it well; labscript generates device hardware instructions, runmanager modifies parameters, runviewer shows a faithful representation of output from a set of hardware instructions, BLACS interfaces with hardware, and lyse analyses results. These programs communicate through well defined standards that comprise of text based streams over ZeroMQ network sockets, and HDF5 files

for large datasets such as hardware instructions and acquired data. When we can't avoid the creation of a single large program, we build it from modular components (for example lyse uses pluggable analysis scripts, and BLACS implements modular hardware support as well as plugins for implementing custom behaviour). As the Unix philosophy advocates, this allows one or more components to be easily replaced by a new custom program. In the labscript suite, this is already occurring. For example, mise (detailed in [8]) has been deprecated and replaced by complex analysis scripts (such as analysislib-mloop [130]) and BIAS will soon be deprecated in favour of a Python alternative[8].

Adhering to this philosophy also allows us to easily distribute components across multiple PCs, and allows lab members to simultaneously interact with different components of the suite. For example while one person manages shot creation via runmanager, another can be interpreting results via an instance of lyse running on a separate PC. This philosophy has also allowed us to rewrite components one at a time, in small steps, rather than needing to rewrite everything in one go. For example, prior to the publication of our paper in 2013, we made significant changes to runmanager and BLACS. Since 2013, we have progressively rewritten every component of the suite. This would not have been possible without a set of distinct programs coupled by well defined interfaces, and our control system features and stability would be poorer for it.

Where possible, we apply the rule of generation to simplify the work needed to implement our control system on an apparatus. This rule is particularly applicable to adding support for new hardware devices. For example, we help developers by providing standardised interfaces for commanding output via labscript, and automatically generate manual control interfaces from a standard set of graphical widgets in BLACS. For users, the high-level interface for commanding output via labscript also removes the need for them to think in terms of hardware instructions, which are automatically generated by labscript and device specific code. This is a significant improvement over line-based systems that describe experiment logic at a level close to the level of raw hardware instructions (and are often unreadable for the uninitiated).

Our development process was also influenced by the rule of optimization. We always rapidly deploy changes to our apparatus in order to test and debug, prior to prematurely optimising our code. We believe this is significantly aided by our dedication to maintaining a complete record of experiments performed (See §4.3.4). The labscript suite is designed to assume there is no special state of devices or software that is not known to the system, and that a comprehensive record of all device and software states in the experiment will be automatically stored in each shot file. For example, we store a record of how all hardware devices are connected together, and the set of configuration parameters needed to configure the device correctly, so that we can reproduce any given configuration at a later date. This gives us freedom to rapidly modify and create solutions without fear of permanently breaking something or changing a configuration that was not recorded anywhere. We have seen several labs that have been impressed by the labscript suite, but have been reluctant to adopt it for fear of breaking what they have now, due to the lack of those systems providing a complete record of the current configuration. However it is my belief that if you can't do something twice, then you can't be sure you've done it once. It is my hope that the labscript

8. In fact, several laboratories using our control system around the world are already using their own custom (usually Python based) imaging system.

suite (underpinned by the Unix philosophy) will encourage research labs to become more agile in their development (of both code and science), and that this will in turn lead to more robust and repeatable science.

Finally, we spent a significant amount of time planning for the future (the rule of extensibility). Most of our focus here was on maximising the flexibility of the suite and ensuring any prescriptions were reduced to a minimum. We discuss this further in the following section.

4.3.2 Flexibility

A key aim during the development of the labscript suite was to create a system that is flexible. This aim was born from several factors: the difficulty in adapting other control systems to our experiment, the desire to have a unified control system within our research group at Monash University, and the desire to build a wider community that could contribute to the ongoing development of a control system in order to reduce duplicated effort. Flexibility is thus at the heart on the labscript suite philosophy, and is the most important of all of the design themes presented here. While we strongly believe in the other design themes discussed in the following sections, the flexible nature of our system allows a user and/or developer, with a bit of coding, to override one or more of these themes while still reaping the benefits from other features.

One of the key features of our system is support for heterogeneous hardware. Labscript, runviewer and BLACS all provide comprehensive APIs in order to simplify the addition of new devices. As we'll detail in §7.1, a developer need only write code to translate between labscript and hardware instructions, hardware instructions and runviewer traces, and code to program the device with the hardware instructions. We achieve this by providing a consistent structure for commanding I/O from specific output types, rather than specific hardware devices. Most GUI generation and integration with the text-based labscript API is done through an abstraction layer we provide, allowing developers to build device specific functionality on top of the general purpose code we provide. Once completed, this device code can be shared with other research groups in a modular way, which avoids duplication efforts and allows research groups to pick and choose their preferred hardware. This implementation also has the benefit of providing a consistent user interface for using most devices, which reduces the overhead of integrating a new device into a lab. The labscript suite also supports an arbitrary number of hardware devices. Many control systems have a limitation of the number of hardware devices, either due to hitting the limit of PC resources, or being implemented in a language that does not support dynamical GUI generation (for example LabVIEW). The labscript suite supports both of these, through the use of a text-based interface in labscript, a tabbed interface in BLACS and the ability to add an arbitrary number of secondary control systems that can run on separate PCs or other embedded devices.

Our analysis system, lyse, follows a similar philosophy. We provide an API for accessing acquired data and saving analysis results in order to ensure consistency and interoperability between components of the labscript suite and custom extensions to our system. Users are able to freely create analysis scripts (written in Python) and make use of any additional libraries they like. These scripts can be run from the command line directly (even on remote PCs), or loaded into the lyse GUI where they will be automatically executed as new data

is acquired. Sets of analysis scripts can easily be switched in and out depending on the current experiment being performed. Again, this allows analysis code to be reused, shared, and easily modified between laboratories.

The program BLACS provides configurable behaviour through a plugin system. This allows research groups to modify the behaviour of the experiment execution phase without creating a divergent copy of the software. We have used this functionality to develop optional features such as implementing a 'keep warm' feature that runs an experiment on repeat and deletes the shot file once complete, or monitoring of lab process control systems so that the queue can be automatically paused if one of our interlocks enters a protective state and shuts down the apparatus.

In essence, we have focussed our effort into building well defined frameworks, for experiment preparation, control and analysis, that are easy to use while enforcing minimal restrictions on extensibility. Our frameworks are broken into distinct components, with well defined open communication protocols and file formats (see §4.4.3 and §4.4.2). This opens the possibility for users to replace or upgrade components with custom implementations featuring previously unforeseen capabilities, as they need, without having to modify the entire suite. This is a distinct advantage over a monolithic design.

When designing our APIs and interfaces we specifically focussed on simplifying the realisation of complex scientific experiments. Experiments are increasingly demanding more complex hardware capabilities, and the simultaneous control of multiple outputs. These outputs often change on a non-uniform time-base, or require their output state to change following a complex equation. The labscript API was specifically designed to abstract this away, so that users could focus on defining the experiment logic for the science they wish to execute, rather than the capabilities or interactions of specific hardware devices (see §5.1 for further details). Experiments also often demand traversal of a multi-dimensional and/or densely sampled parameter space. We designed runmanager to automate this traversal by automatically creating shots that span an arbitrarily large parameter space, with an arbitrarily large number of dimensions, as described by parameters containing lists or arrays.

The labscript suite thus has flexibility at all levels, from the combination of frameworks, to the ability to define arbitrary experiment logic. This results in a control system that is applicable to a wide range of experiments. While initially designed for ultracold atom research, the labscript suite has proven useful in several other situations. For example, we used the labscript suite to automate the bench testing of an objective lens (see §8.3). Due to the flexibility built into our system, we believe that any experiment with a distinct start and stop time, that requires precise timing, can be controlled using the labscript suite.

4.3.3 Graphical vs. textual interfaces

Traditional ultracold atom control systems (line-based systems) relied on graphical definitions of both parameters and experiment logic. The benefit of this design is in the rather direct mapping between the representation of the graphical data structure and the hardware instructions to be programmed into the device. While this simplifies the development of such a control system from the architects point-of-view, it introduces significantly more cognitive load when *using* the system. Primarily, this is due to the inability to ever define experiment logic at a higher (more abstract) level, which forces the user to always think

and process the experiment logic in terms of the state at each time step. We contend that such control systems have been deliberately written for their architectural simplicity, and that increased complexity in the experiment design phase, along with limitations on the complexity of experiment logic, is a consequence of this decision. This is of course a reasonable trade-off for individual laboratories to make in the absence of a more advanced (open source or commercial) control system, as the cost of employing someone to maintain complex software systems is prohibitive and assuming a single laboratory will have a steady stream of graduate students with the necessary architectural skills to maintain a complex laboratory control system is optimistic[9]. However, it is not the optimal solution.

The downside to this trade-off is that many of the line-based systems are clumsy to use because they are managed by simple programming logic. For example, an interface with a 2D array of states requires that the state of each output channel be defined at each time point. Re-ordering instructions, offsetting times, or inserting new time points also requires a degree of complex programming, often resulting in these features being left unimplemented. Even saving and loading of different sets of experiment logic requires additional development effort. While such features are not strictly necessary for running an ultracold atom experiment, the manual work-around typically requires significant time to manually update each graphical item. Ultimately such interface limitations in turn limit the number of channels and/or time-points that can be effectively managed. It is possible to overcome some of these limitations by implementing a more complex user interface design (see the Cicero Word Generator in §2.3.2 as an example). However, this requires significantly more architectural complexity behind the scenes just to make the software easier to use, and doesn't typically result in any additional capabilities for defining more complex experiment logic.

Textual interfaces, on the other hand, provide a more natural way to manage experiment logic. Much of the difficulties with graphical descriptions come for free, such as the ability to copy, paste, or reorder commands, without the need for additional development effort. If the textual interface is provided through a fully fledged programming language, then control statements such as `if ... else ...` or `for ... do ...` loops can be incorporated easily. It is also easier to develop higher level interfaces for experiment logic, for example complex sections of experiment logic can be wrapped up in functions for use by less experienced users. The underlying high level interface can also be written such that commands can be specified in a non-linear time order, which often makes sense for I/O that have inherent delays you wish to account for while still maintaining readable experiment logic.

While textual interfaces bring these many benefits, working day-to-day in a purely textual environment is not ideal. For example, defining parameters and creating and scheduling shots is easier and more convenient if performed through a graphical interface. We follow the work of the David Hall [93] and Strontium BEC [85] control systems by separating out the experiment logic from the definition of parameters so that we can provide a textual

9. By "necessary architectural skills" we mean those with the ability to handle the intricacies of event based programming critical to developing graphical applications, the ability to rapidly develop code and build on code developed by previous students. While almost all physics graduate students have programming skills (and indeed our own control system, the labscript suite, relies on this being true), there is a large gap between writing and modifying simple scripts and understanding and robustly modifying the architecture of a complex graphical application or software framework.

interface for defining experiment logic while keeping a graphical interface for the more common actions like updating parameters. We believe our implementation is superior to these prior works, by our application of the Unix philosophy to the problem, and by the use of a modern high-level programming language. For instance, while the Strontium BEC Control software provides a graphical interface for updating the values of parameters, these parameters must first be defined in the textual interface. In our software, the parameters are both defined and updated from a graphical interface. Both David Hall's and the Strontium BEC Control software use C++ for the programming language, which requires experiment logic be precompiled into an executable before it can be used. By using a modern interpreted language (Python, see §4.4.1), we avoid this step.

We apply similar concepts to our other components. For example, runviewer (for viewing expected output traces) and BLACS (scheduling of shots and manual control of hardware) lend themselves best to graphical control. Analysis (in lyse) lends itself to a mix of graphical and textual, where we believe analysis logic is best described textually, and the visualisation of results and best viewed in tables or plots.

We believe our approach provides a natural mixture for users. Power and flexibility are maintained through reusable text-based scripts, and the ease-of-use of graphical interfaces is maintained for the more frequent interactions with the control system.

4.3.4 Record keeping

The final key design theme of the labscript suite was ensuring the system maintained a complete record of the experiment, from conception to analysis. Humans are notoriously bad at keeping a sufficient record of experiments. We often leave out key details because, at the time, we believe that detail is obvious and a written record is not needed, only to realise months or years later that we've forgotten the detail and should have written it down. For those that can record sufficient details for reproduction, the act of recording those details is often a significant time sink. We were thus heavily motivated to automate this process as much as possible.

Our research group quickly settled on a plan to store experiments on a per shot basis, with one HDF5 file per shot (the file format is discussed further in §4.4.2). Much of the information storage is also a requirement of implementing the Unix philosophy, as it is the primary mechanism for communicating data between isolated components. The main items we currently[10] store in each shot file are records of:

- the parameters used in both their unevaluated form (which contains information about the entire parameter space to be scanned in a sequence of shots) and their evaluated form (specific to the given shot),
- the high level description of the experiment logic,
- the low level hardware instructions to be programmed into the device,
- how the hardware devices and channels are connected together and to the apparatus, and any configuration information needed for these devices,

10. Increasing the coverage of what we store is one of the future improvements we aim to make. For example, we currently do not store a record of the analysis script logic (but this would be easy to do given we already do it for the experiment logic).

- all acquired data (such as images and traces),
- all analysis results, and
- metadata on the experiment, for example shot creation time, shot execution time, and the state of the hardware outputs prior to executing the experiment[11].

Such a complete record can be invaluable during publication review processes by making it easier to respond to unforeseen questions raised by reviewers. The record would also be helpful in identifying how past research has been affected by a recently discovered bug in the control system or experiment logic, ensuring the past results can be reconfirmed or investigated again if necessary. We provide further details on the storage when discussing individual components in more detail, see chapters 5–7.

An important consideration was of course disk space, and storage costs may have been a factor in older control systems not implementing such a comprehensive record. However, storage costs are now minimal. Many institutions provide free (or centrally funded) storage to research groups. For those who need to purchase their own storage, multi-terabyte enterprise level redundant array of inexpensive disks (RAID) systems are now surprisingly cheap. A 30 TB RAID5 network attached storage device can be purchased for under USD 3000[12], which we estimate would provide sufficient storage for a BEC lab for over a decade. Cloud archive storage vaults are also cheap. For example, Amazon Glacier costs approximately $50 USD per TB per year. As storage costs are only going to get cheaper, we believe there is no impediment to automatically storing as much information as possible on scientific experiments.

4.4 Technologies

The labscript suite is built around several key technologies. Here we will introduce those technologies and outline why they were chosen, including the benefits they bring, and whether they can (or will) be replaced by other technologies.

4.4.1 Programming language: Python

The choice of programming language is typically the most important decision made when starting a software project. We contemplated several language choices during the initial stages of the project, including C++ and LabVIEW, before settling on Python. However, while the majority of the labscript suite is written in Python, the choice of the other key technologies was heavily weighted towards those that supported interoperability between multiple programming languages. As such, the labscript suite as a whole is programming language agnostic (as far as that is possible) despite most components being implemented in Python. Indeed the imaging system presented in our paper, BIAS, is implemented in LabVIEW.

Python was chosen primarily due to the low entry bar for understanding and modifying user side code, such as experiment logic and analysis scripts, which is a key factor in enabling

11. The initial hardware output state has the potential to change the experiment sequence in some instances (for example by creating digital edges when the first hardware timed instruction is output).
12. 4x 10TB WD101KRYZ HDDs inside a QNAP TVS-473 NAS.

us to strike the correct balance between graphical and textual control (see §4.3.3). Python syntax relies on explicit indentation to define code blocks [131], rather than special symbols (like `{` or `}` in C++). This enforces a specific coding style, leading to readable code. Python also favours readable keywords such as `not` and `and` for Boolean operations and `in` for checking containment (for example `item in a_list`). This results in code that is often readable by non-experts.

Another key feature of Python is its comprehensive support for interacting directly with other languages through a foreign function interface. Python natively supports calling C libraries [132] and there are third party libraries for calling C++ [133]. This is particularly important since Python is an interpreted language. Software written in an interpreted language is much quicker and easier to modify on-the-fly, when compared to compiled languages like Visual C++ (used in the Strontium BEC control system, see §2.3.3), but execution speed of the software is often the primary trade-off. Foreign function interfaces allow you to write computationally complex algorithms in a performant language (such as C) for use in a less performant language such as Python. Indeed we exploit this for dynamically resampling traces within runviewer (see §5.3.2), as do many other third party Python libraries (such as numpy [81])

Python is an object-oriented language (a key requirement for well designed graphical applications[13]) and contains a comprehensive set of standard libraries [134]. These include libraries for threading, sockets (fundamental networking), data type handling and manipulation, dates and times, and operating system interfaces (among many, many others). Python also has a rich repository of third party libraries in the Python Package Index (PyPI) [135], which can be easily installed via a command line package manager such as pip [136]. We utilise several third party libraries in the labscript suite, some of which we will discuss in the following sections. As part of the labscript suite, we have contributed two libraries to PyPI for use by the wider Python community, qtutils [137] and zprocess [138], which we discuss further in §4.4.5.

These features make Python a popular choice amongst physicists for a range of software projects, and this popularity also played a part in our decision to choose Python as the language of choice for the labscript suite. As of 2018, the labscript suite supports both major versions of Python: Python 2.7 and Python 3.

4.4.2 HDF5

For data storage, we chose the hierarchical data format version 5 (HDF5) file format [139]. HDF5 is a hierarchical data format where tables of data (datasets) are stored within nested groups. Metadata can be added to groups or datasets via the creation of associated attributes. This results in a data structure that resembles a file system, but is contained within a single file.

These features allow us to create self-documenting files; files that contain a complete record of everything related to an experiment, grouped hierarchically, and with human readable names. We create one HDF5 file per shot and build up the contents of the file

13. All good GUI libraries follow an objected oriented design pattern as trying to keep track of the large amounts of data associated with each graphical item without encapsulating it in a hierarchy of objects would be a nightmare.

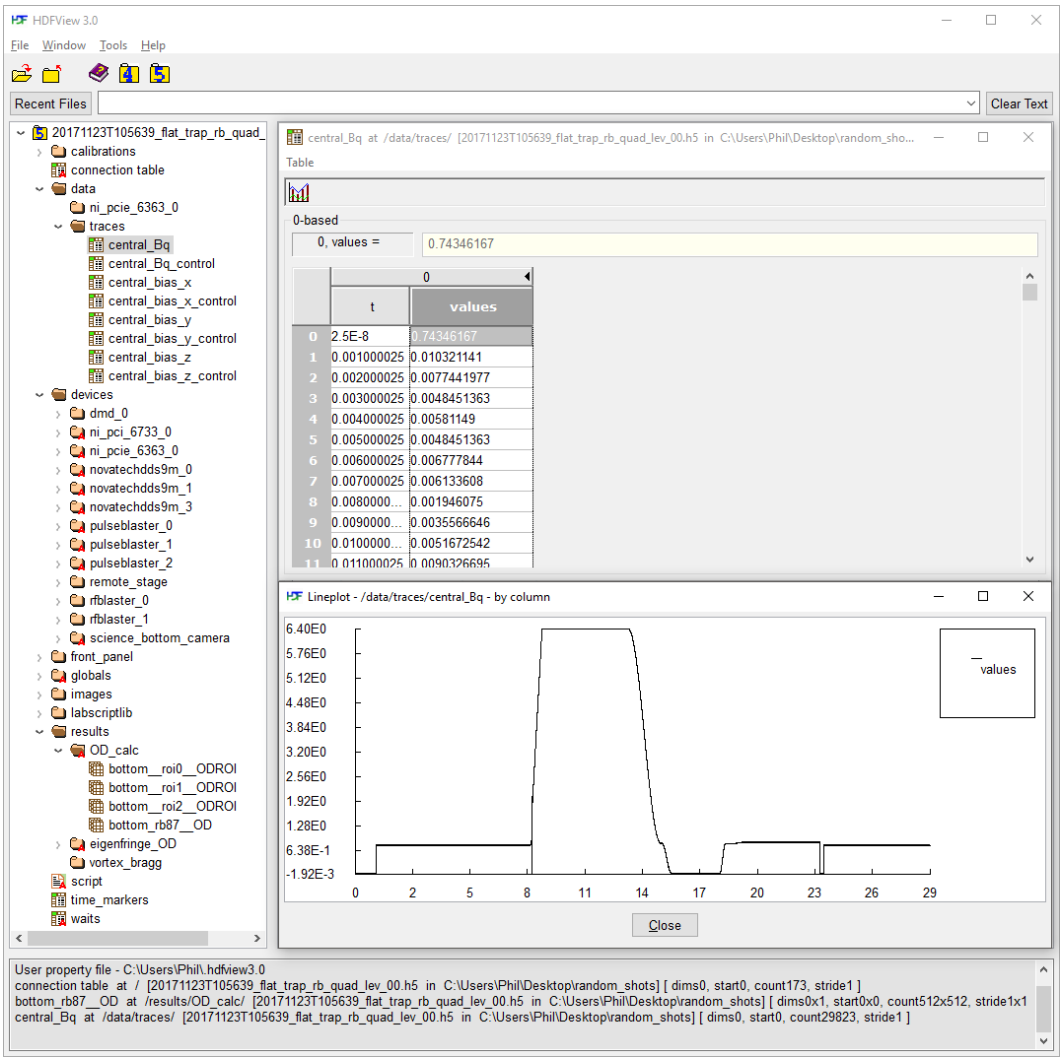


Figure 4.7: An example HDF5 file that has been produced by the labscript suite (created initially by runmanager before passing through all components, ending with lyse), displayed within the HDFView software. On the left, the hierarchical structure of the stored data is shown. We have expanded the 'data' group (and the 'traces' group within), the devices group, and the results group. The devices group contains one subgroup per hardware device used in the experiment (which themselves contain the hardware instructions). Similarly, the results group contains one subgroup per analysis script (which themselves contain saved results). The 'traces' subgroup (within 'data') contains acquired time series acquisitions. We have opened the raw data for the 'central_BQ' trace (which monitored the current through our quadrupole coils) in the top right window, and have displayed a plot of this (generated by HDFView) in the bottom right window. As can be seen, the HDF5 format allows for complex data to be arranged in an easy to read format.

(including runmanager parameters, experiment logic script, acquired data and analysis results) as the file progresses through the various programs in the suite. The structure of a typical HDF5 file produced by the labscript suite is shown in figure 4.4.2.

HDF5 has support across a wide variety of programming languages including common languages like C, C++, Python, LabVIEW, MATLAB, Mathematica, Java, IDL and .Net. When accessing HDF5 file from Python, we use the h5py library [140, 141]. This wide support for HDF5 allows labscript experiments to interact with other software (particularly custom software written by members of a given research group) with minimal effort, and aligns with our goals of maximal extensibility for the labscript suite.

4.4.3 ZeroMQ

For interprocess communication, we chose ZeroMQ (also known as ZMQ, 0MQ and ØMQ). Like HDF5, the ZeroMQ library can be used from a wide variety of programming languages including common languages like C, C++, Python, LabVIEW, Java, and .Net [142]. ZeroMQ support many different communication schemes. We typically use the request-reply model [143] however communication between the worker processes of the heterogeneous hardware in BLACS is delegated to a publish-subscribe model to remove interdependencies between device implementations.

In the labscript suite, ZeroMQ is used for three main tasks. The first, and simplest task, is sending the paths to HDF5 files between components of the labscript suite. We also use ZeroMQ in our custom multiprocessing implementation via the zprocess library we have created (see §4.4.5). Finally, we use ZeroMQ to ensure safe access to HDF5 files when they are accessed from multiple processes. HDF5 files can only be accessed serially, and may become corrupted if multiple processes attempt to access it. To prevent accidental corruption, we have monkey-patched a locking system, using ZeroMQ, over the top of the h5py library. This ensures that any h5py call from within our software first acquires a lock on the specific HDF5 file from a centralised ZeroMQ server, before attempting access.

4.4.4 Qt

As the labscript suite includes a significant graphical component, the choice of third party library for generating the graphical interface is particularly critical. Our primary criteria for this was cross-platform support (the ability to run on Windows, Linux and Mac OSX). Initially we chose GTK version 2 as our graphical toolkit, due to the cross-platform support and integrated tools for rapid graphical application development. As our project developed however, we became aware of a significant memory leak under Windows that we reported to the GTK developers [144]. While this was ultimately fixed, it became apparent that the GTK project had lost much of the knowledge regarding Windows support [145] (which was ultimately demonstrated with the significant delay for Windows support of GTK version 3). Compounding this, the maintainers of the Python language bindings for GTK version 2, PyGTK, were not actively building new versions that included the recent critical bug fixes. The result of this was that our complex programs, such as BLACS, would regularly hard-crash after 3 days of continuous operation. Given our aim of creating a control system suitable for remote operation and/or 24/7 use, this was a significant setback.

After much consideration, we decided to rewrite all of our software using the Qt graphical toolkit [146]. We initially chose PySide [147] for the Python language binding due to the freer licensing constraints, however I again discovered a serious memory leak [148] during the redevelopment of BLACS. PySide development had stalled in much the same way as PyGTK, so we changed the language bindings to PyQt4 [149] which only required minimal changes since the API for PyQt4 was very similar to PySide. Runmanager, lyse, and runviewer were then redeveloped solely for PyQt4. We now support PyQt4 and PyQt5 through an abstraction layer in qtutils, have tentative support for PySide, and aim to support PySide2 (now known as "Qt for Python" [150]) in future releases.

Our experience underlines the need to consider items beyond the feature set when choosing a graphical toolkit. The development processes, patch timeframe (of both the core project and language bindings) and community support for the toolkit must also be a critical factor in determining the toolkit to use. It is worth noting that both Qt and PyQt4/5 are developed and maintained by commercial companies, which we believe will help ensure the continuing success of those projects.

4.4.5 Multithreading and multiprocessing

The labscript suite heavily uses both multithreading and multiprocessing to parallelise the execution of code. Multiprocessing provides isolated containers (processes) for code to run in, and come with their own dedicated memory allocation. The PC operating system handles the allocation of resources and the sharing of the CPU(s) between processes. Multithreading is a parallelisation technique that exists within a single process. The operating system is still responsible for sharing the CPU between threads (and other processes), but other resources (like memory) are shared between all threads in a process.

Multithreading is often considered difficult or risky, due to race conditions, where the possibility exists of simultaneous access or modification of variables across threads, which can result in undefined behaviour of the program. While the risks can be mitigated via the use of synchronisation primitives, such as mutexes (locks) or semaphores, these add complexities to the system and come with risks of their own (such as mutex deadlocks). Python limits these risks somewhat by imposing the global interpreter lock (GIL), which effectively dynamically serialises multithreaded code. This eliminates the computational benefits of multithreading, but still provides encapsulation benefits for tasks that are I/O bound rather than CPU bound. However, even with these restrictions, many Python libraries are not thread-safe, including most graphical interface libraries.

In the labscript suite, we only use threads for tasks that we consider isolated. This include tasks that involve queues (generating or executing shots, or communicating with other processes) and/or can run in the background (such as forwarding shots for analysis to lyse). Such a structure results in the self-containment of a task within a single thread, reducing the risk of unforeseen race conditions. Of course, communication between threads is always necessary to some degree, for which we use Python queues, which are thread-safe. When a thread needs to control the GUI (for example to update a label with a status message) we use the thread-safe method of posting events to the Qt event loop (which runs in the Python main thread). Our library, qtutils [137], has abstracted this procedure away, allowing arbitrary Python functions and methods to be executed in the main thread (via

the Qt event loop) to ensure thread-safety. Where shared access to variables is required, we serialise access transparently by defining the variables as Python properties, and ensuring the defined get and set methods for the property are only called in the main thread via the previously described feature of qtutils. In rare cases where operations are not atomic[14], and we do not want to defer execution into a function to be executed in the main thread, we use Python locks or events to ensure thread-safe serialisation of the task.

We also heavily rely on the use of multiprocessing. While multiprocessing provides a potential speed-boost on multicore systems (since it does not suffer from the Python GIL), and does not share the risks of multithreading, we primarily use it as a means of sandboxing code. We created our own cross-platform multiprocessing library and integrated it into our zprocess library. Primarily authored by Chris Billington, zprocess provides convenience methods for launching Python classes or scripts in a separate Python process, and automatically sets up two way communication via ZeroMQ sockets. BLACS and lyse are the primary users of this approach, although runmanager also uses a separate process for shot generation.

In BLACS, we use a separate 'worker process' when interacting with each hardware device. This prevents the failure of one device from taking down the entire control interface. This is particularly important given that most devices interface using a third party Python library and/or DLL, both of which may exhibit memory leaks, or worse, cause the calling process (our control system) to 'segfault' (where the entire process unexpectedly, and immediately, terminates). By sandboxing these libraries in dedicated processes, we can present the user with debugging information regarding the crash and provide a means for restarting the failed process. The user also continues to maintain control over the remaining devices that are functioning correctly. This is particularly critical for complex experiments where not all hardware is in use in every experiment, as it prevents an unused, but still connected, device from taking down a working system.

In lyse, we use a similar sandboxing technique for each analysis process. Here however, the main benefit is the separation of figure managers for the set of plot windows associated with each analysis script. Similarly to BLACS, sandboxing prevents the failure of one script from taking down the rest of the analysis scripts (which would result in the closure of all plot windows, rather than just those associated with a single script).

4.5 Summary

In this chapter we introduced the labscript suite and discussed the underlying technologies and design themes we incorporated. We largely follow the Unix philosophy in our development. The labscript suite is made up of several distinct programs, which are designed to perform a specific aspect of the experiment lifecycle. We rapidly tested our development on

14. Not to be confused with the scientific use of the word atomic, in software engineering 'atomic' refers to the indivisibility of an instruction. In this case, we refer to the indivisibility of a Python command, which is defined by whether the Python GIL can switch between the current thread context being executed part way through a command. If the GIL cannot, then the command is atomic. If the GIL can switch the currently executing thread midway through the command, then it is not atomic. In such cases, data being used by the command could be modified by another thread part way through execution (if the GIL switches to it), which would result in undefined behaviour when the GIL returned execution to the command unless steps are taken to prevent this behaviour.

our apparatus, and ensured that the design was informed by how we used the control system to perform scientific experiments. Flexibility was also a key requirement of our development process, so that we could ensure our control system was useful to other laboratories, as well as future proofed for new and complex experiments that we researchers are yet to conceive. We also outlined our decision to balance graphical interfaces with textual programming interfaces, and contrasted this to some of the systems reviewed in chapter 2. We then discussed the modern technologies we used in the development of our software framework, specifically the use of the modern programming language Python, the cross-platform and cross-language libraries we use as an interface between components of the labscript suite, and the comprehensive use of multi-threading and multiprocessing to both speed up our framework and to ensure it is robust to failure. In the next chapters we will dive into the detail of how we implemented each component of the labscript suite, and continue to discuss how these choices help perform novel scientific experiments.

Chapter 5

Preparing experiments with the labscript suite

The labscript suite can be broken into two broad categories. The first is the preparation of an experiment, which we will discuss in this chapter. In the following chapter, we'll discuss the second category: execution of the experiment including automating the analysis of results.

The core labscript suite applications for preparing an experiment (labscript, runmanager, and runviewer) were introduced in the previous chapter. In this chapter, we'll discuss the implementation of these programs and how this leads to better control of an experiment.

5.1 Labscript API

The labscript API is the heart of the labscript suite, as it provides the interface for defining how the hardware will behave during an experiment[1]. The labscript API was designed around the premise that the set of heterogeneous hardware to control would be buffered; that is the hardware would be preprogrammed with a set of instructions that define the output states at a series of times. These hardware instructions would then be stepped through on the edges of a digital clocking signal from a pseudoclock. Hardware instructions are typically esoteric, thus difficult for humans to both create them, as well as read them, by hand. The aim of the labscript API is to provide a high level interface for lab users, which will in turn create the low level instructions needed for each piece of hardware, including the clocking signal.

Labscript is built around the use of a pseudoclock, or variable frequency clock. All labscript experiments must use at least one pseudoclock, called the master pseudoclock, which controls the timing of all other devices. In general, these other devices are stepped through a table of hardware instructions at a rate determined by the pseudoclock (see §2.2.3). However, for experiments with a large number of devices we also provide support for secondary pseudoclocks. Secondary pseudoclocks are synchronised to the master pseudoclock via an initial external trigger, but otherwise run independently of the master pseudoclock. This allows groups of devices to be split up across multiple pseudoclocks, minimising the num-

1. We use 'experiment' in this chapter as defined in §4.1.

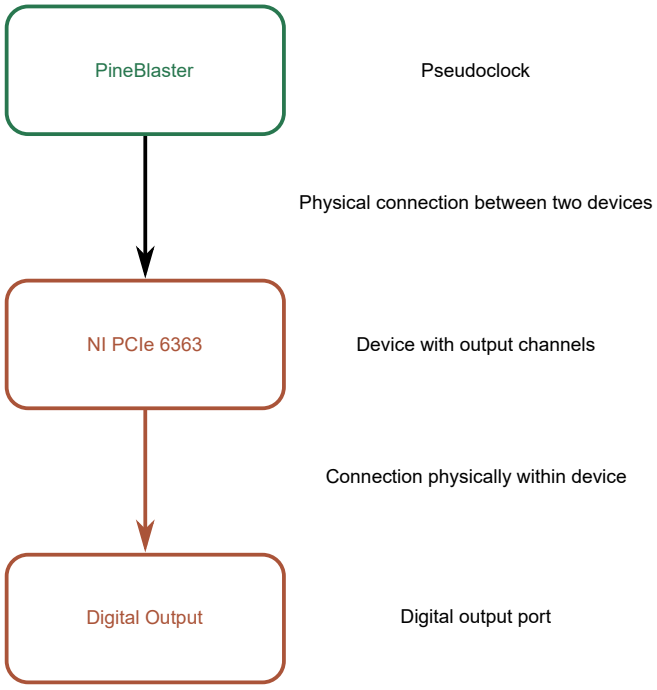


Figure 5.1: The simplest hierarchy of devices supported by labscript. Here, a PineBlaster pseudoclock (detailed in [8, 82, 83]) is connected to a NI general purpose output card, in this case with only a single digital output port configured for use. Arrows show the hierarchy of device control, and point from the controlling device that dictates timing events (parent) to the controlled device (child). The NI card will update the output state of the digital output on each rising edge of the clocking signal from the PineBlaster.

ber of unnecessary instructions in devices that do not require output state changes when another does. We discuss pseudoclocks further in §5.1.4 and §5.1.5.

The labscript suite asks users to define experiment logic inside a Python file (a Python script), using the labscript API. We describe these files as 'experiment logic files' or 'labscript files', and they are stored in a folder called labscriptlib, which is created during the labscript suite installation process. When this script is executed (by runmanager, see §5.2), the calls to the labscript API are executed and the hardware instructions are produced and saved in a file (to be programmed into the hardware at a later time). By convention, we associate experiments with their own independent Python file, although it is important to note that this file is free to import code from any other Python module including those created by the user (which may also utilise the labscript API). This allows common experiment logic to be easily shared between different experiments. The experiment logic file consists of two parts: the 'connection table', which we cover in §5.1.1, and the 'experiment logic', which we cover in §5.1.2. The features of the labscript API are then covered in sections §5.1.3 to §5.1.9

5.1.1 Connection table

The labscript API assumes a particular hierarchy of devices[2] and their connections. At the simplest level, labscript requires two devices: a single device for generating a clocking signal, known as the pseudoclock, and a device with output channels (such as digital or analog outputs) that is fed the clocking signal. Such a hierarchy is shown in figure 5.1. The pseudoclock thus controls the timing of each output, albeit indirectly. There are very few limitations[3] to the complexity of the hierarchy of devices labscript supports. For example, multiple output devices can be attached to a single pseudoclock, and these can, in turn, trigger one or more secondary pseudoclocks that may have their own output devices attached (see figure 5.2). This leads to an architecture that supports an almost unlimited set of devices, which is a very powerful tool for experiments (such as those using ultracold atoms) that are rapidly increasing in complexity.

As shown in figures 5.1 and 5.2, the hierarchy of devices and outputs naturally follows the lines of control as indicated by the arrows in those figures. For example, a pseudoclock controls the timing of other devices, which in turn control the state of their outputs. The natural way to represent this hierarchy of devices and I/O channels, from within a programming language, is to use an object-oriented approach where each object maintains references to its parent and children (as indicated by the aforementioned lines of control). We term this the 'hierarchy of control'. The labscript API defines a set of Python classes[4], which can be instantiated by the user (or by internal labscript code) to define this hierarchy as part of the connection table definition in the experiment logic file. Each class expects a reference to a parent object, along with a description of the connection to the parent, to be passed as arguments at object instantiation time. This defines the parent-child relationship, where parents guide their child objects, which in turn guide their children, *ad infinitum*. The hierarchy of Python objects is more detailed than the device hierarchy shown previously, although they do map to each other. An example is shown in figure 5.3 and the mapping of the Python objects to real hardware is shown in figure 5.4.

In addition to the hierarchy of control, there is an additional hierarchy of class inheritance that allows existing labscript functionality to be extended through subclasses (see figure 5.5). At the base level, every labscript object is an instance of the `Device` class, which provides basic management of the parent-child relationship between objects in the control hierarchy. Labscript then provides generic subclasses of `Device` for items in the control hierarchy, such as `IntermediateDevice`, `TriggerableDevice`, `Pseudoclock`, `ClockLine`, `AnalogIn`, and `Output`, which all inherit from `Device`. Some of these classes may also be subclassed further by labscript, for example to implement specific output types such as `DigitalOut` and `AnalogOut`. These each provide a general purpose implementation of functionality for their level of control (as introduced in figure 5.3). For example, `Pseudoclock` contains

2. It is important to note that we use the word 'device' loosely here. A device does not necessarily have to correspond to one physical hardware device, a point on which we will elaborate shortly.
3. The current known limitations are artificial and relate to the connection of the 'wait monitor', and support for tertiary, or higher, pseudoclocks. These restrictions will likely be removed in the next major version of labscript.
4. For readers unfamiliar with object oriented programming in Python, a 'class' is, loosely, a template for a data structure and associated functions that are to act on that data. Instantiating a class creates an object that follows the template defined by the class. Objects are thus known as 'instances' of a class, and multiple instances of a single class are allowed (for example two physical PulseBlaster devices could be represented by two separate instances of the `PulseBlaster` class).

Figure 5.2: A more complex version of the device hierarchy supported by labscript. Here, the clocking signal from the PineBlaster is simultaneously fed into three separate NI cards. These cards are configured with a variety of digital and analog outputs. As in figure 5.1, the NI cards will update the state of each output on the rising edge of the clocking signal from the PineBlaster, as shown by the arrows that indicate the hierarchy of control. Digital output ports can also be configured as 'Triggers', which can be connected to other pseudoclock like devices. In this example, one of the digital outputs of the first NI card is configured as a trigger, and connected to a PulseBlaster. The PulseBlaster operates as an independent pseudoclock to the PineBlaster, but is synchronised by a pulse at the start of the shot. In turn, the PulseBlaster can provide a clocking signal to other devices, in this case three Novatech DDS9m devices, which can each have up to two DDS outputs configured whose output state is updated by the clocking signal (in addition to two static DDS outputs that can only be set once at the start of a shot). As the PulseBlaster has multiple clocking outputs, these can be used to segregate devices into groups, which can be beneficial for reducing the required number of instructions to be programmed into the devices (see §5.1.4 and §5.1.5).

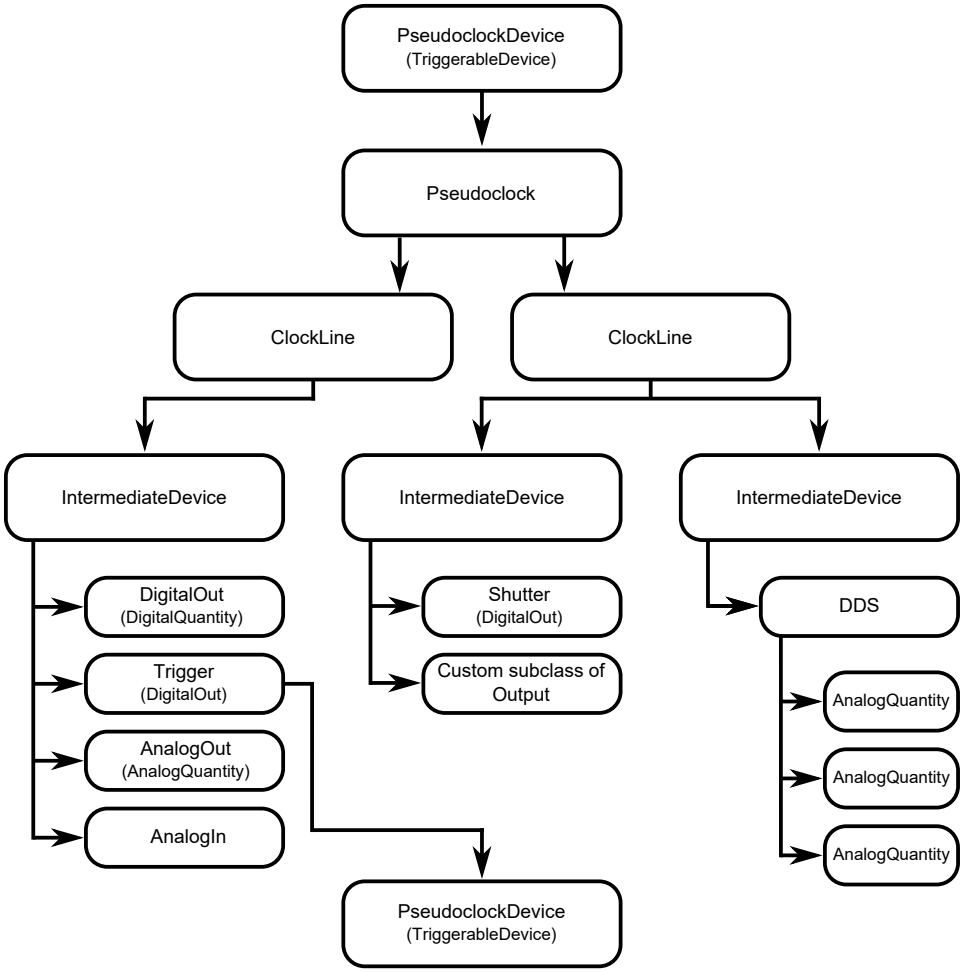


Figure 5.3: An example hierarchy of the Python objects in a connection table definition. The labscript control hierarchy always follows the pattern of a `PseudoclockDevice` with one of more `Pseudoclock` children, which in turn have one or more `ClockLine` children. Each `ClockLine` object can then have one or more `IntermediateDevice` attached. An `IntermediateDevice` then has one or more children that represent an input or output channel of a device. We show how the hierarchy of control maps to hardware devices in figure 5.4. `PseudoclockDevice`, `IntermediateDevice`, and the `AnalogQuantity` classes are not directly used in an experiment logic file, but are subclassed when adding support for new devices in order to implement specific device or I/O behaviour. All objects derive from Python classes that subclass `Device`, which provides basic parent child relationship management between the objects. For devices that indirectly subclass `Device`, we show the parent class in brackets (with the exception of `AnalogQuantity` and `DigitalQuantity`, which have a parent class of `Output` that in turn is a subclass of `Device`). The full hierarchy of class inheritance is shown in figure 5.5. Note that in practice, many of the classes shown in this figure would be replaced with a device specific subclass when constructing the connection table. We discuss subclassing of labscript classes in §7.1. It should also be noted that device specific code may also internally create some of the control hierarchy so that users are not required to explicitly instantiate objects they do not need to access.

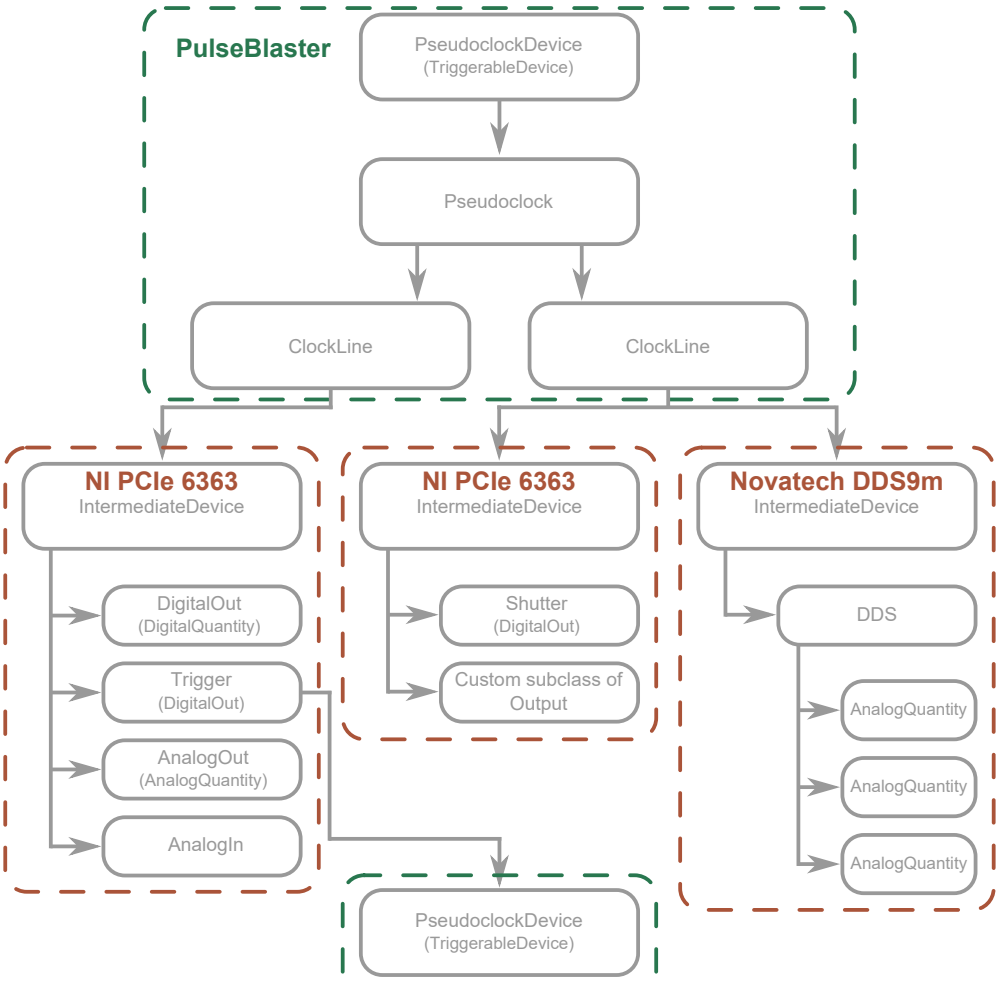


Figure 5.4: Here we show how the Python classes provided by the labscript API (shown in figure 5.3) map to real hardware. Additional devices (with complex control hierarchies) can be connected via secondary pseudoclocks, as indicated by the unbounded and unlabelled `PseudoclockDevice` at the bottom of the figure. Note that PulseBlasters (and other similar devices) may be configured to use digital outputs (and DDSs on some models) as direct outputs (in a similar way to those attached to the NI or Novatech devices). In this case, the `PulseBlaster` device code (provided in the labscript suite) internally creates a `ClockLine` and `IntermediateDevice` object (exposed via the `direct_outputs` attribute of a `PulseBlaster` object) to connect the `DigitalOut` or `DDS` outputs to. This ensures all objects correctly follow the prescribed hierarchy by labscript, even if the hardware is not physically separable into pseudoclock devices and devices that are clocked by a pseudoclock. For brevity, these internally created objects are not pictured.

the algorithm for generating a representation of a pseudoclock signal, based on how the child outputs are commanded, which can be later converted into hardware instructions for a specific device. The subclasses provided by labscript can then be subclassed again by developers when implementing device specific behaviour (see §7.1). Ultimately it is these device specific subclasses, along with the labscript provided I/O classes that are instantiated as part of the connection table (see figure 5.6).

Maintaining a record of the hardware configuration in a lab is key to good record keeping.

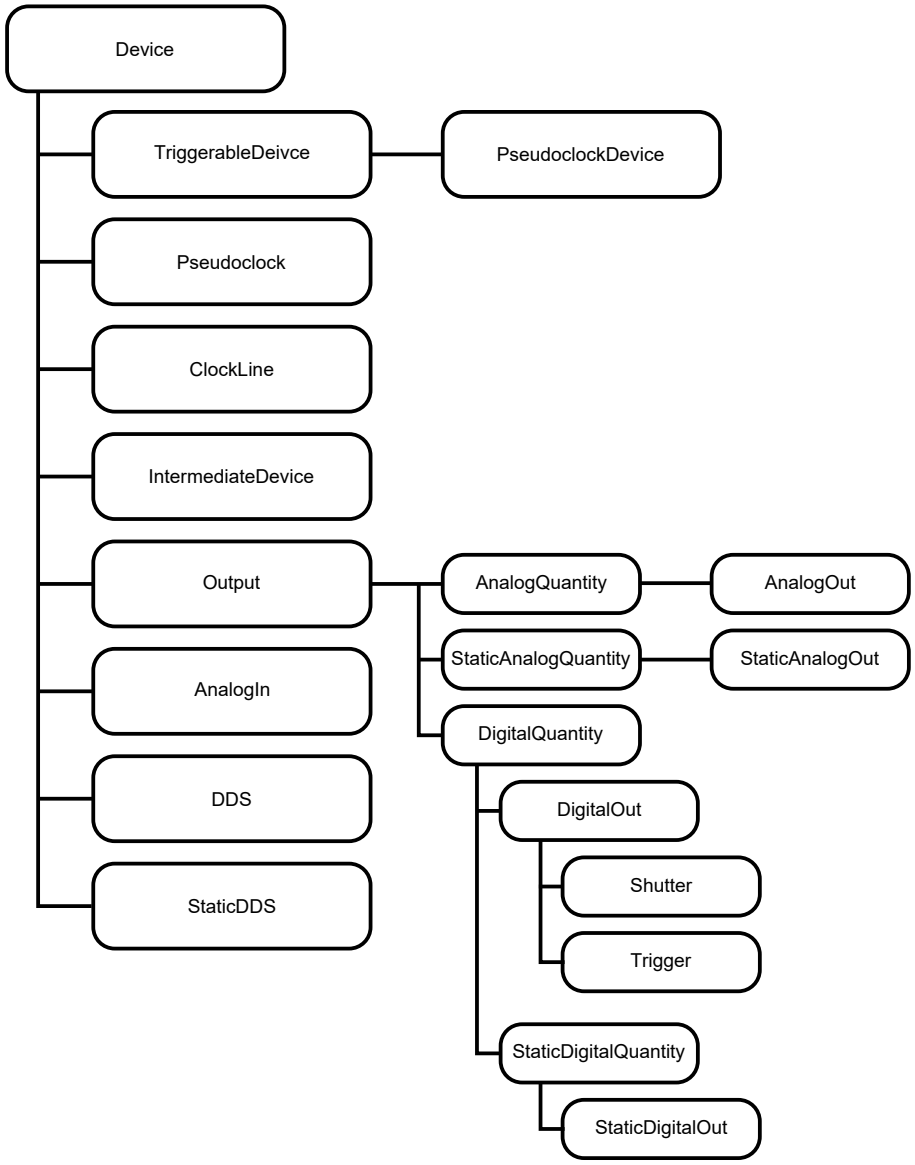


Figure 5.5: The hierarchy of class inheritance within the labscript API. All classes pictured derive (inherit) from the `Device` class. Classes `PseudoclockDevice` and `IntermediateDevice` are subclassed by developers when implementing support for new hardware devices.

Labscript automatically writes a record of the object control hierarchy into each shot file, for any object that derives from the `Device` class. Not only does this record allow you to look back at how hardware was previously configured, it provides framework for determining whether an old experiment can run. This is particularly critical for experiments that have sensitive and/or expensive equipment connected. By maintaining a record of how each device is connected together, and what each output and input is used for, it becomes possible to prevent old experiment logic (written for a previous configuration of the hardware) from

```
# Create the PseudoclockDevice (with internal Pseudoclock)
# and two clocklines
PulseBlaster(name='pulseblaster_0', board_number=0)
ClockLine(name='pulseblaster_0_clock1',
          pseudoclock=pulseblaster_0.pseudoclock,
          connection='flag 0')
ClockLine(name='pulseblaster_0_clock2',
          pseudoclock=pulseblaster_0.pseudoclock,
          connection='flag 1')

# create the NI card (that has its own clockline)
# and the attached I/O
NI_PCIe_6363(name='ni_pcie_6363_0',
             parent_device=pulseblaster_0_clock1,
             clock_terminal='/ni_pcie_6363_0/PFI0',
             MAX_name='ni_pcie_6363_0',
             acquisition_rate=1e3)
DigitalOut('flipper_mirror', ni_pcie_6363_0, 'port0/line0')
AnalogOut ('bias_coil_x',    ni_pcie_6363_0, 'ao1')
AnalogIn  ('bias_x_field',   ni_pcie_6363_0, 'ai13')

# create two devices (that share a single a single clockline)
# and the attached I/O
NI_PCIe_6363 (name='ni_pcie_6363_1',
              parent_device=pulseblaster_0_clock2,
              clock_terminal='/ni_pcie_6363_1/PFI0',
              MAX_name='ni_pcie_6363_1',
              acquisition_rate=2e3)
NovaTechDDS9M(name='novatechdds9m_0',
              parent_device=pulseblaster_0_clock2,
              com_port='com1')
Shutter('imaging_shutter', ni_pcie_6363_1,   'port0/line0')
DDS    ('imaging_AOM',     novatechdds9m_0s, 'channel 0')
```

connection table at / [20180322T143015_connection_table_HDF5_example_0.h5 in C:\labscript_suite\Experiments\labscript27\connection_table_HDF5_example\2018\03\...

Table

0-based

0, unit conversion params = Content-Type: application/json {}

	0						
	name	class	parent	parent port	unit conversion class	unit conversion params	BLACS_connection
0	bias_coil_x	AnalogOut	ni_pcie_6363_0	ao1	None	Content-Type: applica...	
1	bias_x_field	AnalogIn	ni_pcie_6363_0	ai13	None	Content-Type: applica...	
2	flipper_mirror	DigitalOut	ni_pcie_6363_0	port0/line0	None	Content-Type: applica...	
3	imaging_AOM	DDS	novatechdds9m_0	channel 0	None	Content-Type: applica...	
4	imaging_AOM_amp	AnalogQuantity	imaging_AOM	amp	NovaTechDDS9mA...	Content-Type: applica...	
5	imaging_AOM_freq	AnalogQuantity	imaging_AOM	freq	NovaTechDDS9mFr...	Content-Type: applica...	
6	imaging_AOM_phase	AnalogQuantity	imaging_AOM	phase	None	Content-Type: applica...	
7	imaging_shutter	Shutter	ni_pcie_6363_1	port0/line0	None	Content-Type: applica...	
8	ni_pcie_6363_0	NI_PCIe_6363	pulseblaster_0_clock1	internal	None	Content-Type: applica...	ni_pcie_6363_0
9	ni_pcie_6363_1	NI_PCIe_6363	pulseblaster_0_clock2	internal	None	Content-Type: applica...	ni_pcie_6363_1
10	novatechdds9m_0	NovaTechDDS9M	pulseblaster_0_clock2	internal	None	Content-Type: applica...	com1,115200
11	pulseblaster_0	PulseBlaster	None	None	None	Content-Type: applica...	0
12	pulseblaster_0_clock1	ClockLine	pulseblaster_0_pseudoclock	flag 0	None	Content-Type: applica...	
13	pulseblaster_0_clock2	ClockLine	pulseblaster_0_pseudoclock	flag 1	None	Content-Type: applica...	
14	pulseblaster_0_direct_...	ClockLine	pulseblaster_0_pseudoclock	internal	None	Content-Type: applica...	
15	pulseblaster_0_direct_...	PulseBlasterDir...	pulseblaster_0_direct_output_clock_line	internal	None	Content-Type: applica...	
16	pulseblaster_0_pseud...	Pseudoclock	pulseblaster_0	clock	None	Content-Type: applica...	

Figure 5.6: Top: The Python code required to define the hierarchy of classes shown in figures 5.3 and 5.4. Note that we have not included the `Trigger` and additional `PseudoclockDevice` or the custom subclass of `Output`. Bottom: The connection table as stored in the HDF5 file (displayed in HDFView). You will notice entries for the internally created objects, such as the PulseBlaster pseudoclock and DDS analog quantities.

running and damaging equipment. Indeed we implement such a check in BLACS (see §6.1.2 for details on how the connection table, and the device configuration information, is verified).

5.1.2 Experiment logic

Given a set of labscript objects corresponding to the hardware devices and their output channels (see details on constructing a connection table in §5.1.1), a user is now at a point where they can define experiment logic. Each output channel object is an instance of, or inherits from, one of several labscript classes such as `DigitalOut` or `DDS`, which have a set of methods designed specifically for commanding output. These methods always take a time parameter, and for outputs that are analog in nature, an output value parameter and an optional unit (see §5.1.9 for further details on units). The time and value parameter are always specified in SI units (unless a non-SI unit is explicitly specified for the value) and the method names are chosen specifically for readability. For example, objects that inherit from `Shutter` have `open` and `close` methods, `DDS` objects contain `setfreq`, `setamp`, `setphase`, `enable` and `disable` methods, and `DigitalOut` objects have `go_high` and `go_low` methods. This results in commands that look like `imaging_aom.setfreq(t, 83.5, "MHz")` or `imaging_shutter.open(t)`. Thus, each command defining experiment logic is a very readable line of code, despite being written in a general purpose programming language. We believe this lowers the entry barrier for new lab members, as they don't need to wrap their heads around low level grids of dots (like line-based systems) or unreadable code. It also makes debugging faster, and more accurate, for experienced lab users.

The example output commands presented thus far are quite simple, defining only a single change in output value at the specified time. While a user could combine these simple labscript methods with the general purpose control flow statements (such as `for` loops and function definitions) to construct more complex output commands, this is cumbersome and difficult to optimise for use with a general purpose pseudoclock if the user intends to have multiple complex output commands that overlap in time. We demonstrate two such 'poor' examples in figures 5.7(a) and 5.7(b). To improve upon this, the labscript API instead provides methods for *ramping* outputs that are analog in nature. Methods are provided for many common use cases (such as linear, quarter-sine and exponential ramps), however it is also possible to define a custom ramp profile simply by passing a Python function (one that defines the profile as a function of time) to the appropriate labscript API method. An example use of the linear ramp in shown in figure 5.7(c). The key benefit of these inbuilt ramping methods is they delay evaluation of the output state until all experiment logic has been defined. This ensures labscript has all necessary information to correctly segment overlapping ramps and dynamically determine the appropriate sample rate for each ramp segment based on the maximum sample rate requested across concurrent ramps. See §5.1.4 for further details on the internal workings of the labscript pseudoclock generation. Ramping methods thus typically take the parameters time, duration, initial and final values, sample rate and an optional unit, as well as any other parameters required to fully define the output profile (such as an exponential decay constant). Again, this results in very readable code (as shown in figure 5.7(c)).

We have previously discussed the general benefits of textual vs. graphical systems for defining experiment logic (see §4.3.3). We can now present concrete examples of these

```
initial     = 3.0 # Amps
final       = 6.0 # Amps
duration    = 2.0 # seconds
sample_rate = 1e3 # 1 KHz
# linearly ramp the bias_x coil from 3 to 6 amps over the length of
# time specified by duration, with the specified sample rate
for dt in linspace(0, duration, duration*sample_rate):
    bias_coil_x.constant(t+dt, ((final-initial)/duration)*dt+initial,
                         units='A')
```

(a) A bad example of 'ramping' an output. Here, the user has to manually create code that defines the functional form of the ramp, produce the grid of time points at which to evaluate the ramp and iterate over this in order to instruct labscript to produce the required output. Furthermore, it would be difficult for the labscript API to distinguish between a ramp of this form and repeated calls to set the value of the output that were unrelated to each other. Producing an optimised pseudoclock from even this simple example would thus be computationally intensive.

```
# ramp 1 parameters
t_initial      = 1.0 # seconds
initial        = 3.0 # Amps
final          = 6.0 # Amps
duration       = 2.0 # seconds
sample_rate    = 1e3 # 1 KHz
ramp_m = (final-initial)/duration # gradient of ramp 1
# ramp 2 parameters
t_initial2   = 1.5  # seconds
initial2     = 82.0 # MHz
final2       = 84.0 # Amps
duration2    = 1.0  # seconds
sample_rate2 = 2e3  # 1 KHz

# linearly ramp the bias_x coil from 3 to 6 amps over the length of
# time specified by duration, with the specified sample rate
t = t_initial
seg1_d = t_initial2 - t_initial # duration of segment 1
for dt in linspace(0, seg1_d, seg1_d*sample_rate, endpoint=False):
    bias_coil_x.constant(t+dt, ramp_m*dt+initial, units='A')

t += seg1_d
max_rate = max(sample_rate, sample_rate2)
for dt in linspace(0, duration2, duration*max_rate, endpoint=False):
    bias_coil_x.constant(t+dt, ramp_m*(dt+seg1_d)+initial, units='A')
    imaging_AOM.setfreq (t+dt, ((final2-initial2)/duration2)*dt+
                  initial2, units='MHz')

t += duration2
# set endpoint of imaging_AOM
imaging_AOM.setfreq (t, final, units='MHz')
seg3_d = (t_initial+duration)-t # duration of segment 3
for dt in linspace(0, seg3_d, seg3_d*sample_rate):
    bias_coil_x.constant(t+dt, ramp_m*(dt+(duration-seg3_d))+initial,
                         units='A')
```

(b) Continuing the theme of bad examples, this shows how you might write code for two overlapping ramps, using the approach shown in (a). Note that this example does not work if one ramp was not entirely contained by the other or if you want to have more than two ramps overlapping. Generalising this approach produces even more complex code, that would need to be written in each place in the script where you required overlapping ramps.

```
bias_coil_x.ramp(t=1, duration=2, initial=3, final=6,
                 sample_rate=1e3, units='A')

imaging_AOM.frequency.ramp(t=1.5, duration=1, initial=82, final=84,
                           sample_rate=2e3, units='A')
```

(c) An example of how the advanced labscript API allows for the simple definition of complex overlapping ramps of outputs. Here, we reproduce all of the logic of (b) in a far more readable (and shorter) block of code. It can be immediately seen from the experiment logic how the ramps are parameterised, their functional form (based on the method name, e.g. 'ramp' for linear), and how they overlap. The labscript API internally splits the ramps up into sections based on the times at which they overlap and automatically produces the appropriate grid of time points based on the maximum sample rate requested at any given time. Furthermore, the labscript API is able to use the parametrisation of the ramps in order to efficiently determine the required pseudoclock instructions necessary to generate the specified output.

Figure 5.7

benefits as demonstrated by the labscript API. As the experiment logic is written in a general purpose programming language, the user is given access (by default) to all control flow tools available in that language. Conditional logic is thus as simple as calling different labscript API methods from within the `if`/`else` blocks of an `if` statement. If the conditional expression is defined as a runmanager global variable (see §5.1.3 and §5.2), the logic of an experiment can be changed shot-to-shot simply by toggling a Boolean variable from within the graphical interface of runmanager. If a sequence of commands needs to be repeated (and is not suited to analog ramps as described above), they can be easily put inside a `for` or `while` loop. An example of such a sequence is show in figure 5.8. Experiment logic can also be easily commented out, put into reusable functions or copy-pasted at will. Textual definition of experiment logic naturally lends itself to supporting a non-linear temporal definition of output commands. While a text document is by nature linear (sequential lines), the use of labscript API methods do not need to be in sequential time order. This allows the user to call the labscript methods in the order that makes logical sense to a reader, again increasing the readability of the logic. For example, shutters often have opening and closing delays, requiring the digital trigger to be issued prior to the requested open or close time. Similarly, a camera exposure may begin significantly before the event of interest in order to minimise the subsequent inter-frame time. Using the labscript API, the command to expose could be issued as `camera.expose(t-exposure+event_duration, ...)` and be physically located in the text file, next to experiment logic that triggers the event of interest. Alternatively, this behaviour can be incorporated internally within a labscript API class, as is the case for the `Shutter`. The `Shutter` class takes an open and close delay as an optional argument when it is instantiated. Then when the `open` and `close` methods are called at time `t`, the digital pulses are actually offset from `t` automatically such that the shutter is physically opened or closed by the requested time (this can be seen graphically within runviewer, see figure 5.21).

```
# now the source MOT should be on, along with the central MOT
# we begin the pushing sequence:
if verbose:
    print "Starting Rb MOT load at t = %s"%t
# temporarily duplicate "t"
rb_load_t = t
# iteratively load the Rb central MOT
while (rb_load_t + rb_source_MOT_load_time + rb_push_duration
       < t + rb_central_MOT_load_time):
    # let the source MOT load
    rb_load_t += rb_source_MOT_load_time

    # turn off trap light to repump while we push,
    # and turn on push beam
    rb_source_MOT_trap_aom.disable(rb_load_t)
    rb_imaging_push_aom.enable(rb_load_t)

    #wait for the push duration
    rb_load_t += rb_push_duration

    #turn the push off and the MOT back on
    rb_source_MOT_trap_aom.enable(rb_load_t)
    rb_imaging_push_aom.disable(rb_load_t)

#load is now complete
#turn off source
rb_source_MOT_trap_aom.disable(rb_load_t)
rb_source_MOT_repump_aom.disable(rb_load_t)
rb_source_MOT_shutter.close(rb_load_t)
t=rb_load_t
```

Figure 5.8: An example of how standard control statements from a programming language can be used to easily augment experiment logic. Here we show a segment of code from the experiment logic of our dual-species BEC apparatus at Monash University, where we repeatedly load the central rubidium MOT from the rubidium source MOT. This loops the push sequence as many times as will fit within the time defined by the `rb_central_MOT_load_time` global, where the length of a single push sequence is defined by the sum of `rb_source_MOT_load_time` (the length of time it takes to load the source MOT from background vapour) and `rb_push_duration` (the length of time we push for in each iteration).

5.1.3 Global variables

Labscript inserts global variables (defined in runmanager) into a global Python namespace. These global variables are read from a HDF5 shot file (typically created by runmanager). To ensure these global variables are not only accessible to the experiment logic script, but also to any script that the experiment logic script imports, we inject the global variables into the Python `__builtin__` module. This module traditionally houses the basic built-in items of Python (such as standard Exceptions, inbuilt data types and a set of standard functions for working with these data types, see [151]) and is, traditionally, not intended to be modified (although modification is not forbidden). By inserting global variables here, we remove the need for the user to do anything special to access the global variables they have defined, reducing the barrier to entry.

It is also worth noting that we perform an identical trick when defining devices and channels in the connection table. In order to demonstrate why this is necessary, consider the following example of how we would have implemented the connection table definition if we had followed the standard Python format for object instantiation: `pulse_blaster_1 = PulseBlaster(board=0)`. In this scenario, while the PulseBlaster object is assigned to a named variable, the internal methods of this object have no access to the name of the variable it was assigned to. As such, there is no record of a device name. If you wished to use this device inside an imported function, you would also need to manually pass a reference to this device as one of the function arguments (which becomes significantly unwieldy if you abstract to the level of a function like `make_bec(t)` which would then require you pass almost everything in your connection table as arguments to the function!). Instead (as shown in §5.1.1) we pass the name in as a string and don't directly assign the object to a variable in the experiment logic script (for example `PulseBlaster('pulse_blaster_1', board=0)`). However, internally (at instantiation time), we inject a reference to the created object directly into the `__builtin__` namespace so that it is accessible through a variable with the name passed in to the class constructor, in exactly the same way as global variables from runmanager are made accessible. While this represents the most significant departure from Python standard practice in the entire labscript suite, it is still implemented using inbuilt Python tools and mimics the end-result of the standard Python syntax.

5.1.4 Pseudoclocks

As discussed previously, labscript is built around the use of a pseudoclock (and indeed, requires one at the top of the device tree). Labscript was specifically designed to automatically create the required pseudoclock instructions, based on the experiment logic commanded by the user, in order to make it easier to define complex experiment logic.

By default, labscript assumes that pseudoclocks support complex hardware instructions (such as looping instructions, see §2.2.3)[5]. This makes it much simpler to manage the generation of pseudoclock instructions for the 'ramping' functions introduced in §5.1.2. As each requested output ramp has a specified sample rate, it is implied that there is a set of output instructions that will be stepped through at a constant rate for the duration of the ramp. This then maps trivially to a pseudoclock instruction that ticks N times where N can be determined from the length of the ramp and the sample rate.

Of course, as intimated in §5.1.2, labscript supports simultaneous ramps, on multiple channels (that are clocked by the same pseudoclock), that may partially (or fully) overlap in time. The generation of the pseudoclock becomes significantly harder in such cases, but is key to providing the flexibility we demand. We use the following algorithm to generate the pseudoclock instructions, which successfully handles many complex cases (a flowchart of this process is also shown in figure 5.9):

1. A pseudoclock object requests, from all outputs attached to all devices clocked by the pseudoclock, a list of times at which the output state changes. We call these the

5. This is not a strict requirement since it is a simple computation to expand a looping set of hardware instruction to a linear set of hardware instructions. However, most devices without support for complex hardware instructions do not have sufficient memory for holding the linear set of hardware instructions (or are more expensive).

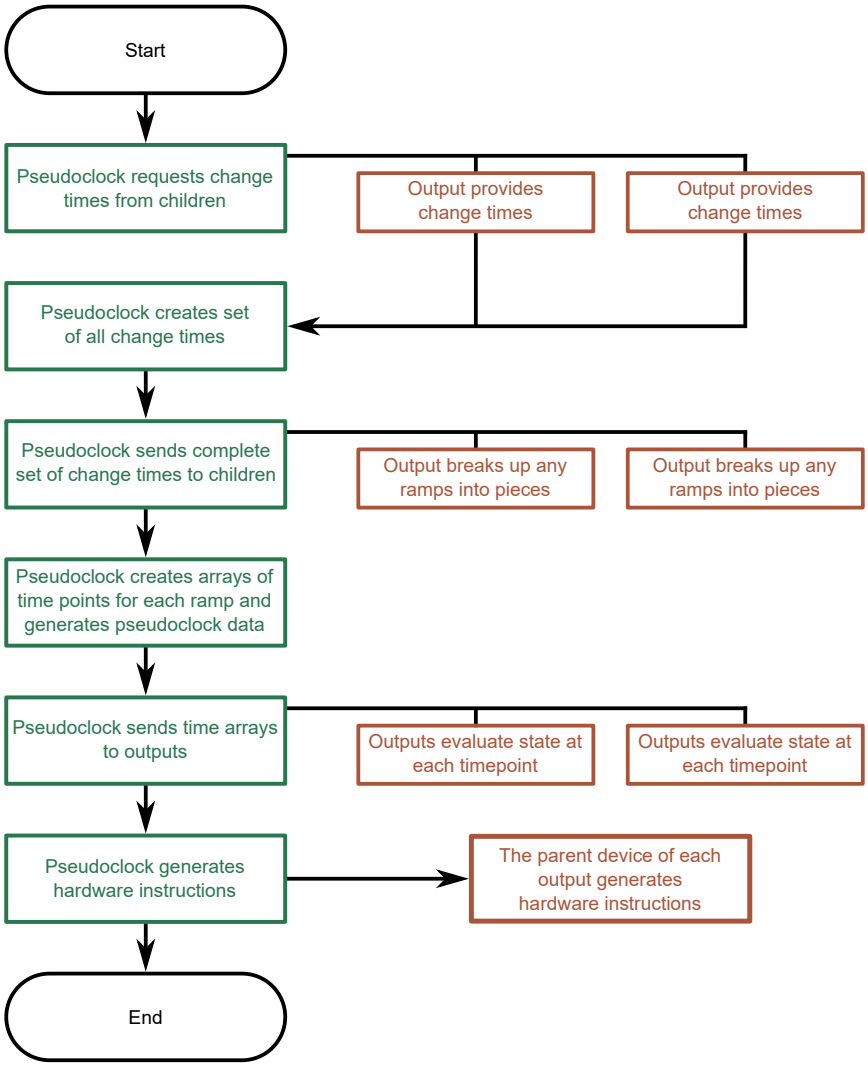


Figure 5.9: A flowchart of the pseudoclock generation algorithm. See main body text of §5.1.4 for further details.

'change times'. At this stage, ramps are only represented by a single entry, corresponding to the start of the ramp. The request is made by calling the `get_change_times()` method on each output object (which the pseudoclock holds a reference too). This method performs a check to ensure that requested output for a given channel does not conflict with itself (for instance, that you have not attempted to command output from a channel at a time when output has already been commanded from that same channel at that same time).

2. The pseudoclock next produces the set of all change times, and ensures there is a change time for the start and end of the shot as well as for any triggers. Change times are then quantised to the timing resolution of the pseudoclock, to minimise timing errors during future calculations. Checks are performed on this set of change times

to ensure that no change time is too close to another, which is defined by the device with the slowest update rate that is attached to the pseudoclock. We also check to ensure no output was commanded after the specified stop time.

3. The set of all change times is then passed back to each output object via a call to the method `make_timeseries()`. This allows each output object to break up any ramps that may intersect with an update on another channel (in the same way that ramps are represented as 1 instruction prior to this step, if ramps must be broken up into segments, each segment is represented by only 1 instruction for now). The result is that each output attached to the pseudoclock now has the same number of instruction as all of the others. Each instruction now also corresponds directly to each element of the change times set, which allows the pseudoclock to ascertain the type of instruction at each point in time.

4. The pseudoclock then iterates over each time point in the change times set. At each point, the pseudoclock looks for instructions on output channels that require a ramp. If found, the pseudoclock takes the maximum rate out of all ramps at the current point in time, and generates an array of clock ticks that is added to a list of times at which the clock ticks (which we call the set of 'all times'). Note that this ramp may only be a portion of the commanded ramp if the ramp was split in step 3. This ensures that overlapping ramps only update at the faster of the specified sample rates, during the overlapping segments. Checks are performed to ensure that the required sample rate does not exceed the capabilities of the pseudoclock or the slowest device (as described in step 2). If no ramps are found for a given time point, then just a single value is added to the set of all times.

   We also, for each time point, generate a set of Python dictionaries that contain information about the required pseudoclock instruction at each step. We call these pseudo-pseudoclock instructions. These instructions are formatted in a device agnostic way and contain information such as the time of the instruction, the time step between clock ticks, and the number of clock ticks to produce.

5. The set of all times, which is a list that contains elements that are either single time points (for single state changes) or an array of time points (for ramps), is then passed to each output object via a call to the method `expand_timeseries()`. Each output object then evaluates the output state at each time point, which may involve evaluating the function that defines a ramp at a set of time points, or duplicating a single value across multiple time points (for example, if a ramp was present on another channel). The output states are then stored in the `raw_output` attribute of each output object.

6. All information about the output states have now been produced. The code specific to the pseudoclock device implementation now runs to convert the pseudo-pseudoclock instructions into hardware specific instructions for the device in use, and these are stored in the HDF5 file (see sections §5.1.8 and §7.1.1). Output devices also collate the instructions from the attached output channel objects, and these are also stored in the HDF5 file.

While the algorithm is quite complex, the object oriented design of labscript allows users to reuse our code trivially in their own pseudoclock device implementations. This brings both simplicity and consistency to the implementation of pseudoclocks in scientific experiments. Encapsulating this algorithm within an object hierarchy also makes the implementation of secondary pseudoclocks possible. Once the master pseudoclock has been generated, the identical algorithm is run for all secondary pseudoclocks (with a minor timing offset to account for triggering delays of secondary pseudoclocks).

5.1.5 Gated clocks

There are some obvious limitations to the pseudoclock instruction generating algorithm described in the previous section. Most notably, the fact that the global update rate is limited by the device with the slowest update rate. While this can be solved by splitting devices across multiple pseudoclocks, such an approach requires more devices, which may encroach on cost or space limitations of certain projects. In discussion with Ian Spielman (NIST) and the Monash development team, I developed a modified algorithm, which we call 'gated clocks', that bypasses this limitation[6].

Some pseudoclock devices (for example, the PulseBlaster) can command multiple digital outputs. While these cannot be used as independent pseudoclocks (since they do not have independent instruction sets per output), they can be used to avoid the update rate limitations of slow devices for the majority of an experiment shot. We conceptualised this through the `ClockLine` class previously introduced as part of the labscript API object hierarchy. By attaching devices to specific clocklines, we have a programmatic means of separating these devices in labscript. The algorithm for generating pseudoclock instructions is then able to generate additional metadata (to be used by the pseudoclock device specific code, see §7.1.1), which indicates the set of clocklines that must tick for any given pseudoclock instruction (and by inference, the set of clocklines that should not tick). We can thus bypass the update rate restrictions imposed by slow devices in time periods where the outputs of slow devices do not need to update. The maximum update rate at any given time is thus restricted only by the slowest update rate out of all devices that update their output state at that time (see figure 5.10).

5.1.6 Controlling data acquisition

So far, we have only discussed control of outputs. However, an integral part of any experiment is, of course, making a measurement. For experiments under computer control, this is done by acquiring one or more sets of data using acquisition devices. These devices may be dedicated, or incorporated into devices that also support output and, at least for ultracold atom experiments, can be broken into two categories: devices that acquire analog time series, and cameras that acquire one or more images[7].

6. This modification is one of the major feature improvements since the labscript suite was published.
7. The labscript suite does not currently support digital inputs, as there has not yet been sufficient demand. We expect to eventually add support, but in a pinch, analog inputs can of course acquire digital signals as well.

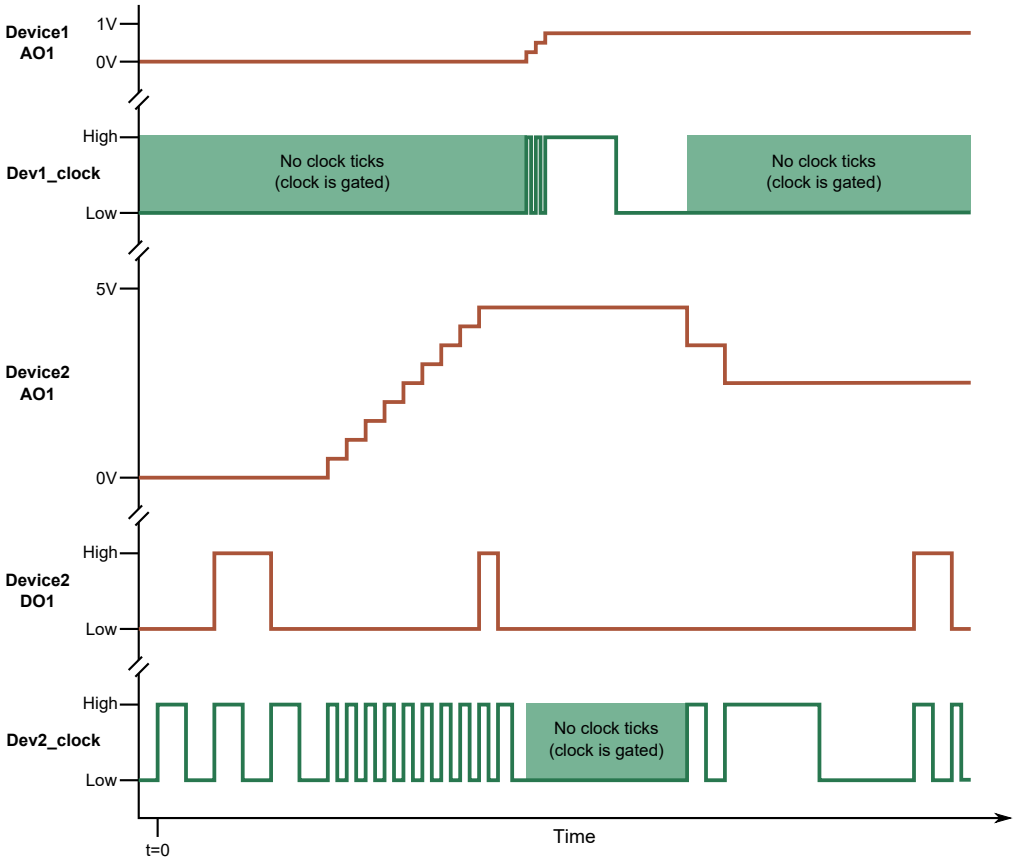


Figure 5.10: An example of gated clocks. Here we have two devices: `Device1` with a single analog output called `Device1AO1`, and `Device2` with an analog output `Device2AO1` and a digital output `Device2DO1`. The maximum update rate of `Device1` is twice that of `Device2`. The two devices are clocked by separate clocklines `Dev1_clock` and `Dev2_clock` respectively. In this scenario, we assume that the first ramp of `Device2AO1` occurs at the maximum update rate for the device and thus half of the maximum update rate for `Device1` We 'gate' each clockline (shaded area) at times where the attached outputs do not update their state. This allows for clockline `Dev1_clock` to tick faster than the limit imposed by `Device2`, as `Device2` does not receive clock ticks during this time. This also saves redundant hardware instructions in a similar way to secondary pseudoclocks. Note that while gated clocks results in asymmetric clock ticks during the shaded regions, devices that can handle a variable frequency clock can typically also handle asymmetric clocking pulses.

5.1.6.1 Analog time series

The acquisition of an analog time series does not require control by a pseudoclock. This is because the vast majority of use cases expect a constant sample rate. While it is often expected that acquisition of analog time series may occur for several brief periods in a shot (rather than continuously throughout the entire shot), this is often achieved through gating the acquisition device via a digital output, rather than preprogramming a set of acquisition parameters prior to the shot commencing. This is an artefact of the way most acquisition devices are implemented, rather than a specific labscript design decision.

The labscript syntax for defining analog time series acquisitions is similar to that of outputs. In the connection table, acquisition devices are defined in the same way as output devices (either as a child of a `ClockLine` if they support output, or as a child of a digital trigger if they are a standalone acquisition device). An additional entry is made for each analog input channel (using the `AnalogIn` labscript class) that is attached to the previously defined device object. Often, a digital output is also defined for triggering the acquisition to begin at the start of the shot (some devices that support both input and output are able to use the first pseudoclock pulse to also trigger the acquisition instead). To define an acquisition, each `AnalogIn` object has an `acquire` method that takes a (unique) name, the start time and the end time as arguments. This results in similarly readable code (as previously shown for the outputs) such as `MOT_fluorescence_monitor.acquire('MOT load', start_time=0, end_time=5)`. Labscript currently assumes that the acquisition sample rate is fixed throughout the experiment, and so this parameter is typically defined in the connection table as part of the parent device instantiation. However, this behaviour could be changed easily by subclassing the relevant labscript acquisition classes.

Labscript also does not currently support hardware gating of analog acquisitions. Instead, analog acquisitions are gated in software after the shot has completed[8]. This has the downside of leaving the initial trigger as the only synchronisation point between the acquisition of analog time series and output devices, the ramifications of which will be explored in §5.1.7.

5.1.6.2 Images

Image acquisition, like analog time series, uses a digital trigger for synchronisation. However, unlike analog time series, this trigger usually occurs during the experiment and may occur multiple times. So while cameras are not controlled by a pseudoclock (that is, their data acquisition process and read out of data is not clocked by the pseudoclock), the times at which acquisitions begin are controlled by a pseudoclock. As such, cameras in labscript are effectively modelled as a digital output of the device, such as an NI card, that provides the trigger (and indeed, `Camera` subclasses the `TriggerableDevice` class, which internally creates a `Trigger` object that is a subclass of `DigitalOut`). The `Camera` class contains additional methods such as `expose`, the arguments to which determines the start time, duration, and other metadata, for each requested acquisition. Labscript internally generates `go_high` and `go_low` commands to generate the required digital pulse to trigger the camera exposure, and the additional metadata is stored in the HDF5 file. It is up to the imaging control system to use this metadata to correctly program the camera prior to the start of the experiment, so it is ready to receive the triggers (see §6.2), and of course up to the user to ensure the camera trigger input is correctly connected to the controlling digital output.

5.1.7 Waits

The core idea underpinning labscript is that the precise timing of the experiment logic is precomputed. However, there are many instances where you may not necessarily know the

8. As with the extensibility of the labscript output classes, the analog acquisition classes are also very extensible. Implementing hardware gating for a new device would not be significantly more time consuming than the act of adding support for a new device to labscript.

precise timing prior to execution. Examples (in ultracold atom research) include loading a MOT until a predefined atom number is reached or synchronisation to an external AC magnetic field (such as the one created by mains electricity). Labscript thus provides support for variable delays between instructions via the use of the `wait` command. A minimalistic labscript example containing a wait is shown in figure 5.11.

The implementation of the wait command is done by preparing all pseudoclocks for retriggering. This effects a resynchronisation of all pseudoclocks (and thus all outputs), the timing of which is determined by an external trigger provided by a piece of hardware not under labscript control. For the previously provided example of loading a MOT until an atom number is reached, this could be independent electronics designed to emit a trigger when MOT fluorescence light captured by a photodiode reaches an appropriate level. For synchronising to the mains electricity magnetic field, this could be independent electronics designed to emit a trigger when the AC line voltage crosses zero. The implementation of waits thus requires that all pseudoclock devices support the ability to halt execution (during a set of hardware instructions), resume on an external trigger, and allow this to occur an arbitrary number of times. A `wait` command necessarily has a minimum length for each wait, defined by the preparation time to place the devices in the retrigger mode, the delay of the master pseudoclock responding to the external trigger, and the minimum time required for the master pseudoclock to successfully trigger all secondary pseudoclocks. In order to prevent hung execution, we also provide a means to define a maximum length of each wait, via the `timeout` keyword argument, the use of which will be elaborated on shortly.

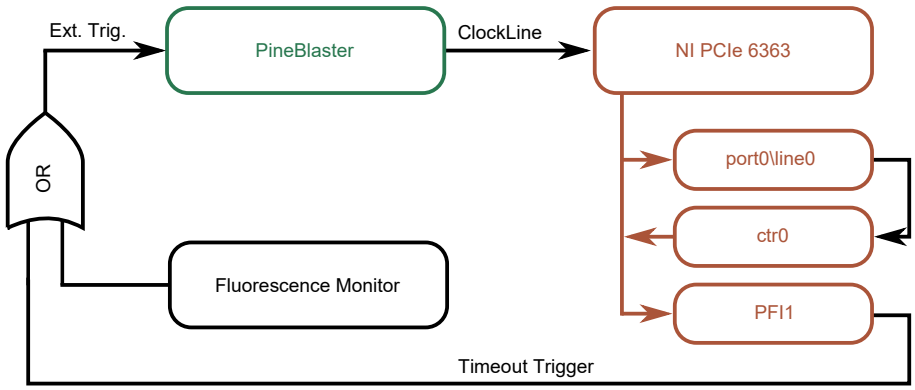
As the length of the wait is (deliberately) under the control of an external device, timing information in labscript experiment logic no longer matches experiment execution. We must thus introduce the concept of labscript time and experiment time. Labscript time refers to the times specified in the experiment logic file, where the length of each wait is considered to be 0 s long (plus the known retriggering time for secondary pseudoclocks). Experiment time then refers to the time during the execution of an experiment, where $t = 0$ corresponds to the start of the first clock tick of the master pseudoclock. This distinction has no effect on device output, which is synchronised by the master pseudoclock, but does affect analog acquisition, which is only synchronised to the start of an experiment. Acquisitions are thus defined in labscript time, but recorded in experiment time.

In order to determine the drift between these two time definitions, we employ a 'wait monitor' to measure the length of each wait. The wait monitor is instantiated like any other device in labscript, but is provided with 3 sets of connection information: a device and digital channel to use to indicate when a wait occurred, a device and channel to monitor the previously mentioned digital channel, and a device and channel to be commanded in software time to retrigger the master pseudoclock if the duration of the wait exceeds the `timeout` mentioned previously. The first set of connection information is used by labscript to internally instantiate a digital output channel, which is commanded (by labscript) to pulse just after every wait instruction. This digital output is then fed into a device that measures the times at which the digital channel goes high, relative to the start of the experiment, allowing the determination of the length of each wait from this information and the specified start times of the waits. A schematic diagram of these connections is shown in figure 5.11(b).

Further implementation details of the wait monitor are discussed in §6.1.1.4, where the

```
PineBlaster(name='pineblaster_0', usbport='COM1')
NI_PCIe_6363(name='ni_pcie_6363_0',
             parent_device=pineblaster0.clockline,
             clock_terminal='/ni_pcie_6363_0/PFI0',
             MAX_name='ni_pcie_6363_0',
             acquisition_rate=1e3)
WaitMonitor('wait_monitor',
            # flag that pulses after a wait
            ni_pcie_6363_0, 'port0/line0',
            # counter that monitors the times the above flag goes high
            ni_pcie_6363_0, 'ctr0',
            # software timed output that retriggers the master
            # pseudoclock if the wait hits the timeout
            ni_pcie_6363_0, 'PFI1')

t = start()
# Science part 1...
# Configure outputs for MOT load
t += 7

# Now that everything is set up, wait until the MOT
# is loaded (or continue anyway after 5 seconds)
t += wait('MOT_load', t, timeout = 5)

# science part 2!
t += 3
stop(t)
```

(a) A minimal labscript example demonstrating a 'wait' command.



(b) A schematic of the hardware configuration.

Figure 5.11: This example demonstrates how you might configure your experiment to wait until a MOT has reached a desired fluorescence. To use a wait, the labscript file must define a `WaitMonitor`, which defines the output port for a digital flag that will pulse immediately after each wait, a counter (that is physically wired to the previously described flag) to measure the length of the wait(s) by recording when each wait ends, and a software timed digital flag (in this case a PFI output on the NI card). The PFI output is electronically combined with the trigger from the fluorescence monitor electronics so that both devices can retrigger the master pseudoclock and resume the experiment (the former if the timeout is reached, the latter if the fluorescence level is reached).

lengths of the waits are measured during an experiment and used within BLACS device code to split (gate) acquired data into the requested acquisitions.

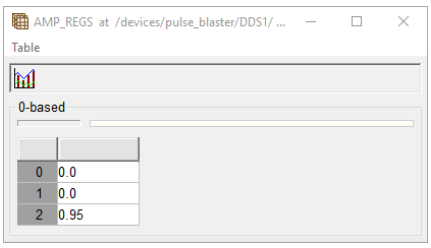
5.1.8 Shot storage

As introduced in §4.3.4, storing a complete record of the entire experimental process is a key philosophy of the labscript suite. Labscript contributes to this by saving a significant amount of data in each shot file. The most obvious datasets labscript saves are tables of hardware instructions. Labscript generates a HDF group (similar to a folder) for each hardware devices, at the internal HDF5 location of `/devices/<device name>` where `<device name>` matches the name passed to the constructor of each device object in the connection table. The format of the data stored is left up to the code for the specific device (as they are only written and read by device specific code, see §7.1) allowing hardware instructions to be stored in the most appropriate format for each device. For example (see figure 5.12), the PulseBlaster implementation stores the main body of hardware instructions in a HDF dataset within the group provided by labscript, but also stores the frequency, amplitude and phase register mappings in separate sub groups. The NovaTechDDS9m, on the other hand, uses two HDF datasets to store the instructions for the two channels that support table mode and the values for the two channels that can only be programmed once at the start of the experiment. Despite the typically unreadable nature of hardware instructions, we encourage developers who add support for devices to labscript to balance the format of the data required for programming the device with readability. For example, the devices we have implemented use human-readable names for the groups and datasets, and we name dataset columns so that a user can hone in on the required piece of data if they need to directly access the hardware instructions. Labscript also saves a representation of the connection table, which we showed previously in figure 5.6. This table contains a record of everything related to the configuration of devices and channels that cannot be reconfigured from shot to shot by BLACS (for example, BLACS is unable to rewire the hardware to select a different clock input port for an NI card).

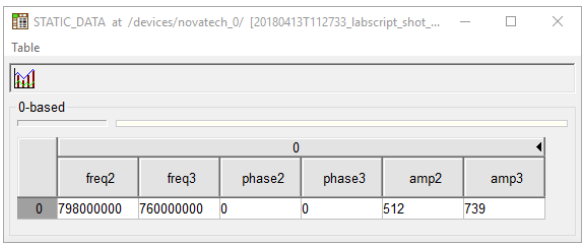
Of course, saving just the hardware instructions and connection table does not retain the intent of the experiment. This is, of course, best described by the Python file containing the experiment logic. Rather than attempting to store the experiment logic in some form of intermediate format, we simply save the entire experiment script file as text within the shot file. We also save a copy of any Python files imported into the main experiment logic file from the labscriptlib Python module (where all experiment logic files should be contained). This allows common code to be modularised, while preserving a complete copy of the Python code used to generate the hardware instructions. Saving the python files thus provides a means to determine the full intent of an experiment if it is reviewed in future years.

Additional metadata that is not related to a specific device or channel is stored in separate groups at the root level of the HDF5 file structure. Currently, these consist of the list of waits in a table, and calibrations (for example shutter delays)[9]. Such information is

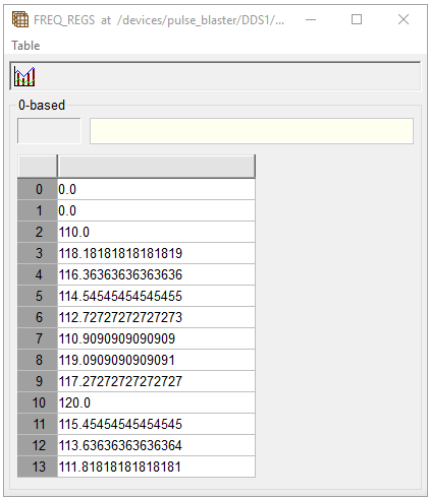
9. The calibrations group is an artefact of the early layout of groups, and there is an ongoing discussion as to whether this information belongs in the connection table for each Shutter channel as a shutter delay is a physical property of the experiment configuration and cannot be configured through software.



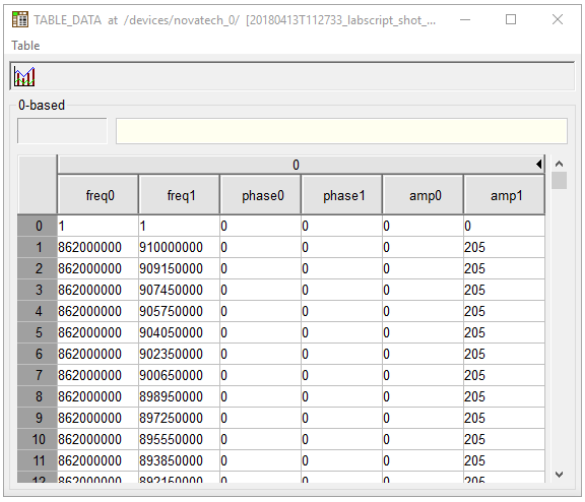
(a) Amplitude registers for a PulseBlaster. The row index corresponds to the index shown in the 'amp1' column in figure 5.12(e) while the value contains the amplitude in an appropriate format for the PulseBlaster programming API (in this case, Volts peak-to-peak).



(b) The values for the two DDS channels of a Novatech DDS9m that support only a single set of values pre-programmed prior to the start of the experiment. Here frequency is stored in multiples of 0.1 Hz as this is the limit of the precision of the device. Similarly, amplitude is stored in a quantised form as an integer with a maximum value of 1023 (which corresponds to 1 Volt peak-to-peak [120]) as this is what the programming API expects.



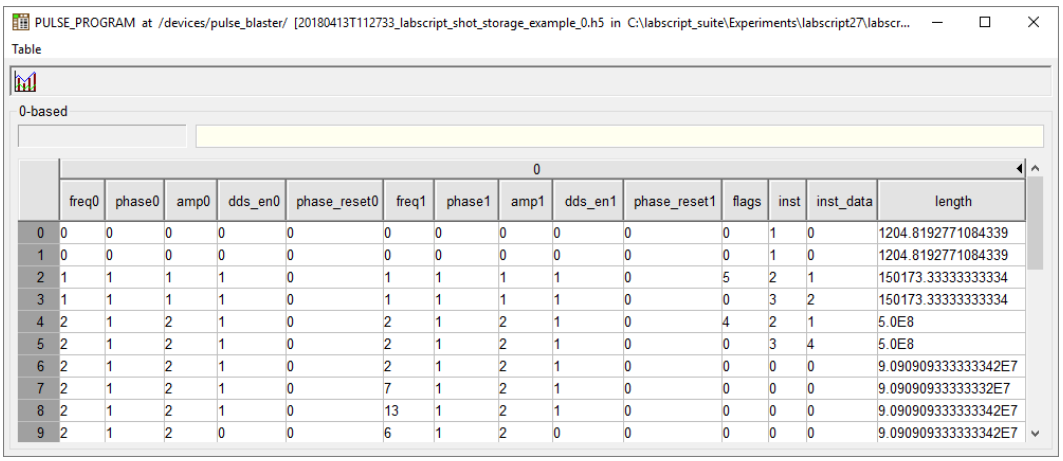
(c) A list of the frequency registers for a PulseBlaster. As in 5.12(a), the row indices correspond to the value stored in the 'freq1' column in figure 5.12(e). The values stored are in units of MHz, an appropriate format for the PulseBlaster API.



(d) The set of hardware instructions for the Novatech DDS9m DDS outputs that support table mode. The format of these instructions is identical to that of figure 5.12(b).

typically not recoverable from the hardware instructions and is primarily used to display metadata in runviewer (see §5.3).

Some versions of labscript also save the version control information from mercurial (Hg) repositories. This provides documentation on the software versions of labscript components used to generate the shot files as well as the status of the labscriptlib repository (if configured as one). Such information is very useful if attempting to understand the impact of bugs in the control software on past shots, or determining the feature set available at the time the shot was created. However, at the time of writing, this functionality is disabled by default as we determined that reading the Hg repository information was slowing down shot generation. Ultimately we will add support for a variety of different version control systems



	0													
	freq0	phase0	amp0	dds_en0	phase_reset0	freq1	phase1	amp1	dds_en1	phase_reset1	flags	inst	inst_data	length
0	0	0	0	0	0	0	0	0	0	0	0	1	0	1204.8192771084339
1	0	0	0	0	0	0	0	0	0	0	0	1	0	1204.8192771084339
2	1	1	1	1	0	1	1	1	1	0	5	2	1	150173.33333333334
3	1	1	1	1	0	1	1	1	1	0	0	3	2	150173.33333333334
4	2	1	2	1	0	2	1	2	1	0	4	2	1	5.0E8
5	2	1	2	1	0	2	1	2	1	0	0	3	4	5.0E8
6	2	1	2	1	0	2	1	2	1	0	0	0	0	9.090909333333342E7
7	2	1	2	1	0	7	1	2	1	0	0	0	0	9.09090933333332E7
8	2	1	2	1	0	13	1	2	1	0	0	0	0	9.090909333333342E7
9	2	1	2	0	0	6	1	2	0	0	0	0	0	9.090909333333342E7

(e) A set of hardware instructions for a PulseBlaster. 'freq', 'amp', and 'phase' columns contain integers that map to the tables of frequency, amplitudes and phase registers (see figure 5.12(a) and 5.12(c) for examples of the amplitude and frequency registers of DDS1). The 'flags' column contains an integer representation of a bit-field that contains the digital output state, of all 12 outputs, for each instruction. The 'inst' and 'inst_data' columns contain information about the type of instruction and how it should be interpreted (as these are 'complex' instructions, see §2.2.3). Finally, the 'length' column contains the length of time that the values of the instruction should be held for before moving to the next instruction (as determined by 'inst' and 'inst_data'). These columns directly map to the parameters of the relevant API call [122] used to program the PulseBlaster.

Figure 5.12: Examples of hardware instructions stored in a HDF5 file. The group name and location (path) within the HDF5 file are shown in the title bar of each image. Where multiple columns of data must be stored, we name the columns so that the data is more comprehensible by a human observer.

in a way that does not impact the speed of the shot generation.

5.1.9 Unit conversions

As mentioned previously in §5.1.2, much of the labscript API supports the specification of output values in user-determined units. By default, labscript API functions (such as `analog_output.constant(t, value)`) expect the value to be in SI units that correspond to the type of output (for example, volts for analog outputs and Hertz for frequency). We call these units 'base' units. However, these quantities often control another physical quantity (such as magnetic field strength or current). When writing experiment logic in base units (in any control system), you ultimately end up with a lot of 'magic' calibration numbers being used whose relationship to real-world units is not immediately obvious. This is often because it is easier to just adjust the magic base unit number until you get the result you want, rather than spending the time doing the calculation to real world units. The labscript API attempts to simplify this by automating the conversion between physical units and the base units that labscript works in.

All subclasses of `AnalogQuantity` (such as `AnalogOut`, and the frequency, amplitude and phase channels of a `DDS`) allow a reference to a unit conversion Python class, and a dictionary of parameters (called unit conversion parameters), to be passed in as optional arguments to the constructor used to create the object in the connection table. An example of this is shown in figure 5.13. The class passed in must define the SI base unit and a

```
# define unit conversion class
class BidirectionalCoilDriver(UnitConversion):
    base_unit = 'V'
    derived_units = ['A']

    def __init__(self, calibration_parameters=None):
        if calibration_parameters is None:
            calibration_parameters = {}
        self.parameters = calibration_parameters

        # I[A] = slope * V[V] + shift
        # Saturates at "saturation" Volts
        self.parameters.setdefault('slope', 1) # A/V
        self.parameters.setdefault('shift', 0) # A
        self.parameters.setdefault('saturation', 10) # V

        UnitConversion.__init__(self,self.parameters)

    def A_to_base(self,amps):
        shift = self.parameters['shift']
        slope = self.parameters['slope']
        volts = (amps - shift) / slope
        return volts

    def A_from_base(self,volts):
        volts = numpy.minimum(volts, self.parameters['saturation'])
        shift = self.parameters['shift']
        slope = self.parameters['slope']
        amps = slope * volts + shift
        return amps

# define connection table
AnalogOut('bias_coil_x', ni_pcie_6363_0, 'ao1',
          unit_conversion_class=BidirectionalCoilDriver,
          unit_conversion_parameters={
              "slope": bias_x_coil_slope,
              "shift": bias_x_coil_shift,
              "saturation": bias_x_coil_saturation
          })
```

Figure 5.13: An example of a unit conversion available in the labscript suite [152]. Unit conversions are defined by subclassing the `UnitConversion` class available in `labscript_utils.unitconversions.UnitConversionBase`. Unit conversion classes, along with a set of parameters to use (which may be loaded from labscript global variables) are then passed as optional arguments to the output constructors in the connection table definition. See main body text of §5.1.9 for further details.

list of the new units the user can use (the `derived_units`). The parameters passed to the `AnalogQuantity` constructor are passed in to the unit conversion class constructor when it is created internally by labscript. The unit conversion class constructor can then save these parameters for use in the methods defined for converting between units. For each derived unit, two methods must be defined in the class; one to convert between the base unit and the derived unit, and the other to convert between the derived unit and the base unit. For example, if the derived unit was Amperes (with the unit being defined as `'A'`) then you

would define methods such as `A_to_base(self, amps)` and `A_from_base(self, volts)`, which would return volts and amps respectively.

Many unit conversions follow similar patterns. For instance coil drivers typically have a linear relationship between the base unit of volts and current through the coil. Helmholtz coils also have a linear relationship between field strength (in Gauss or Tesla) and current. By writing a generic linear calibration class between the common derived units and the expected base unit, a user can abstract away the need to manage multiple calibrations. For example, a single unit conversion class, written for a bi-directional Helmholtz coil driver, can be used across multiple analog output channels that control different pairs of coils by specifying a different set of unit conversion parameters in the channel constructor. All channels can then be used using the following (or similar) syntax: `analog_output.constant(t, value_in_amps, 'A')`.

The calibration classes must all live within the `labscript_utils.unitconversions` module so that they are importable by the labscript suite. The calibration class name (and parameters) are stored in the connection table so that BLACS can instantiate the unit conversion class. This allows BLACS to provide manual control of each channel in physical units as well (see §6.1).

`DDS` channels also automatically instantiate a set of labscript unit conversions specified by the parent devices `get_default_unit_conversion_classes` method. This allows a parent device to provide default unit conversion classes (which can of course be overridden by the user). For example, the NovaTechDDS9m device provides a default unit conversion class for the frequency of the DDS that contains 'MHz' as the derived unit (as Hz is difficult to read for such large values). Similar behaviour could be extended to other types of labscript output classes if the need arises.

Defining experiment logic in physical units has several advantages. These include improved record keeping and the ability to see if the magnitude of quantity you are using is unphysical. There is an increased overhead in maintaining the calibration between units, however this trade-off is worth it for many research groups. Calibrations can also be automated if necessary. For example, since calibrations are defined in experiment logic, they can be parameterised by global variables (just like any other part of the experiment logic). Calibration experiments, combined with an appropriate lyse analysis script, can then be written to perform, measure, and analyse a unit conversion relation. Results from such an analysis can be placed into runmanager globals by the analysis script, ensuring any other experiment that shares these globals has access to the latest calibration data.

5.2 Runmanager

Runmanager is the primary means of defining and managing the set of experiment parameters (global variables - see §5.1.3) used in the labscript experiment logic. Runmanager also handles the creation of each HDF5 shot file, and the invocation of labscript via the execution of a user specified labscript experiment logic file.

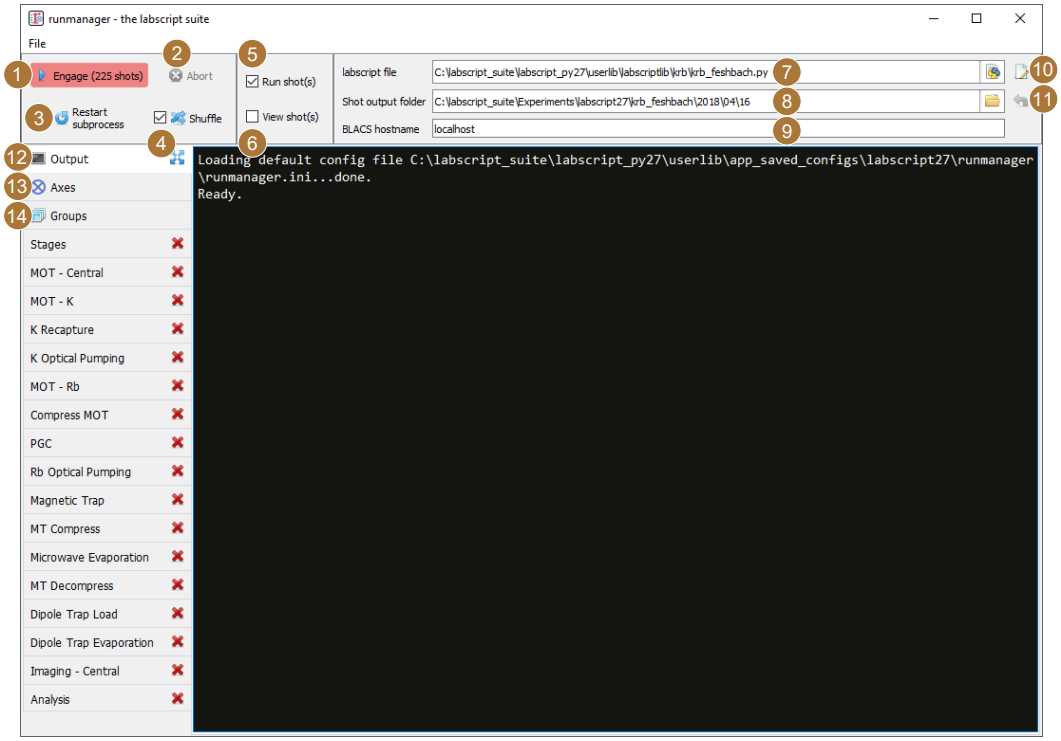


Figure 5.14: The runmanager graphical interface with the main controls labelled as per the main body text in §5.2.1. The 'output' tab is currently shown.

5.2.1 The graphical interface

As discussed in §4.3.3, we believe that the manipulation of parameters, along with controls for producing shots, are best implemented in a graphical interface. Critical information on the current runmanager configuration, along with controls for generating new shots, are located in a always visible toolbar at the top of the runmanager interface. These comprise (as labelled in figure 5.14):

1. The engage button: This begins production of the appropriate number of shot files. The number of shot files that will be produced is displayed prominently in the button text so that any mistakes made when defining the parameter space scan can be quickly corrected prior to beginning shot generation. This button can also be 'clicked' via the F5 key on a keyboard.

2. The abort button: This stops the production of shot files prematurely.

3. The restart subprocess button: Primarily for debugging and for use during labscript development, this button restarts the subprocess that manages the execution of the labscript experiment logic file, which in turn generates and stores hardware instructions inside the HDF5 file (see §5.2.5).

4. The shuffle checkbox: This checkbox controls the global setting for whether parameter space scans are shuffled or not. This is a tri-state checkbox (all-some-none) displaying

the current shuffle state on the axes tab. Clicking the checkbox will overwrite the state of each entry on the axes tab with the new state of the global checkbox. For more details, see §5.2.3.

5. The run shots checkbox: If ticked prior to clicking the engage button, shot files will be sent immediately to the BLACS queue once the hardware instructions have been generated by labscript.

6. The view shots checkbox: If ticked prior to clicking the engage button, shots will be sent to runviewer once the hardware instructions have been generated by labscript. Runviewer is assumed to be running locally, and will be launched if it is not already running once the first HDF5 file has been generated.

7. The labscript file: The Python file containing the experiment logic to be compiled into hardware instructions (see §5.1).

8. The shot output folder: The location to store the HDF5 shot files. By default, the location in specified by the combination of a value in the laboratory PC configuration file (see labconfig in the glossary), the name of the experiment logic Python file and the current date. The location automatically updates, at midnight, to a new folder for the day provided the folder location is left as the default.

9. The BLACS hostname: The network hostname of the PC the HDF5 shot files are to be sent to if the 'run shots' checkbox is ticked. It is expected that BLACS is running on the specified PC, and that network access (including firewalls and other network access controls) is configured appropriately.

10. The open in editor button: This button open the specified labscript experiment logic file in the text editor specified in the laboratory PC configuration file (see labconfig in the glossary).

11. The reset shot output folder button: This button resets the shot output folder to the default. This will re-enable the auto incrementation of the folder (based on the current date), which is disabled for custom locations.

These controls provide rapid access to the key functionality of runmanager (creating and distributing shot files) at all times, making for an efficient workflow.

The rest of the runmanager interface exists within a set of tabs. The first 3 tabs contain further runmanager specific controls:

12. The output tab: This tab contains the terminal output of the shot creation process including the terminal output produced during the execution of the labscript experiment logic file (see figure 5.14). For example, Python `print` statements included in the experiment logic code will appear here during shot creation. This makes it easy to debug the experiment logic code using simple methods common to general purpose programming. Warnings and error messages generated by the labscript API also appear here in red text, so that any issues are immediately noticed and can be actioned. As this output is useful for debugging purposes, we allow the tab to be 'popped out' into a separate window so it can be visible at the same time as another tab (to avoid

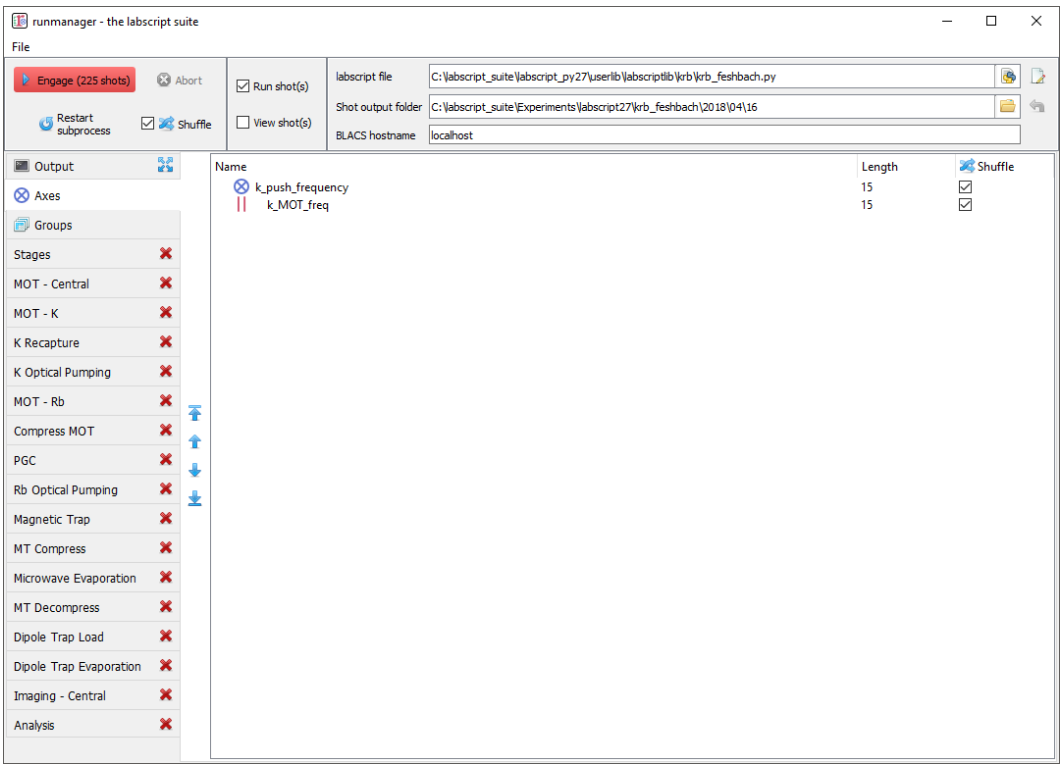


Figure 5.15: The 'Axes' tab of runmanager. This tab displays a list of all global variables (indicated by the blue outer product icon) or groups of global variables (indicated by the icon with red parallel bars) that form axes of the parameter space that will be scanned over (see item 13 in the main body text and §5.2.3 for further details). The order of the axes can be changed using the controls to the left of the list, which sets the order in which the outer product of the axes is performed (when generating the shot files).

the need to frequently switch between the output and the tab containing the global variable(s) you are currently modifying).

13. The axes tab: This tab allows the user to control the iteration order of the parameters in the defined parameter space (see figure 5.15). The length of each axis of the parameter space is displayed, as is a shuffle checkbox for determining whether the points along that axis should be shuffled before the parameter space is expanded into the set of shots to be created. The global shuffle control (see item 4 in this list) is linked to the state of the shuffle checkboxes on the axes tab. This feature, along with the many benefits, is detailed further in §5.2.3 (see feature 3 and the paragraphs following).

14. The groups tab: This tab manages the HDF5 files that store the globals (see figure 5.16). Further details on managing global variables will be discussed in §5.2.2.

These tabs are then followed by an arbitrary number of tabs containing sets of global variables, which will be discussed further in §5.2.2.

In addition to this, runmanager can save and restore the entire GUI state via the relevant menu items in the 'File' menu. This allows rapid switching between different types

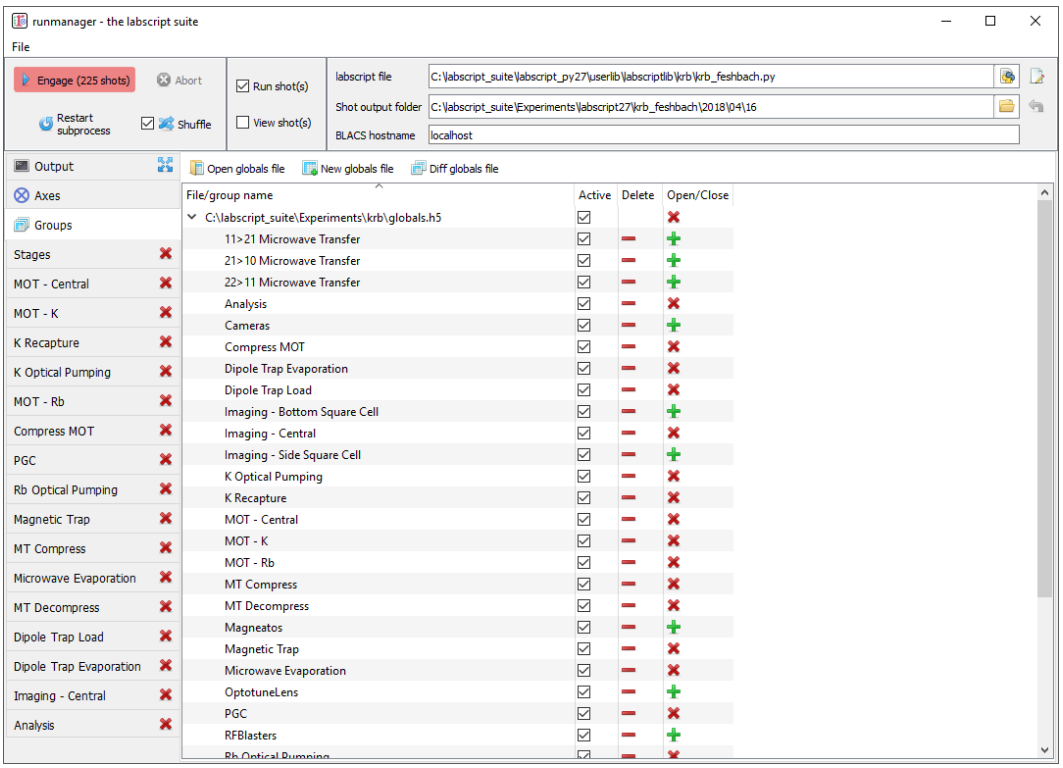


Figure 5.16: The 'Groups' tab of runmanager. This tab displays the groups of global variables (stored in HDF5 files) that have been loaded into runmanager. From this tab, users can enabled/disable the use of these globals when compiling shots (using the 'active' checkboxes) and open/close an editing tab for each group. The editing tabs, when open, are displayed as additional tabs on the left most edge of the runmanager interface. See §5.2.2 for further details on managing globals.

of experiment logic and/or globals files[10]. This is particularly useful for shared experiment apparatuses, where different users want to run different experiments, and for the cases where a user wishes to rapidly switch between one of more diagnostic configurations they have previously saved.

5.2.2 Managing global variables

Runmanager provides a simple interface for grouping and modifying global variables. As mentioned previously, the 'groups' tab in runmanager handles creating and opening the HDF5 files that store the global variables. There are two levels of organisation for global variables:

- at the file level (globals can be stored across multiple files, the union of which is used to generate shots), and
- groups within each file.

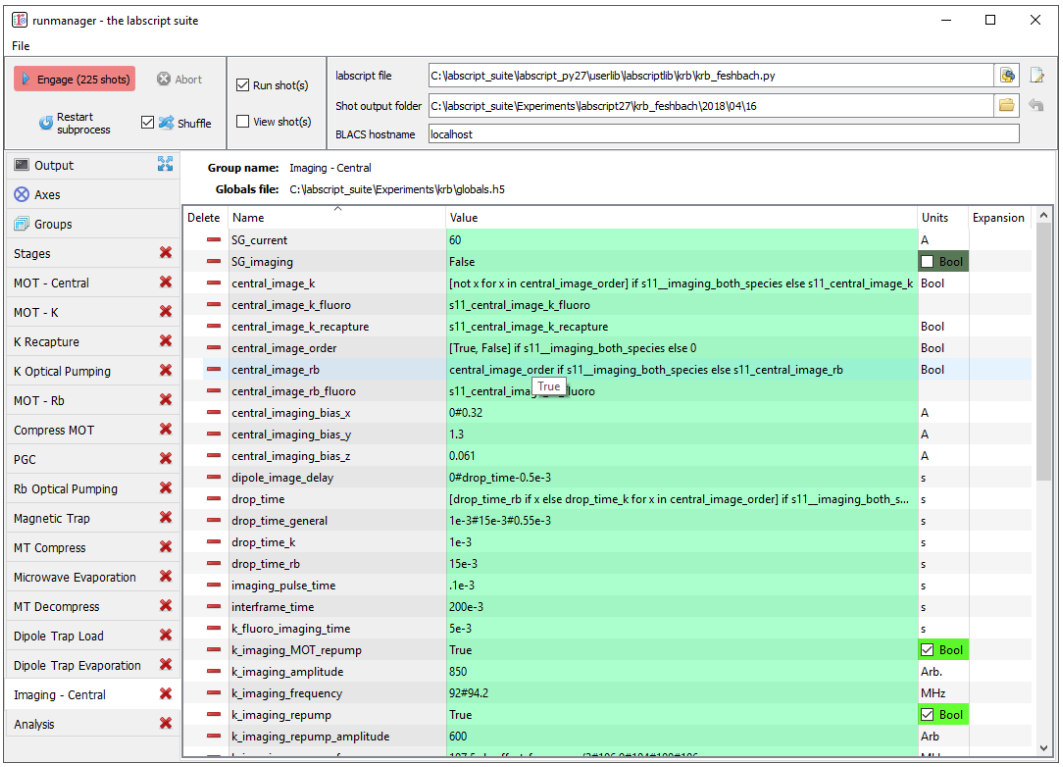
10. For clarity, the values of the globals are not saved in this configuration file, but simply the location of the HDF5 file containing the globals. This means that any globals in files shared between saved runmanager configurations will share their values. For cases where global values should differ between runmanager configurations, separate globals files should be used.

Globals groups are created from the 'groups' tab in runmanager and can have arbitrary names (including spaces and special symbols). The only requirement is that a group name is unique within its file (it can however have the same name as a group in a different file). Globals within a group are then only used in the creation of shots if the 'active' checkbox for the group is checked on the groups tab (see figure 5.16). This provides a simple way of switching between different groups of globals, allowing labs to maintain a common set of parameters for their experiments as well as individual parameter sets for specific users and/or experiments. For example, rather than modifying a set of globals in a group, a user could instead deactivate the group containing those globals, and instead ask runmanager to pull those globals from a separate file.

Each group of globals can be opened for editing in a new tab. We provide columns for the global name, value and units. The global name must be a valid Python variable name [153], and must not conflict with any member of the pylab library, python keywords, or existing items in the Python `__builtin__` module. This ensures that it can be injected into the labscript experiment logic (see §5.1.3) without conflicting with existing Python functionality. The global name must also be unique across all active groups, as global groups are joined into a single set before passing the globals into the labscript experiment logic.

The value of a global can be any Python expression (including the use of functions from the numpy module), that evaluates to a datatype supported by HDF5, such as, but not limited to:

- a number: `1234` or `780e-9`,
- a string: `'N_atoms'`,
- a list or numpy array (which will be treated as an axis of a parameter space to scan, where the global variable will contain only one of the elements of the list or array in each shot): `[1, 2, 3]` or `array([1, 2, 3])`,
- a tuple (which despite being list like, will **not** be treated as an axis of a parameter space to scan and will instead be passed into labscript as the tuple specified): `(1, 2, 3)`,
- a Boolean: `True` or `False`,
- an equation: `1+2`,
- a Python inbuilt, or numpy, function call that returns a valid value: `linspace(0, 10, 10)`,
- an expression that references another defined global variable by name (the value of this global variable is used in its place): `2*other_global` or `linspace(0, 10, other_global)`,
- a Boolean expression: `(other_global1 and other_global2) or (other_global3 == 7) or (other_global4)`, or
- any of the above plus a Python comment: `780e-9 #This was previously 781e-9`.



(a) An example of complex global variables that utilise Python expressions to define their value. Note, for example the `drop_time` global variable, whose full expression is shown in (b). The `drop_time` used is always drawn from one of three global variables, but the global variable selected is determined by a separate global variable (a Boolean) and may contain a list of drop times if the user wishes to image multiple species. In the case of the expression generating a list, this global becomes an axis of a parameter space, running two shots for every other data point in the parameter space (one shot to image each of the two species our experiment supports). Such an expression could not be defined within experiment logic as parameter spaces must be defined within runmanager, not labscript. In order to simplify the view of globals with complex expressions, the tooltip (shown for the `central_image_rb` global) shows the value(s) the global will take in the next compiled shot(s).

```
# code entered into runmanager global
[drop_time_rb if x else drop_time_k for x in central_image_order] if
s11__imaging_both_species else (drop_time_k if central_image_k == True
 else (drop_time_rb if central_image_rb == True else drop_time_general
))
```

(b) The full expression of the `drop_time` global is shown.

Figure 5.17

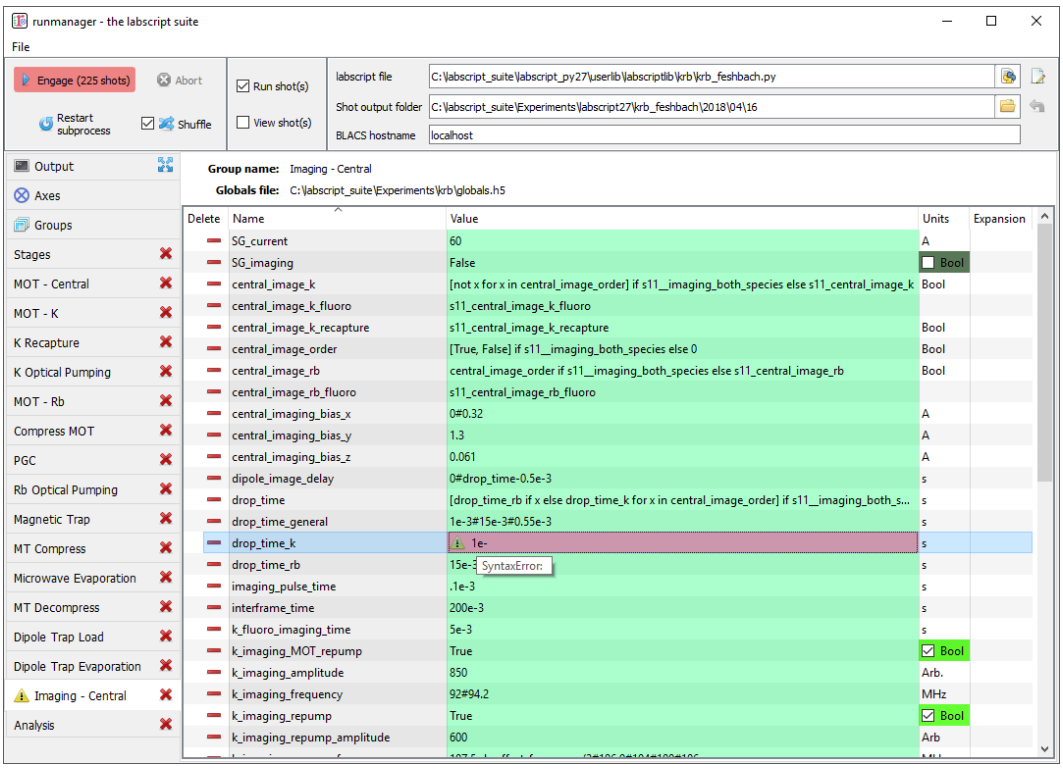


Figure 5.18: An example of an evaluation error in a global variable. The user is notified of the error in two places: an icon appears next to the tab name and the global in question is highlighted in red. The tooltip displays the cause of the error, in this case a Python syntax error.

As these expressions can become quite complex (see figure 5.17), the tooltip for the value cells displays the evaluated result of the Python expression. The value cell is also colour coded to the successful evaluation of the expression, so that mistakes can be easily identified (see figure 5.18).

The units of the global are not currently passed into the labscript experiment logic code, but are a way to provide context to the user within runmanager. For example, if the labscript experiment logic multiplied a global variable for a frequency by `1e6` everywhere it was used (or the keyword argument `units="MHz"` was used everywhere), then you could type 'MHz' into the units column of runmanager so that a later user would know that the global was expected to be of that magnitude and would not accidentally enter it in kHz or Hz. In addition to this, globals whose values are explicitly specified as either `True` or `False` have their units automatically set to 'Bool', a checkbox is placed in the units column for easy toggling, and the units cell is colour coded to this checkbox for easy observation of the state. We frequently use this functionality to enable/disable various stages of our experiment logic file (see figure 5.19).

While we recommend storing globals in a dedicated set of files, the storage format for the globals is identical to that in any shot, which allows a user to easily load in globals from existing shots (even ones that have been executed and analysed). However, once pointed at an existing shot file, any modification to globals will modify that shot file, thus partially

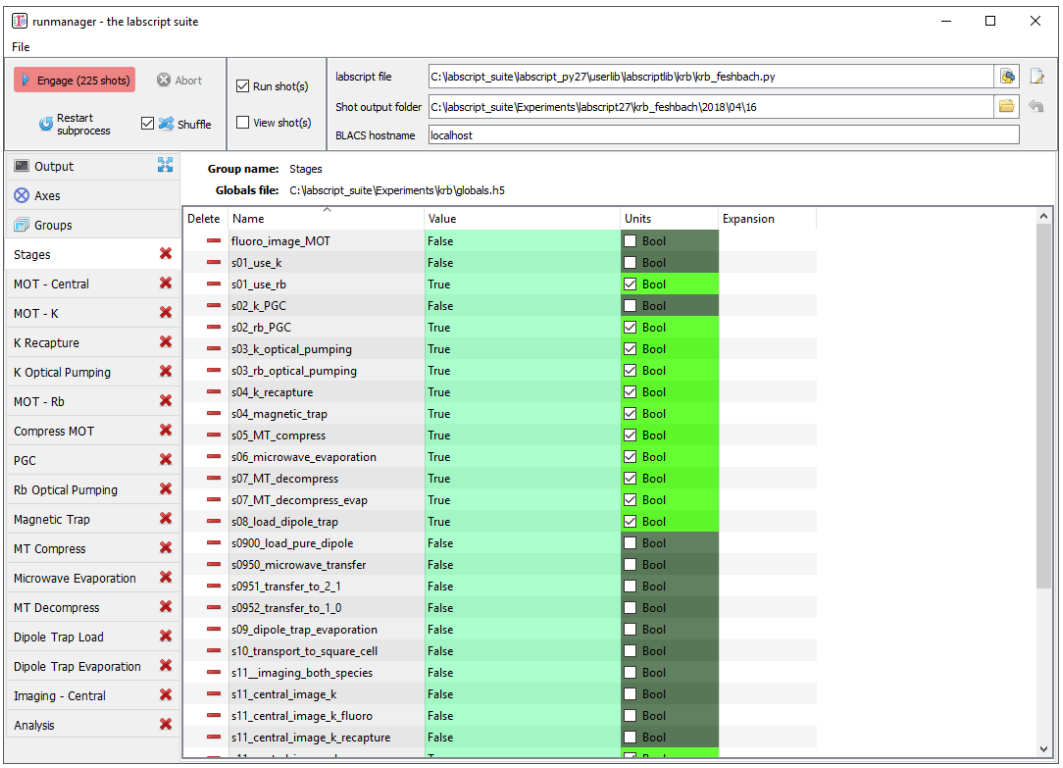


Figure 5.19: An example of how a labscript experiment can be parameterised by a series of Boolean global variables. Here we split up the production of a BEC into several stages. We name each global with a prefix that increments in order to keep the globals in an appropriate sort order. Runmanager detects the Boolean type of the global, and provides a simple checkbox toggle in the units column, By using these global variables in our labscript experiment logic file, as the Boolean expression for an `if` statement, we can quickly turn on/off various stages of the BEC production process (which is very useful when debugging or optimising the BEC production process).

destroying the complete record of the experiment[11]. Thus, we encourage this feature to only be used for the cases where you wish to look at the globals from an old shot or where you wish to use the globals, without modification, to compile new shots.

5.2.3 Parameter space scans

One of the key features of runmanager (and critical goals of our scientific control system) is the ability to easily automate the traversal of a large parameter space, an increasingly important requirement for performing modern ultracold atom experiments. Runmanager provides four features for managing parameter space scans:

1. The automatic detection of global variables that are defined as a list[12]. Such globals are labelled 'outer' in the expansions column as all such globals will be combined, via

11. Note that in an executed shot file, globals exist in two formats: the evaluated format (one point in the parameter space) used by labscript, and the raw strings as displayed in runmanager. Only the latter would be overwritten if globals were edited in the manor described in the main body text.
12. Runmanager considers both Python lists and numpy arrays to be what we refer to as 'lists' in this section.

an outer product, into the parameter space to be scanned. The number of shots to be generated, which is simply the product of the lengths of all 'outer' product globals, is displayed next to the engage button.

2. The ability to define what we term 'zip groups' after the Python function `zip`. Two (or more) globals (specified as lists) can be grouped together so that they iterate through values in lockstep. In this instance, the zip group is used as a single axis of the outer product rather than one axis for each global.

3. The third feature is the ability to define the order in which the axes of the parameter space are iterated over when producing individual shots (see the 'Axes' tab discussed previously in §5.2.1).

4. The ability to randomly shuffle the order of values within each global (or zip group) defined as a list. This can be done on a per global basis or on the entire set of shots that spans the defined parameter space.

These features provide a powerful basis for performing complex experiments.

Consider the following example. Many of the early stages of BEC production (for instance the MOT or magnetic trap stages) should be optimised for best phase-space density. Phase space density is calculated from several parameters; the most important being atom number and atom cloud temperature. While atom number can be easily measured from an absorption image from a single shot, temperature is most commonly determined from analysing the result of multiple shots. In this case, the drop time (the time between releasing the atoms from the trap and taking the absorption image) is varied for each shot and the temperature determined by fitting to the linearised relationship between atom cloud size and drop time. Already, it can be seen that measuring the phase space density for a single set of parameters requires several shots, which can be easily automated via the feature 1 described above.

Now consider the optimisation of MOT or magnetic trap parameters. Many of these are coupled and can not be independently optimised. As such, it is preferable to optimise two or three variables at once, measuring the phase-space density at each point to determine the optimal set of parameters. Such a parameter space typically takes several hours to complete due to the large number of shots that must be run. A BEC apparatus is likely to undergo systematic drifts during this time, which may invalidate the results. However, with careful thought, features 3 and 4 can be used to counteract this. For example, systematic drift will effect the linearity of the data when determining temperature, especially if the acquisition of each data point is separated by a significant period of time. However, by defining the drop time to be the inner most item of the outer product, you ensure that all shots needed to determine the phase-space density for a single set of MOT parameters are executed as close together in time as possible. Shuffling the order of the drop time then eliminates short term systematic drift, as does separately shuffling the order of the values in each remaining axes of the outer product (the MOT parameters). If long term systematic drifts need to be quantified, then an additional axes to the outer product can be added at the outer most layer in order to repeat each of the shots a prescribed number of times (by defining an additional 'dummy' global variable as `range(N)` where `N` is the number of times to repeat each shot).

While the above example may seem complicated, runmanager makes it trivial to implement. A user simply defines the list of values to scan over for each parameter, sets the order in which the outer product should use each axis, and specifies whether the values for each axis should be shuffled. Once done, clicking the engage button generates the sequence of shots and sends them to BLACS to be executed on the experiment.

5.2.4 Evaluation of globals

All global variable expressions are automatically evaluated after a change to any global variable. This serves to both update the tooltip with the result of the expression, detect axes of a parameter space to scan (and group them into zip groups if appropriate) and warn the user of any errors during the evaluation of the globals. As discussed previously, runmanager allows these global variable expressions to reference other global variables. This allows a user to maintain a record of a set of parameters, and all relevant quantities derived from one or more of those parameters, without ever storing a parameter more than once. This ensures that important quantities need not be derived (from globals) in the labscript experiment logic script, and that they are accessible directly during the analysis stage (see chapter 6).

To implement this, we take advantage of the Python built-in function `exec` which not only evaluates a string containing a Python expression, but can do so from within a controlled namespace. This has a two-fold benefit. The first is that it allows us to provide access to a specific set of functions that can be used from within the Python expressions (such as numpy functions like `linspace`). The second is that it allows us to keep track of the relationship between global variables, which is critical for both descriptive error messages and automatically detecting which globals should be combined into a zip groups.

The Python `exec` function is given access to a namespace to work in via an optional argument in the form of a dictionary. Keys and values in this dictionary correspond to variable names in the namespace and their associated values respectively. Rather than using a native Python dictionary for the namespace, we subclass the Python dictionary and override the built-in dictionary method for looking up entries in the dictionary. When combined with `exec`, this translates to our `dict` subclasses tracking each time the `exec` function requests the value of a variable in the namespace. This then provides us with a mapping of each global variable, and the names of global variables that it depends on. In order to resolve both the order in which global variable expressions are evaluated in, and detection of any recursive relationships, we begin by evaluating all global expressions and then recursively re-evaluate the set of globals that did not evaluate in the previous iteration. The first iteration will evaluate any (correctly defined) independent globals, and subsequent iterations will then be able to evaluate globals that depend on other globals (once those other globals have been evaluated by a previous iteration).

The hierarchy of global interdependencies is then used to determine automatic zip group names, which are based on the name of the global in the hierarchy that does not depend on any other. If a global depends on multiple other globals, then the zip group name is chosen semi-randomly based on the order of the items in the Python dictionary (which depends on a hash of the dictionary key names and the size of the dictionary). However, it is of course always possible to overwrite the automatic zip group name with something else should our

algorithm choose incorrectly.

We believe that this complex evaluation of global variables is only possible due to the use of an interpreted language that has tools for parsing its own syntax. As such, the choice of Python as our programming language has allowed us to implement extremely useful, advanced features that might otherwise be too difficult to produce in more low level languages such as C++.

5.2.5 Shot creation

The internal process for generating shot files is quite complex. This is primarily motivated by the desire for modularity (for example, to separate shot file generation from hardware instruction generation) and the desire for robustness. As runmanager ultimately initiates the execution of user code (the labscript experiment logic file), there is a risk that problems in the user code could crash runmanager. We mitigate this by using a multi-process architecture (see §4.4.5).

We originally spawned a new Python process for each shot (in order to guarantee the internal state of labscript was fresh). However the time required to start a Python process (especially on Windows) was a considerable fraction of the entire shot generation time. As such we now use a single, long-lived, Python process and clean-up the internal state of labscript and Python explicitly after each shot.

To generate shot files, runmanager:

1. Re-evaluates all globals (see §5.2.4). This both determines the number of shots to produce, and generates the evaluated set of global variables for each shot.

2. The globals are then written to HDF5 files, one file for each shot. We also write the unevaluated globals into every HDF5 file, in order to provide a complete record of the experiment (the unevaluated globals contain information about the parameter space that is not available when looking at the single point of parameter space in the evaluated globals of a single shot file).

3. In a thread (in order to keep the GUI responsive), we iterate over the set of files and send their file paths to a long-running subprocess (launched by runmanager at startup) that is used to execute labscript code in an isolated environment. We call this process the 'compilation subprocess'.

4. The subprocess, which has the labscript API imported, calls an initialisation method to inform the labscript API of the HDF5 file to write hardware instructions to.

5. The subprocess loads the global variables from runmanager into the `__builtin__` dictionary.

6. The subprocess then executes the labscript experiment logic file (using the Python function `exec`) in an isolated namespace, which invokes the labscript API via the users experiment logic and generates the required hardware instructions and saves them in the HDF5 file. Terminal output (for example, `print` statements) are sent back to the parent runmanager process and placed in the output tab.

7. The subprocess restores the `__builtin__` dictionary to its original state to prevent globals from polluting subsequent shots. A clean-up method from the labscript API is also called so that the internal state of the labscript Python module is also reset.

Once shot files are created, the file paths are sent to runviewer or BLACS, as determined by the checkboxes in the runmanager GUI, for viewing and/or executing the shots respectively.

This architecture also has several unrealised benefits:

1. If the need arose, we could easily parallelise the generation of hardware instructions by instantiating multiple instances of the compilation subprocess.

2. We could use runmanager as a generic parameter (space) management software by replacing the compilation subprocess with something else. For example, runmanager could be used to manage parameters for simulations, producing one shot file per simulation to be run in the same way we do for real experiments. These files could then be sent to a scheduling program (like BLACS) that feeds them to the simulation software.

5.3 Runviewer

While the textual based interface of labscript is ideal for defining experiment logic in 'high level' terms, there can often be a discrepancy between what was intended and what was actually commanded. This is particularly prevalent in situations where more complex control flow features (such as while loops or parameterised function) are used as this increases the abstraction between the language used to command the output and the actions of the output. Runviewer exists to bridge this gap, allowing us to maintain the benefits of textual control of the experiment without losing the benefits of the graphical representation of the experiment logic. Runviewer achieves this by producing a series of plots containing the output state for each channel. These plots are 'reverse-engineered' from the hardware instructions stored in the HDF5 file, and are thus a faithful representation of what each output channel should do during an experiment (provided of course that the reverse engineering code is accurate).

There are thus several uses for runviewer. The most important is the ability to graphically observe the experiment logic. This allows a user to easily observe experiment features such as the shape of complex ramps or the synchronisation between events on different channels. Runviewer also supports simultaneous display of traces from multiple shots providing, for example, a means to see how an output trace changes when a global variable is adjusted. Finally, comparisons between expected output in runviewer, and observed output on an oscilloscope can make debugging hardware problems quicker.

5.3.1 Generating output traces

Runviewer generates the displayed output traces by processing the hardware instructions stored in the HDF5 shot file. We specifically reconstruct the output from the lowest level description of the experiment logic in order to accurately represent what the output hardware will do during an experiment. In order to support a diverse range of hardware, part

of the reconstruction process is handled by device specific code that must be written by a developer when adding support for a new device. This device specific code simulates how the device processes hardware instructions and updates output states. It is discussed in more detail in §7.1.4, so for the purposes of this section we'll only cover generally what such code should do. The reconstruction algorithm is then as follows:

1. The master pseudoclock device is identified from the HDF5 file.

2. We import the device specific runviewer class for the master pseudoclock and request that it generate the traces for its outputs. As this is the master pseudoclock, we instruct the device specific code that there is not anything controlling the timing of this device (the need to do this will become apparent in a later step).

3. The device specific code generates a set of output traces (as it sees fit) and returns these traces to runviewer by calling a provided runviewer method, indicating that these traces should be available for display. This allows the device to produce as many traces as it likes, without limitation by the runviewer architecture. This is critical, as it removes the need for runviewer to support specific output types. Instead, this support is baked-in to the device specific code, which should already be aware of the output capabilities of the device.

   If timing information was provided by runviewer (which is the case for all devices except the master pseudoclock, see step 5 below), then it is used by the device code to generate the correct timing of the output traces. For example, a Novatech DDS9m only stores a table of output state changes, so the timing information of the parent `ClockLine` is needed. Similarly, the timing of a secondary pseudoclock is dependent on state changes to the parent `Trigger` line.

4. The device specific code then returns, to runviewer, a dictionary of traces for any `ClockLines` or `Triggers` assigned to digital outputs of the device (which may or may not have already been provided to runviewer for display in the previous step).

5. Runviewer iterates over this dictionary, and finds all devices that are children of each `ClockLine` or `Trigger`. For each device, the device specific code is imported and called as in step 2, except that this time we provide the device specific code with the trace for the `ClockLine` or `Trigger` so that it can generate output traces with the correct timing. The device specific code then follows step 3 and runviewer repeats steps 3 to 5 recursively until all devices have been processed.

5.3.2 The graphical interface

The graphical interface of runviewer comprises 3 sections (see figure 5.20). The first section manages the loading of shots into runviewer. Here you can enable (or disable) shots for plotting, choose the plot colour, and choose whether markers for shutter open and close times should be displayed. The second section manages the channels that are to be plotted. These channels can be reordered using the controls to the left, which then affects the order in which the plots appear. The list displays the union of all channels from shots that

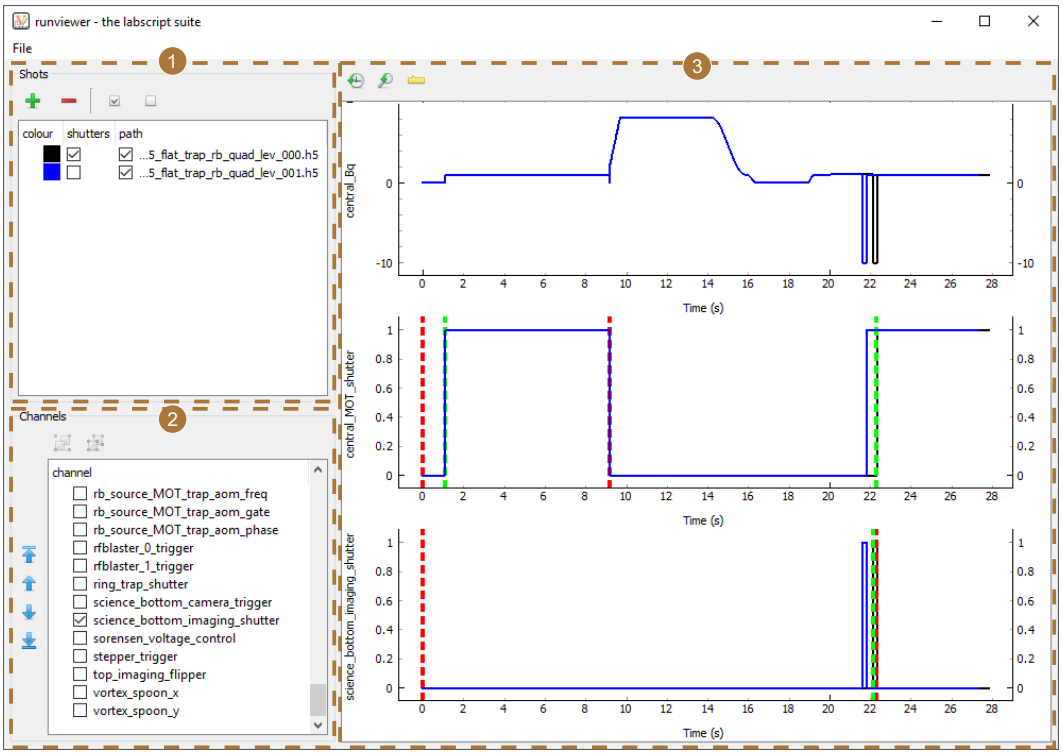


Figure 5.20: The runviewer interface consists of 3 main sections. (1) Controls for loading shots, selecting the colour of traces, selecting whether shutter open/close markers are to be shown, and whether the output traces from this shot should be shown. Note that we have only enabled shutter markers for one of the two shots loaded (the black trace). (2) A reorderable list of channels contained within the loaded shots. The order here determines the order of plots in (3). Only enabled channels will be displayed in (3). (3) Plots of the output traces for the selected channels in the selected shots. Here we show data from 2 shots of a real experiment sequence from our lab used to study vortex clustering dynamics [127]. The two shots loaded demonstrate how you can observe differences in output between shots in a sequence (in this case due to varying the time between stirring and imaging the vortex clusters). In this figure we display the entire length of the trace, which makes it difficult to distinguish between the shutter open/close events (red and green dashed, vertical lines) and the digital output trace. The discrepancy between these events becomes more apparent when zooming in (see figure 5.21).

are currently enabled or have been previously enabled. This ensures runviewer remembers selected channels, even if they do not exist in the current shot, removing the need for a user to constantly re-enable channels when switching between different types of experiments. The configuration of enabled channels can also be saved and loaded from the 'File' menu, which is a useful aid when switching between regularly-used experiments.

The third section comprises the plotting region. We use the Python plotting library `pyqtgraph` to generate the plots. This choice was primarily made due to the performance of `pyqtgraph`, which is significantly faster than other common Python plotting libraries

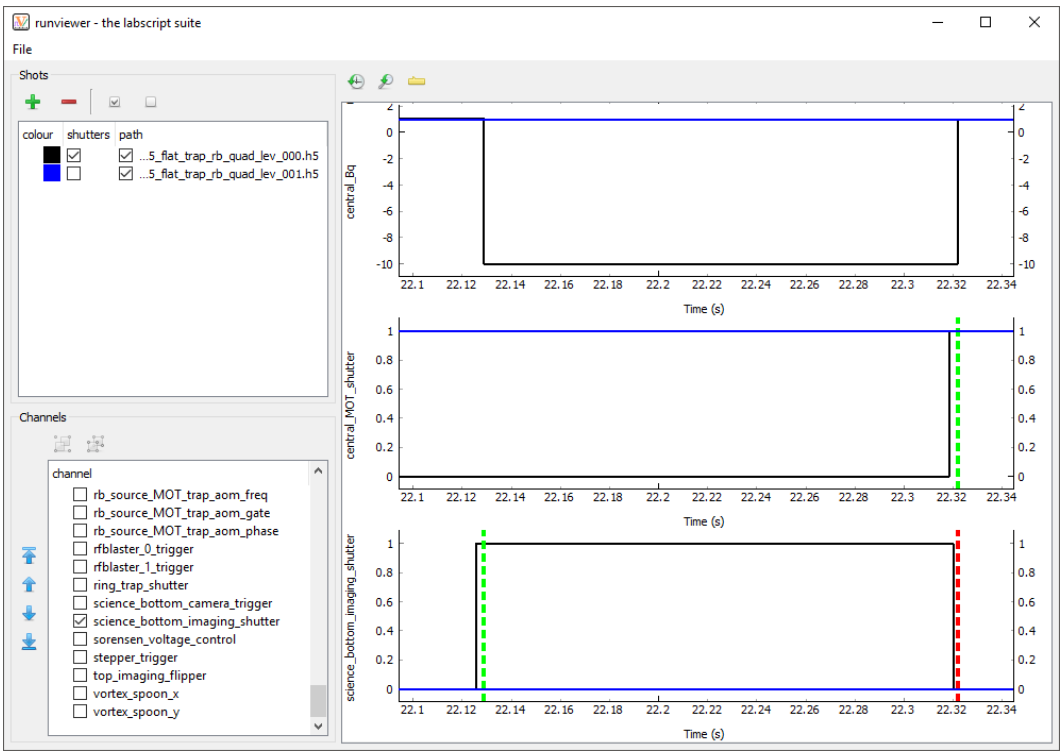


Figure 5.21: Here we show the same traces as in figure 5.20, but zoomed just after the 22 s mark. We can now clearly see the difference between the change in digital state (black trace) used to open and close the shutter, and the time at which the shutter was actually commanded to open and close (green and red dashed, vertical lines respectively). In this case, the shutter (open, close) delay was specified in the labscript file as $(3.11, 2.19)$ ms for the `central_MOT_shutter` and $(3.16, 1.74)$ ms for the `science_bottom_imaging_shutter`.

such as `matplotlib`[13]. The user can pan and zoom the plots produced by `pyqtgraph` using the mouse (by holding left or right mouse button respectively while moving the mouse). The time axes of each plot are linked together so that multiple output traces can be easily compared to each other. Two buttons are then provided at the top of the interface for resetting the axes to the default scaling.

As discussed previously, the output traces are generated directly from the hardware instructions. This creates two problems: information about the timing of certain events may not be contained within the hardware instructions, and the output trace may contain too many data points to plot efficiently (even when using `pyqtgraph`). The first problem we solve by plotting vertical markers at points of interest. For example, the `Shutter` class automatically accounts for the open and close delay of a shutter. The output trace thus only captures the time at which the digital output goes high or low and does not capture when the shutter will be open or closed. Runviewer reverse engineers these missing times from metadata stored within the HDF5 (see §5.1.8) so that they can be plotted as markers

13. We typically use `matplotlib` in the labscript suite as it is a widely known package with an almost identical syntax to MATLAB. This means that many users are already familiar with the syntax needed to create plots. As the user is not required to write or modify the code that generates the plots in runviewer, this benefit was not applicable and so it was worth using pyqtgraph for the increased performance.

of interest (see figure 5.21).

The second problem is solved by dynamically resampling the output traces depending on the zoom level of the x-axis of the plots. I wrote a feature-preserving algorithm for this purpose to avoid the many down-sampling algorithms that miss features faster than the sampling rate. This ensures that zoomed out plots accurately represent the trace, even when resampled. The algorithm starts by creating an output array of points that is 3 times the maximum width, in pixels, that the plot is expected to be displayed at[14]. We fill every third data point in the output array using 'nearest neighbour on the left' interpolation, using only the section of the output trace that is currently visible. We then fill the other two data points with the highest and lowest value between the first data point and the 4th data point (which will also be determined using 'nearest neighbour on the left' interpolation). These two data point are placed in the order in which they appear, the reason for which will become clear shortly. This is repeated until the output array is full. The output array is then passed to `pyqtgraph` for plotting. Fast features thus exist in three data points of the array, which pyqtgraph correctly plots in one pixel as a vertical line. This is similar to the way digital oscilloscopes display acquired signals.

Despite our optimisation efforts, resampling still takes a significant period of time, particularly if there are many plots displayed. We thus perform the resampling in a thread in order to keep the GUI responsive. However, because the resampled data has more points than can be displayed, and these points are in the correct order, zooming in still immediately shows a reasonable approximation of the trace while the user waits for the resampling to complete in the background.

5.4 Summary

In this chapter we have covered the labscript suite components involved in preparing an experiment. Labscript formed the core of this, with a high-level API for defining how hardware is connected together and for commanding experiment logic. The use of this API allows us to simplify the control interface for heterogeneous hardware and, behind the scenes, automatically generate complex clocking signals. We also discussed how we use this API to automatically document as much of the experiment preparation stage as we can. While experiment logic greatly benefits from being defined in a text-based environment, we introduced graphical programs for managing parameters and the creation of shots (runmanager), as well as visualising the expected output signals of the hardware (runviewer). We also showed how runmanager makes it easy to traverse complex parameter spaces while countering systematic drifts of the experiment apparatus, an important requirement is most modern scientific experiments. Combined, these programs form a powerful trio for preparing complex, precisely timed, scientific experiments.

14. It is currently set to `3*2000` but will be dynamically linked to the screen resolution in the future.

Chapter 6

Executing experiments with the labscript suite

While most existing control systems (at least for ultracold atom research) consider experiment execution to involve only the running of an experiment on the apparatus, we consider analysis to also be a critical part of experiment execution. This is particularly important if you wish your control system to 'close the loop' by feeding the results of analysis back into the experiment automatically.

In this chapter we discuss the execution of a shot on the experiment hardware using BLACS in §6.1, followed by a discussion on the integration of existing control systems and distributed hardware in §6.2. We then discuss automated analysis using lyse in §6.3 followed by 'closing the loop' in §6.4, which ties together the entire control system presented thus far.

6.1 BLACS

BLACS is the primary interface between experiment shot files created by runmanager, and the hardware devices that control the apparatus. BLACS provides a graphical interface for users to manage the execution of shots, and manually control the output state of hardware devices. In order to support heterogenous hardware, the functionality of BLACS can be extended by developers (who implement support for custom devices) through the provided BLACS API. BLACS thus broadly consists of a set of device code that interfaces with the hardware and provides programmatic and manual control of that hardware, which we discuss in §6.1.1, and a shot management routine that receives shot files from runmanager and schedules their execution on the apparatus, which we discuss in §6.1.2.

6.1.1 Device tabs

BLACS creates a tab, in the GUI, for each device it is to control. This information is sourced from a lab connection table, defined using the labscript API, which is kept up to date with the current configuration of hardware in the lab. Much of the BLACS GUI is thus dynamically generated, creating an interface suited to a particular apparatus configuration rather than enforcing a particular style. These tabs encapsulate three components: the code

that produces the graphical interface, the worker process(es) (which communicate with the actual hardware), and a state machine which handles communication between the GUI and the worker process(es).

6.1.1.1 The graphical interface

Each tab GUI is generated from a set of standard components in order to bring uniformity to the control of heterogeneous hardware. This also simplifies the process of adding support for new hardware devices (see §7.1.2) as the author of the device code does not require knowledge of the GUI widget toolkit. Each tab comprises the following sections (see figure 6.1):

1. device management shortcuts (such as restarting the device),
2. a region (usually hidden) for displaying error messages from the worker process,
3. arrays of 'manual' controls for interacting with each of the device's input and output channels when shots are not running,
4. custom controls specific to a particular device (for example status indicators), and
5. the current state of the state machine (see §6.1.1.3).

The most prominent feature is the arrays of manual controls. These are particularly useful for manual debugging of an experiment apparatus outside of running shots. For easy identification, each channel is automatically named with both the hardware output port, and any assigned name from the lab connection table. All analog values also have an associated dropdown list, where the current unit is displayed. Unit conversions are automatically determined from the lab connection table (where they are defined using the labscript API, see §5.1.9). This makes debugging simpler as you can immediately be sure of the output quantity in real world units (for example, the strength of a magnetic field). All output controls can be locked via a right-click context menu to prevent accidental change of their state, which is particularly important when controlling sensitive equipment that can be damaged. For analog quantities, the default step size used when incrementing or decrementing the value[1] can also be customised via the right-click context menu.

The values displayed in the manual controls are also coupled to the hardware device capabilities. The device code that programs the hardware (see worker processes in §6.1.1.2) has the ability to return a new value for each channel, each time the device is programmed, allowing the quantised, rounded or coerced value to be returned such that the manual control faithfully displays the output state. BLACS also provides an architecture to periodically poll device values for devices that support such queries. This is particularly important for devices that are not physically restricted to being controlled by a single user (for example, devices controlled via a web interface) or devices that don't remember their state after

1. Incrementing or decrementing the value can be done using the up/down arrows next to the value, the mouse scroll wheel, or the arrow keys on the keyboard. The page up/down keys can also be used, which will adjust the value by 10 times the step size. This is distinct from typing a value directly into the widget, which is not affected by the step size. However both incrementing/decrementing and typing a value in will be equally affected by any quantisation demanded by the hardware device, which we discuss in the following paragraph.

being power cycled. For such devices, BLACS continually compares the device state with the values displayed in the GUI. If a difference is detected, BLACS presents the user with options to select either the device state or the local state on a per output basis (see figure 6.2).

6.1.1.2 Worker processes

For each device, BLACS launches at least one separate Python process (a worker process) for communicating with the hardware. BLACS communicates with the worker process through our own remote procedure call (RPC) protocol. The python process(es) run a thin wrapper around a specified Python class, which allows the parent process (in this case BLACS) to remotely call methods of the class in the worker process. A method in the worker process is invoked by the tab state machine (see §6.1.1.3), via a message sent over a ZMQ socket. The only task of a worker process is to process any data that is sent to it (via the invocation of one of its methods), interact appropriately with the hardware device(s) it manages, and return any relevant data to the state machine. A third party software library, used to interact with a hardware device (typically provided by a hardware manufacturer), is then only loaded within the isolated worker process. There are several benefits to this 'sandboxing' model, which have been previously outlined in §4.4.5. Details on writing the code for a worker process can be found in §7.1.3.

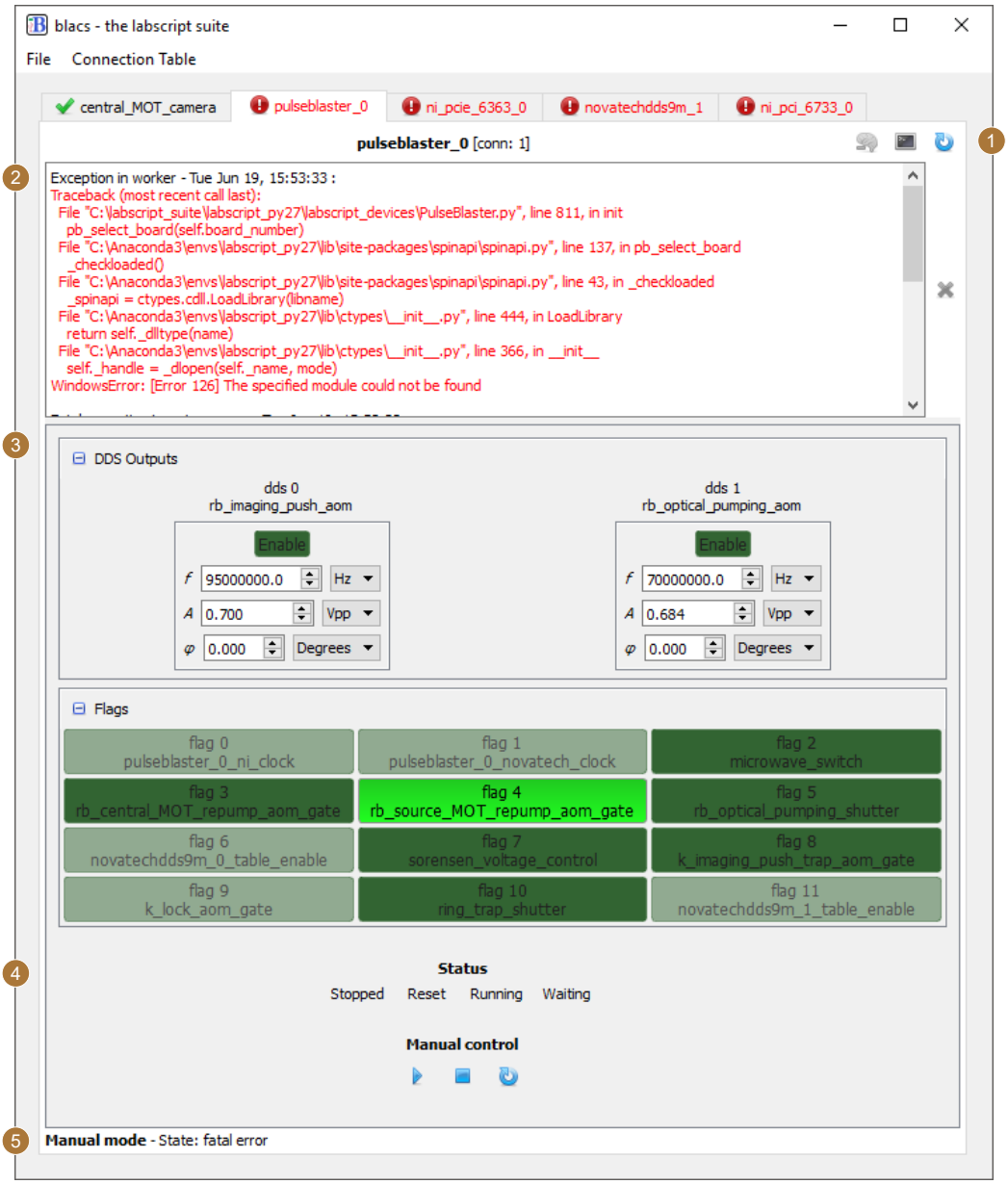
As previously implied, we have implemented the ability for a BLACS device tab to spawn multiple worker processes. This is particularly useful for devices that handle both inputs and output, and whose API allows these inputs and outputs to be separated and managed by separate processes. An example of such a device is a National Instruments acquisition card such as the NI PCIe-6363. For this device, we spawn three worker processes: the first handles analog and digital outputs, the second handles analog acquisition and the third handles monitoring of a counter in order to measure the lengths of any waits.

Multiprocessing also results in a reduction in device programming time prior to the start of an experiment shot. Most device programming is I/O bound (not limited by the processing power of the PC). Simultaneously programming all devices used in a shot thus typically completes in the time it takes to program the longest device (rather than the sum of all programming times for sequential programming).

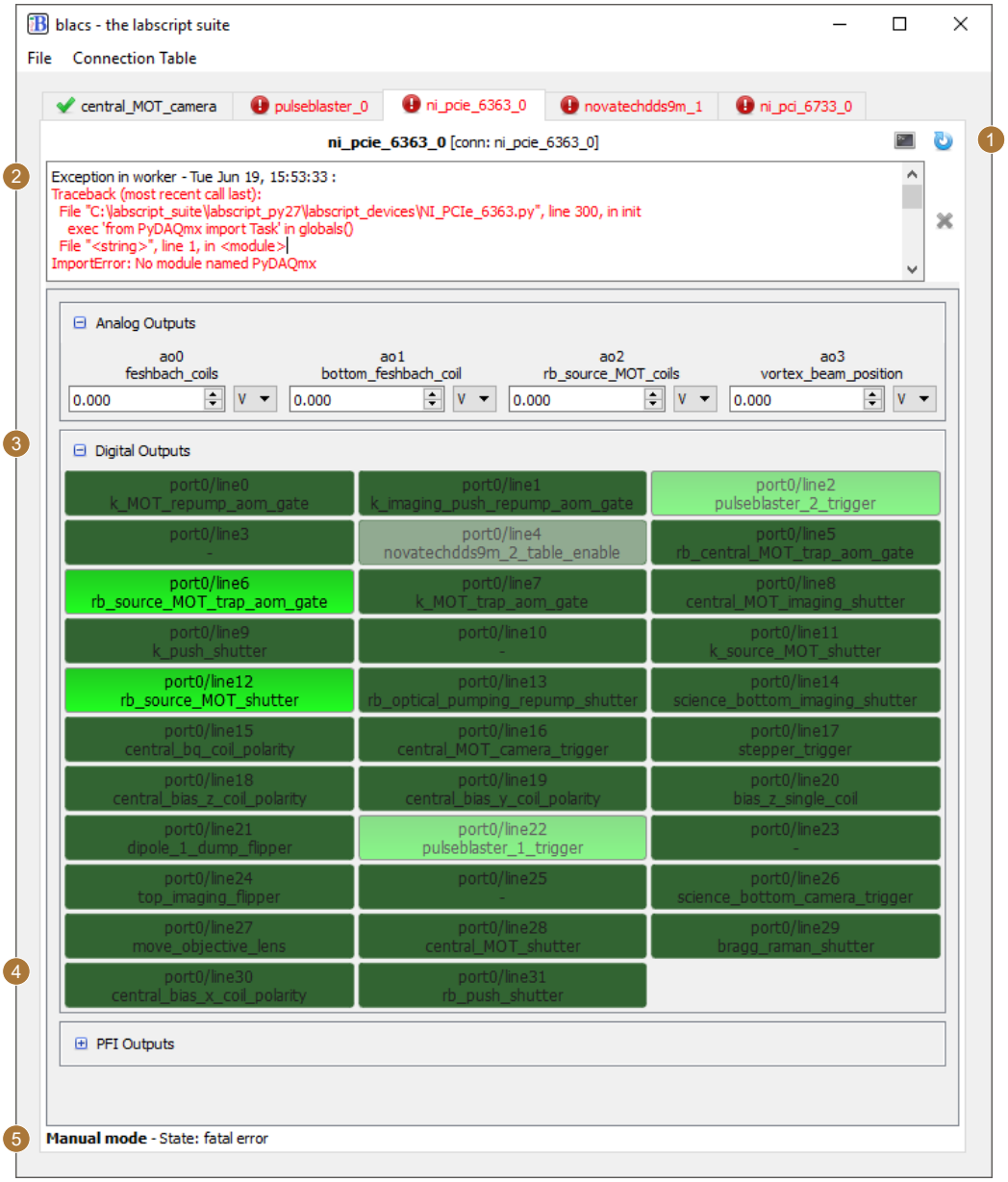
6.1.1.3 State machine

One of the major changes in BLACS v2.0 (written and released after our paper [8] was published) was the introduction of a more advanced state machine for each device tab. State machines are an important tool in building complex systems as they enforce a workflow (in this case, for GUI-hardware interaction) which improves the stability of the control system. By using a state machine, we enforce control over what actions can be taken at any given time, improving the robustness of our control software. For example, manual controls on the BLACS front panel should not be able to control hardware devices that are under precision timing while executing a shot. A state machine allows such events to either be discarded or queued until an appropriate time, under a consistent set of easily defined rules.

The aim of this state machine is to manage the execution of the device-specific code described previously, which falls into the categories of GUI code and worker-process code.



(a) An example of a BLACS tab for a PulseBlaster DDS-II-300-AWG device. The numbered labels match the listing in the main body text of §6.1.1.1.



(b) An example of a BLACS tab for an NI PCIe-6363 device. The numbered labels match the listing in the main body text of §6.1.1.1.

Figure 6.1: (a) and (b): An example of devices tabs for two different hardware devices. Despite the entirely different capabilities of the devices, we provide a unified interface for control of those devices. Note that we have hidden the BLACS queue in this image along with the other 3 device tab regions that were visible in figure 4.5.

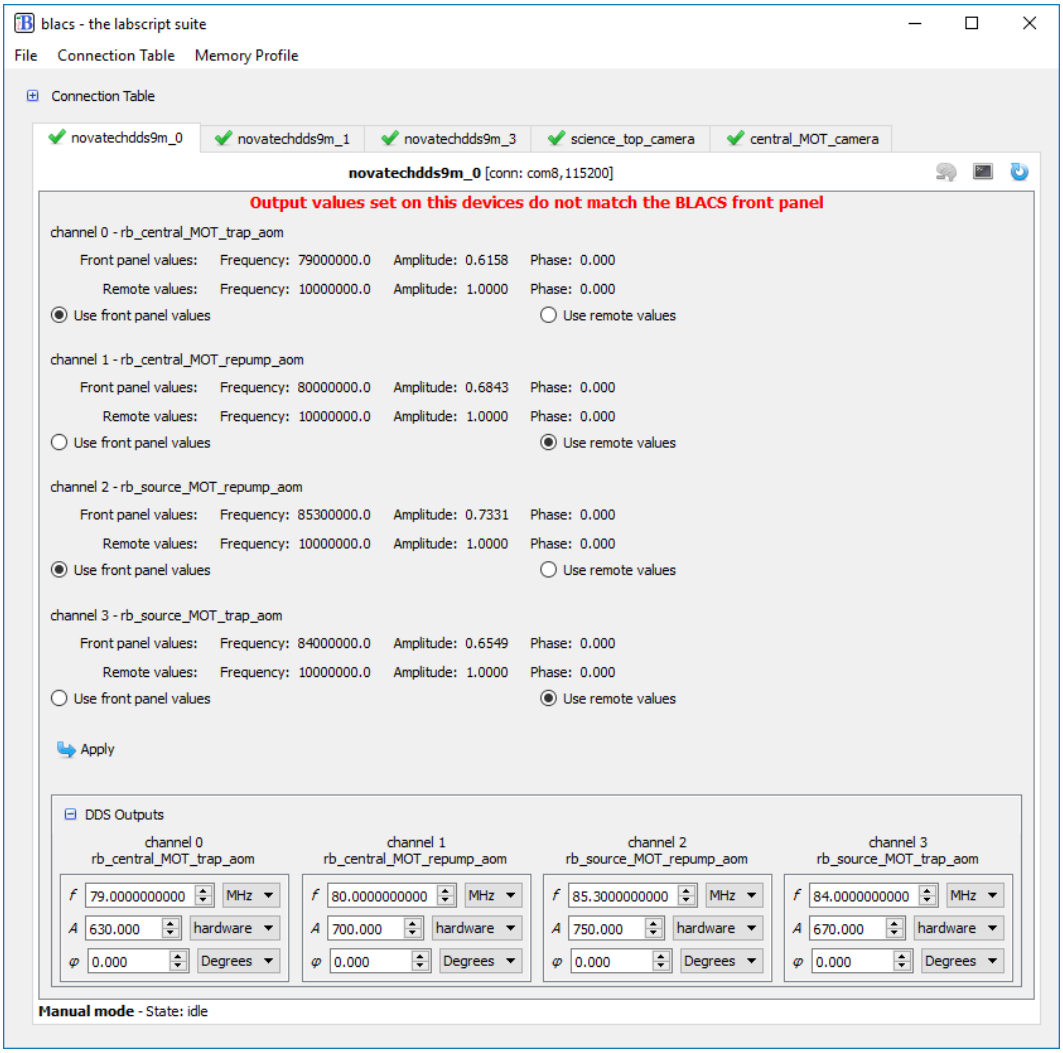


Figure 6.2: An example showing how devices in BLACS can monitor the consistency between the front panel state and the output of the device (when not running a shot). Here we show a Novatech DDS9m device that has just been power cycled, which causes the output states to reset to a default setting. BLACS detects an inconsistency between the front panel values of BLACS and the output state reported by the device, and presents the GUI pictured above. The user can then to choose either use the local or remote value for each output channel. Once selected, the front panel values of BLACS are updated to the selected value and the device is reprogrammed to match, restoring consistency.

This code exists within Python methods of the device classes (which we cover later in §7.1), and so will be referred to in this section as the execution of a 'GUI method' and a 'worker process method' respectively. We have implemented a non-standard nested state machine, for which we will coin the term *2D state machine*. It consists of two orthogonal sets of states (which we term the inner and outer states) which are linked by the device code. This architecture differs from a standard nested state machine in that it is not hierarchical (events are not passed to the parent state machine as in the hierarchical finite state machine). Our implementation is also unique in that the workflow of the inner (dimension of the) state machine is identical regardless of the outer state.

The outer dimension follows a classical state machine. There are four possible states (which we call *modes* to distinguish them from the states on the inner dimension):

- mode_manual,
- mode_transition_to_buffered,
- mode_buffered, and
- mode_transition_to_manual.

These four modes represent the two possible modes of operation for the hardware; manual control from BLACS or stepping through instructions during execution of an experiment shot, and the transitions between these modes (where the programming required to change the mode of the device, for example the programming of hardware instructions, usually takes place).

The inner dimension of this two-dimensional state machine is similar to the state machine that existed in BLACS v1.0. There are 5 possible states:

- idle,
- execute (part of) GUI method,
- request execution of worker process method via ZMQ,
- wait for results from worker process method via ZMQ, and
- fatal error.

The inner state machine spends the majority of time in the idle state where it waits for an event to become available from a queue. Events are typically placed in the queue in response to user actions (for example clicking one of the manual control buttons), the 'queue manager' processing a shot (see §6.1.2), or the timeout of a timer (for example for regular status checks of the hardware).

We define GUI methods that may be queued in the inner state by using a Python decorator, which abstracts away the state machine so that users can call the Python method as normal and be assured that it will always run as part of the state machine (although this enforces the requirement that such methods will return immediately and not return a result to the caller). The decorator itself takes arguments that indicate the modes (of the outer state machine) that the GUI method is allowed to run in, and whether the method should remain in the event queue until the outer state machine enters a mode where it can run, or

if it should just be deleted once it reaches the head of the queue. We also provide an option to only run the most recent entry for a method if duplicate entries for the GUI method exist in the queue (albeit with different arguments). This is particularly useful for methods that take a long time to complete but which may be queued up rapidly, for instance a user rapidly changing output values of a device that is slow to program. An example of how you might use the state machine is shown in the definition of a GUI method in figure 6.3.

The state machine for each device tab runs in its own thread and follows a well defined workflow (also shown graphically in figure 6.4) which can be influenced by the device code. When an event is available in the queue (that can run in the current mode of the outer state machine), the inner state machine transitions to the 'execute GUI method' state, and calls the Python method that was queued up. As this method likely interacts with the GUI, the method is executed in the main thread (via a request to the GUI event loop provided by the Qt widget toolkit). This method executes (temporarily blocking interaction with the GUI) until it either completes, or hits the `yield` Python keyword. The `yield` keyword in Python returns control of the program to the calling code (in this case our state machine). The `yield` keyword also allows the called method to return information to the calling code (for example the `data` variable would be returned if called as `yield(data)`). We utilise this to allow the GUI method to request that the inner state machine transition through the inner states relating to the worker process, in this case by calling:

```
yield(self.queue_work('worker_name', 'worker_method_name'))
```

If such a statement is encountered, the inner state machine enters the 'request execution of worker method' state and requests the named worker process execute the named method. It then immediately transitions to the 'wait for results from worker' state. Once results have been received from the worker process (after it has run the worker method), the inner state machine re-enters the 'execute GUI method' state, passing in the results from the worker process as the return value of the `yield` keyword, and continues with the execution of the GUI method from the point it left. This continues until the execution of the GUI method is complete, where the state machine then enters the 'idle' state again. The exception to this is if a Python `Exception` is raised in the GUI method, in which case the state machine enters the 'Fatal error' state. The GUI method may also request a change in the outer state machine mode, which then determines which events can be processed when the inner state machine next returns to the 'Idle' state.

This results in a flexible state machine architecture that allows the device code to control some portions of the state machine, while maintaining a fixed state machine structure across device tabs. By exposing the internals of the state machine only via the BLACS tab GUI methods, we can abstract away much of the state machine implementation and simplify the necessary coding skills needed to implement support for new devices. We believe this is a critical requirement of meeting the flexibility goal of our control system, and further details on the simplicity of adding support for new devices is discussed in §7.1. Our state machine architecture also allows us to provide a consistent and responsive GUI to a user by obscuring hardware specific details and offloading these to a separate process.

```
class MyDevice(blacs.device_base_classes.DeviceTab):
    # only run in MODE_MANUAL and keep the state in the queue until
    # the mode condition is met
    @define_state(MODE_MANUAL, True)
    def transition_to_buffered(self, h5_file, notify_queue):
        # set the mode to MODE_TRANSITION_TO_BUFFERED
        self.mode = MODE_TRANSITION_TO_BUFFERED

        # define the set of arguments and keyword arguments
        # to be passed to the worker processes
        args, kwargs = (h5_file,), {}
        # Yield to the state machine so that the worker process
        # can be run
        result = yield(self.queue_work(self.primary_worker,
            'transition_to_buffered', *args, **kwargs))

        # check that everything was successful...
        if result:
            # success!
            # update the mode and notify the caller
            self.mode = MODE_BUFFERED
            notify_queue.put([self.device_name, 'success'])
        else:
            # Failure!
            # notify the caller
            notify_queue.put([self.device_name, 'fail'])
            # queue up a method in the state machine
            # to return to MODE_MANUAL and instruct the
            # worker to program the device ready for
            # manual operation
            self.abort_transition_to_buffered()

    @define_state(MODE_TRANSITION_TO_BUFFERED, False)
    def abort_transition_to_buffered(self):
        ...
```

Figure 6.3: An example of how one might define the GUI method for triggering the programming of devices so that they are ready for buffered execution of a shot (ready to step through hardware instructions). The GUI method `transition_to_buffered` has been decorated in order to ensure it is only run as part of the state machine, which means the method will sit in the inner state machine's event queue until the outer state machine mode is set to 'MODE_MANUAL'. When finally executed by the state machine, the method updates the mode of the outer state machine, and yields to the inner state machine in order to tell a worker process to transition into buffered mode (which typically involves programming the table of hardware instructions from the HDF5 shot file). If successful, the outer state machine mode is updated again and the caller of the method (in this case the 'Queue Manager') is notified of the result. If unsuccessful, we call the `abort_transition_to_buffered` method (which is also decorated as a GUI method) which queues up a new event for the inner state machine. In practice, common functionality like these methods are abstracted away from the user and contained within the `blacs.device_base_classes.DeviceTab` class. They are implemented in a similar (but more generalised) way to the code shown here. For example `transition_to_buffered` is actually written to support an arbitrary number of worker processes. Further information on adding support for new devices (and how to use the state machine architecture) is included in §7.1.2.

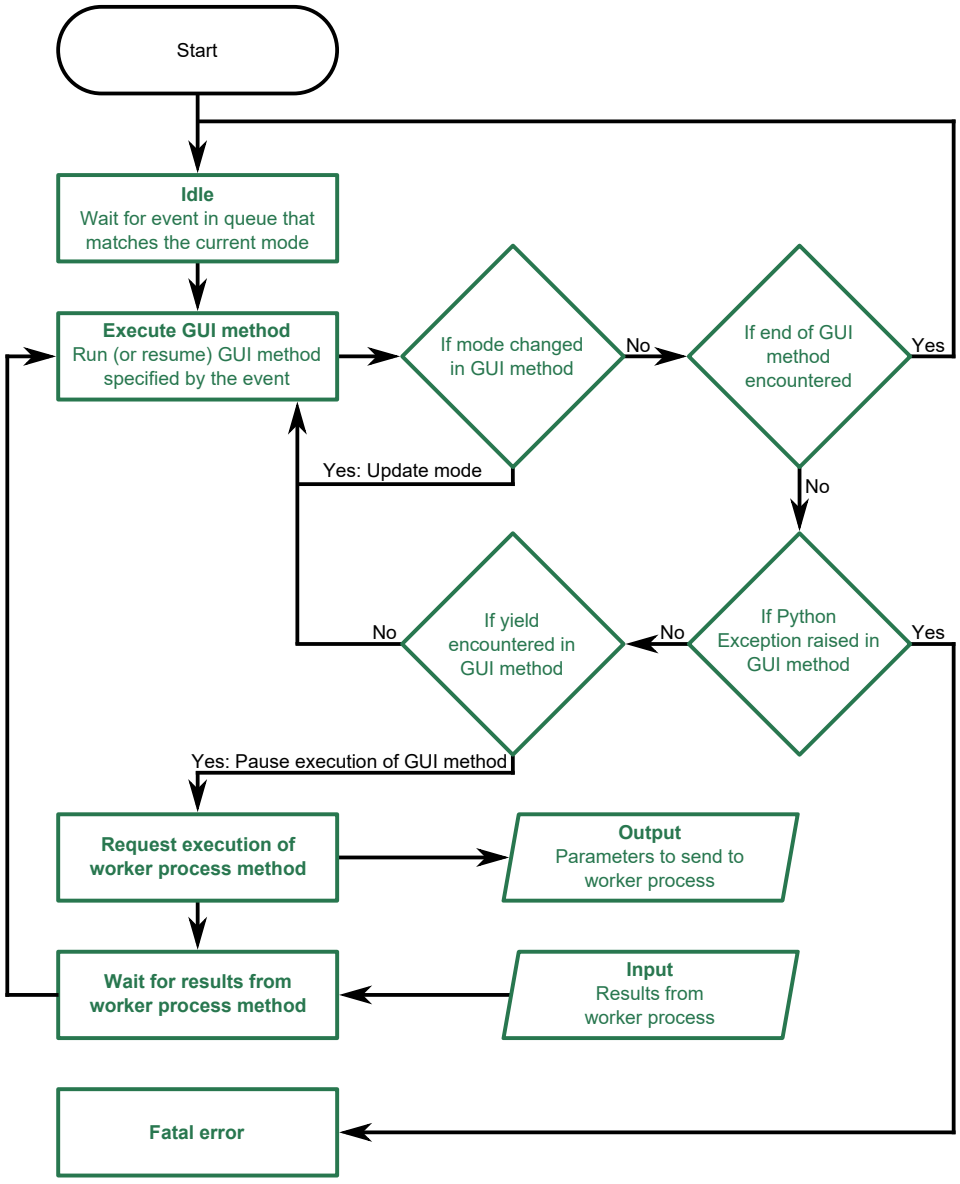


Figure 6.4: A flowchart of the logic for the BLACS state machine as described in the main body text in §6.1.1.3.

6.1.1.4 Handling waits

As introduced in §5.1.7, in order to use waits, one of the devices in BLACS must monitor the length of each wait so that a mapping between labscript time and experiment time can be made. This mapping is then used by analog acquisition devices in BLACS to correctly break up acquired traces into the requested acquisitions. The length of the wait is also used in real (software) time by the wait monitor in order to ensure the experiment is not stuck in a wait forever.

The current wait monitor implementation uses one of the inbuilt counters that are in many National Instruments acquisition devices, however other implementations are of course possible if support is added when creating the device classes (see §7.1). At the completion of an experiment shot, the wait monitor calculates the durations of each wait (based on data it acquired during the shot) and writes these to the shot file. The wait monitor then broadcasts a ZMQ event indicating this has been completed. Device code that relies on the wait information (for example, for breaking up analog acquisitions into the requested set of traces), waits for this event to be received during the transition to manual mode, before performing any action. This ensures that the measured lengths of the waits are always available in the HDF5 file when required.

6.1.2 Shot management

The primary purpose of BLACS is to execute experiment shots on the lab apparatus. File paths to shots are typically received by BLACS over ZMQ, but can also be loaded directly through the BLACS GUI (either via the file menu, or by dragging and dropping onto the queue). Prior to accepting the shot, BLACS compares the connection table of the shot to the lab connection table and ensures that the shot is compatible with the current configuration of the laboratory hardware. Connection table compatibility requires that the shot connection table is a subset of the lab connection table. This ensures that old experiments can not be run on hardware that is no-longer configured correctly, preventing damage or unexpected results. Shots that pass this check are added to a queue, which is visible in the BLACS GUI (see figure 6.5).

The queue is processed by a thread in BLACS, which we term the 'queue manager', that takes the top-most shot in the queue and, in turn, executes it. Shot execution follows the following pattern (a flowchart of this process is also shown in figure 6.6):

1. For each device in use in the shot, a message is sent to the corresponding device tab state machine indicating that the device should program the device for hardware timed execution of a shot. These messages are sent asynchronously, which ensures devices program in parallel if possible (subject to the state machine being available to process the message). Included in this message is the path to the HDF5 shot file, which each device tab ultimately passes to a worker process that in turn, reads out the hardware instructions and programs the hardware. During this programming, the device tab enters the mode 'transition_to_buffered' (see §6.1.1.3).

2. The queue manager then waits until all devices have reported they have programmed, at which point all device tabs in use should be in the 'buffered' state machine mode. If a device does not report it has completed within a 5 minute timeout, or a device

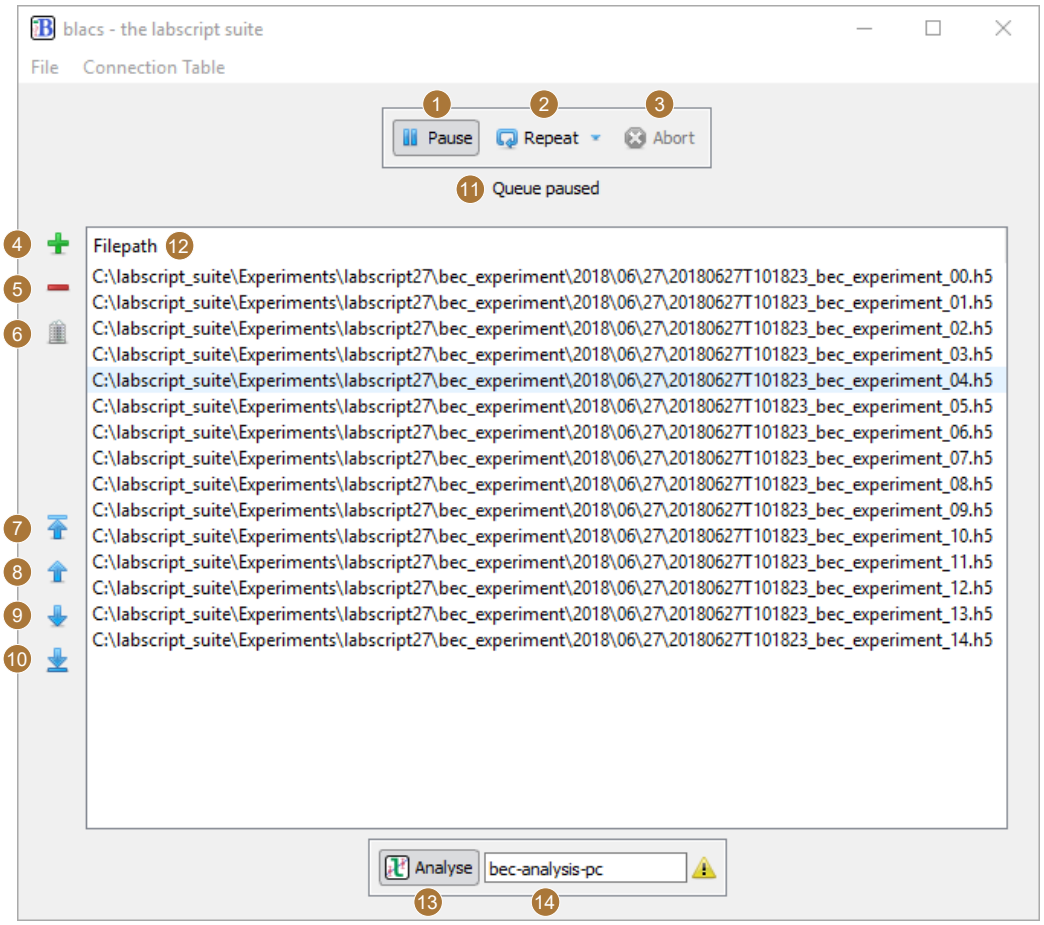


Figure 6.5: The queue manager GUI within BLACS (with the device tabs hidden). (1) The pause button stops the queue from processing new shots (a currently running shot will finish). (2) The repeat button, when enabled, will duplicate a completed shot and either place the duplicate at the bottom or the top of the queue (depending on the mode selected). (3) The abort button immediately stops the execution of the current shot and returns hardware to manual mode. (4-5) Buttons to add or delete selected shots from the queue. (6) A button to clear the entire queue. (7-10) Buttons to reorder selected shots within the queue. (11) The current status of the queue is displayed here. For example, the status may indicate that devices are currently being programmed, the master pseudoclock has been triggered and the experiment is running, or that acquired data is currently being saved into the HDF5 shot file. (12) The list of shot files in the queue, in the order they will be executed (the topmost is executed first). (13) A button to enable or disable the forwarding of shots to lyse for analysis. (14) The network hostname of the PC running lyse.

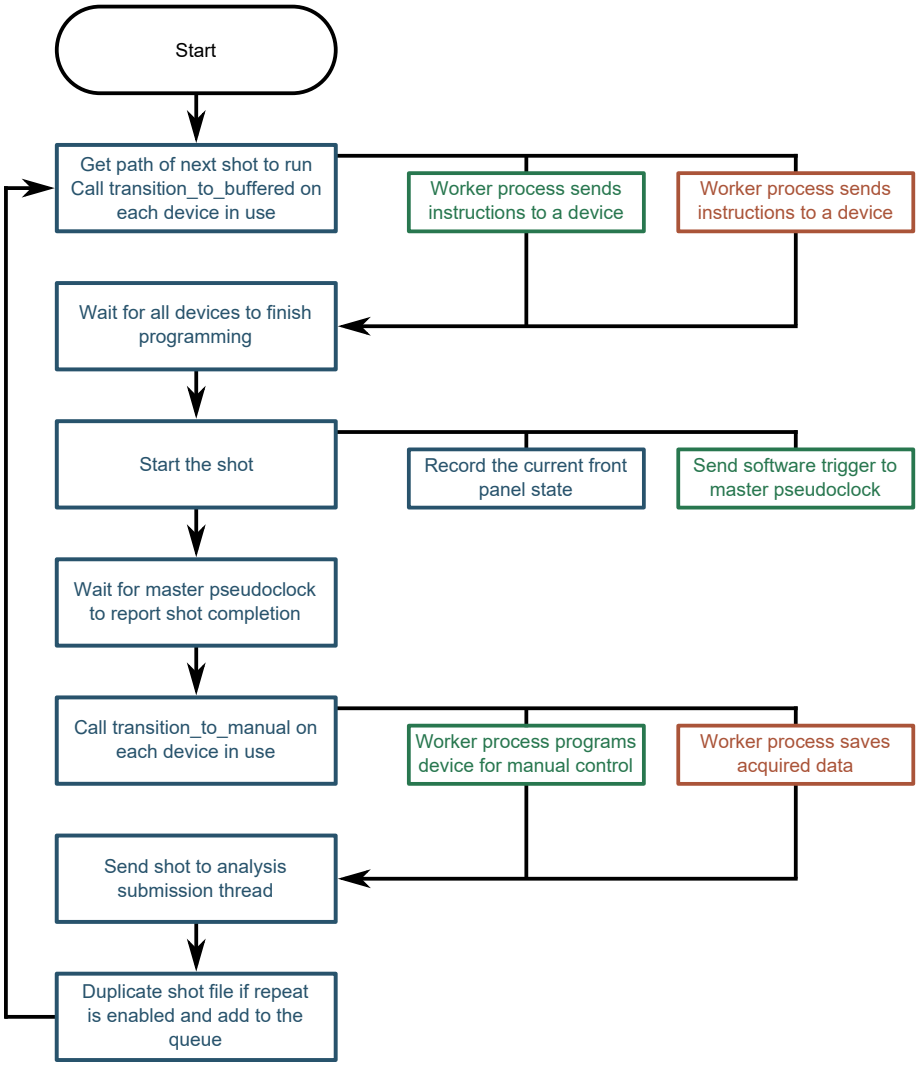


Figure 6.6: A flowchart of the logic for the BLACS queue manager. For brevity, we have not included the logic for pausing the queue via the GUI or handling error conditions. See the listing in the main body text of §6.1.2 for further details.

reports an error has occurred during programming, the queue manager aborts the shot by pausing the queue, instructing all device tabs to abort, and replacing the shot at the top of the queue.

3. Provided all devices report they are ready, the queue manager proceeds with starting the shot. This involves recording the current state of all manual controls (as these usually affect the initial values of the shot and may affect results in certain experiments) and then instructing the master pseudoclock to begin execution of the programmed instructions.

4. The queue manager then waits for the master pseudoclock to report that the experi-

ment shot has completed. If an error occurs in a device tab during a shot, the queue manager aborts the shot (as previously described) and pauses the queue.

5. Once a shot has completed, the queue manager instructs all device tabs to 'transition to manual' mode. At this stage, device tabs enter the 'transition_to_manual' state machine mode where they save any acquired data and reprogram the hardware device for manual operation via the BLACS GUI. Again, if errors occur during this process, the queue manager aborts the shot as before, but with the additional step of cleaning any saved data from the HDF5 file (so that the shot file is returned to the state prior to execution).

6. The path to the shot file is now sent to a separate thread that runs a routine for managing submission of shots to lyse for analysis. This routine forwards the shot file paths to the lyse server specified in the BLACS GUI if analysis submission is enabled (see figure 6.5 (13–14)). If lyse does not respond to these messages, the shot file paths are buffered until such time as lyse does respond, to ensure no shots are missing from analysis.

7. Finally, the queue manager checks the state of the repeat button in the BLACS GUI and, if required, duplicates the shot (minus the acquired data) and places the duplicate in the appropriate place in the queue.

6.1.3 Plugins

To cater to the variety of lab environments, BLACS supports custom user code (that is not covered by a device implementation) in the form of plugins. Plugins allow custom graphical interfaces to be created through the addition of menu items, notifications, preferences editable through a common preferences panel, and custom tabs that sit alongside device tabs. Plugins are also provided access to a variety of internal BLACS objects, such as the shot queue, and can register callback functions to be run when certain events happen (such as a shot completing). This provides a powerful basis for customising the behaviour of BLACS in a way that is both modular and maintainable, providing a way to include optional conflicting features without needing to resolve the incompatibility. Plugins can be easily shared between groups, allowing for a diverse variety of control system interfaces that are all built on a common platform. We have developed several plugins at Monash, which are detailed in the following sections, and demonstrate the broad applicability of the plugin system.

6.1.3.1 The connection table plugin

This plugin is included in the default install of BLACS, and provides a clean interface to manage the lab connection table that BLACS uses to automatically generate the device tabs and their graphical interfaces. The plugin inserts a menu item that provides shortcuts for:

1. editing the connection table Python file,

2. initiating the recompilation of the connection table, and

3. editing the preferences that control the behaviour of the plugin.

The preferences panel allows you to configure a list of HDF5 files containing runmanager globals to use during the compilation of the connection table (commonly used as unit conversion parameters), as well as a list of unit conversion Python files, to watch for any changes. At startup, the plugin launches a background thread that monitors changes to these files, as well as the connection table Python file and compiled connection table HDF5 file. If any modifications are detected, a notification is shown at the top of BLACS informing that the lab connection table should be recompiled. If recompilation is chosen by the user, the plugin manages the recompilation of the connection table using the runmanager API and output from this process is displayed in a window. Once recompilation is successful, the plugin relaunches BLACS so that the new connection table is loaded. This ensures that BLACS is using the same knowledge of the experiment apparatus as any future shots will when they are created by runmanager (assuming they share the globals files used).

6.1.3.2 The labwatch plugin

The labwatch plugin is designed to link the process control systems in our lab with our scientific control system. The labwatch plugin was created by fellow student Shaun Johnstone, in consultation with myself (on the BLACS side) and Martijn Jasperse (on the process control side). As discussed in §3.3, our process control systems log to a standard Linux syslog server. This server however, has also been configured to mirror the syslog messages over a ZMQ pub/sub socket. The labwatch plugin subscribes to these messages from within BLACS and monitors them based on a user-supplied configuration.

The user-supplied configuration is made through a preferences panel created by the plugin. Here, the user specifies a list of identifiers that allow the plugin to filter the syslog messages for a certain device and/or quantity (such as temperature or flow rate) to be observed. Allowed states or upper and lower bounds for values are then specified for each quantity that is being observed. When the syslog messages indicate that the quantity is out of bounds, the BLACS shot queue is automatically paused an a notification is presented to the user. This ensures that BLACS does not allow experiments to run if the apparatus is not functioning correctly, a key requirement of automatic data taking systems that are not under constant observation by human operators. The monitoring of process control systems is, however, read-only, preserving the separation between process and scientific control systems.

6.2 Secondary Control systems

While BLACS was designed to be a comprehensive interface for all hardware, the practicalities of this are often more complex. For example, some devices do not provide libraries or drivers that can easily be interfaced with Python or may come bundled with a control program of their own that would be easier to use than writing an interface in Python. It is also common to see control interfaces running across multiple PCs, either to create a multi-user control interface or due to PC-to-device connection requirements that necessitate the use of more than one PC. Additionally, operational laboratories may already have control software that they would like to continue using as part of the labscript suite. Fortunately

the labscript suite provides a simple solution to all of these scenarios via the concept of 'secondary control systems'.

While the labscript suite model requires that all hardware devices have an associated device tab in BLACS (that manages the programming of the device), there is no requirement that the device tab and worker process actually do the programming of a hardware device directly. We can thus interact with almost any other system (which we term a 'secondary control system') by writing a BLACS device tab as an intermediate communication broker between BLACS and the other system, provided that other system has a communication mechanism that is accessible through Python. For systems that cannot interface directly with Python, it is often possible to write additional intermediate brokers to ultimately pass messages between a BLACS worker process and the other system. The labscript suite is thus not limited to controlling devices with direct support, like most control systems, but can instead be used as a powerful tool for coordinating other control systems as well as directly attached hardware devices.

6.2.1 BIAS

BIAS was the first program we interfaced with the labscript suite as a secondary control system. BIAS was written by fellow student Martijn Jasperse, with development commencing at approximately the same time as BLACS. BIAS was designed as a standalone image acquisition program, implemented in LabVIEW due to the wide compatibility with a variety of commercial camera drivers. It incorporates both manual and triggered acquisition, along with optical density calculations for absorption images of ultracold atoms, multiple region-of-interest selection and 2D Gaussian fitting to images within these regions.

As the projects were conceived at similar times, they were initially developed independently of each other. It was also thought that independent, asynchronous control of region of interest selection and the atom cloud fitting would be beneficial in order to improve cycle time. Ultimately, we have moved much of the fitting into lyse, however we still use BIAS as an interface to our cameras and to select regions-of-interest. In order to integrate BIAS with the rest of the labscript suite, we developed a simple protocol for communicating between the two programs[2]. BLACS communicates to BIAS over a raw network socket, via a device tab, sending information such as the path to a HDF5 shot file for the current shot, and messages to indicate when the shot is complete or has been aborted. BIAS then acts like any other worker process would, controlling the hardware as appropriate, and reading and saving data to and from the shot file. Further details on the underlying implementation of BIAS can be found in Martijn Jasperse's thesis [106].

6.2.2 Others

While not as comprehensive as BIAS, developers in our research group have written several small programs that also act as secondary control systems. Chris Billington wrote a Python script that imitates the communication protocol of BIAS, which can be used to build custom imaging software or interface with existing software that a lab may currently use. For

2. It should be noted that his protocol is only defined within BIAS and within the device tab code. It is not built into BLACS, which means that other secondary control systems can use whatever protocol is appropriate.

example, he used this at another institution to copy images, saved independently by their camera software, into the HDF5 file on shot completion. This meant that no customisation was needed for the existing software (at the expense of losing the ability to control camera parameters via labscript). This code can be found in the labscript_utils repository [154].

In our labs at Monash, Shaun Johnstone created a secondary control system to provide semi-hardware timed control of Zaber translation stages. A Raspberry Pi was configured with a custom Python script that was commanded by BLACS as a piece of custom hardware. The custom Python script was written to also respond to digital triggers received via the Raspberry Pi GPIO pins, coordinating the movement of two translation stages to positions informed by previous communications from BLACS (which are typically shot specific). Further details of this can be found in Shaun Johnstone's thesis [100].

6.3 Lyse

Lyse (short for 'analyse') was developed to provide a modular analysis framework for performing an arbitrary set of analyses on shots that have been executed by BLACS. The lyse application provides a graphical interface (see figure 4.6) for managing the analyses to perform, and displaying a table of data from executed shots that have been loaded into lyse for analysis. It is designed to run continuously as new data is acquired, or as a standalone system for analysing data that was previously acquired. Analysis routines are defined as Python scripts, which the lyse application queues up for execution as new shot data arrives (or on user request). Lyse maintains the table of shot data, such as runmanager globals and analysis results, as a Pandas DataFrame, which it updates automatically as new shots arrive from BLACS and/or analysis routines complete. This DataFrame is made available to analysis routines and is also used as the data store backing the previously mentioned table in the GUI.

The graphical interface is only designed to facilitate the analysis of data and display a table of results for a set of loaded shots. It is important to stress that the graphical interface is not designed to be used to actually configure or define the analyses beyond indicating which analysis routines to use and the order in which they should run. This is distinct from other analysis software that often provide a unified interactive graphical interface for the entire analysis process (for example, Igor Pro). The main benefit of this approach is the flexibility to control and customise every aspect of your analysis workflow, including the underlying implementation of analysis techniques. This design decision follows the philosophy we applied to runmanager, BLACS, and the labscript API, which was previously described in §4.3.3 and chapter 5. The definition of an analysis routine is best described textually, using a programming language with an API that provides the user with helpful tools to simplify the mundane and/or standard aspects of analysis. The automation of performing analysis on acquired data, including the types of analysis and the order in which separate analysis routines are performed, is best managed via a graphical application. Unlike, for example, MATLAB, we don't attempt to embed a text editor into our graphical application, instead leaving this to projects specifically designed to produce outstanding text editors.

Analysis routines are defined as Python scripts, by the end user, and make use of the

lyse API, which provides convenience methods for obtaining data from the shot files or the Pandas DataFrame. We allow two types of analysis scripts to be loaded into lyse: those designed to analyse data from a single shot (see §6.3.2) and those that analyse data from a set of shots (see §6.3.3). While lyse (the application) and the lyse API are tightly coupled, it is important to note that the lyse API can also be used independently of the graphical interface, which we discuss in §6.3.4.

The lyse application launches a separate worker process for each analysis script loaded. This follows the same model we used in BLACS and brings the same stability improvements to the application. When a new shot file is loaded into lyse (either by the user or received directly from BLACS), lyse sequentially notifies each worker process that the analysis script should be run. The worker process then executes the Python code contained in the user-specified analysis script. Finally, the worker process sends a dictionary of analysis results saved using the lyse API back to the lyse application where it is added to the Pandas DataFrame and displayed in the GUI.

The worker process also contains code that modifies the default plotting behaviour of matplotlib, the well-known 3rd party Python graphing library we use in lyse. By default, a script using standard matplotlib calls would block execution while displaying the plots as this in an intrinsic requirement of all software that displays a graphical interface. This would result in analysis hanging while plots for one analysis script were inspected and would require plots to be opened and closed each time an analysis script ran. To work around this, the user is instructed to use the standard matplotlib library but not call the `show()` function, which creates and displays any previously defined plots. The worker process code is then responsible for instantiating and drawing each plot, after the user's script has finished executing. This allows the worker process to handle communication with the lyse application asynchronously, provide a custom matplotlib toolbar for the user, and seamlessly update plots with data from the latest run of the analysis script without requiring the plot windows to be recreated. This also allows us to provide options to preserve the zoom level of each plot window, or other custom behaviour, without the need for a special plotting API.

The result of all of these features is a robust framework that can be used to implement arbitrary analysis and produce publication quality plots, using the standard matplotlib library, that update in real time as new data is acquired.

6.3.1 Pandas

Pandas is a general purpose data analysis library for Python. Lyse uses pandas for mediating access to most experiment data, with the exception of time series acquisitions and images which would take up an inordinate amount of space if cached within a Python data structure.

Pandas itself is best described by this quote from the documentation [155]:

> *pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with "relational" or "labelled" data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, real world data analysis in Python. Additionally, it has the broader goal of becoming the most powerful and flexible open source data analysis / manipulation tool available in any language. It is already well on its way toward this goal.*

pandas is well suited for many different kinds of data:

- *Tabular data with heterogeneously-typed columns, as in an SQL table or Excel spreadsheet*
- *Ordered and unordered (not necessarily fixed-frequency) time series data.*
- *Arbitrary matrix data (homogeneously typed or heterogeneous) with row and column labels*
- *Any other form of observational / statistical data sets. The data actually need not be labelled at all to be placed into a pandas data structure*

We use Pandas in lyse to hold a 2D table (a DataFrame) of shot data, where rows correspond to shots and columns correspond to the various data contained in the shots such as global variable values and analysis results. Columns of the DataFrame can also be hierarchical, which we utilise to group analysis results together. DataFrames can be easily indexed and/or sliced to access individual values or subsets of the DataFrame, and pandas provides convenience methods for grouping and iterating over data, which we'll cover further in the following sections. In short, pandas provides a convenient framework for storing and using data, an important basis for our general purpose analysis system.

6.3.2 Single-shot analysis

Single-shot analysis scripts are designed to be run with data from a single shot file. Messages between lyse and an analysis script worker process thus also contain the path to the relevant shot file. The shot file path is then made available to the analysis script via the lyse API attribute: `lyse.path`. Beyond this, the user is free to construct their analysis script as they see fit, with minimal constraints placed on them by lyse. The only real constraint is that plots should be made using the matplotlib library so that lyse can correctly manage the display of plots produced.

While it is possible to directly read experiment data out of the shot file, the lyse API does also provide convenience methods to help with this. The lyse API provides a class that can be instantiated by: `run = lyse.Run(lyse.path)`[3]. This `run` object then provides convenience methods to pull out information from the shot file, such as globals, acquired traces, and acquired images. This object can also be used to save analysis results, for instance by calling `run.save_result_array('OD', a_numpy_array)` or `run.save_result('N_atoms', N)` to save arrays or single values respectively. Globals and analysis results from other (previously run) scripts, can be also access through the Pandas DataFrame in the same way as for multi-shot analysis scripts (discussed in the following section). Results are saved in a HDF group determined from the filename of the script to avoid naming conflicts. When accessing the results through the Pandas DataFrame (for example from a different single-shot analysis script), columns are indexed with a tuple containing the filename and the name of the result, again preventing name conflicts with other results or global variables. For example, a result `'N_atoms'` from a script called `OD_calc.py` could be accessed from a DataFrame object `df` using `df[('OD_calc', 'N_atoms')]`.

3. Named `Run` for unfortunate historical reasons. It should really be called `Shot`.

6.3.3 Multi-shot analysis

Multi-shot analysis routines, as the name suggests, are designed to act on a set of shots. The lyse API provides a method `lyse.data()` which fetches the Pandas DataFrame from the lyse application over a ZMQ socket. Typically, multi-shot analysis routines then slice the DataFrame to contain one or more sequences of shots (usually the last N sequences) and perform analysis on that subset only, although this is heavily dependent on the analysis to be performed. One of the most useful features of the Pandas DataFrame is the ability to group the rows of the DataFrame by a common value in one or more columns using the `pandas.DataFrame.groupby()` method [156]. As the DataFrame contains a column for every global variable and every single-shot analysis result that was saved, this provides a very easy way to collate data with minimal code. For example, a sequence of shots that alternated between rubidium and potassium absorption images would be first be grouped by the global name storing the alkali type, and then these two subsets of shots would be iterated over for further analysis. Higher-order analysis is also possible by iteratively nesting calls to `pandas.DataFrame.groupby()`. This complements the arbitrary parameter space definition from within runmanager, by providing an analysis framework with the flexibility to easily analyse results from such a parameter space investigation. Further real world examples of this will be presented in chapter 8.

6.3.4 Command-line usage

While we recommend running analysis scripts from within lyse, this is not a requirement. Single-shot analysis routines (Python files) can be executed in any Python environment that has the lyse API installed by executing the script from a terminal. In such situations, the `lyse.path` variable would be set to `sys.argv[1]` (the command line argument after the Python filename). Such scripts should explicitly call `matplotlib.show()` in order to produce the plots. As such a call is incompatible with running the script from within lyse (see §6.3), we provide a variable that indicates whether the script has been launched from lyse or not (`lyse.spinning_top`, a reference to the movie Inception) so that the call to `matplotlib.show()` can be placed within a conditional statement.

Multi-shot analysis routines can be executed from the terminal in a similar way. In this case, the Pandas DataFrame should be fetched from a running instance of lyse (via the `lyse.data()` method, which optionally takes `host` and `port` keyword arguments for connection to remote PCs) or loaded from a locally stored DataFrame (for example a previously 'pickled' DataFrame that was written to a file). The latter is particularly useful for users with large data sets as lyse does not provide DataFrame persistence between application sessions. Again, `lyse.spinning_top` can be used to make the script compatible with both the lyse application and the terminal.

Alternatively, if no script is available, the lyse API can be used from an interactive terminal such as IPython. In such a case, the single-shot analysis API will not provide a path in `lyse.path` and results will not be able to be saved to a shot file without first calling `Run.set_group('script_name')` in order to identify the HDF5 group name where results should be stored.

6.4 Mise

In our paper [8], we presented a program called 'mise' (short for optimise) which demonstrated the ability to perform closed-loop optimisation using the labscript suite. As the other components of our system have evolved, we have decided to deprecate mise, and it is currently not supported by the latest version of the labscript suite. Closed-loop optimisation remains an important goal of the labscript suite, but incorporating support for all possible optimisation methods seemed impractical with the architecture of mise. Instead, we have worked to provide appropriate API features so that optimisation routines can be written as lyse analysis scripts. To this end, we now provide a lyse API attribute called `lyse.routine_storage` which allows analysis routines to store persistent information across multiple executions of an analysis routine run by the lyse application. This allows analysis routines to keep a record of the optimisation process (for example, a list of shots in the current generation of a genetic algorithm) which is necessary for many optimisation algorithms to determine when a stage of the optimisation has completed and new shots should be generated. Analysis routines are free to then use the existing runmanager API for modifying global variables, generating shots, and submitting those shots to BLACS. The analysislib-mloop routine [130] demonstrates these new features, adding support for the M-LOOP [157, 158] machine learning package to the labscript suite in less than 300 lines of code.

We believe our latest approach provides the most flexibility by allowing for arbitrary optimisation routines to be implemented at the analysis stage without having to build an entirely custom, standalone application. This is in part, made possible, by our design of accessible APIs that follow the Unix philosophy. We are however continuing to work on improving the functionality of closed-loop optimisation within the entire labscript suite framework, which we discuss further in §9.2.

6.5 Summary

In this chapter we detailed the components of the labscript suite dedicated to performing an experiment. We first covered BLACS, which interfaces with the hardware devices controlling the apparatus. BLACS was designed specifically to provide the robustness needed for continuous operation, which we achieved by utilising techniques from software engineering and process control such as sandboxing and state machines. We also designed BLACS to be extensible, so that it can adapt to future research directions, by providing a plugin framework for new features and a modularised hardware interface, which we'll detail further in the next chapter. BLACS also facilitates the integration of existing or custom control software into the labscript suite workflow, through the secondary control system architecture. We then introduced our analysis framework, lyse, which allows the user to create analysis routines to run on shot files that have been executed by BLACS. The routines can be written to perform analysis on single and/or multiple shot files, allowing complex parameter spaces to be analysed. These analysis scripts can also produce publication quality plots, which update automatically in near real time, as data is acquired. By utilising an API provided by runmanager, these analysis scripts can also be used to create new shots, with globals

based on analysis results, closing the loop and allowing for semi-autonomous control of the apparatus.

Chapter 7

Extending the labscript suite

While we have demonstrated the flexibility of much of the user facing aspects of the control system in the previous chapter, it is also critical to assess the support for the hardware that forms the interface between the control system and the physical apparatus. In this chapter we demonstrate how our architecture makes it easy for developers to add support for new hardware devices.

7.1 Adding support for new hardware devices

As detailed in chapters 5 and 6, the labscript API, BLACS, and runviewer are designed specifically for extensible hardware support. In this section we will detail the minimum requirements for adding support for custom hardware, and demonstrate the simplicity of our architecture.

The labscript suite stores device-specific code in a dedicated Python module called `labscript_devices`. The main code for each device model is then stored in a single Python file within this module, named after the device in question[1]. This Python file then contains, at a minimum, four Python classes: one for the labscript API, BLACS GUI, BLACS worker process, and runviewer. With the exception of the labscript API class, these classes can be named as the developer wishes. The labscript API class must be named the same as the Python file it exists in. For example, the `PineBlaster` labscript class exists within the `PineBlaster.py` file. The four classes must be registered with the labscript suite using class decorators [159] which can be imported from `labscript_devices`: `@labscript_device`, `@BLACS_tab`, `@BLACS_worker`, and `@runviewer_parser`. This allows the labscript applications to find and import the classes when necessary[2]. As we will see in the next sections, the four classes subclass either a labscript-suite-provided class, or classes imported from an existing device, to extend the already provided behaviour.

1. It is important to note that the filename must follow Python guidelines for module names (for example no spaces are allowed in the filename).
2. At the time of writing, this method for registering classes is about to be deprecated. The new method is similar but, instead of using a class decorator, requires that the classes are registered by specifying a fully qualified import path (as a string) in an argument to the `labscript_devices.register_classes()` function. This will enable code for each labscript suite component to be separated across multiple files so that, for example, the BLACS code is not imported within runviewer. It also enables device code to be bundled up into folders, which is more amenable to modular distribution of device code not included within the master repository of the labscript suite.

7.1.1 Labscript

As introduced in §5.1.1, devices fall into one of two categories: pseudoclocks and devices that update their output based on (or are clocked by) an external pseudoclock signal. Labscript provides provides the classes `PseudoclockDevice` and `IntermediateDevice` respectively to help implement new devices of these types.

7.1.1.1 Implementing pseudoclock devices

The most versatile pseudoclock device is one that supports the following features:

1. software triggering, used by BLACS to initially start the experiment shot,
2. support for complex instructions including loops and waits, and
3. external triggering, used to synchronise pseudoclocks to each other and external events.

For the purposes of this section, we will assume that the hypothetical hardware has all of these features so that it can be used as either a master or secondary pseudoclock. In practice, the first feature is only necessary for master pseudoclocks. The second feature is strongly recommended for all pseudoclocks, but is not strictly necessary provided the hardware can be made to output an arbitrary pulse sequence. The third feature is only required if the device is to be used as a secondary pseudoclock or the experiment uses waits.

The labscript class should inherit from `PseudoclockDevice` (or another class that ultimately inherits from `PseudoclockDevice`) and override the `__init__` and `generate_code` methods. Both overridden methods should typically immediately call the base class implementation on the first line of the respective methods. Labscript expects several class attributes to be defined, which detail the specifications and limitations of the device. We show these for the PineBlaster below:

```
@labscript_device
class PineBlaster(PseudoclockDevice):
    # A human readable name for device model used in error messages
    description = 'PineBlaster'
    # The maximum rate (in Hz) at which the clock can tick
    clock_limit = 10e6
    # The smallest unit of time the device can process
    clock_resolution = 25e-9
    # The delay between receiving a trigger and the first clock pulse
    trigger_delay = 350e-9
    # The minimum length of a wait (or the minimum time between
    # reaching a wait instruction and responding to a trigger to
    # resume)
    wait_delay = 2.5e-6
```

The `__init__` method should, at a minimum, be defined as below. Note that the keyword argument `usbport` may be alternatively named, as appropriate for the device. We discuss the passing and storage of additional parameters at instantiation time in §7.1.1.3.

```
def __init__(self, name, trigger_device=None, trigger_connection=
             None, usbport='COM1'):
    # Run the labscript suite base class method
    # (we assume inheritance from PseudoclockDevice here)
    PseudoclockDevice.__init__(self, name, trigger_device,
             trigger_connection)

    # stores the connection information in the BLACS_connection
    # column of the connection table
    self.BLACS_connection = usbport

    # any additional device specific code
    ...
```

In addition to the above, some pseudoclock devices also internally instantiate `Pseudoclock` and `ClockLine` objects if there is always a fixed number of these present in the device. For example, the PineBlaster instantiates one `Pseudoclock` and one `ClockLine` object as it only has one output. The PulseBlaster instantiates one `Pseudoclock`, but leaves the creation of `ClockLine` objects to the connection table definition in the experiment logic file because PulseBlaster digital outputs can be assigned as either clock lines or digital outputs.

The `generate_code` method is called internally by labscript[3]. The base class method (provided by labscript) is responsible for invoking the parts of labscript that collate the requested I/O from the experiment logic, and generate internal (general purpose) data structures that contain a representation of the required pseudoclock signal(s) and a list of state changes for each output (that matches with the generated clocking signal(s)). As such, when overriding the `generate_code` method from inside a device specific class, it is important to first call the base class method:

```
def generate_code(self, hdf5_file):
    # Run the labscript suite base class method
    # (we assume inheritance from PseudoclockDevice here)
    # This generates instruction data for every child and
    # subsequent generation of children that is attached
    # to this device
    PseudoclockDevice.generate_code(self, hdf5_file)

    # Create the HDF5 group for storage of
    # device attributes and instructions
    group = self.init_device_group(hdf5_file)

    # device specific code here
    ...
```

Following the call to the base class method, and the creation of the device group in the HDF5 file, the `generate_code` method should generate hardware instructions for the pseudoclock device, in an appropriate format for the device, and save them into the previously created device group. There is no specific format enforced; the format should be chosen to best

3. This occurs as part of the function call `labscript.stop(t)`, which a user calls at the end of their experiment logic file.

suit the programming of the device. A representation of the desired clocking signal (the pseudo-pseudoclock instructions) is available from the `Pseudoclock` object(s), which will be a child of this device. This is accessible either through `self.get_all_children()` or an internal variable if the `Pseudoclock` was instantiated internally. If, for example, the `Pseudoclock` was created internally and saved in `self.pseudoclock`, then the pseudoclock clocking instructions are stored in `self.pseudoclock.clock`. The format of this data is a list of dictionaries, of the form:

```
[
    # start: The time (in seconds since the initial device trigger) of
    #        the instruction
    # reps: The number of times to tick the clock for this instruction
    # step: The period of each clock tick
    # enabled_clocks: A list of ClockLine objects that should tick for
    #                 this instruction
    {'start':0.0, 'reps':1, 'step':100e-6, 'enabled_clocks': [<
                ClockLine object>, <ClockLine object>, ...]},
    ...
]
```

This then forms a basic implementation of a pseudoclock device. We believe our API provides one of the easiest to master when it comes to adding new pseudoclock devices as the majority of the implementation is taken care of by the base class. This only leaves the developer with the task of writing code to translate from the pseudo-pseudoclock instructions to the format required for programming the device.

7.1.1.2 Implementing pseudoclocked devices

Adding labscript support for pseudoclocked devices (devices that update their output state from an externally provided clock tick, such as the Novatech DDS9m or an NI card) is very similar to that of the pseudoclock described in the previous section. The labscript API provides a class `IntermediateDevice` that should be subclassed and the `__init__` and `generate_code` methods overridden. Class attributes indicating the type of outputs and maximum clock rate should be set:

```
@labscript_device
class NovaTechDDS9M(IntermediateDevice):
    # A human readable name for device model used in error messages
    description = 'NT-DDS9M'
    # The labscript Output classes this device supports
    allowed_children = [DDS, StaticDDS]
    # The maximum update rate of this device (in Hz)
    clock_limit = 9990
```

The `__init__` method is much the same, except the method signature is changed:

```
    def __init__(self, name, parent_device, com_port = "", baud_rate=
                115200):
        IntermediateDevice.__init__(self, name, parent_device)
        self.BLACS_connection = '%s,%s'%(com_port, str(baud_rate))
```

We again save information regarding the device connection to the control PC in the `BLACS_connection` attribute. The `generate_code` method has the same method signature as described in the pseudoclock implementation, and should write a table of instructions to the HDF5 file in a format appropriate for programming this device. Instead of accessing the pseudoclock instructions though, subclasses of `IntermediateDevice` should iterate over a list containing the attached output object(s) stored in `self.child_devices`. The list of output states for each output is then available in the `raw_output` attribute of each output object[4].

7.1.1.3 Storing configuration attributes

The labscript suite provides a common API for saving and retrieving device configuration metadata (properties) from the shot file. There are two locations for this data; the first is a HDF attribute to the device group introduced above, and the second is a column of the connection table (where the data is stored as a JSON formatted string). Device metadata that is critical to defining the device or channel configuration and cannot be changed shot to shot via software configuration, should be saved in the connection table properties column. For example, the frequency of a reference clock connected to a device would usually be stored in the connection table properties. Device metadata that can be configured through software should be saved as an attribute of the device group[5]. For example, most analog acquisition devices have a software configurable acquisition rate that would be stored in the device properties.

These parameters are usually specified when device objects are instantiated using the labscript API. Labscript provides a Python decorator that allows you to easily assign arguments from the `__init__` constructor method to either the device properties or connection properties locations[6]. For example, this demonstrates a potential modification to the previously introduced `PineBlaster` class:

```
@set_passed_properties(property_names = {
    "connection_table_properties": ["device_config"],
    "device_properties": ["channel_config"]
})
def __init__(self, name, trigger_device=None, trigger_connection=
                None, usbport='COM1', device_config=None,
                channel_config=None):
    PseudoclockDevice.__init__(self, name, trigger_device,
                    trigger_connection)
    self.BLACS_connection = usbport
```

The optional instantiation arguments `device_config` and `channel_config` are then stored in the connection table and as a device group attribute respectively, without any additional

4. `DDS` and `StaticDDS` objects are actually an amalgamation of multiple analog outputs, and the lists of instructions should instead be accessed through `my_dds_output.frequency.raw_output`, `my_dds_output.amplitude.raw_output`, and `my_dds_output.phase.raw_output`.
5. It should be noted that, currently, you cannot save device group attributes for channels attached to a device, due to the fact that only devices (and not their channels) receive a device group in the HDF file. It is expected that this will be addressed in a future update [160].
6. The original idea for this was proposed and implemented by myself, albeit limited to the connection table [161]. The decorator interface was developed by Ian Spielman and integrated after discussions with our development team at Monash [162].

work. There are also dedicated methods for saving data in these locations if an internal calculation is required before the value can be saved (for example, if the value to be saved is derived from the combination of two constructor arguments). These properties can then be retrieved from shot files within runviewer, BLACS and lyse using the provided `get` function in the `labscript_utils.properties` Python module.

7.1.2 BLACS graphical user interface

Creating GUIs is usually considered difficult. While a standalone Python script executes (reasonably) linearly, graphical applications host an 'event loop' that is typically outside of the control of the developer. The event loop calls blocks of code the developer has written in response to events generated by a user in the graphical interface. This paradigm shift of program execution is often difficult for new developers to follow. Furthermore, long computations can no longer run in the primary thread, as this would block the execution of the event loop (thus locking up the graphical interface).

We help developers easily create graphical interfaces in BLACS by providing almost all of the required functionality in a Python class (`blacs.device_base_classes.DeviceTab`) which can be subclassed for each device. For a `IntermediateDevice` like an NI card, the subclass of `DeviceTab` need only override a single method (`initialise_GUI`). The overridden method is then responsible for defining the hardware properties of each channel, instantiating 'output objects' (which cache the current output states, handle unit conversions, and updating the values of GUI widgets), creating and placing the GUI widgets and creating the worker process for communicating with the hardware. The developer can also take the opportunity to state if the device supports remotely checking the values programmed into the device (to ensure they match the current front panel values) and/or whether the hardware timed instruction table in the device can be updated without reprogramming the entire table. Most of this can be achieved using provided methods of `DeviceTab`, as shown below:

```
@BLACS_tab
class NI_PCI_6733Tab(DeviceTab):
    def initialise_GUI(self):
        # Capabilities
        self.num_AO = 8
        self.num_DO = 8
        ao_prop, do_prop = {}, {}

        # Define channel hardware parameters.
        # ao_prop and do_prop dictionaries should be keyed by the
        # channel name as used in the labscript 'connection' string
        for i in range(self.num_AO):
            ao_prop['ao%d'%i] = {'base_unit':'V',
                                 'min':-10.0,
                                 'max':10.0,
                                 'step':0.1,
                                 'decimals':3
                                }
```

```
for i in range(self.num_DO):
    do_prop['port0/line%d'%i] = {}

# Create the BLACS output objects
self.create_analog_outputs(ao_prop)
self.create_digital_outputs(do_prop)
# Create widgets connected to the BLACS output objects
dds_widgets,ao_widgets,do_widgets = self.auto_create_widgets()

# and place the widgets in the GUI
self.auto_place_widgets(('Analog Outputs', ao_widgets),
                        ('Digital Outputs', do_widgets))

# Store the Measurement and Automation Explorer (MAX) name
self.MAX_name = str(self.settings['connection_table'].
         find_by_name(self.device_name).BLACS_connection)

# Create and set the primary worker.
# Here we set the worker name, worker process class to use,
# and a dictionary of parameters to be passed to the worker
# process.
self.create_worker("main_worker", NiPCI6733Worker, {
    'MAX_name':self.MAX_name,
    'num_AO':self.num_AO,
    'num_DO': self.num_DO
})
self.primary_worker = "main_worker"

# Set the capabilities of this device
self.supports_remote_value_check(False)
self.supports_smart_programming(False)
```

As can be seen from the above example, only around 50 lines of code are needed to generate complex GUIs that contain all of the previously described features in §6.1.1 (and much of this code is either comments or boiler-plate code). This includes code that automatically names controls based on a name from the BLACS connection table, the state machine architecture, and a robust worker process model for communicating with a hardware device.

For pseudoclock devices, i.e. the devices that control the timing of every shot, two additional methods should be defined. The first is the `start_run` method and the second is a method for checking whether the shot has finished. Both must be written to use the device tab state machine and the latter should be configured to run periodically during experiment execution. Again, the BLACS device tab base class provides significant help here, and the methods generally look as follows:

```
    # A method, called by the queue manager, used to command the
    # master pseudoclock to begin the shot
    @define_state(MODE_BUFFERED,True)
    def start_run(self, notify_queue):
        success = yield(self.queue_work(self.primary_worker,
                                        'start_run'))
        if success:
            self.statemachine_timeout_add(100, self.status_monitor,
                notify_queue)
        else:
            raise RuntimeError('Failed to start run')

    # A method, called by the statemachine repeatedly (as
    # configured by start_run above) that monitors the
    # master pseudoclock and notifies the queue manager when
    # the shot is complete
    @define_state(MODE_BUFFERED,True)
    def status_monitor(self,notify_queue):
        finished = yield(self.queue_work(self.primary_worker,
                                         'status_monitor'))
        if finished:
            notify_queue.put('done')
            self.statemachine_timeout_remove(self.status_monitor)
```

As can be seen, these methods are thin wrappers around calls to the worker process, and are used by the BLACS queue manager to interact with the primary (master) pseudoclock.

Additional customisation to the device class is of course possible, using other inbuilt `DeviceTab` methods or by overriding the default behaviour. For example, a developer could add additional GUI widgets to the BLACS device tab such as status icons or plots. However, as shown, basic GUI functionality can be produced using very little code (and zero knowledge of GUI programming) which we believe is an important requirement of a flexible control system capable of integrating with the ever evolving hardware requirements of experiments.

7.1.3 BLACS worker processes

Creating a BLACS worker process amounts to filling-in-the-blanks with code that uses the API provided by the device manufacturer to configure and program the device. Methods within the worker process class are called by the BLACS device tab state machine in response to GUI methods using the `yield` keyword. While we have shown (in §7.1.2) how the BLACS GUI class may have developer defined methods that follow this pattern, for example `start_run()`, there are several methods within the base class (`DeviceTab`) that also do this. These methods (such as `DeviceTab.program_device()`) call corresponding methods in the worker process (such as `program_manual`) which must also be written in addition to any device-specific worker process methods introduced by a developer. Here we show the skeleton of a worker process class, subclassing `blacs.tab_base_classes.Worker`, with comments that indicate what each method should be written (by a developer) to do:

```
@BLACS_worker
class NiPCI6733Worker(Worker):
    def init(self):
        # Once off device initialisation code called when the
        # worker process is first started.
        # Usually this is used to create the connection to the
        # device and/or instantiate the API from the device
        # manufacturer

    def shutdown(self):
        # Once off device shutdown code called when the
        # BLACS exits

    def program_manual(self, front_panel_values):
        # Update the output state of each channel using the values
        # in front_panel_values (which takes the form of a
        # dictionary keyed by the channel names specified in the
        # BLACS GUI configuration

        # return a dictionary of coerced/quantised values for each
        # channel, keyed by the channel name (or an empty dictionary)
        return {}

    def transition_to_buffered(self, device_name, h5file,
                               initial_values, fresh):
        # Access the HDF5 file specified and program the table of
        # hardware instructions for this device.
        # Place the device in a state ready to receive a hardware
        # trigger (or software trigger for the master pseudoclock)
        #
        # The current front panel state is also passed in as
        # initial_values so that the device can ensure output
        # continuity up to the trigger.
        #
        # The fresh keyword indicates whether the entire table of
        # instructions should be reprogrammed (if the device supports
        # smart programming)

        # Return a dictionary, keyed by the channel names, of the
        # final output state of the shot file. This ensures BLACS can
        # maintain output continuity when we return to manual mode
        # after the shot completes.
        return final_values

    def transition_to_manual(self):
        # Called when the shot has finished, the device should
        # be placed back into manual mode
        # return True on success
        return True
```

```
    def abort_transition_to_buffered(self):
        # Called only if transition_to_buffered succeeded and the
        # shot if aborted prior to the initial trigger

        # return True on success
        return True

    def abort_buffered(self):
        # Called if the shot is to be abort in the middle of
        # the execution of the shot (after the initial trigger)

        # return True on success
        return True
```

Additionally, if the current output state of the device can be read at any time (and `self.supports_remote_value_check(True)` was called in the BLACS GUI class), then the `check_remote_values` method should also be defined:

```
    def check_remote_values(self):
        # Return the output state of the device in a dictionary,
        # keyed by the channel name
        return remote_values
```

Pseudoclock devices must also define worker process methods for starting the shot via a software trigger to the device and checking whether the shot has completed. In this case, the method names and return values should match those used in the associated BLACS GUI methods (which can be anything the developer wishes).

Again, we can see that implementing the BLACS worker process for a device comes down to data wrangling between the instruction storage in the HDF5 file, the BLACS dictionaries of values (keyed by channel name) and whatever format is needed by the device-specific programming API (which is usually provided by the device manufacturer). This is effectively as minimal as possible, while still maintaining the many benefits outlined previously, such as multiprocessing sandboxes for device communication.

7.1.4 Runviewer

As discussed in §5.3.1, runviewer walks the connection table and calls device specific code in order to generate the output traces to display. To add support for a new device in runviewer, a class must be registered using the `@runviewer_parser` Python decorator. This class takes the following format:

```
@runviewer_parser
class RunviewerClass(object):
    def __init__(self, path, device):
        self.path = path
        self.name = device.name
        self.device = device
```

```
def get_traces(self, add_trace, clock=None):
    # device specific code

    # return a dictionary of traces for all clocklines
    # and triggers that this device manages
    return clocklines_and_triggers
```

The registered Python class must define a method called `get_traces` which is called internally by runviewer. Runviewer passes in a reference to a runviewer function, `add_trace`, and a tuple called `clock`. The `clock` variable is provided as a 2-tuple of NumPy arrays (themselves 1D arrays of equal length) which contain the times and state changes of the clocking signal or external trigger (depending on whether the device is implemented in labscript as an `IntermediateDevice` or `PseudoclockDevice`). The device specific code should then access the HDF5 file (specified in `self.path`), load the stored instruction set from the devices group, and reverse engineer output traces from the hardware instructions and the `clock` variable for every output that the device manages. Each output trace should be produced in the same format as the `clock` variable; that is, a 2-tuple of NumPy arrays where the first stores the list of times at which the output changes state and the second stores the values of the state change.

Any trace that should be available to a user for display is then sent to runviewer by calling the provided `add_trace` method. As discussed in §5.3.1, this callback structure allows device-specific code to control how many traces are made available to a user. For example, a device with rf outputs might return 3 traces per output: one for frequency, amplitude, and phase. The method signature for `add_trace` is `add_trace(name, trace, parent_device_name, connection)`, where `name` is a string containing the display name for the output, `trace` is the 2-tuple, and `parent_device_name` and `connection` are strings specifying the relationship between the trace and the device. It is likely that `name`, `parent_device_name`, and `connection` should be determined from the connection table. Runviewer provides access to the section of the connection table relevant to the device via the `self.device` attribute.

Finally, runviewer requires a dictionary of output traces that correspond to clock lines or digital outputs that were instantiated in labscript as `Trigger` objects. The dictionary should be keyed by the name of the clock line or trigger, as defined in the connection table, and the value should be a 2-tuple of the output trace. These will ultimately be passed in to other devices as the `clock` variable of the `get_traces` method.

As with labscript and BLACS, adding support for a new device in runviewer comes down to implementing a small amount of device specific code. In this case, code to translate between hardware instructions and expected output traces is all that is required. The graphical presentation of shots and traces is all handled internally by runviewer.

7.2 Summary

In this chapter we have demonstrated how easy it is for developers to add support for new hardware devices. Developers only need to create a small number of subclasses for labscript, BLACS, and runviewer. By abstracting away common code into base classes, we

have engineered our software so that the code to be written for each subclass is focussed around device specific implementation details, like the format of hardware instructions and the interaction with a manufacturer API. This makes it easy to support a wide variety of devices while still maintaining a common API and interface for those who wish to control these devices. The labscript suite thus has good support for heterogenous hardware, and is well placed to support custom hardware devices as they are needed.

Chapter 8

Case studies

The success of a control system must be measured by its application to experiments. As outlined in chapter 1, we believe that a key indicator of this is the flexibility of the control system; that is, the ability of the control system to adapt to a wide range of different apparatuses and the ever-evolving experiments that are run on them.

In this chapter we'll detail some specific uses of the labscript suite where I contributed directly to the research. We'll first demonstrate how we control and parameterise experiments using our BEC apparatus and provide some examples of how we use the labscript suite on a day-to-day basis. Following this, we'll show how we have used the labscript suite to perform high impact research, which required control of non-standard hardware devices and complex multi-shot analysis scripts. Finally, in order to demonstrate the flexibility of the labscript suite, we'll show how we used it to acquire and analyse data that did not require precision timing and how the labscript suite was still extremely beneficial in such a use case.

8.1 BEC apparatus control & optimisation

As mentioned previously, we have used the labscript suite to control several ultracold atom experiments at Monash University. These are the dual-species (K-Rb) BEC lab (which I detailed the apparatus construction of in chapter 3), the spinor BEC lab, and the second generation dual-species (K-Rb-2) BEC lab currently under construction. The labscript suite has been instrumental in getting these labs operational and producing ultracold atom clouds. In this section I will first detail the experiment control hardware we use (that interfaces with the labscript suite), followed by some examples of how we have used the power of the labscript suite for rudimentary optimisation of the experimental sequences.

8.1.1 Experiment control hardware

One of the goals we achieved with the labscript suite was support for a wide range of heterogeneous hardware. This is reflected in the mixture of commercial and in-house hardware we use to control the three ultracold atoms experiments at Monash University (summarised in table 8.1). As can be seen, we use devices with a range of communication interfaces (and protocols) and with a range of different I/O capabilities.

Manufacturer	Model	Type	Interface	Hardware timed?	I/O
National Instruments	PCIe-6363	Multifunction I/O	PCIe	Yes	32 digital I/O 4 analog outputs 32 analog inputs
National Instruments	PCI-6733	Analog output	PCI	Yes	8 analog outputs
Novatech	DDS9m	DDS	RS-232	Yes	2 rf outputs (additional 2 software-timed rf outputs)
SpinCore	PulseBlaster DDS-II-300-AWG	DDS & pulse train	USB	Yes	2 rf outputs 4-12 digital outputs (firmware dependent)
PhaseMatrix	QuickSyn FSW-0010	Frequency synth.	RS-232 over USB	No	1 microwave output
Monash (in-house)	RFBlaster	DDS	Ethernet	Yes	2 rf outputs
Zaber Technologies	Various linear stages	Translation stage	RS-232	No	N daisy chained linear stages
Alazar Technologies	ATS9462	DAQ	PCIe	Yes	2 analog inputs (180 MS/s)
Texas Instruments	DLP LightCrafter	DMD	Ethernet via USB	Yes	608x684 pixel DMD
Photonfocus	MV1-D1312(I)	Camera	Ethernet (GigE)	Yes	Camera
Allied Vision Technologies	Various	Camera	Ethernet (GigE)	Yes	Camera
Andor Technology	Various	Camera	various	Yes	Camera

Table 8.1: A table of experiment control hardware, and the capabilities of each device, used with our BEC apparatuses at Monash University.

We exclusively use SpinCore PulseBlasters to provide the pseudoclocks for our experiments, due to their support for multiple clock lines. They primarily clock the NI cards and Novatech DDS devices. Each lab has at least two pseudoclocks and one of each NI card listed in table 8.1. These devices provide all of our analog and digital outputs. The analog outputs are used primarily to control coil drivers (see §3.4.3) while the digital outputs are used for controlling shutters, camera triggers, and gating rf outputs. Rf is produced exclusively by our DDS devices: the Novatech DDS9m, PulseBlaster, and RFBlaster[1]. The K-Rb experiment uses four Novatech devices in total; three are used to drive the AOMs on the laser table detailed in §3.1, while the other is used to drive the dipole trap AOMs on the vacuum table. We use the PulseBlaster DDS outputs to control some AOMs as well. We then use RFBlasters for driving atomic state transitions either directly with the rf produced, or via our microwave quadrature modulator discussed previously in §3.4.4, due to their superior agility in arbitrary waveform generation. The spinor BEC lab uses a similar combination of devices, however they use fewer Novatech DDS devices as their experiment is only single species, but more PulseBlaster and RFBlaster devices due to their more complex atomic state manipulation requirements. Each lab also typically has at least one sort of device that is only used within that lab, for example the Alazar data acquisition card or the DMD are currently only used in the spinor and K-Rb labs respectively.

Both K-Rb and spinor BEC labs also use devices that cannot be controlled under hardware timing[2]: the PhaseMatrix microwave generator, the Zaber linear translation stages,

1. A custom device produced in-house at Monash University.
2. I.e. they have no pseudoclock or trigger inputs and can only store a single value per channel (the current output value) at any given time.

and the two software-timed channels of the Novatech DDS9m. For these devices, labscript supports setting the output state prior to starting the master pseudoclock, providing shot-to-shot control of these devices (which is typically sufficient for our use cases).

A range of cameras are also used across the three labs. Each lab has at least two cameras in use, from different manufacturers, which are managed by separate instances of BIAS. Most of our cameras communicate over gigabit Ethernet, however some also use Camera Link or proprietary interfaces that require dedicated capture cards with either a PCI or PCIe interface.

For PCI and PCIe devices (such as NI cards or proprietary camera capture cards), we use Adnaco fibre range extenders [163] to minimise ground loops and allow our control PCs to be physically separated from the experiment. These range extenders work transparently at the operating system level (and so do not interact with the control system), and are a useful addition to our control hardware.

As we can see, while many of the devices are used commonly between the labs, the specific set that is in use varies considerably. However, the labscript suite has been designed to easily handle this, by requiring each lab to specify the set of hardware in use in the connection table definition. We include a diagram of the K-Rb connection table in appendix D.3, which demonstrates the complexity and variety of the hardware we need to manage for our experiments.

8.1.2 K-Rb apparatus experiment logic & globals

We designed our K-Rb apparatus so that students could work on separate projects at the same time. As all past and expected projects work with ultracold atoms, much of the experiment logic, and many of the globals, for producing the ultracold atoms are shared between users. As an example, we include the globals and experiment logic for the K-Rb experiment in appendix D.1 and D.2 respectively. These give an 'unsanitised' view of how one might use the labscript suite for day-to-day operation of an ultracold atom apparatus. The development of the experiment logic, and set of optimised globals, comprised a significant portion of both mine, and Shaun Johnstone's, PhD projects. This was not helped by the fact that much of the optimisation had to be re-done after moving our experiment to a new building mid-way through our projects[3].

We divide our experiment logic into stages, which are enabled/disabled by Boolean global variables in runmanager (see table D.49), by placing experiment logic inside `if` statements (see for example line 708 in appendix D.2). We then typically have one group of globals for each stage, for example 'Rb Optical Pumping' in table D.42. This is just a convention though, and often we will split globals across multiple groups where appropriate, for example the MOT load stage is split across 4 globals groups, one for each species-location combination, which are shown in tables D.17, D.26, D.27, and D.41.

When doing research using ultracold atoms, we typically 'fork' (make a copy) of the experiment logic and globals, and add our required 'science' experiment logic between the load of the cross-dipole trap and imaging. While we could directly share the experiment logic by moving common code to a shared Python module, the rate of change of our experiment logic appears to be slow enough that this is premature optimisation.

3. See [164] for a video of the moving process for our vacuum table!

A similar system is also employed in the spinor BEC lab, where Alex Wood [123], Martijn Jasperse [106], and Lisa Starkey [165] successfully shared an apparatus concurrently (separate days of the week).

8.1.3 Apparatus optimisation

We'll now detail the use of a general purpose lyse multi-shot analysis script that allows us to quickly gather information regarding optimal values for one or two experimental parameters at a time. We have found this to be invaluable both for developing the experiment logic and for rapid diagnostics of our apparatus when things go wrong[4]. The script is designed to create a set of plots of measured quantities as a function of a set of parameters. The list of measured quantities are specified in a runmanager global (for easy adjustment[5]) as a tuple[6], which is accessible via the Pandas DataFrame from lyse. One plot is created per item in the list of measured quantities, with the measured quantity used as the dependent axis. The axes of the parameter space are automatically detected from globals with an expansion type (which is accessible through the `lyse.Run` API) indicating it was part of the parameter space scan. These globals are then used for the independent axis (or axes) of the plots. We thus call this script 'y_vs_auto', and the full code for the multi-shot analysis script is located in appendix E. The script can handle parameter space scans with dimensions between zero and two[7].

A 0D parameter space is simply a single shot (no parameter space scan). In such cases, the shot repeat number becomes the x-axis (independent axis) of a X-Y plot. The graphs in this mode then allow you to look at the stability of the measured quantities as a function of time, simply by turning on the 'repeat' mode of BLACS and generating a single shot from runmanager.

A 1D parameter space is perhaps the most frequently used, which simply plots the measured quantity as the function of a single global variable. This is very useful for quickly determining if a runmanager global is set to its optimum value. For instance, we regularly scan globals that correspond to AOM frequencies, in order to compensate for drifts in the resonant laser frequency, which often occur due to mechanical movement of coils that are poorly attached to our apparatus.

A 2D parameter space scan produces a 2D plot using a custom version of the `imshow` matplotlib function that was created by Chris Billington to handle irregular grids[8]. The two globals that form the parameters space axis are used as the x and y axes of the plot, and the measured quantity is used as the z-axis of the plot (the colour at each grid point).

The y_vs_auto script relies on several specially named runmanager globals that control its functionality. They are always pulled from the last row of the DataFrame (the last shot to be run) and are:

4. A regular occurrence for experimental ultracold atom physicists!
5. This is a 'hack' because we have not yet implemented a separate set of 'analysis globals' in lyse that can be manipulated through a graphical interface. This hack also allows the behaviour of this lyse script to be controlled by the operator of runmanager, even if lyse is running on a separate PC.
6. The use of a tuple allows a list of values to be specified without it being detected as an axis of a parameter space.
7. We welcome contributions beyond two dimensions from beings capable of simultaneous observation of multiple depth planes!
8. This was necessary so that we could correctly generate the plots prior to the completion of all shots in a parameter space, and for instances where the globals were not regularly spaced.

- `no_sequences_to_analyse`: This determines how many separate sequences of shots should be used. This is usually one, but it can be set to two or higher if, after the first sequence has run, you decide you wish to acquire additional data for the same parameter space and have it displayed on the same set of plots.

- `repeats`: We use this global to 'trick' runmanager into creating a fixed number of repeats[9] for each shot by specifying it as a list (for example `range(3)` would produce 3 repeats of each shot). As this global would form an axis of the parameter space, we specifically instruct our script to ignore it so that it does not end up on the axis of the plots.

- `image_order`: We use this global in our K-Rb apparatus to indicate that both species are being imaged. This global is set to be derived from other globals, as previously detailed in figure 5.17, so we do not need to explicitly set it (see also table D.13 in appendix D). Ultimately, the global is controlled by the user through `s11__absorption_image_k` and `s11__absorption_image_rb` Booleans in the 'Stages' globals group (see table D.49). As we cannot image both species in a single shot, we generate two otherwise identical shots for each point in the parameter space, but image a different species in each. This global is thus a list of values, and is an axis of the parameter space as far as runmanager is concerned. It is `True` for shots where we image rubidium and `False` for shots where we image potassium. This global, however, is also excluded from the parameter space to be plotted, and we instead generate separate plots for each image by grouping the DataFrame by this global, iterating over the two groups, and running the bulk of the script logic (which produces the plots) for each group (see lines 273-279 of appendix E).

- `absorption_image_k` and `absorption_image_rb`: If we are not imaging both species, then these globals indicate which single species we are imaging. These are also derived from the same set of globals as `image_order`, so that you only have to choose the species to image in one place.

- `plot_measurement_k`, `plot_measurement_rb`, and `plot_measurement`: These contains a tuple of the measurements to be used as the dependent variables of the plots. The first two are used when performing absorption images of potassium, rubidium, or both. The script produces one plot for every item in the tuple(s). If we are not absorption imaging (for example, instead using fluorescence imaging), then we use the list of measurements in `plot_measurement` instead. The entries in the list can be any DataFrame column name, or an alias of a DataFrame column name stored as an attribute of our analysislib Python module[10]. By using separate lists of measurements for the two species, we are able to reference species-specific aliases for measured quantities, and plot measurements of interest for a specific species only.

9. This contrasts to the repeat functionality in BLACS, which will repeat a set of shots continuously until stopped.
10. Aliases are helpful for accessing the results produced by single shot analysis scripts. For example, instead of the plot measurement being specified as `('atom_cloud_fit', 'number_of_atoms')`, you might specify the alias as `n_atoms_rb = ('atom_cloud_fit', 'number_of_atoms_rb')` so that you can specify the plot measurement as `'n_atoms_rb'`.

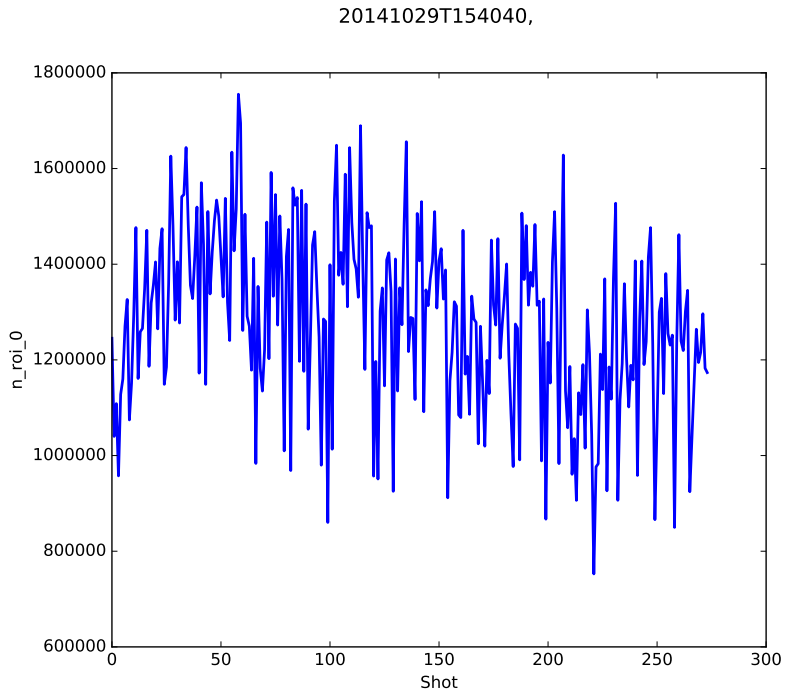


Figure 8.1: An example of a 0D parameter space scan, where a single shot is repeated many times and the measured quantities are plotted against shot number. Here we show a stability check of the atom number in our dipole trap after an attempt at transporting rubidium atoms to one of our science cells (see §3.2.1 and figure 3.4). The mean atom number transported was 1.29×10^6, however the standard deviation was 0.19×10^6, or 14%. This was unacceptably large, to the point where optimising the subsequent parts of the experiment logic proved impractical due to the inability to discern optimal values for parameters. Ultimately we resorted to performing our science experiments in the central chamber, where the shot-to-shot variation in atom number was much smaller.

For 1D parameter spaces (that are thus plotted on an X-Y scatter plot), each sequence is plotted using a different colour. This allows a user to easily see whether variation in the y-axis is due to random fluctuations or is the result of a longer term drift that resulted in a similar shift in all data points from a subsequent sequence. We also plot the average for any values that are located at an identical x-value, and include error bars equal to the standard deviation of the points. Unfortunately we have not yet determined an appropriate way to display similar information for 2D parameter spaces, since colour is already used for the z-axis value.

We show real-world examples of the use of this analysis script in figures 8.1-8.3. In figure 8.1 we show an example of a long term stability measurement of our atom number after transport to a science chamber. In order to generate this graph, we simply created a single shot from runmanager (with whatever fixed set of parameters we were interested in observing) and placed it on repeat in BLACS. A single-shot analysis script performed atom cloud fitting, and saved the results in the shot file and lyse DataFrame. The 'y_vs_auto' script then produced the included figure, updating in real-time as new data arrived in lyse.

In figure 8.2 we show the results of a 1D parameter space scan of the AOM driving

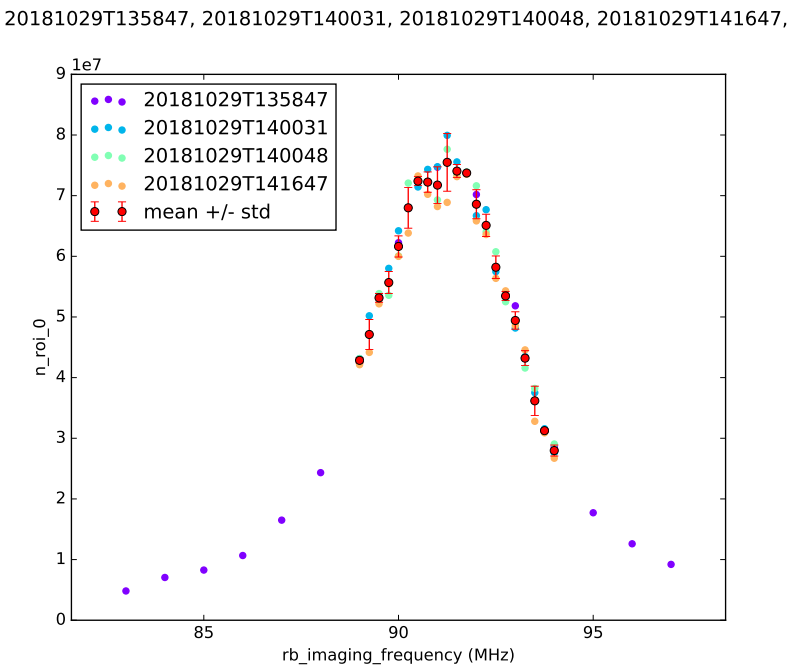
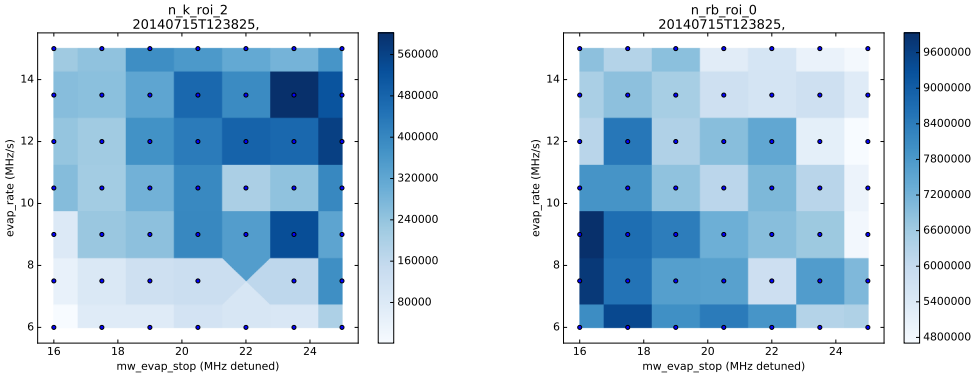


Figure 8.2: An example of a 1D parameter space scan used to determine the optimal imaging frequency. Four separate sequences were generated from runmanager, each of which is assigned a unique colour automatically by the 'y_vs_auto' script. For points of the parameter space where multiple shots were run, the mean and standard deviation is shown in red. Note: The blue and green sequences have very similar timestamps due to their sequential creation from runmanager. The timestamps are only indicative of when the sequence was generated, not when it was executed by BLACS (which is not shown, as each shot within a sequence has a different execution time).

frequency for the rubidium imaging beamline of our apparatus (see §3.1.1.3). This graph was generated by running four sequential sequences of shots, where each sequence contained multiple shots with a varied `rb_imaging_frequency` global in runmanager specified using the `linspace` function. The first sequence was a coarse scan from 83 MHz to 97 MHz over 15 shots (a 1 MHz step between shots). Once the approximate location of the peak was identified we ran three additional sequences, each containing 21 shots, with the same runmanager global set to `linspace(89, 94, 21)` (a step of 0.25 MHz between shots, aligning with the parameter space points from the first sequence). In this case, we imaged the atoms after evaporation in the magnetic trap (prior to transfer into a pure optical trap). As in the previous example, the atom number was determined from a fit to atom cloud absorption image in lyse, and the 'y_vs_auto' script then produced the included figure.

In figure 8.3 we show the results of a 2D parameter space scan of two key parameters of our forced magnetic trap evaporation stage. In order to generate this graph, we defined a parameter space in runmanager by setting the `evap_rate` global to `linspace(6, 15, 7)` and the `mw_evap_stop` global to `linspace(16, 25, 7)`. These globals control the rate at which the microwave field is swept and how far (in MHz) we sweep the field, respectively, forming a 49 point parameter space. We also set both `s11__absorption_image_k` and `s11__absorption_image_rb` to True, which ensures that the global `image_order` is set



(a) (b)

Figure 8.3: An example of a 2D parameter space scan automatically detected and plotted by our 'y_vs_auto' analysis script. Here we have scanned two parameters that control the forced evaporation in the hybrid trap, in order to see how this affects the atom number in the decompressed hybrid trap. These parameters are `evap_rate`, which controls the rate at which the microwave knife frequency is changed, and `mw_evap_stop`, which controls the stopping frequency of the microwave sweep (the start point is fixed between shots). We repeat every point in the parameter space twice in order to produce shots that image potassium and rubidium atoms (figures (a) and (b), respectively). The two parameters form the axes of the graph, and the colour surrounding each data point represents the fitted atom number after time-of-flight (TOF) absorption imaging. As we can see, potassium and rubidium prefer opposite corners of this parameter space, which is not ideal.

correctly to act as a third axis of the parameter space, repeating the previously mentioned 49 shots twice, in order to image rubidium atoms in one and potassium atoms in the other (for a total of 98 shots). These 98 shots were generated by runmanager (in a randomised order), executed by BLACS, and then analysed in lyse (first in order to determine the atom number as in the previous examples and then by the 'y_vs_auto' script).

The plots in figure 8.3 were the first indication we had that our apparatus was going to have trouble producing dual-species ultracold atom clouds. We eventually concluded that the significantly disparate atom numbers between the two species (shown here as somewhere between 1 and 2 orders of magnitude depending on the chosen point of parameter space) was the primary culprit of our problems. Even with efficient optical pumping of rubidium atoms, there would always be a population, left in a state that has unfavourable collisional interactions with potassium, of a similar magnitude to the number of potassium atoms. We believe this is corroborated by the above plots, which show that potassium favours shorter, faster evaporation parameters (which would result in less time for the unfavourable collisions to occur). A recent preprint [166] also suggests this as a potential loss mechanism, along with several other possibilities. However it is worth noting that they were ultimately successful despite having disparate atom numbers in the initial load (likely due to a successful molasses phase, which we have not achieved for potassium).

As can be seen, this script demonstrates the powerful combination of our analysis framework with our graphical parameterisation of experiment logic. We can easily produce complex diagnostic results simply by setting the value of one or more globals to a list of values to scan over, and waiting for the data to come in as shots are executed on the apparatus.

8.2 Vortex dynamics

One of the prominent demonstrations of our control system is the recent research lead by Shaun Johnstone in our laboratory, which I (among several others) also contributed to. In this research, we investigated how the dynamics of vortex clustering changed in a 2D quantum gas (a 2D ultracold atom cloud), as a function of evolution time and vortex initial conditions. This experiment was conducted using our K-Rb apparatus, but using only rubidium atoms.

8.2.1 Experiment design

In addition to the apparatus described in chapter 3, we also required a 2D trap and a pair of Bragg beams, which are already described in detail in [100, 127]. Vortices are created in our 2D atomic cloud by dragging a grid of obstacles through it. We are thus able to vary the initial conditions of the vortices by adjusting the size of the obstacle grid, which allows us a degree of control over whether the initial distribution of vortices is dominated more by clusters or dipole pairs, or somewhere in-between.

The 2D trap is formed using repulsive (blue detuned) dipole traps, consisting of a Hermite-Gaussian (HG) mode trap propagating perpendicular to a ring trap (see figure 8.4). The HG mode is produced using a holographic element, as described in [41]. The obstacle grid is produced by a DMD, which is also used to simultaneously create the repulsive ring trap. The DMD used is the Texas Instruments DLP LightCrafter previously mentioned in §8.1.1. This DMD provides a buffer of up to 96 frames that can be stepped through using a digital trigger, making it an ideal candidate for use with the labscript suite. We believe the labscript suite is the first (publicly available) control system to support the hardware timed output of images, which is a testament to the flexibility and extensibility of our design[11].

The support for the DMD was added by Shaun, building on top of previously unreleased code I had written for software-timed image output for SLMs[12]. The DMD is implemented in the labscript suite as an `IntermediateDevice`, with a single output. As the output is an image (an array of pixels), a new output type was created for this device. It should be noted that this was quite straight-forward, and was done by subclassing the existing `Output` labscript class and adding a method called `set_image()`. Users can then create a `LightCrafterDMD` device and `ImageSet` output in the connection table of their experiment logic, and call `ImageSet.set_image(t, raw=image_data)` to command output of image data at time `t`. This mimics the similar commands already part of the labscript suite, such as `AnalogOut.constant(t, value)`. Similar modification were made to BLACS, to provide support for output types that are images[13].

Because experiment logic is written in a text-based language, we are able to utilise 3rd party Python libraries to help programmatically create the image frames for the DMD (see

11. This is distinct from image acquisition (image input), which has been part of control systems for a long time.
12. The SLM support was originally added for producing the HG mode for the 2D trap, so that we could adjust the dimensions of the 2D trap from shot-to-shot. However, the poor diffraction efficiency ultimately led us to use a fixed holographic element as described above.
13. As manual mode is software timed, and thus only single images can be programmed in manual mode, this just directly utilised the previously mentioned SLM support for working with a single static images.

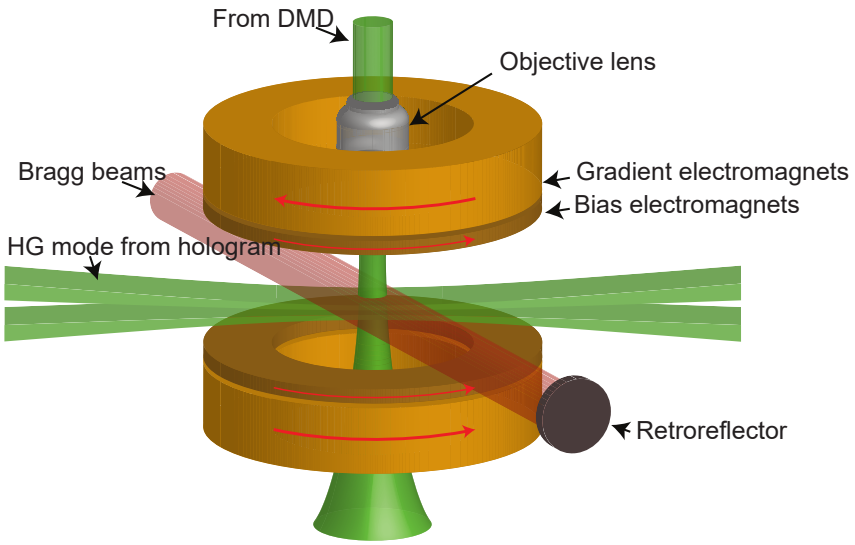


Figure 8.4: The trapping geometry for the vortex dynamics experiment, with the Bragg beams needed to implement the velocity selective imaging technique. The geometry is centred on the crossed-optical dipole trap (not shown) previously described in §3.2.5. A HG mode produced using a holographic element is focused onto the centre of our trap, and a radial trap is produced using the DMD and objective lens shown. The gradient electromagnets are the same ones used to produce a quadrupole field during the MOT load and forced magnetic evaporation stages (red arrows indicate the direction of current through the coils). The gradient coils are used to produce a magnetic field of sufficient strength to cancel gravity as the HG beam is not sufficient to hold the atoms by itself. The bias electromagnets are used to spatially remove the field zero from the atoms, ensuring there is no radial confinement from the quadrupole field and that the atoms remain in a consistent quantum state throughout the experiment. The gradient electromagnets are also kept on during TOF expansion (prior to performing the absorption image) to ensure that the cloud only expands radially while staying in the imaging plane (the objective shown is also used as the absorption imaging system objective). This figure was adapted from Shaun Johnstone's thesis [100] with permission.

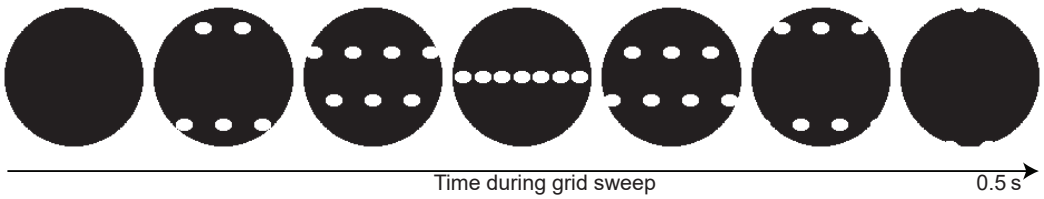


Figure 8.5: Here we show 7 example frames for the DMD (every 15th frame used in the shot) produced dynamically by Python code within our experiment logic. White pixels correspond to regions of high intensity ('on' pixels) of our blue-detuned (repulsive) trapping potential. This figure was adapted from Shaun Johnstone's thesis [100] with permission.

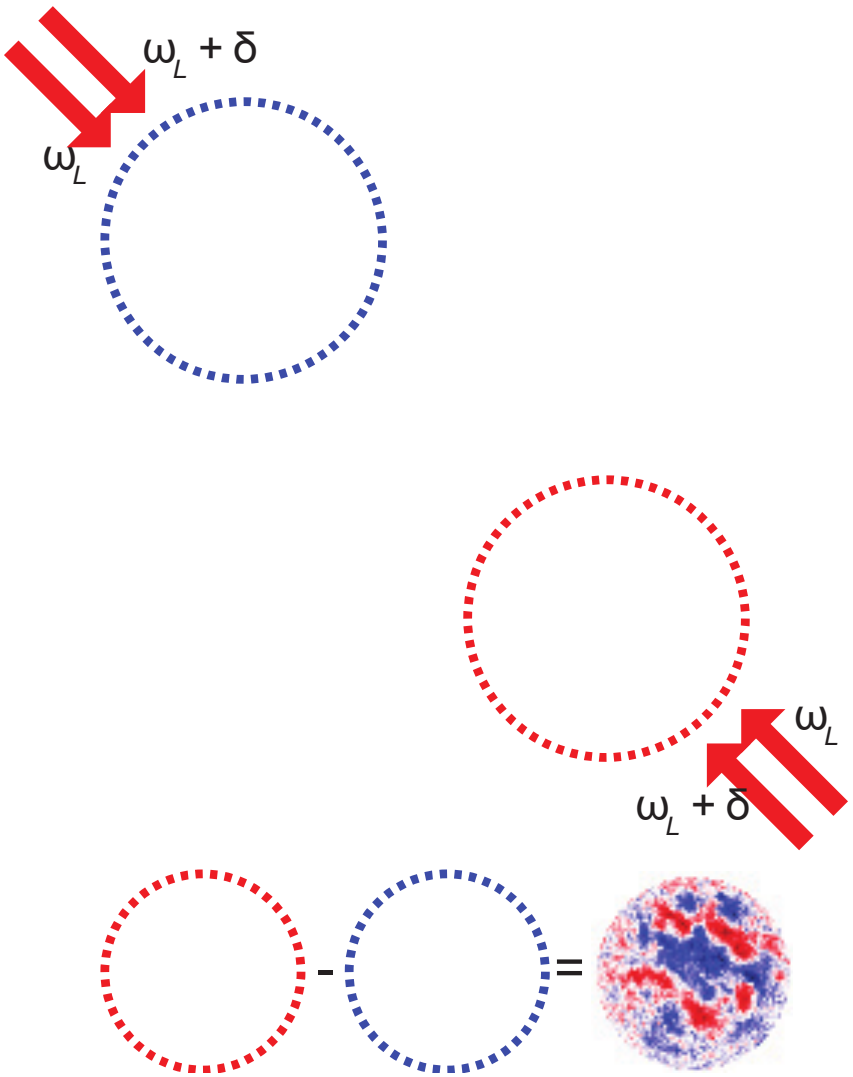


Figure 8.6: An example absorption image produced from our implementation of the velocity selective Bragg spectroscopy technique detailed in [67]. The red arrows correspond to two overlapped Bragg beams as shown previously in figure 8.5, detuned from each other by δ, and have a frequency approximately $\omega_L = 6.6\,\text{GHz}$ from the atomic resonance with the rubidium-87 cooling transition. The Bragg beams scatter the components circled in blue and red that, when subtracted, produce a map of the velocity field (where the red/blue colour in the resultant map indicates velocity in the direction of the component circled with the same colour in the absorption image), allowing the sign of each vortex to be determined. This figure was adapted from Shaun Johnstone's thesis [100] with permission.

figure 8.5). In this case, Shaun used the Pillow [167] Python library to draw ellipses onto a copy of the ring trap potential frame, using the `PIL.ImageDraw.Draw.ellipse()` method, which handles the complicated task of drawing shapes for you. The size of the ellipses (and thus the number that interact with the atom cloud) is parameterised via a runmanager global, so that it can be used as an axis of a parameter space scan. Again, this reinforces the fact that a control system written using an object-oriented language, with text based experiment logic, provides a rich environment for implementing custom requirements with

minimal effort, and in a way that allows consistent interfaces to be developed for use by non-experts.

The measurement of the atomic cloud is performed via an absorption image of the atoms. However, in order to recover enough information to deduce both the position and sign of the vortices, we utilise a velocity selective Bragg spectroscopy technique [67] that outcouples part of the atomic cloud in both the forward and reverse propagation directions of the Bragg beams (see figure 8.6).

In order to generate sufficient statistics of the vortex clustering dynamics, we ran 25 repeats of each shot in a parameter space spanning 5 obstacle grid sizes and 10 evolution times. This totalled 1250 shots, of which 70% ultimately proved useful (the remaining 30% were too distorted to accurately determine the vortex locations).

8.2.2 Single-shot analysis

The analysis for this experiment was likewise complex, consisting of five single-shot analysis scripts and four multi-shot analysis scripts. The single-shot analysis starts with a script for calculating the optical density (OD) of the atoms from the absorption image (a standard first step for many ultracold atom experiments) computed from three image frames: one with the imaging light and atoms present (the atoms frame), one with just the imaging light (the flat frame), and one with no imaging light (the dark frame). Due to the formation of one or more etalons between the flat surfaces of our vacuum windows and/or imaging system components, we observe a significant set of fringes across the atoms and flat frames. Unfortunately, the fringes are not removed through a standard calculation of OD due to the temporal delay between frames during which the locations of the fringes change. To counteract this, we utilise an eigenbasis method [168, 169] to compute a flat frame that best matches with the fringe locations in the current atoms frame. In our case, our eigenbasis is constructed from a rolling set of the last 200 flat frames. This computed flat frame is then used along with the atoms frame and an average of the last 200 dark frames, to calculate an image of the (saturation corrected) OD of the atom cloud(s) using the formula [100]:

$$\mathrm{OD}_{(x,y)} = -\ln\left(\frac{\mathrm{atoms}_{(x,y)} - \mathrm{avg\_dark}_{(x,y)}}{\mathrm{computed\_flat}_{(x,y)} - \mathrm{avg\_dark}_{(x,y)}}\right) + \frac{\mathrm{computed\_flat}_{(x,y)} - \mathrm{atoms}_{(x,y)}}{I_{\mathrm{sat}}}.$$

It should be noted that this analysis script is not unique to this experiment, indeed it is used to calculate the OD for all of our shots. The modular nature of lyse allows us to reuse a single analysis script over multiple experiments, ensuring our current best practice is consistently applied to our research.

Once the OD has been calculated (and saved in the HDF5 file), we proceed with identifying the location and sign of the vortices. We start by identifying the location of vortices by analysing the primary (central) cloud shown in figure 8.6, following the technique outlined in [170]. We utilise the OpenCV library [171] to perform the required image transformations, which provides computationally efficient implementation of the required algorithms. Utilising external libraries means the implementation of the technique is quite straightforward, requiring that we just call `cv2.GaussianBlur`, `cv2.Laplacian`, `cv2.threshold`, and `cv2.findContours` functions in sequence (with the appropriate parameters) in order to determine the location of the vortices. We also calculate image moments of each 'vortex' found

(using the `cv2.moments` function) in order to determine if the detected vortex is actually a closely bound pair of vortices. All predicted vortex locations are then saved in the HDF5 file.

The vortex identification routine is not 100% robust, so we next use a script designed to allow a user to review (and edit if necessary) the detected vortex locations. We start by overlaying the detected vortex locations over the OD frame, and present this to the user in a plot generated with matplotlib. Custom controls are added to the matplotlib window[14] to allow vortex locations to be added or deleted (by the user) via a GUI. This was achieved using the picker API of matplotlib, which provides an easy way to program custom interactions between users and objects in a matplotlib figure. We also added a button to mark the entire shot as 'bad' (for example if the atom number was too low to accurately resolve vortex locations at all, or the previously mentioned fringe removal was ineffective). The new updated vortex locations were saved separately to the HDF5 file. This information, along with the 'good'/'bad' flag, allowed us to later quickly generate statistics on the overall quality of our data, which we included in the supplementary material of our publication [127]:

> *Of the 1250 runs of the experiment (25 repeats of 10 hold times after 5 different grids), 5% had vortex locations manually added but none removed (an average of 1.3 vortices were added to each of these shots), 19% had automatically detected vortex locations removed but none added (an average of 1.3 vortices were removed from each of these shots), 2% required minor adjustments to be made to vortex positions (vortices moved less than a vortex diameter, usually correcting a vortex dipole pair detection), while 5% required vortices to be both added and removed. A further 30% of shots were rejected entirely, in cases where the vortices were too hard to distinguish (this can occur, for example, if the fringes in the imaging probe are particularly strong, or if atoms become trapped outside the main nodal line of the HG mode, obscuring the BEC), or the atom number was significantly less than the average.*

Once the positions of the vortices have been determined, we use the next analysis script to determine their sign. In order to do this accurately, we developed a new technique for analysing a differential Bragg image like that shown at the bottom of figure 8.6. This technique involves calculating a theoretical velocity flow field (based on a point vortex model) for each possible vortex sign configuration, discretising this to the pixel size of our image frame and projecting the velocity profile along the Bragg beam propagation direction in order to produce a set of theoretical differential Bragg images. Each theoretical image is then compared to the measured image in order to determine the configuration of vortex signs that best match our data.

This process is computationally intensive, requiring the generation and comparison of 2^N theoretical differential Bragg images for an image with N vortices. As previously mentioned, lyse does not yet have inbuilt multiprocessing capabilities. However, the general purpose nature of our analysis system allows individual scripts to be written to take advantage of

14. This was originally done by modifying the core of lyse, but later moved into the analysis script itself once the required features were exposed via the lyse API, maintaining the modularity of the analysis system.

multiple cores. To speed up the vortex sign determination, I wrote a new analysis script that launched a copy of the original script (using the Python subprocess module) for each CPU core, effectively mimicking running the script from a terminal (outside of the lyse GUI, see §6.3.4). These scripts were passed the path to the shot file as the first command-line argument, and a fixed vortex sign configuration for the first two vortices. Each process then ran in parallel, computing the best match out of the quadrant of parameter space it was assigned. The best vortex configuration from each script was returned to the parent lyse script, which then repeated the computation for the four returned configurations to determine which of the four was most optimal. This resulted in a factor of four speedup in the computation time. We then implemented an additional factor of two speedup by recognising that our measure of 'goodness' was symmetric; that is, that the opposite sign configuration of the worst match would be equivalent to the best match. These combined speedups allowed us to analyse shots with up to 25 vortices in a practical time frame, however additional techniques to minimise runtime will need to be developed if higher vortex numbers are generated in future research. Finally, in the parent lyse script, we save the optimal vortex configuration to the shot file, along with a determination of the number of free, clustered, or dipole vortices, and the polarisation and first order correlation function.

In the remaining single-shot analysis script we then calculated the energy spectra of the vortex configuration.

8.2.3 Multi-shot analysis

The multi-shot analysis scripts were dedicated to each producing one figure for the main body text of our publication, along with any figures for the supplementary material that were derived from the same data as used for the primary figure. This gave us the flexibility during the manuscript preparation stage to adjust our main figures independently of each other, simply by (un)loading analysis scripts in lyse. Only minor additions to the first figure (due to the complex legend requirements at the bottom of the figure) were added outside of lyse, using Adobe Illustrator.

For our first publication figure (multi-shot analysis script) we show images of our computed optical density, measured Bragg signal, computed velocity flow field, and classification of the identified vortices for three specific shots, as an example of the procedure we follow for extracting vortex information from our experiments. As we are showing data from a fraction of the available shots, we directly pull out the required information (the results of several of the single-shot analysis scripts) from the three HDF5 shot files rather than accessing it from the DataFrame of all shots. This is primarily because there is no need to utilise any of the DataFrame functionality (for example slicing) and we would have to access each shot individually anyway as arrays stored by single shot analysis scripts are not stored in the DataFrame due to size constraints. Fortunately, using a general purpose programming language for our analysis scripts (Python) allows us to do anything we like, for example pulling out the required data using either the lyse provided `Run` object API, or by using the h5py Python library directly.

The general structure of the other three multi-shot analysis scripts are quite similar (despite producing distinct plots). In each case, we access the DataFrame, and nest calls to `groupby` in order to create slices of the DataFrame along the axes of the obstacle grid

size and the evolution time (the two main axes of our parameter space in runmanager). The graphs then usually display the mean of measured quantities from each slice of the DataFrame, as extracted by the previously run single-shot analysis scripts.

8.2.4 Dataset publication

The full dataset, including experiment logic, raw data, analysis results, analysis scripts, and both good and bad shots, is available online [172]. This allows interested parties to study our results and verify they are correct, and will help other groups build upon this avenue of research. While this is technically possible to do without the labscript suite, it is far more cumbersome. Interested parties now need only install Python, install the labscript suite, and download our raw data. Once loaded into lyse, they will be able to immediately reproduce all of our figures from our publication, and see how our results were built up from the raw data. We believe that the flexibility of the labscript suite, combined with its standardised interface, provides unprecedented simplicity for release of supplementary material (such as raw data and experimental techniques). We suspect there is a large amount of untapped potential here for using the labscript suite to simplify the transparent publication of research and for generating future research collaborations.

8.3 Objective lens development

As an example of the versatility of our control system for research outside of the field of ultracold atoms, we used it to bench test a high-resolution objective lens. While the precision timing capabilities of our system were not needed, the workflow from experiment preparation to experiment analysis was still very useful.

The projects aim was to design an objective lens with both a high-resolution and large field-of-view (FOV), for imaging BECs. For greater flexibility, and improved optical access, we required that the lens be situated outside the vacuum chamber. This requires an objective that can also correct for aberrations introduced by the vacuum glass window. The objective lens was designed by Lisa Starkey in Zemax, starting from the work of Alt [173]. Lisa's design improves on the work of Alt by obtaining a higher numerical aperture (NA) while only using readily available catalogue lenses, and is published in reference [174].

Our publication includes data on the bench testing of the lens, which was collected and analysed using the labscript suite[15]. In order to measure the FOV of the objective lens, we imaged a 1µm pinhole as a function of pinhole position with respect to the optic axis. The pinhole was mounted to 3 Zaber T-LS28M motorised translation stages under labscript control, in an x-y-z configuration (with z as the optic axis). The pinhole was first focused by adjusting the z translation stage, which then remained fixed for later shots. We used runmanager to create a parameter space for the x-y axes that defined a 60 by 60 grid spaced by approximately 12.6 µm and 10.4 µm, respectively. The parameter space scans across the range of y-coordinates (in order) for a given x-coordinate, before repeating for an incremented x-coordinate. We waited 1 s before exposing the camera in order to allow any

15. I am the second author of the publication, due to my contribution to the data collection and analysis for the bench testing.

```
from labscript import *

from labscript_devices.PulseBlaster import PulseBlaster
from labscript_devices.ZaberStageController import
                ZaberStageController, ZaberStageTLS28M
from labscript_devices.Camera import Camera

# MAIN DEVICE DEFINITIONS
PulseBlaster('pulseblaster_0', board_number=0)
ZaberStageController('ZaberController', com_port='com1')
ZaberStageTLS28M('y', ZaberController, '1')
ZaberStageTLS28M('x', ZaberController, '2')
ZaberStageTLS28M('z', ZaberController, '3')

# IMAGING SYSTEM
Camera('andor_neo_0', pulseblaster_0.direct_outputs, 'flag 2',
        BIAS_port=42518, serial_number="01512", SDK="Andor3",
        effective_pixel_size=2.24e-6,
        exposure_time=exposure_time, orientation='top')

# set position of Zaber stages
z.constant(z_pos)
x.constant(x_pos)
y.constant(y_pos)

# start the shot
start()

# Wait a little over 1 second before taking the image.
t=1e-3
t+=1

# take the image
andor_neo_0.expose('pinhole', t, 'atoms')
t+=exposure_time

# stop a little after the exposure has finished
stop(t+4e-6)
```

Figure 8.7: The experiment logic to perform the pinhole raster scan across the objective lens FOV. It should be noted that we have updated the experiment logic to be compatible with the current labscript version.

vibrations from the translation stage movement to damp out. The full experiment logic is shown in figure 8.7.

The analysis of the acquired images consists of a single-shot analysis script and a multi-shot analysis script. The single shot analysis script locates the pinhole by looking for a pixel of maximum intensity, and uses the location as the initial conditions for a 2D Gaussian fit. A multi-shot analysis script then calculates the geometric mean of the x and y widths of the pinhole image, in order to determine the spot size of the imaged pinhole as a function of position. From this we determine the FOV, based on the region in which the spot size is less than the design resolution of the object (1.3 µm), see figure 8.8 (a)-(c). We also programmatically extracted the theoretical point spread function from Zemax, and

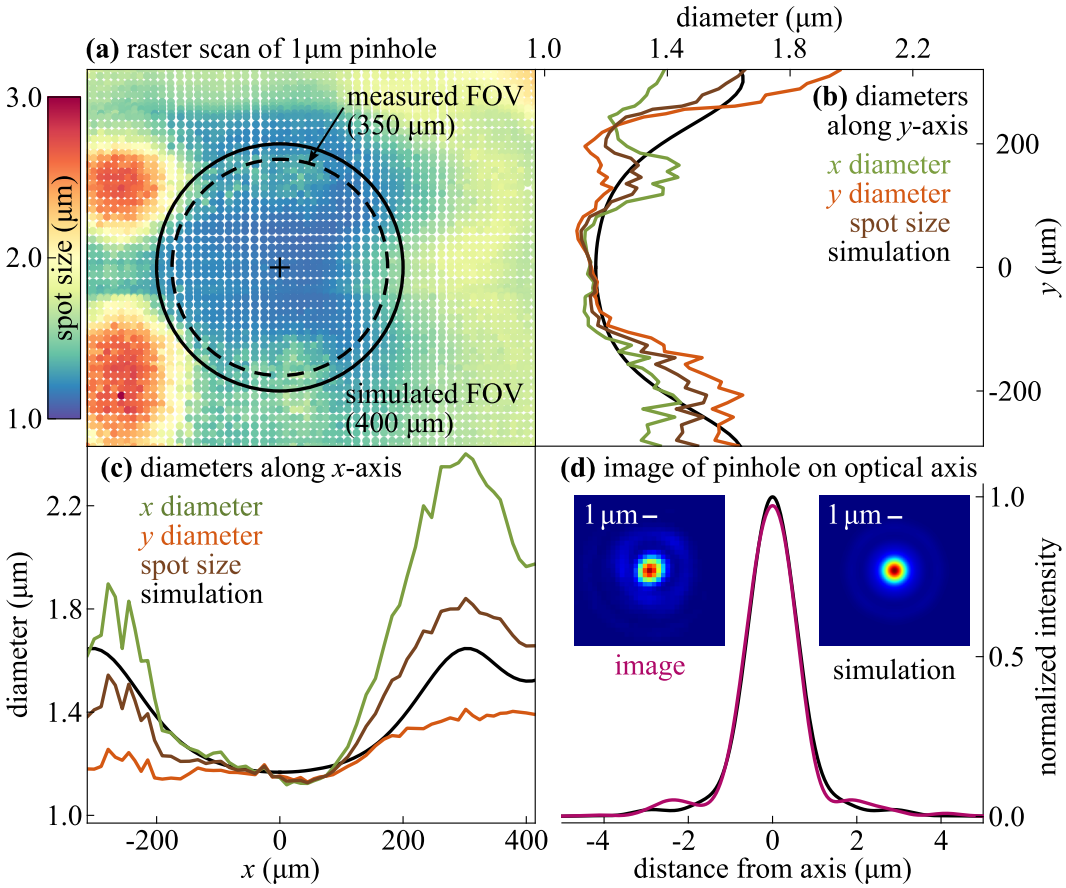


Figure 8.8: The final figure included in our publication [174], which closely matches the figures produced by lyse shown in figure 8.9. Original caption as written in the paper [174]: "(a) The spot size of a 1 µm pinhole measured at 3600 positions across the object plane. (b, c) The spot size and constituent diameters across the y- and x-axis respectively through the optic axis (cross in (a)). The measured spot size is in good agreement with our simulation within the FOV. (d) The on-axis pinhole image compared to the simulated point spread function convolved with a 1 µm pinhole. The curves are the azimuthal averages of the inset images."

convolved it with a 1 µm top-hat function in order to simulate what the spot should look like on-axis. We can see from figure 8.8 (d) that they are in close agreement.

While the step size of the translation stages is listed as 0.09921875 µm[16] (much smaller than our grid) the unidirectional accuracy is listed as 29 µm (larger than our grid) [175]. We thus rely on the observed location of the pinhole on the camera chip in order to extrapolate the x-y location of the pinhole. We can see from figure 8.8 (a) that the pinholes are located in bands along the x and y axes, as indicated by the x-coordinate spacing between adjacent lines in the y-direction. We see similar banding in the orthogonal dimension (although it is slightly less obvious). This indicates that the translation stages are moving non-linearly in the y-dimension, but consistently so for each x coordinate. We suspect this is due to imperfections in the drive rod, which causes the stage to skip by more than the expected

16. While it might seem odd that Zaber list the step size to a precision of 10 fm, I suspect this value is more accurately described as the average step size.

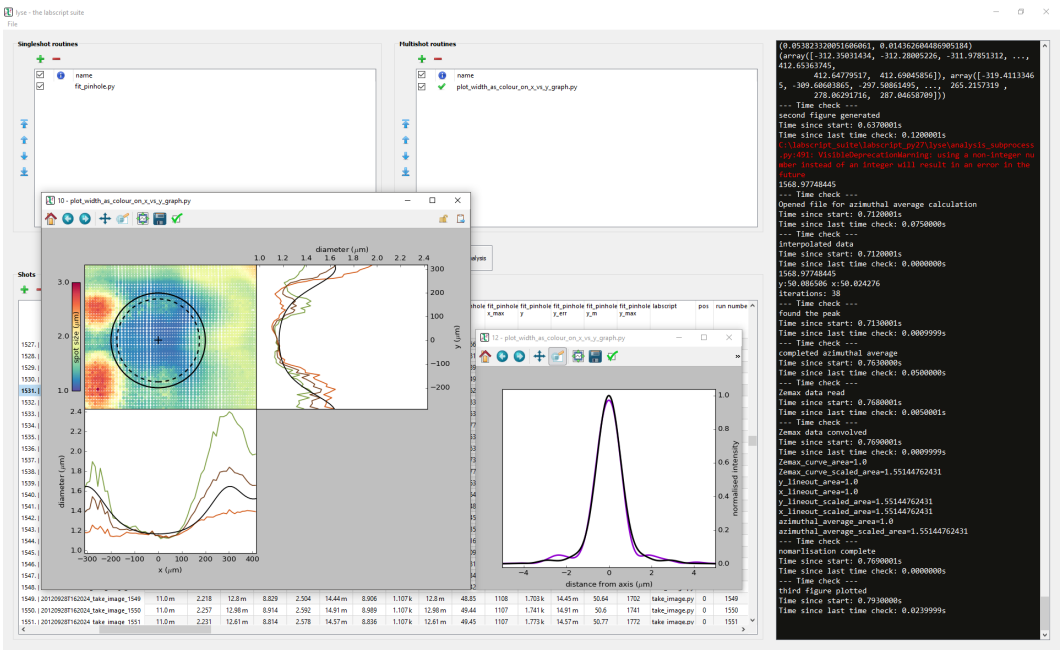


Figure 8.9: Here we show how we used lyse to produce publication quality figures from our custom Python analysis scripts. The two figures shown were exported as in a vector format (.svg) and combined using the free software Inkscape. The resultant figure (including some minor formatting changes) is shown in figure 8.8. Ultimately these formatting changes could have been applied as part of the analysis script, however as they were made during our revision of the manuscript, we opted to apply them to the combined figure by hand rather than re-producing the combined figure from scratch again.

amount at certain locations. We do also note that there is an unexplained curvature to the observed x-y coordinates for points above $y = 250\,\mu\text{m}$. However, given this is outside our designed FOV and the curvature is consistent across the entire range of x-coordinates, we do not believe it is indicative of an issue with the lens design or our analysis technique.

As we have previously stated, lyse is capable of producing publication quality figures. We demonstrate this in figure 8.9, where we show the raw plots produced by our multi-shot analysis script. As can be seen, we produced figure 8.8 (a)-(c) in one window, and figure figure 8.8 (d) in another. These were exported in .svg format, and combined in Inkscape (a vector graphics editor). In this instance, the combination in Inkscape was done primarily as it was the simplest way to combine 8.8 (d) without the axes joining (a)-(c) and while maintaining no spacing between (a)-(c). In most cases, however, complete figures can be produced directly from lyse. That said, often minor formatting changes during late revision stages are easier to apply directly from a graphics editing program, for instance to meet specific journal guidelines.

Overall, the labscript suite proved extremely useful for this research, despite it being somewhat outside the target demographic. The automation of the parameter space scan meant we could acquire a larger quantity of data (3600 shots) than was practical by hand[17],

17. In fact, due to the large quantity of closely spaced data in figure 8.8 (a), we had to add a clarification to the manuscript stating that this figure was in fact the result of 3600 measurements and not the result of a "strange interpolation" as the reviewer had thought.

and the analysis of this data was aided by the ability to structure arbitrary analysis scripts in lyse, to generate publication quality figures.

8.4 Summary

In this chapter we covered examples of how to use the labscript suite. We first described the implementation of the labscript suite in our dual-species BEC laboratory at Monash University, including the hardware in use, examples of the experiment logic, and the parameterisation of that logic via runmanager global variables. We then showed how general purpose analysis scripts can be written to aid the day-to-day work that often goes on in-between taking publication quality data. Finally, we showed how we used the labscript suite in the publication of two journal articles. In both cases, we were able to utilise the features of the labscript suite to automate parameter space scans, control non-standard equipment, and create publication quality figures. In the case of the vortex dynamics paper [127], we also published the shot files and analysis scripts online allowing other research groups to investigate our findings in detail.

Chapter 9

Conclusion

9.1 Summary

In this thesis I have presented the labscript suite, a comprehensive scientific control system for running precisely timed experiments. We have improved upon previously published control systems by striking a balance between graphical and text-based interfaces, providing interfaces for hardware abstraction to support custom hardware for novel experiments, integrating an analysis framework into the control system to automate the generation of publication ready graphics, automating record keeping, and by designing the entire suite around the need to automate the preparation, execution, and analysis of experiment shots spanning multi-dimensional parameter spaces.

In chapter 2 I reviewed the physical processes behind the production of an ultracold atom cloud, and some of the novel systems that can be studied. I then used that information to inform the hardware device requirements, which in turn informs the design of the control system. Other published control systems were reviewed against these requirement, and found lacking in several areas.

In chapter 3 I detailed the ultracold atom apparatus I constructed with colleague Shaun Johnstone and others. This apparatus is designed to produce dual-species ultracold atom clouds of rubidium and potassium, which requires a complex set of lasers, optics, magnetic coils, and hardware control devices to operate. Our apparatus, along with the spinor-BEC apparatus in a neighbouring lab, was developed in parallel with our control system, ensuring that we developed a general purpose control system applicable to many experiments. I also detailed the parts of the process control system I constructed, which runs independently of our scientific control system and ensures sensitive aspects of the apparatus are not damaged.

Designing a control system is difficult, not least because the initial choice of underlying technologies it is built upon influences the entire design and can lock a control system architect into hard-to-use features. In chapter 4 I introduced the technologies we chose and explained the careful decisions behind their selections. We chose Python as our primary programming language due to its wide range of 3rd-party libraries, popularity, low entry-bar for new users, interoperability with other languages, and its object oriented design. We then chose Qt for creating graphical interfaces, ZeroMQ for network communication, and HDF5 for data storage. Importantly, these three technologies have bindings for many other languages, ensuring the labscript suite can communicate with additional components written in a broad

range of programming languages. We also designed the labscript suite to be flexible so that it is suitable for controlling a wide range of experiments and can adapt to future needs. This was primarily achieved by following the Unix philosophy by making the components of the labscript suite modular, and ensuring experiment logic and analysis routines are defined using a general purpose programming language (Python). We create one HDF5 file per experiment shot, and the labscript components automatically store metadata in these shot files as part of the comprehensive experiment record. This includes sufficient information to reproduce the entire shot, including low-level hardware instructions (for reproducing a shot on an appropriately configured apparatus) and the high-level experiment logic and parameters (for interpreting the intention of the shot).

In chapter 5 I detailed the components of the labscript suite used to prepare experiment shots for execution. Experiment logic is choreographed in Python scripts using the labscript API. Labscript is a high-level API ensuring experiment logic scripts are readable by humans, and produces low-level hardware instructions for supported devices. When choreographing an experiment, the labscript API is first used to define the set of hardware and connections in a connection table. This creates Python objects, for each I/O channel, containing Python methods specific to an input or output type, which can be called to define the experiment logic. By designing experiment logic around output types, we simplify the mechanism for adding support for new hardware devices. This design also simplifies experiment logic by providing a consistent user interface across all device types.

Experiment logic is parameterised by global variables, which are managed from runmanager. Runmanager generates shot files by inserting the global variables into a selected experiment logic script and executing the labscript code, which generates device hardware instructions and saves them to the HDF5 shot file. Complex multi-dimensional parameter spaces are defined in runmanager by setting global variables to lists, informing runmanager to create a shot from the experiment logic for each point in the parameter space.

Runviewer was then created as a tool to create a visual representation of the experiment logic from the hardware instructions stored in a shot file, providing a faithful representation of what each hardware output should do.

In chapter 6 we detailed the components of the labscript suite used to execute experiments. This includes BLACS, which manages the execution of experiments on the apparatus, and lyse, which manages the analysis of acquired data. BLACS maintains a queue of shots to execute and provides manual control of each hardware devices when shots are not being executed. The manual device controls are dynamically generated at runtime, ensuring BLACS can handle arbitrary hardware configurations. BLACS can command secondary control systems, such as BIAS (our camera imaging software written in LabVIEW), which may run on separate PCs. Executed shots are analysed in lyse, which runs a user-specified set of analysis routines as new shots arrive. As these analysis routines are general purpose Python scripts, they can be written to generate new shots, enabling custom closed-loop optimisation routines.

In chapter 7, we show how our hardware abstraction layers allow developers to easily add support for new hardware devices. Developers only need to create four small subclasses, extending functionality provided by the labscript suite, which handle programming the device using the manufacturer's API and the format of hardware instructions.

Finally, in chapter 8, we detail examples of how we have used the labscript suite to

automate complex experiments. This includes the optimisation of our ultracold atom apparatus, and the experiments detailed in two publications [127, 174]. These experiments both involved the collection of data over a multi-dimensional parameter space, and the generation of publication quality graphics from lyse analysis routines.

We believe this thesis demonstrates that the labscript suite is currently the most comprehensive scientific control system for precisely timed experiments, and will be of significant benefit to experimentalists working with shot based experiments including those in the ultracold atom community.

9.2 Future work

Despite being, we believe, the most comprehensive control system for precisely timed experiments, we have a roadmap of significant features we would like to develop in the near future. Some of these features have been inspired by research groups at other institutions, and indeed we have already had feature contributions to our code base from researchers at several institutions including Technische Universität Darmstadt and NIST.

The most pressing is perhaps the rewrite of significant portions of the labscript API, specifically those that work with instruction timing. While the object hierarchy is now stable, the labscript timing code is some of the oldest code in the entire labscript suite, and has not kept up with our current hardware expectations. For example, while we account for delays and trigger offsets for secondary pseudoclocks, the way we account for this is by overwriting the internal timing information within labscript with a set of offset times. This means we lose information regarding the original labscript time at which the instruction was commanded. We also currently avoid floating point rounding errors by rounding all times to the nearest 0.1 ns. While what we currently do is sufficient, it has resulted in subtle bugs that we have had to fix. A more modern approach would be to represent each instruction as a Python object that keeps track of timing offsets and stores instruction times in integer multiples of the clock resolution of the master pseudoclock.

We are planning improved functionality for feedback between analysis results and future shots. We expect to improve the runmanager API, and BLACS, so that quick analysis scripts can be run immediately on shot completion, update global variables in runmanager, and remotely trigger the compilation of the next shot. We will also be looking into just-in-time compilation, which would allow a precompilation of most of a shot, with a final modification to be done by BLACS just before executing the shot, allowing fast feedback of results into future shots.

While the labscript suite supports secondary control systems, they are currently expected to be standalone programs. Currently this means that there is no easy way to spread already supported hardware across multiple PCs without writing custom software. We are thus working on the ability to launch BLACS as a secondary control system, which would allow any supported hardware device to be launched on any number of appropriately configured PCs. We have already defined the syntax for specifying this in a labscript connection table, and work to allow BLACS to understand this syntax is ongoing. As a related feature, we are working on the ability to launch the existing BLACS worker processes on remote PCs, allowing the graphical interface for the device, and the hardware connection, to be physically

separated.

Our extensive use of lyse has indicated to us that a missing feature is an interface for managing analysis global variables, in a similar vein to the way runmanager handles experiment logic globals. We currently work around this by specifying some analysis globals either in runmanager (prior to the shot running), or in a Python file we import into each analysis script. However, we believe there is much room for improvement here, and are looking at how we can improve the lyse GUI along these lines.

Another improvement for lyse we are looking at is better support for multi-core CPUs, for which there are several use-cases for. The first is for computationally intensive tasks. While parallel processing is possible with lyse, as discussed in §8.2, this was done just by launching subprocesses rather than using existing parallelisation libraries, which are currently incompatible with lyse. Correcting this incompatibility would be beneficial to many users. We are also considering how we can use multiple CPU cores to speed up analysis of each shot by allowing multiple analysis scripts to run at once (each on a different core). This would be fairly straightforward for multi-shot analysis scripts, but would require inter-script dependency resolution for single-shot analysis scripts. Finally, we are considering adding support for parallelising the analysis of multiple shots at once (which would not run foul of the single-shot inter-script dependency resolution). This is particularly useful if you need to re-run an updated single-shot analysis script on all shots in a sequence that has been previously acquired and reloaded into lyse (for example if you have just corrected a bug in your single-shot analysis code).

We are interested in exploring the use of the labscript suite framework outside of running experiments. For example, much of the labscript suite may be useful for theorists who wish to run simulations that span significant parameter spaces. By replacing labscript and BLACS appropriately, the suite could be made to compile shots with information on the simulation to run, to be executed later either locally or on a cluster, with the results sent to lyse where analysis scripts could parse the results and produce publication quality graphs. As well as managing the lifecycle of the simulation, you also get the benefits of being able to perform quick parameter space scans and a comprehensive record of each simulation within a self contained HDF5 file. To this end, I have begun to modify runmanager to support arbitrary APIs other than labscript, in order to turn it into a general purpose parameter space management program.

As we hope you can see, there is still much scope for continued research into useful features for aiding precisely timed experiments, which we hope to accomplish with future developments of the labscript suite.

Appendix A

Glossary

analysislib A Python module (library) for storing Python analysis files and helpful functions and/or attributes for use during analysis. 159

AOM acousto-optic modulator. 14, 32, 33, 36, 37, 38, 39, 43, 45, 47, 49, 50, 52, 155, 158, 160

API application programming interface. 5, 17, 22, 27, 56, 58, 60, 61, 68, 69, 75, 79, 80, 81, 84, 87, 94, 99, 100, 101, 105, 114, 119, 121, 122, 123, 134, 137, 138, 139, 140, 141, 143, 146, 147, 150, 152, 153, 158, 167, 168, 176, 177, 178, 179, 180

BEC Bose–Einstein condensate. 6, 11, 24, 26, 27, 31, 32, 35, 40, 52, 56, 64, 72, 73, 87, 108, 112, 155, 156, 157, 169, 173, 175

connection table A set of Python calls to the labscript API that define the set of hardware, including attached inputs and outputs, in use for an experiment along with information on how these devices are connected together and to the control PC(s). See 5.1.1 for further details. 58, 61

DAQ Short for 'data acquisition'. Usually used to describe a data acquisition device. 155

DCS distributed control system. 3

DDS direct digital synthesiser. 1, 14, 61, 81, 84, 99, 103, 155

DMD digital micromirror device. 14, 155, 163

ECDL external cavity diode laser. 32, 35, 36, 39

experiment logic The definition of what happens with the experiment, as a function of time. This can take many forms, such as high level descriptions like `make_bec(t)`. 5, 17, 18, 55, 56, 58, 80, 180

experiment time Time, as experienced by the experiment, where $t = 0$ corresponds to the master pseudoclock outputting the first change in output (usually a pulse to trigger secondary pseudoclocks). No further adjustment is made to recorded times, thus the time of an event after a wait will be dependent on the length of the wait in that specific shot. 97, 128

FOV field-of-view. 169, 170, 171

FPGA field-programmable gate array. 3, 24, 28

FSM finite state machine. 2

gate Refers to the ability to turn on or off a process. For example, you could (naively) gate a signal by connecting that signal through a relay. Controlling the relay (via a separate digital line) allows you to turn on or off (to gate) the original signal. 94

GUI graphical user interface. 5, 17, 22, 56, 60, 61, 64, 68, 73, 76, 106, 114, 115, 119, 121, 122, 123, 127, 128, 131, 134, 137, 138, 143, 148, 149, 150, 152, 167, 178

hardware device An I/O device that is used to control an apparatus. Hardware devices are programmed by the application BLACS (or a secondary control system) based on instructions produced by the labscript API. 1, 4, 5, 6, 17, 18, 28, 56, 58, 56, 60, 81, 121, 180, 182

hardware instructions A set of data, representing the experiment logic to be performed, formatted for programming into a specific hardware device. These are typically very low level instructions, in a format close to what the device programming API requires, and are typically not easily read by humans. 14, 17, 56, 99, 100

HDF5 hierarchical data format version 5. v, vi, 48, 56, 60, 61, 64, 66, 71, 73, 75, 84, 89, 93, 96, 99, 100, 103, 104, 105, 106, 105, 107, 106, 108, 114, 115, 116, 118, 128, 131, 134, 136, 140, 145, 146, 152, 153, 166, 167, 168, 175, 176, 178, 182

HG Hermite-Gaussian. 163

HMI human-machine interface. 3

I/O input and output. 3, 4, 5, 12, 14, 15, 48, 49, 55, 61, 68, 70, 76, 81, 123, 145, 155, 176, 180, 190, 214, 231

IDE integrated development environment. 25

IGBT insulated-gate bipolar transistor. 51, 52

JSON Short for JavaScript Object Notation, JSON is a cross-language format for passing hierarchical data such as dictionaries (key, value pairs) and/or arrays. The JSON specification is available at [176]. 147

labconfig We store configuration settings for the labscript suite in a '.ini' style file. In order to avoid a bootstrapping issue, the file is named after the local network hostname of the PC so that the labscript suite knows where to find it when combined with a list of known locations for the file (such as the users home directory or the location created by the installer). Settings in this file include the network ports over which components communicate and file or folder paths to expected locations of user scripts, shot files, and certain 3rd party software (such as a preferred text editor). 105

labscriptlib A Python module (library) for storing experiment logic files. Experiment logic files imported from this module are automatically saved in the shot file by labscript (see §5.1.8). 80, 99

labscript time Time, as defined in labscript, where $t = 0$ corresponds to the master pseudoclock outputting the first change in output (usually a pulse to trigger secondary pseudoclocks) and the length of every wait is assumed to be 0 (when excluding the retriggering delays of all pseudoclocks). 97, 128

lock In software engineering, a lock typically refers to a mechanism for serialising access to a resource that cannot handle simultaneous access from multiple sources. The resource could be anything from an internal data structure to an external file on a network drive. Lock implementations typically have methods for checking if a lock is in use, acquiring the lock (marking it as in use) when it becomes free, and releasing the lock when it is no longer needed (so that it can be acquired somewhere else when required). There is no way to force the use of the lock (since it is external to the resource), so this form of locking only works if all code accessing the resource is explicitly written to use the locking system. 75, 76

LVTTL low-voltage transistor-transistor logic. 50

master pseudoclock A pseudoclock that is triggered by BLACS and ultimately dictates the timing of an entire shot. If waits are used in a shot, this pseudoclock is hardware triggered to resume the experiment at the appropriate time, which in turn triggers all secondary pseudoclocks. 182

monkey-patch A broad term to describe local changes made to software by an external actor. In Python, this term typically refers to the addition or modification of object attributes or methods, after instantiation, by code external to the object's class definition. The modifications are thus local to the modified objects, and are technically unsupported by the internal logic of the object (although careful monkey-patching can integrate seamlessly). Similar modifications can also be applied to classes, in which case any subsequent object instantiated using that class would be created with those modifications. 75

MOT magneto-optical trap. 9, 10, 11, 13, 33, 37, 38, 39, 40, 42, 43, 44, 45, 46, 47, 51, 52, 56, 87, 96, 97, 112, 157, 163

NA numerical aperture. 169

namespace The scope in which a unique variable name is valid (can be accessed by name). For example, a variable defined for the first time inside a Python function exists in the namespace of the function, but cannot be accessed from outside of that function (unless explicitly specified). 89, 113, 114

NI National Instruments. 14, 18, 24, 26, 56, 80, 81, 96, 97, 99, 123, 146, 148, 155, 157

NPBS non-polarising beam splitter. 44, 45, 46

OD optical density. 166, 167

PBSC polarising beam splitter cube. 43, 44

PGC polarisation gradient cooling. 10, 13, 40

PID proportional–integral–derivative. 2, 48

PLC programmable logic controller. 2, 3, 48, 49

pseudoclock A device which provides a non-uniform clocking signal, which is used to trigger other devices to update their output state. The master pseudoclock and any secondary pseudoclocks are categories of pseudoclocks used in the labscript suite. See 5.1.4 for further details. 5, 15, 18, 24, 27, 60, 79, 80, 91, 181, 182

RAID redundant array of inexpensive disks. 72

rf radio-frequency. v, 1, 11, 13, 14, 32, 49, 50, 52, 153, 155, 193

RPC remote procedure call. 123

sandbox A model for isolating software from other software and/or PC resources. The most common implementation is at the PC operating system level, which prevents a process from directly accessing (and modifying) the memory of another. We use sandboxes in the labscript suite to isolate user code and hardware device drivers from the main labscript suite applications. 6, 77, 123, 141, 152

SCADA supervisory control and data acquisition. 3

secondary control system An additional application for controlling hardware devices that runs on a separate PC, but is controlled (during shot execution) by BLACS. See 6.2 for further details. 64, 180

secondary pseudoclock A pseudoclock that is triggered by a digital signal produced by the master pseudoclock at the start of an experiment shot (and the end of each wait) but otherwise runs independently. 60, 181, 182

shot A single execution of an experiment. This refers to a collection of hardware instructions that will be preprogrammed into a set of devices, triggered to start on command, and determine a distinct end-point to the experiment. More generally, we use this term to refer to this single experiment at all stages, from conception to analysis. 4, 7, 18, 55, 56, 64, 77, 179, 181, 182

shot file The HDF5 file containing all data related to a single shot. 56, 60

SLM spatial light modulator. 1, 14, 24, 163

TA tapered amplifier. 32, 33, 35, 36, 39

TOF time-of-flight. 159, 163

TTL transistor-transistor logic. 50

UHV ultra-high vacuum. 4, 10, 40

widget A button or value control such as a textbox. More generally, this refers to any of the building blocks that make up a graphical interface. 18

worker process An isolated software process that is spawned to perform a specific task and is under the control of a parent application. For example, BLACS spawns a worker process for each device, which handles communication with a single hardware device. Lyse spawns a worker process for each analysis script loaded, where the analysis script can be executed in an isolated environment. Worker processes can take advantage of multi-core CPUs in a PC, and provide protection to the main application from crashes caused by 3rd party code (such as device drivers or user analysis scripts). 56, 61, 123, 138, 139, 143

zip group A group of two or more experiment parameters which together form a single axis of a parameter space and iterate in lock-step. 22, 60, 112

Appendix B

Coil interlock design

The Galil code for the coil interlock, followed by the circuit board layout (figure B.1) and schematic diagram (figure B.2) of the custom Galil breakout board we designed. Note, the Ethernet reconnection code was provided by Martijn Jasperse [106].

```
REM ======================[ KRB COIL INTERLOCK ]=======================
REM simple monitor to check coil water cooling and disable current drivers

#AUTO; 'resume from here on power-cycle
REM @AN[3] = Top T sensor 1
REM @AN[4] = Top T sensor 2
REM @AN[5] = Bottom T sensor 1
REM @AN[6] = Bottom T sensor 2
REM @AN[0] = Top coil flow
REM @AN[1] = Bottom coil flow
REM @AN[2] = Bypass flow
REM @AN[7] = NOT IN USE
REM (See Caps+numbers. Eg. AITOPT1 vvvvvvv)
REM Analog In TOP Temperature sensor 1 or 2
AITOPT1 = 3
AITOPT2 = 4
REM Analog In BOTtom Temperature sensor 1 or 2
AIBOTT1 = 5
AIBOTT2 = 6
AITOPFL = 0; 'Analog In TOP FLow sensor
AIBOTFL = 1
AIBYPFL=2;'AnalogIn BYPass FLow sensor

REM @OUT[0] = Top T sensor 1 not OK
REM @OUT[1] = Top T sensor 1 OK
REM @OUT[2] = Top T sensor 2 not OK
REM @OUT[3] = Top T sensor 2 OK
REM @OUT[4] = Bottom T sensor 1 not OK
REM @OUT[5] = Bottom T sensor 1 OK
REM @OUT[6] = Bottom T sensor 2 not OK
REM @OUT[7] = Bottom T sensor 2 OK
REM @OUT[8] = Top coil flow not OK
REM @OUT[9] = Top coil flow OK
REM @OUT[10] = Bottom coil flow not OK
REM @OUT[11] = Bottom coil flow OK
REM @OUT[12] = Bypass flow not OK
REM @OUT[13] = Bypass flow OK
REM @OUT[14] = Enable PSU
REM @OUT[15] = Enable Water

REM Digital Out Top coil Temperature sensor 1 (or 2) Not OK led
DOTT1NOK = 0
REM Digital Out Top coil Temperature sensor 1 (or 2) OK led
DOTT1OK = 1
DOTT2NOK = 2
DOTT2OK = 3
REM Digital Out Bottom coil Temperature sensor 1 or 2) Not OK led
DOBT1NOK = 4
DOBT1OK = 5
DOBT2NOK = 6
DOBT2OK = 7
REM Digital Out Top coil FLow Not OK led
DOTFLNOK = 8
DOTFLOK = 9
DOBFLNOK = 10
DOBFLOK = 11
REM Digital Out bYpass FLow Not OK led
DOYFLNOK = 12
DOYFLOK = 13
DOENPSU = 15; 'Digital Out ENable PSU
DOENWTR = 14; 'Digital Out ENable WaTeR

REM @IN[0] = Start Interlock
REM @IN[1] = Stop Interlock
REM @IN[2] = PSU relay open?
REM @IN[3] = PSU relay close?
REM @IN[4] = Water relay open?
REM @IN[5] = Water relay close?
DISTRT=0;'Digital Input STaRT interlock
DISTOP=1;'Digital Input STOP interlock
DIPSUO=4;'Digital Input PSU relay Open
'Digital Input PSU relay Closed
DIPSUC=5
'Digital Input WaTeR relay Open
DIWTRO=2
'Digital Input WaTeR relay Closed
DIWTRC=3

REM Whether we use the flow/T sensors
REM If we are not using any flow sensors then we have a flow error (this is so you don't get a warning about solenoids failing)
REM This means you MUST use at least one flow sensor if you want this to interlock to be able to turn on!
USETT1 = 1
USETT2 = 1
USETB1 = 0
USETB2 = 0
USEFLT = 1
USEFLB = 1
USEFLY = 0
```

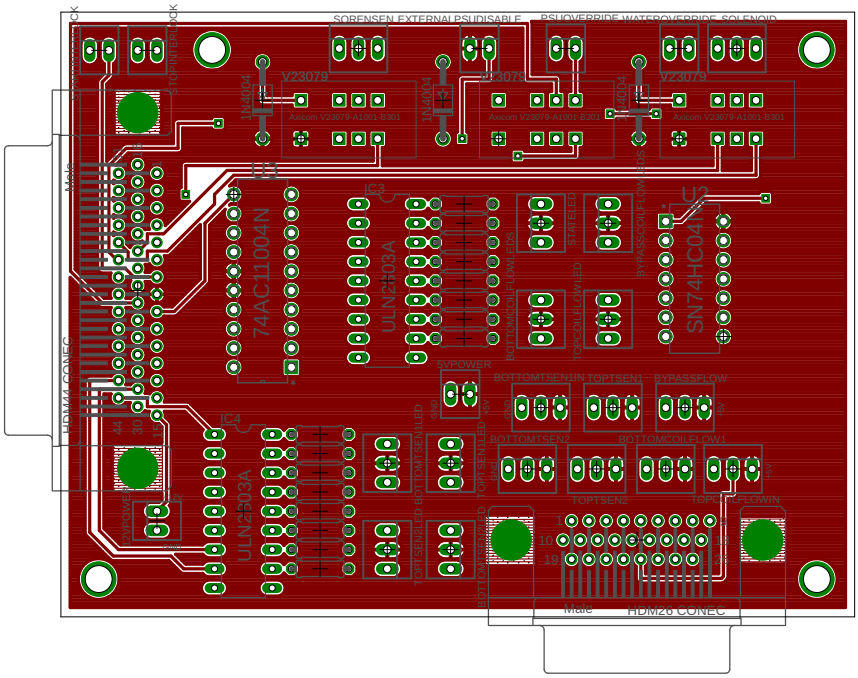
```
89
90  REM =======[ ANALOG LIMITS]=======
91  'conversion factor: 1L/min = 2.6V
92  FLTOPM = 0.7; 'Minimum Flow
93  FLBOTM = 0.7
94  FLBYPM = 0.2
95  FLTOPMA = 1.0; 'Maximum Flow
96  FLBOTMA = 1.0
97  FLBYPMA = 4.0
98  TEMPT1M = 25; 'Maximum Temp in degrees
99  TEMPT2M = 35
100 TEMPB1M = 30
101 TEMPB2M = 30
102 MINT = 4; 'Minimum T in degrees
103
104 'exponential averaging overlap factor
105 OVRLAP=0.99
106 V=OVRLAP
107 LOGCOUNT=0
108 ERRCOUNT=0
109 PEERRCNT=0
110
111 ' Power cycle LEDs
112 SB DOTT1NOK
113 SB DOTT2NOK
114 SB DOBT1NOK
115 SB DOBT2NOK
116 SB DOTFLNOK
117 SB DOBFLNOK
118 SB DOYFLNOK
119 CB DOENPSU
120 CB DOENWTR
121 AO0,0
122 WT1000
123 CB DOTT1NOK
124 CB DOTT2NOK
125 CB DOBT1NOK
126 CB DOBT2NOK
127 CB DOTFLNOK
128 CB DOBFLNOK
129 CB DOYFLNOK
130 SB DOTT1OK
131 SB DOTT2OK
132 SB DOBT1OK
133 SB DOBT2OK
134 SB DOTFLOK
135 SB DOBFLOK
136 SB DOYFLOK
137 AO0,5
138 WT1000
139 CB DOTT1OK
140 CB DOTT2OK
141 CB DOBT1OK
142 CB DOBT2OK
143 CB DOTFLOK
144 CB DOBFLOK
145 CB DOYFLOK
146 AO0,0
147 WT1000
148
149 'initialise variables
150 FLTOP = @AN[AITOPFL]
151 FLBOT = @AN[AIBOTFL]
152 FLBYP = @AN[AIBYPFL]
153 TEMPT1V = @AN[AITOPT1]
154 TEMPT2V = @AN[AITOPT2]
155 TEMPB1V = @AN[AIBOTT1]
156 TEMPB2V = @AN[AIBOTT2]
157 JS#CALCT
158
159 LED=0; 'Set the LED red
160 LEDSTATE=0; 'the state of the LED when flashing
161 STATE = 0
162 PSUEXT = 0; 'Is the PSU external disable tripped (0 =
    normal operation, 1 = PSU disabled externally)
163
164 XQ#RECONN,1;'connect to the network
165 XQ#RUNLED,2; 'Run LED (so that we can make it blink!)
166
167 TERR=0
168 REPORT=0
169
170 REM ********************
171 REM ***** OFF STATE *****
172 REM ********************
173 #OFF
174 STATE = 0
175 REM check for start signal
176 ' Has Start been called?
177 IF(TERR=0)&(@IN[DISTRT]=1)
178     JP#RUN
179 ENDIF
180 ' Check sensors
181 JS#CHKSTAT
182 WT10
183 ' Loop!
184 JP#OFF
185
186
187 REM *************************
188 REM ***** RUNNING STATE *****
189 REM *************************
190 #RUN
191 STATE = 1
192 REPORT=1
193 ' start the interlock
194 SB DOENWTR
195 SB DOENPSU
196 LED=1; 'Set the LED green
197 WT3000
198 FLERR=0
199 FLTOP=@AN[AITOPFL]
200 FLBOT=@AN[AIBOTFL]
201 FLBYP=@AN[AIBYPFL]
202 IF(WTRRUN=0)&(PSURUN=0)
203     MG"NOTICE,Interlock Started"
204 ENDIF
205
206 #RUNLOOP
207 ' Check for stop signal
208 IF(@IN[DISTOP]=1)
209     WT1000; 'Wait 1s for the person to take their finger
        off the button
210     ' Jump to COOL state
211     JP#COOL
212 ENDIF
213
214 ' Check sensors
215 JS#CHKSTAT
216 ' Should we go to SHUTDOWN or COOL state?
217 IF(WTRRUN=1)|(PSURUN=1)
218     IF (FLERR=1)|(TERR=1)|(PSUEXT=1)
219         F=FLERR
220         T=TERR
221         P=PSUEXT
222         MG"CRITICAL,FlowError=",F{F1.0}{N}
223         MG", TempError=",T{F1.0}{N}
224         MG", PSUExtDisable=",P{F1.0}{N}
225         IF (PSUEXT=1|TERR=1)&(FLERR=0)
226             MG", status='Cooling'"
227         ELSE
228             MG", status='E. Shutdown'"
229         ENDIF
230         IF(FLERR=1)
231             JP#SHUTDWN
232         ENDIF
233
234         IF (FLERR=0)&(TERR=1|PSUEXT=1)
235             ' Only Temperature error, or PSU Ext-disable,
                then go to cool state
236             JP#COOL
237         ENDIF
```

```
    ENDIF
ENDIF

' Loop!
WT10
JP#RUNLOOP



REM ************************
REM ***** COOLING STATE *****
REM ************************
#COOL
STATE = 2
REPORT=1
LED=2
'Disable PSU
CB DOENPSU; 'Shutdown PSU
MG"NOTICE,Interlock Cooling"

LOOP = 0
#CLOOP
IF (LOOP < 2000)
    ' Loop for 30s while cooling, but only if the PSU is actually off
    ' Check for stop signal
    IF(@IN[DISTOP]=1)
        ' Immediately jump to SHUTDWN state
        JP#SHUTDWN
    ENDIF

    ' Check sensors
    JS#CHKSTAT
    WT10

    ' Should we go to SHUTDOWN state?
    IF (FLERR=1)
        JP#SHUTDWN
    ENDIF

    LOOP = LOOP + 1
    JP#CLOOP
ENDIF

' Now go to shutdown state
JP#SHUTDWN



REM **************************
REM ***** SHUTDOWN STATE *****
REM **************************
#SHUTDWN
REPORT = 1
STATE = 3
CB DOENWTR; 'Shutdown water
CB DOENPSU; 'Shutdown PSU
LED=0; 'Set running LED red

' Wait 3s for the relay to change state, and the flow to stop (this avoids solenoid failure messages!)
LOOP = 0
#SDLOOP
IF (LOOP < 200)
    JS#CHKSTAT
    WT10
    LOOP = LOOP + 1
    JP#SDLOOP
ENDIF

' Get Relay status
PSUORLY = @IN[DIPSUO]
PSUCRLY = @IN[DIPSUC]
WTRORLY = @IN[DIWTRO]
WTRCRLY = @IN[DIWTRC]
IF(PSUORLY=1)&(PSUCRLY=0)&(WTRORLY=1)‘
  &(WTRCRLY=0)
  'We are fine
ELSE
MG"ALERT,E.Shutdown Failed to dis",‘
"engage relays. Water and/or "{N}
MG"Sorensen PSU may still be on."
ENDIF

' Send a log message now with the current state!
REPORT=1
JS#CHKSTAT


MG"NOTICE, Interlock Stopped"
' Now go to off state!
JP#OFF




REM ************************ SYSTEM CHECK ************************
#CHKSTAT
WTRRUN = @OUT[DOENWTR]
PSURUN = @OUT[DOENPSU]

'exponential averaging to smooth data
FLTOP=(V*FLTOP)+((1.0-V)*@AN[AITOPFL])
FLBOT=(V*FLBOT)+((1.0-V)*@AN[AIBOTFL])
FLBYP=(V*FLBYP)+((1.0-V)*@AN[AIBYPFL])
TEMPT1V=(OVRLAP*TEMPT1V)+((1.0-OVRLAP)‘
       *@AN[AITOPT1])
TEMPT2V=(OVRLAP*TEMPT2V)+((1.0-OVRLAP)‘
       *@AN[AITOPT2])
TEMPB1V=(OVRLAP*TEMPB1V)+((1.0-OVRLAP)‘
       *@AN[AIBOTT1])
TEMPB2V=(OVRLAP*TEMPB2V)+((1.0-OVRLAP)‘
       *@AN[AIBOTT2])
JS#CALCT

' CHECK FLOW SENSORS AND UPDATE LEDs
FLERR = 0

' Check flow of top coil
IF(FLTOPM<FLTOP)&(FLTOP<FLTOPMA)
  CB DOTFLNOK
  SB DOTFLOK
ELSE
  IF(USEFLT=1)
    FLERR = 1
    SB DOTFLNOK
    CB DOTFLOK
  ELSE
    CB DOTFLNOK
    CB DOTFLOK
  ENDIF
ENDIF

' Check flow of bottom coil
IF(FLBOTM<FLBOT)&(FLBOT<FLBOTMA)
  CB DOBFLNOK
  SB DOBFLOK
ELSE
  IF(USEFLB=1)
    FLERR = 1
    SB DOBFLNOK
    CB DOBFLOK
  ELSE
    CB DOBFLNOK
    CB DOBFLOK
  ENDIF
ENDIF

' Check flow of bypass loop
'IF flow OK, show OK, otherwise if flow not OK show error if in use
IF(FLBYPM<FLBYP)&(FLBYP<FLBYPMA)
    CB DOYFLNOK
    SB DOYFLOK
```

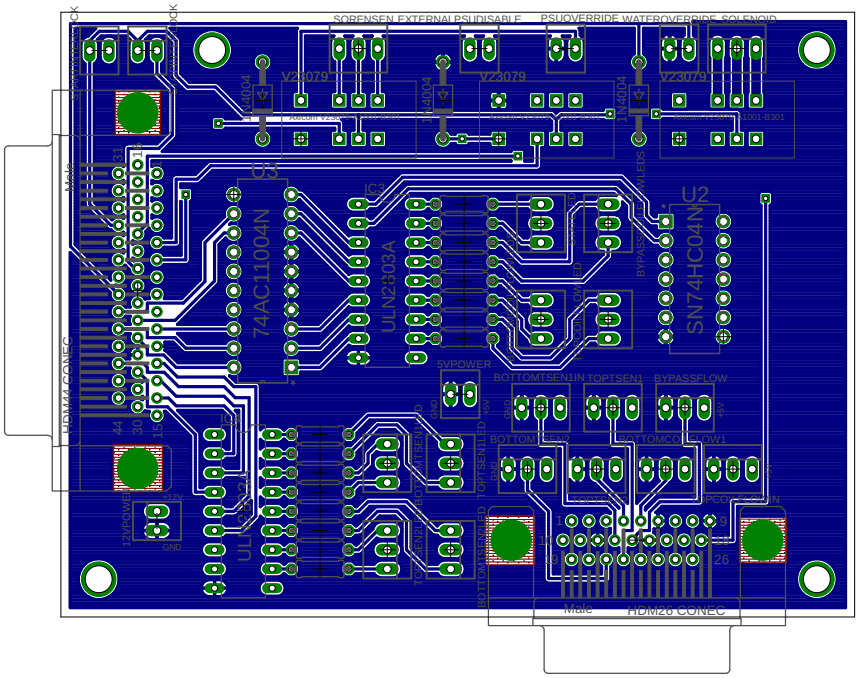
```
ELSE
  IF(USEFLY=1)
    FLERR = 1
    SB DOYFLNOK
    CB DOYFLOK
  ELSE
    CB DOYFLNOK
    CB DOYFLOK
  ENDIF
ENDIF


REM If we are not using any flow sensors then we have a flow error (this is so you don't get a warning about solenoids failing)
REM This means you MUST use at least one flow sensor if you want this to interlock to be able to turn on!
IF(USEFLY=0)&(USEFLB=0)&(USEFLT=0)
    FLERR=1
ENDIF

' If WTRRUN is 0 then ignore flow errors
IF (WTRRUN=0)
    FLERR=0
ENDIF

TERR=0
IF(USETT1=1)
    IF(TEMPT1>TEMPT1M)|(TEMPT1<MINT)
        TERR=1
        SB DOTT1NOK
        CB DOTT1OK
    ELSE
        CB DOTT1NOK
        SB DOTT1OK
    ENDIF
ELSE
    CB DOTT1NOK
    CB DOTT1OK
ENDIF

IF(USETT2=1)
    IF(TEMPT2>TEMPT2M)|(TEMPT2<MINT)
        TERR=1
        SB DOTT2NOK
        CB DOTT2OK
    ELSE
        CB DOTT2NOK
        SB DOTT2OK
    ENDIF
ELSE
    CB DOTT2NOK
    CB DOTT2OK
ENDIF

IF(USETB1=1)
    IF(TEMPB1>TEMPB1M)|(TEMPB1<MINT)
        TERR=1
        SB DOBT1NOK
        CB DOBT1OK
    ELSE
        CB DOBT1NOK
        SB DOBT1OK
    ENDIF
ELSE
    CB DOBT1NOK
    CB DOBT1OK
ENDIF

IF(USETB2=1)
    IF(TEMPB2>TEMPB2M)|(TEMPB2<MINT)
        TERR=1
        SB DOBT2NOK
        CB DOBT2OK
    ELSE
        CB DOBT2NOK
        SB DOBT2OK
    ENDIF
ELSE
    CB DOBT2NOK
    CB DOBT2OK
ENDIF

' Get Relay status
PSUORLY = @IN[DIPSUO]
PSUCRLY = @IN[DIPSUC]
WTRORLY = @IN[DIWTRO]
WTRCRLY = @IN[DIWTRC]

' Is the PSU override on?
IF(PSUCRLY=1)&(PSURUN=0)
    PSUOVR=1
ELSE
    PSUOVR=0
ENDIF

' Is the Water override on?
IF(WTRCRLY=1)&(WTRRUN=0)
    WTROVR=1
ELSE
    WTROVR=0
ENDIF

' Is the PSU disabled via the external input to the coil-interlock box?
IF (PSUORLY=1)&(PSURUN=1)
    ' Debounce this, as the relay takes some time to switch
    IF (PEERRCNT=200)
        PSUEXT=1
    ELSE
        PEERRCNT=PEERRCNT+1
    ENDIF
ELSE
    PEERRCNT=0
    PSUEXT=0
ENDIF

'LOG STATUS MESSAGE EVERY 30s
LOGCOUNT = LOGCOUNT + 1
REM Only message every 30 seconds (this number takes into account the length of time it takes to execute the code, in this loop, as well as the WT<milliseconds> commands)
IF (LOGCOUNT=2000)
    REPORT=1
    LOGCOUNT=0
ENDIF

REM Report errors immediately, but then fall back to standard logging every 30s
IF(FLERR=1)|(TERR=1)|(PSUEXT=1)
    IF(ERRCOUNT=0)
        REPORT=1
    ENDIF
    ERRCOUNT=1
ELSE
    ERRCOUNT=0
ENDIF

IF(REPORT=1)
    ' What status should this be?
    IF (FLERR=1)|(TERR=1)|(PSUEXT=1)|‘
    (PSUOVR=1)|(WTROVR=1)
        MG"WARNING,status="{N}
    ELSE
        MG"INFO,status="{N}
    ENDIF

    'What state are we in?
    IF (STATE=0)
        MG"Off."{N}
    ENDIF
```

```
530
531     IF (STATE=1)
532         MG"Running."{N}
533     ENDIF
534
535     IF (STATE=2)
536         MG"Cooling."{N}
537     ENDIF
538
539     IF (STATE=3)
540         MG"Shutting down."{N}
541     ENDIF
542
543     IF (FLERR=1)|(TERR=1)|(PSUEXT=1)|‘
544     (PSUOVR=1)|(WTROVR=1)
545         ’What message should we send?
546         IF (FLERR=1)
547             MG" Flow Error."{N}
548         ENDIF
549         IF (TERR=1)
550             MG" Temperature Error."{N}
551         ENDIF
552         IF (PSUEXT=1)
553             MG" PSU externally disabled."{N}
554         ENDIF
555         IF (PSUOVR=1)
556             MG" PSU override engaged."{N}
557         ENDIF
558         IF (WTROVR=1)
559             MG" Water override engaged."{N}
560         ENDIF
561     ENDIF
562
563     ’MG"’"{N}
564     MG",PSUon=",PSURUN{F1.0},‘
565     ", WaterOn=",WTRRUN{F1.0}{N}
566
567     IF(USEFLT=1)
568 MG",TopCoilFlow="{N}
569 MGFLTOP{Z2.3}{N}
570     ENDIF
571
572     IF(USEFLB=1)
573 MG", BottomCoilFlow="{N}
574 MGFLBOT{Z2.3}{N}
575     ENDIF
576
577     IF(USEFLY=1)
578 MG", BypassCoilFlow="{N}
579 MGFLBYP{Z2.3}{N}
580     ENDIF
581
582     IF(USETT1=1)
583     MG", TopTemp1="{N}
584     MGTEMPT1{Z2.3}{N}
585     ENDIF
586
587     IF(USETT2=1)
588     MG", TopTemp2="{N}
589     MGTEMPT2{Z2.3}{N}
590     ENDIF
591
592     IF(USETB1=1)
593     MG", BottomTemp1="{N}
594     MGTEMPB1{Z2.3}{N}
595     ENDIF
596
597     IF(USETB2=1)
598     MG", BottomTemp2="{N}
599     MGTEMPB2{Z2.3}{N}
600     ENDIF
601 MG", PSUExtDisable=",PSUEXT{F1.0}{N}
602 MG", WaterOverride=",WTROVR{F1.0},‘
603     ", ","PSUOverride=",PSUOVR{F1.0}
604 ENDIF
605 ’ Set REPORT=0 so that we won’t report next time unless
    an error, or a state requests it
606 REPORT = 0
607 EN
608
609
610 #DOTCALC
611 IF (CALV<0.1)
612   CALT=0
613 ELSE
614   CALTMP = 5/CALV
615   IF (CALTMP=1)
616     CALT = 0
617   ELSE
618 CALTMP=@SQR[29.5/((5/CALV-1)*9.98)+1]
619     IF (CALTMP < 1.377)
620       CALT = 0
621     ELSE
622       CALT = 53.85*@SQR[CALTMP-1.376]
623     ENDIF
624   ENDIF
625 ENDIF
626 EN
627
628 #CALCT
629 ’ Get TEMPT1V and put in variable CALV
630 ’ Run temp routine
631 ’ place result from CALT into TEMPT1
632 CALV = TEMPT1V
633 JS#DOTCALC
634 TEMPT1 = CALT
635
636 CALV = TEMPT2V
637 JS#DOTCALC
638 TEMPT2 = CALT
639
640 CALV = TEMPB1V
641 JS#DOTCALC
642 TEMPB1 = CALT
643
644 CALV = TEMPB2V
645 JS#DOTCALC
646 TEMPB2 = CALT
647 EN
648
649 REM *************** RUN LED updater ****************
650 #RUNLED
651 IF (LED = 0)
652     AO0,0; ’Set running LED red
653     LEDSTATE=0
654 ENDIF
655 IF (LED = 1)
656     AO0,5; ’Set running LED green
657     LEDSTATE=0
658 ENDIF
659
660 ’ Blink the LED
661 IF (LED = 2)
662     IF(LEDSTATE=0)
663         LEDSTATE=1
664         AO0,0; ’Set running LED red
665     ELSE
666         LEDSTATE=0
667         AO0,5; ’Set running LED green
668     ENDIF
669 ENDIF
670 ’Wait 200ms
671 WT200
672 JP#RUNLED
673 EN
674
675
676 REM ************************ ETHERNET RECONNECTION
    ************************
677 #RECONN
678 IHC=>-3
```

```
CFC
ETH0=0
#KPALIVE
  #RCLP
    IF(ETH0=0)
      IHC=>-3
      IHC=130,194,171,188 <518>1;'syslog server
      WT150
    ENDIF
    ETH0=(_IHC2=-1)&(_IHC3=0);'udp master && arp success
  JP#RCLP,ETH0=0;'keep trying
  CW2;'disable copy-bit
  'MG{EC}"bec-coils-krb:INFO,ping"
  IF(_IHA2>0)&(@ABS[_IHA0-_IA0]>$FF);
    IHA=>-3
  ENDIF
  IF(_IHB2>0)&(@ABS[_IHB0-_IA0]>$FF);'require lab subnet
    IHB=>-3
  ENDIF
  WT60000
JP#KPALIVE
EN
```



(a)



(b)

Figure B.1: The custom circuit board layout for breaking-out the analog and digital I/O on the Galil. The circuit board is dual layer, with the top layer shown in (a) and the bottom layer shown in (b). The Galil connects via two high-density D-subminiature connectors (DA26 and DB-44) which are broken out into Molex connectors, connected to LED drivers or used to engage relays and monitor the relay state.

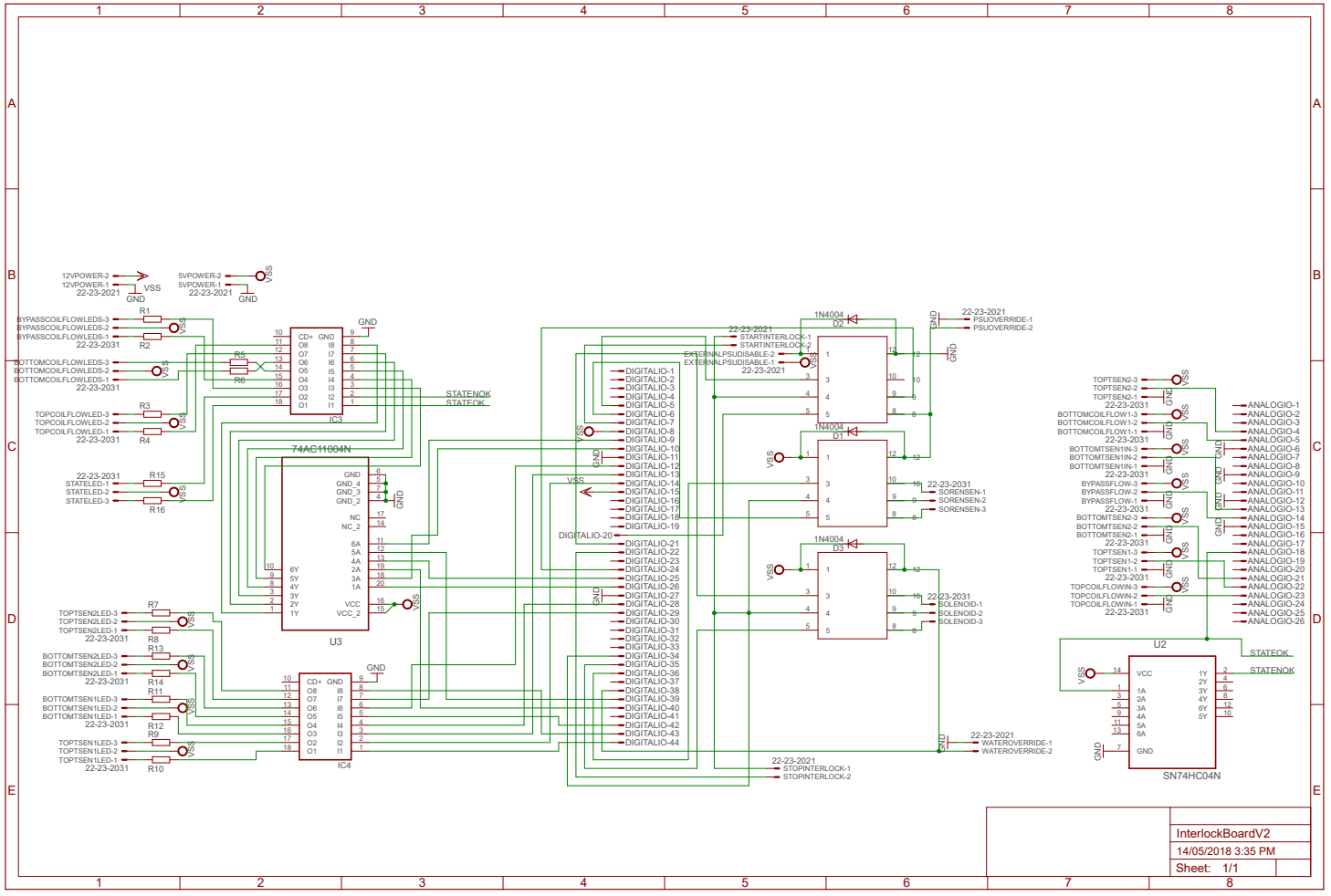


Figure B.2: The schematic diagram for the coil interlock circuit.

Appendix C

Supernova design

The circuit board layout and schematics for the Supernova.

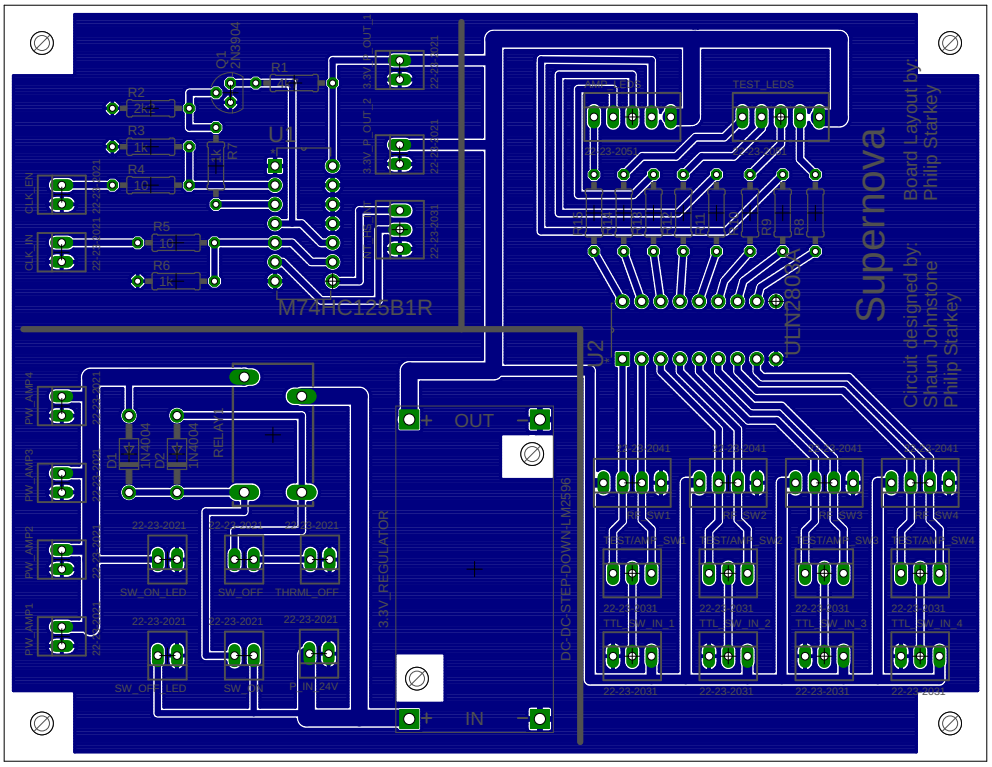
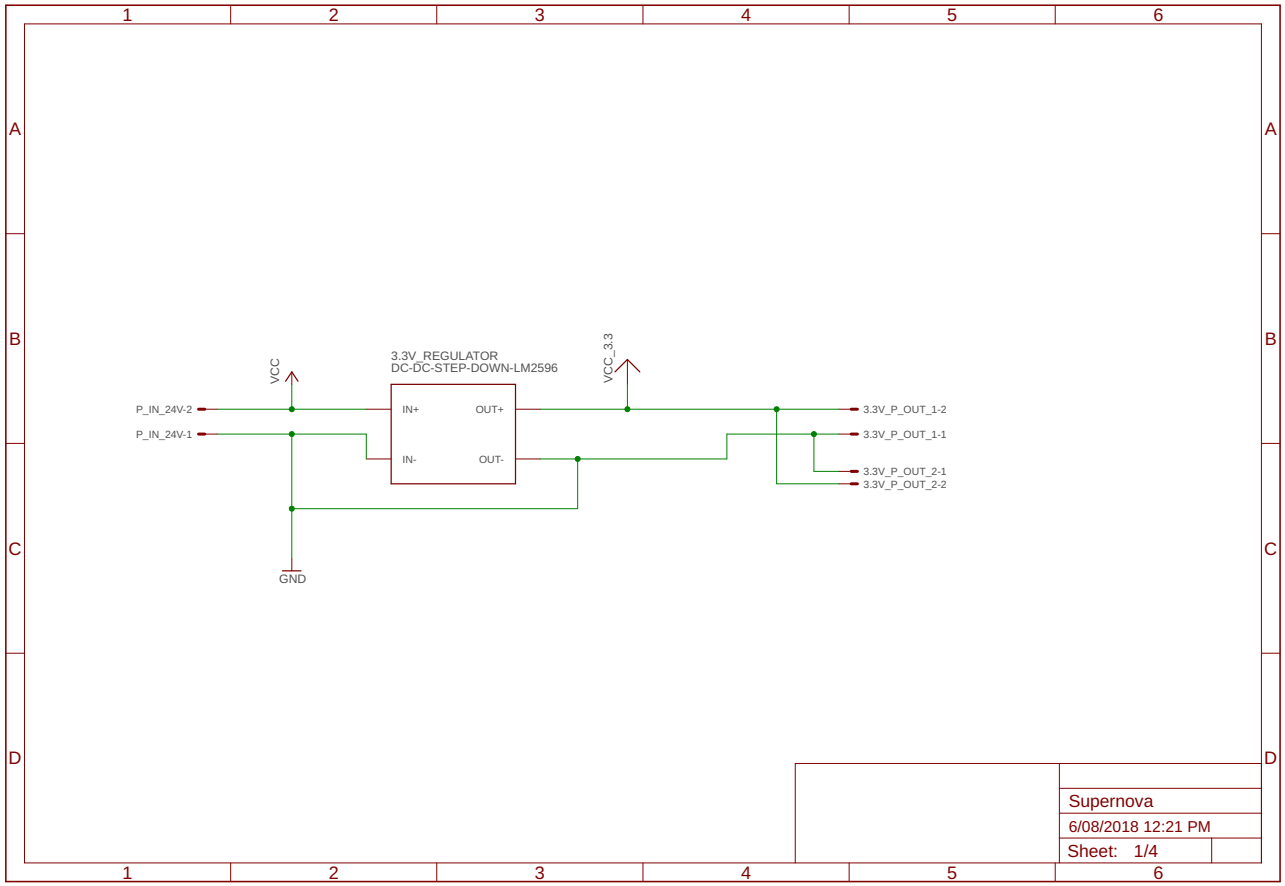
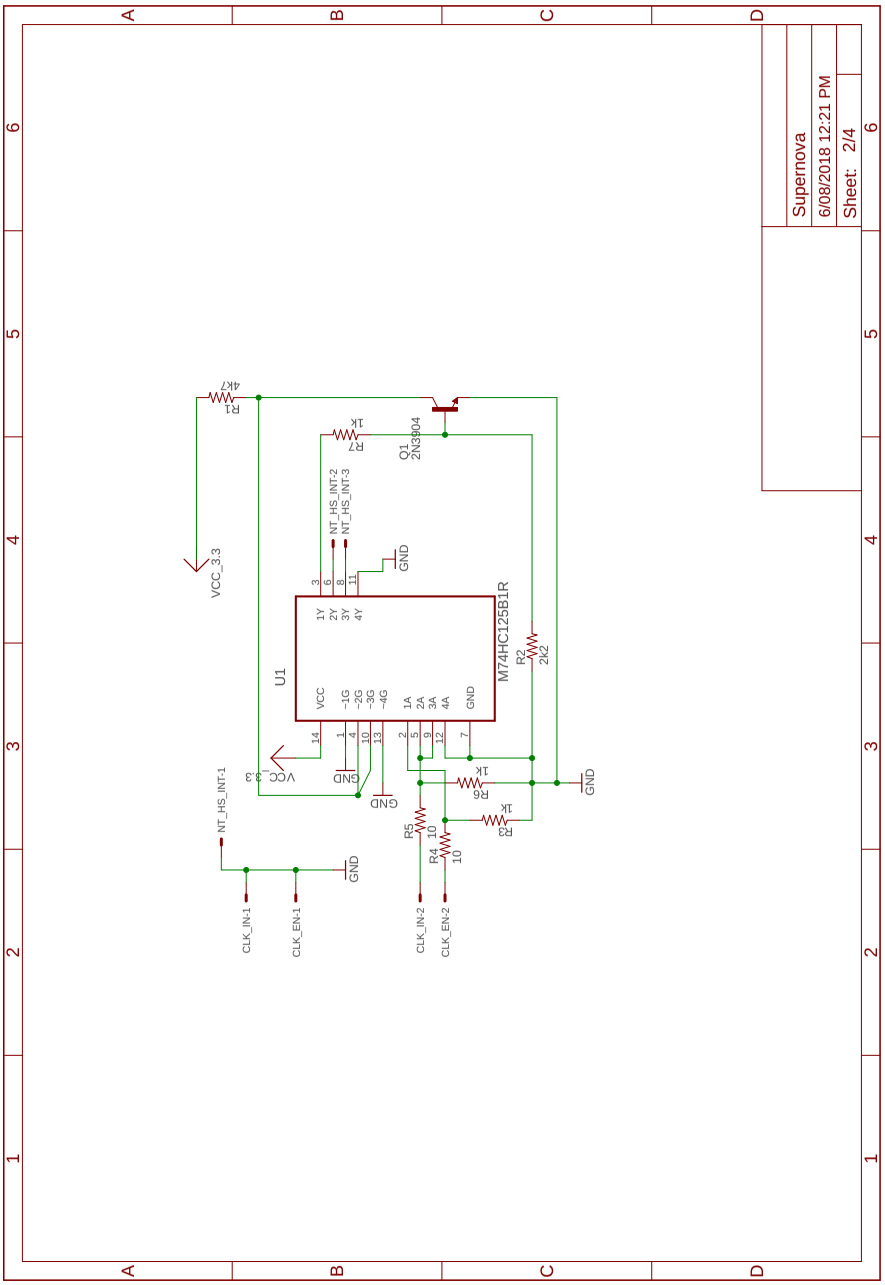
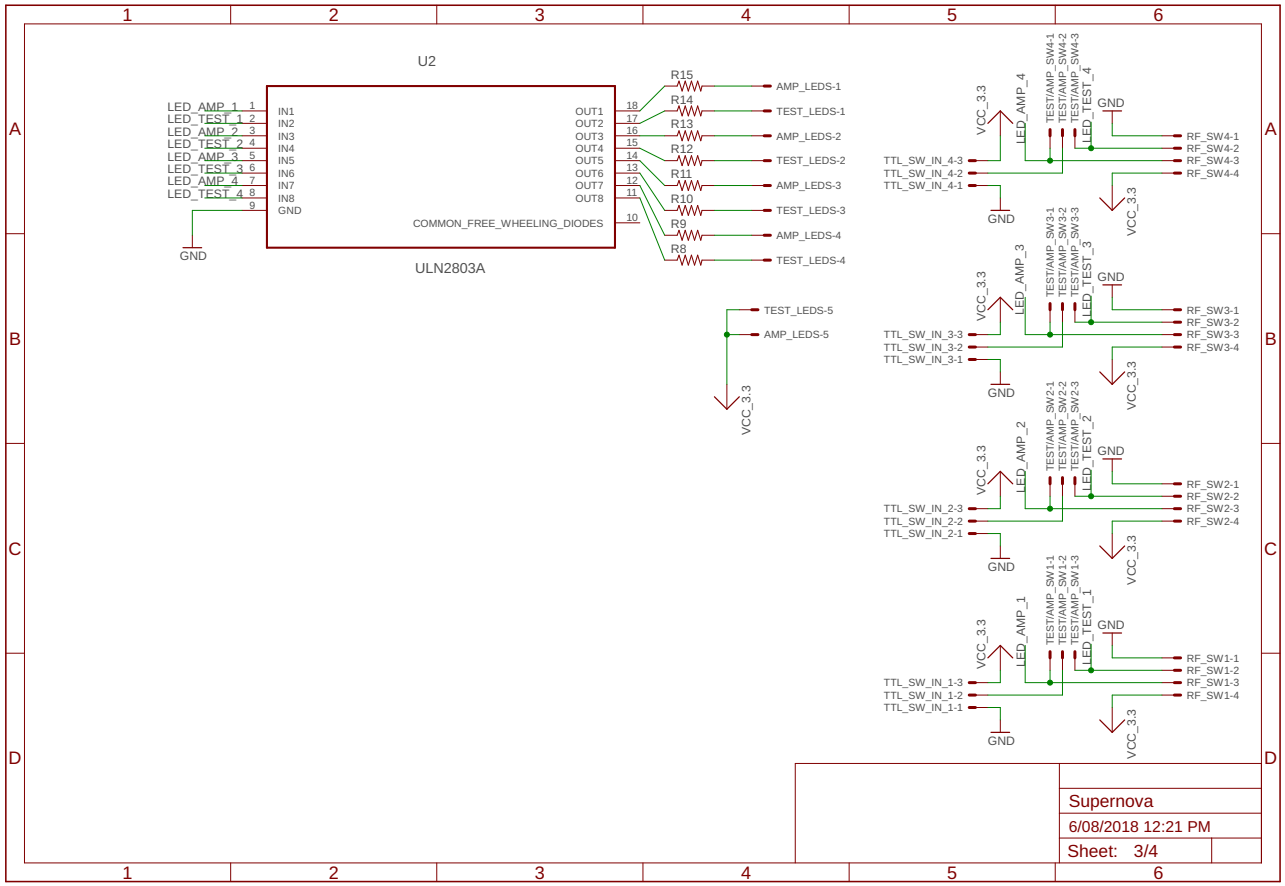
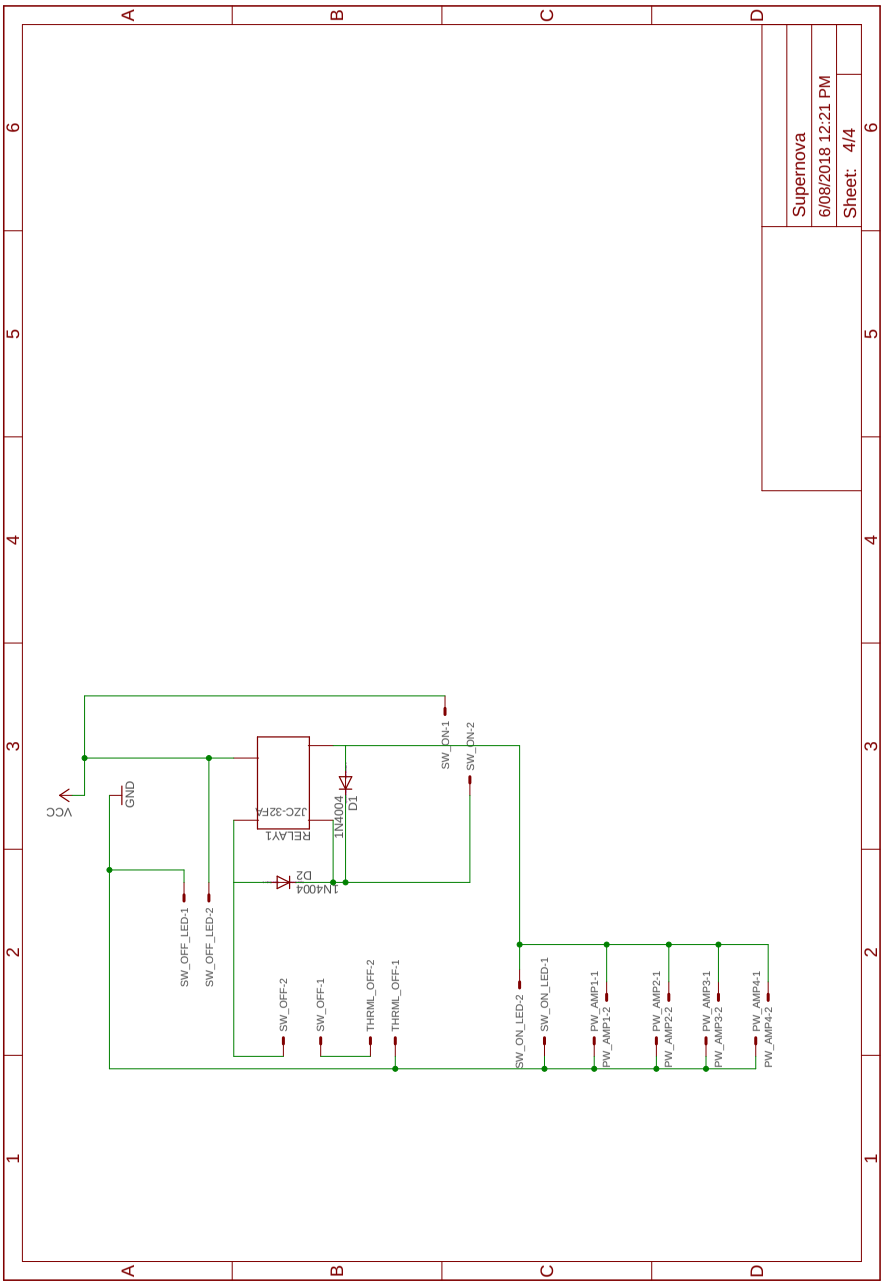


Figure C.1: Top left: The circuitry for the tri-state buffer used to connect the clocking signals to the Novatech DDS9 board. Bottom left: The circuitry for managing the power to the rf amplifiers. We also include a 3.3 V regulator here to supply power to the tri-state buffer, LEDs and rf switches. It is possible to also use this for powering the Novatech DDS9m board, however the power supply is noisier than a standard linear regulator and we have not tested whether this noise is successfully filtered by the Novatech board or if it affects the rf frequencies generated by the board. Right: The circuitry for connecting the front panel switches, rf switches and LEDs (see main body text, §3.4.1, for further details on the front panel).


1
2
3
4
5
6
A
B
C
D
3.3V_REGULATOR
DC-DC-STEP-DOWN-LM2596
VCC
VCC_3.3
P_IN_24V-2
P_IN_24V-1
IN+
IN-
OUT+
OUT-
3.3V_P_OUT_1-2
3.3V_P_OUT_1-1
3.3V_P_OUT_2-1
3.3V_P_OUT_2-2
GND
Supernova
6/08/2018 12:21 PM
Sheet: 1/4


A
B
C
D
1
2
3
4
5
6
Supernova
6/08/2018 12:21 PM
Sheet: 2/4
R1
4k7
R7
1K
Q1
2N3904
NT_HS_INT-2
NT_HS_INT-3
GND
VCC_3.3
U1
M74HC125B1R
R2
2k2
VCC
GND
R6
1K
R3
1K
R5
10
R4
10
NT_HS_INT-1
CLK_IN-1
CLK_EN-1
CLK_IN-2
CLK_EN-2


U2
ULN2803A
LED_AMP_1
LED_TEST_1
LED_AMP_2
LED_TEST_2
LED_AMP_3
LED_TEST_3
LED_AMP_4
LED_TEST_4
IN1
IN2
IN3
IN4
IN5
IN6
IN7
IN8
GND
OUT1
OUT2
OUT3
OUT4
OUT5
OUT6
OUT7
OUT8
COMMON_FREE_WHEELING_DIODES
R15
R14
R13
R12
R11
R10
R9
R8
AMP_LEDS-1
TEST_LEDS-1
AMP_LEDS-2
TEST_LEDS-2
AMP_LEDS-3
TEST_LEDS-3
AMP_LEDS-4
TEST_LEDS-4
TEST_LEDS-5
AMP_LEDS-5
VCC_3.3
TTL_SW_IN_4-3
TTL_SW_IN_4-2
TTL_SW_IN_4-1
TTL_SW_IN_3-3
TTL_SW_IN_3-2
TTL_SW_IN_3-1
TTL_SW_IN_2-3
TTL_SW_IN_2-2
TTL_SW_IN_2-1
TTL_SW_IN_1-3
TTL_SW_IN_1-2
TTL_SW_IN_1-1
TEST/AMP_SW4-1
TEST/AMP_SW4-2
TEST/AMP_SW4-3
TEST/AMP_SW3-1
TEST/AMP_SW3-2
TEST/AMP_SW3-3
TEST/AMP_SW2-1
TEST/AMP_SW2-2
TEST/AMP_SW2-3
TEST/AMP_SW1-1
TEST/AMP_SW1-2
TEST/AMP_SW1-3
RF_SW4-1
RF_SW4-2
RF_SW4-3
RF_SW4-4
RF_SW3-1
RF_SW3-2
RF_SW3-3
RF_SW3-4
RF_SW2-1
RF_SW2-2
RF_SW2-3
RF_SW2-4
RF_SW1-1
RF_SW1-2
RF_SW1-3
RF_SW1-4
Supernova
6/08/2018 12:21 PM
Sheet: 3/4


A
B
C
D
1
2
3
4
5
6
Supernova
6/08/2018 12:21 PM
Sheet: 4/4
VCC
GND
SW_DN-1
SW_DN-2
RELAY1
JZC-32FA
1N4004
D1
1N4004
D2
SW_OFF_LED-1
SW_OFF_LED-2
SW_OFF-2
SW_OFF-1
THRML_OFF-2
THRML_OFF-1
SW_ON_LED-2
SW_ON_LED-1
PW_AMP1-1
PW_AMP1-2
PW_AMP2-1
PW_AMP2-2
PW_AMP3-1
PW_AMP3-2
PW_AMP4-1
PW_AMP4-2

Appendix D

K-Rb experiment details

This appendix provides a copy of the experiment logic and runmanager globals used in our dual-species apparatus. We also include a graphical representation of the hierarchy of device control to emphasise the scale of the hardware we must control.

D.1 Globals

Here we provide the set of experiment parameters (global variables) defined in runmanager. This emphasises the complexity of our experiments, and the number of parameters we have control over and must optimise in order to make an ultracold atom cloud in a dual-species apparatus.

Name (units)	Value	Name (units)	Value
kick_off_field_delay (s)	20e-3	mw_transfer_2_1_bias_z (A)	0
kick_off_magnetic_atoms (Bool)	False	mw_transfer_2_1_centre_freq (MHz)	1.05
kick_off_magnetic_atoms_time (s)	20e-3	mw_transfer_2_1_range (MHz)	0.2
kick_off_quad_current (A)	50	mw_transfer_2_1_rate (MHz/s)	200
mw_transfer_2_1_bias_y (A)	0	transfer_to_2_1	s0951_transfer_to_2_1

Table D.1: The globals in the group "11>21 Microwave Transfer"

Name (units)	Value	Name (units)	Value
blow_2_1 (Bool)	True	mw_transfer_1_0_centre_freq (MHz)	0.527
clock_mw_detuning (MHz)	2e-3	mw_transfer_1_0_range (MHz)	0.5
clock_pi (Bool)	False	mw_transfer_1_0_rate (MHz/s)	100
clock_pi_time (s)	2e-5	transfer_to_1_0	s0952_transfer_to_1_0

Table D.2: The globals in the group "21>10 Microwave Transfer"

Name (units)	Value	Name (units)	Value
blow_2_2 (Bool)	s0952_blow_away_22	mw_transfer_bias_z (A)	0
microwave_transfer (Bool)	s0950_microwave_transfer	mw_transfer_centre_freq (MHz)	1.65
mw_transfer_after_time (s)	0	mw_transfer_field_time (s)	8e-3
mw_transfer_bias_x (A)	0	mw_transfer_range (MHz)	0.35
mw_transfer_bias_y (A)	0	mw_transfer_rate (MHz/s)	100

Table D.3: The globals in the group "22>11 Microwave Transfer"

Name (units)	Value	Name (units)	Value
fit_clip	4	plot_measurement (Measurement)	('n_fluoro')
fit_od0 (Bool)	True	plot_measurement_k	('n_k_roi_2', 'n_k_sum', 'OD_max_roi_2', 'wx_k_roi_2', 'wy_k_roi_2', 'x_k_roi_2', 'y_k_roi_2',)
fit_rois (-)	('roi0',)	plot_measurement_rb (8)	('n_roi_0','n_sum_roi0','OD_max_roi_0','x_roi_0','y_roi_0','wx_roi_0','wy_roi_0')
fit_rois_k	('roi2',)	repeats	[0]*num_repeats if num_repeats > 1 else 0
fit_rois_rb	('roi0',)	waveplate (Degrees)	190
no_sequences_to_analyse (Sequences)	1	x_axis (Global)	'y_roi_0'
num_repeats	1	y_axis (Global)	'None'

Table D.4: The globals in the group "Analysis"

Name (units)	Value	Name (units)	Value
after_anti_gravity_time (s)	0.5	anti_gravity_current (A)	20
anti_gravity (Bool)	False	anti_gravity_ramp_time (s)	0.1

Table D.5: The globals in the group "Anti gravity"

Name (units)	Value	Name (units)	Value
bottom_camera_pixel_size (m)	0.535e-6	side_k41_saturation_intensity	2858.0
bottom_rb87_saturation_intensity (adu)	3800	side_rb87_saturation_intensity (adu)	1800
central_chamber_pixel_size (m)	5.916e-6	square_cell_side_pixel_size (m)	3.06e-6

Table D.6: The globals in the group "Cameras"

Name (units)	Value	Name (units)	Value
atomlaser (Bool)	False	dipole_recompress_amplitude (arb)	100
atomlaser_amp (arb)	42	dipole_recompression (Bool)	False
atomlaser_time (s)	20e-3	dipole_recompression_duration (s)	10e-3
central_dipole_evap_type ("linear" or "exp")	'exp'	dipole_trap_evaporation (Bool)	s09_dipole_trap_evaporation
cross_dipole_evap_trunc (Arb)	0.99	dipole_trap_evaporation_ramp_lens_position (Bool)	False
cross_dipole_evaporation_end_amplitude (Arb)	0.1	dipole_trap_evaporation_ramp_lens_position_duration (s)	0
dipole_evap_central_bias_z (A)	0.01	dipole_trap_evaporation_ramp_lens_position_final (distance)	0
dipole_evap_trunc (arb)	0.98	dipole_trap_evaporation_time_constant (s)	3
dipole_evaporation_duration (sec)	5.5	evaporated_dipole_hold_time (s)	0.5
dipole_evaporation_end_amplitude (arb)	130	magnetic_fall (Bool)	False
dipole_pre_evaporation_amplitude (arb)	250	magnetic_fall_time (s)	5e-3
dipole_pre_evaporation_duration (s)	0.8	quad_turnoff_time (s)	4.5

Table D.7: The globals in the group "Dipole Trap Evaporation"

Name (units)	Value	Name (units)	Value
MT_load_pure_dipole_bias_time (s)	0.1	pgc_dipole_trap (Bool)	False
MT_load_pure_dipole_decompress_time (s)	0.1	pure_dipole_amplitude (arb)	dipole_trap_power
dipole_trap_aom_frequency (MHz)	99	pure_dipole_central_bias_x (A)	0
dipole_trap_power (Arb)	1023	pure_dipole_central_bias_y (A)	0
load_dipole_trap (Bool)	s08_load_dipole_trap	pure_dipole_central_bias_z (A)	0
load_pure_dipole_trap (Bool)	s0900_load_pure_dipole	pure_dipole_hold_time (s)	.1

Table D.8: The globals in the group "Dipole Trap Load"

Name (units)	Value	Name (units)	Value
expand_in_hg (Bool)	True	hg_expansion_kick_time (s)	5e-3
expansion_coil_turn_time (s)	3e-3	hg_expansion_side_bias (A)	5.037
green_sheet_amp (arb)	0.445	hg_expansion_time (s)	20e-3
hg_expansion_bottom_quad_current (A)	35	hg_expansion_top_quad_current (A)	0
hg_expansion_end_bias (A)	1.6		

Table D.9: The globals in the group "Expand in HG"

Name (units)	Value	Name (units)	Value
green_alignment_kill_after_time (s)	10e-3	green_hg_turn_on_time (s)	0.05
green_alignment_kill_amp (arb)	0.04	hg_hold_time (s)	0
green_alignment_kill_time (s)	0	kill_for_green_alignment (Bool)	False
green_hg_amplitude (arb)	0.445	load_green_hg (Bool)	True
green_hg_load_red_off_time (s)	0.3		

Table D.10: The globals in the group "Green HG"

Name (units)	Value	Name (units)	Value
flat_trap_bottom_imaging_amp (arb)	0	rb_square_fluoro_amplitude (Arb)	700
flat_trap_dither_amplitude (MHz)	0	rb_square_fluoro_frequency (MHz)	rb_science_bottom_imaging_ frequency
flat_trap_dither_freq (Hz)	2000	rb_square_imaging_repump_ amplitude (arb)	700
flat_trap_turn_off_delay_ bottom (s)	0	rb_square_imaging_repump_ frequency (MHz)	85
iXon_exposure_time (s)	1e-3	science_bottom_focus_position (microsteps)	194000
iXon_interframe_time (s)	50e-3	science_bottom_imaging_field (A)	0
quad_turn_off_delay_bottom (s)	0	science_bottom_imaging_pulse_ time (s)	0.1e-3
rb_science_bottom_imaging_ amplitude (Arb)	0.12	science_drop_time_k (s)	1e-3
rb_science_bottom_imaging_ frequency (MHz)	91	science_drop_time_rb (s)	0.5e-3
rb_science_fluoro_imaging_time (s)	0.03e-3	square_cell_fluoro (Bool)	False
rb_science_imaging_repump (Bool)	False	square_cell_fluoro_k (Bool)	False

Table D.11: The globals in the group “Imaging - Bottom Square Cell”

Name (units)	Value	Name (units)	Value
SG_current (A)	60	k_recapture_imaging_time (s)	10e-3
SG_imaging (Bool)	False	k_repump_fluoro_amplitude	700
central_drop_time_k (s)	0.5e-3	k_repump_fluoro_frequency (MHz)	104.5
central_drop_time_rb (s)	1e-3	k_repump_recapture_imaging_ amplitude (Arb)	700
central_image_k_fluoro	s11_central_image_k_fluoro	k_repump_recapture_imaging_ frequency (MHz)	102.1
central_image_k_recapture (Bool)	s11_central_image_k_recapture	k_saturation_intensity	2858.0
central_image_rb_fluoro	s11_central_image_rb_fluoro	k_trap_fluoro_amplitude	773
central_imaging_bias_x (A)	0	k_trap_fluoro_frequency (MHz)	89.8
central_imaging_bias_y (A)	1.3	k_trap_recapture_imaging_ amplitude (Arb)	773
central_imaging_bias_z (A)	0.061	k_trap_recapture_imaging_ frequency (MHz)	78.4+k_offset_frequency
dipole_image_delay (s)	0	rb_fluoro_amplitude (Arb)	700
drop_time_general (s)	1e-3	rb_fluoro_frequency (MHz)	rb_imaging_frequency
imaging_pulse_time (s)	.1e-3	rb_fluoro_imaging_time (s)	5e-3
interframe_time (s)	200e-3	rb_imaging_amplitude (Arb)	0.25
k_fluoro_imaging_time (s)	5e-3	rb_imaging_frequency (MHz)	91.15
k_imaging_MOT_repump (Bool)	False	rb_imaging_repump (Bool)	False
k_imaging_amplitude (Arb.)	850	rb_imaging_repump_amplitude (Arb)	700
k_imaging_frequency (MHz)	94.5	rb_imaging_repump_frequency (MHz)	85
k_imaging_repump (Bool)	False	rb_saturation_intensity	2858.0
k_imaging_repump_amplitude (Arb)	600	red_camera_exposure_time (s)	1e-3
k_imaging_repump_frequency (MHz)	107.5	side_imaging_expand_in_beam_1 (Bool)	False
k_recapture_imaging_MOT_load_ time (s)	5e-3	side_imaging_expand_in_beam_2 (Bool)	False
k_recapture_imaging_quad (A)	20	side_imaging_expanded_power (arb)	400

Table D.12: The globals in the group “Imaging - Central”

Name (units)	Value	Name (units)	Value
absorption_image_k (Bool)	[not x for x in image_order] if s11__absorption_image_both_species else s11__absorption_image_k	drop_time_rb (s)	central_drop_time_rb if central_imaging else (science_side_drop_time_rb if square_cell_side_imaging else (science_drop_time_rb if square_cell_bottom_imaging else 0))
absorption_image_rb (Bool)	image_order if s11__absorption_image_both_species else s11__absorption_image_rb	image_order (True=Rb, False=K, 0=not imaging both species)	(([True]*len(drop_time_rb) if isinstance(drop_time_rb, ndarray) or isinstance(drop_time_rb, list) else [True]) + ([False]*len(drop_time_k) if isinstance(drop_time_k, ndarray) or isinstance(drop_time_k, list) else [False])) if s11__absorption_image_both_species else 0
central_imaging (Bool)	s11__central_absorption_image	square_cell_bottom_imaging (Bool)	s11__square_cell_bottom_imaging
drop_time (s)	[drop_time_rb[i] if x and (isinstance(drop_time_rb, ndarray) or isinstance(drop_time_rb, list)) else (drop_time_rb if x else (drop_time_k[i-len(drop_time_rb)] if (isinstance(drop_time_rb, ndarray) or isinstance(drop_time_rb, list)) and (isinstance(drop_time_k, ndarray) or isinstance(drop_time_k, list)) else (drop_time_k[i-1] if isinstance(drop_time_k, ndarray) or isinstance(drop_time_k, list) else drop_time_k))) for i, x in enumerate(image_order)] if image_order else (drop_time_k if absorption_image_k == True else (drop_time_rb if absorption_image_rb == True else drop_time_general))	square_cell_side_imaging (Bool)	s11__square_cell_side_imaging
drop_time_k (s)	central_drop_time_k if central_imaging else (science_side_drop_time_k if square_cell_side_imaging else (science_drop_time_k if square_cell_bottom_imaging else 0))		

Table D.13: The globals in the group “Imaging - Common”

Name (units)	Value	Name (units)	Value
flat_trap_drop_amplitude (arb)	0.5	science_side_imaging_boost_flat (Bool)	False
flat_trap_turn_off_delay (s)	0	science_side_imaging_expand_in_cross (Bool)	False
imaging_expansion_beam_amplitude (arb)	800	science_side_imaging_expand_in_trap (Bool)	False
imaging_expansion_beam_ramp_time (s)	1e-3	science_side_imaging_field (A)	0
imaging_expansion_cross_beam_amplitude (arb)	0.99	science_side_imaging_pulse_time (s)	0.1e-3
rb_science_side_imaging_amplitude (arb)	0.7	side_imaging_bias (A)	5
rb_science_side_imaging_frequency (MHz)	94.5	side_imaging_hold_in_quad (Bool)	False
science_side_drop_time_k (s)	1e-3	tof_anti_gravity (Bool)	False
science_side_drop_time_rb (s)	18e-3		

Table D.14: The globals in the group “Imaging - Side Square Cell”

Name (units)	Value	Name (units)	Value
k_compressed_MOT (Bool)	s021_k_compressed_MOT	k_compressed_MOT_time (s)	60e-3
k_compressed_MOT_bias_x (A)	0.9	k_repump_compressed_MOT_ amplitude (Arb)	1023
k_compressed_MOT_bias_y (A)	0.001	k_repump_compressed_MOT_ frequency (MHz)	104
k_compressed_MOT_bias_z (A)	1.15	k_trap_compressed_MOT_ amplitude (Arb)	811
k_compressed_MOT_current (A)	27	k_trap_compressed_MOT_ frequency (MHz)	82
k_compressed_MOT_hold_time (s)	0		

Table D.15: The globals in the group "K CMOT"

Name (units)	Value	Name (units)	Value
k_compressed_MOT_cooling (Bool)	s022_k_compressed_MOT_cooling	k_repump_compressed_MOT_ cooling_frequency (MHz)	103.3
k_compressed_MOT_cooling_time (s)	20e-3	k_trap_compressed_MOT_cooling_ amplitude (Arb)	773
k_repump_compressed_MOT_ cooling_amplitude (Arb)	250	k_trap_compressed_MOT_cooling_ frequency (MHz)	84.3

Table D.16: The globals in the group "K CMOT Cooling"

Name (units)	Value	Name (units)	Value
central_MOT_hold_time (s)	0	k_central_MOT_load_bias_z (A)	.69
k_central_MOT_load_bias_x (A)	0	k_central_MOT_load_current (A)	10.6
k_central_MOT_load_bias_y (A)	0		

Table D.17: The globals in the group "K MOT - Central"

Name (units)	Value	Name (units)	Value
k_MT_compressed_bias_x (A)	0	k_MT_compression_time (s)	100e-3
k_MT_compressed_bias_y (A)	0.01	k_MT_hold_time (s)	0 if k_spin_purification else 2
k_MT_compressed_bias_z (A)	0	k_initial_MT (Bool)	s031_k_initial_MT
k_MT_compressed_quad (A)	135		

Table D.18: The globals in the group "K MT"

Name (units)	Value	Name (units)	Value
k_MT_decompressed_hold_time (s)	0	k_MT_decompression_time (s)	0.6
k_MT_decompressed_quad (A)	15	k_spin_purification (Bool)	s032_k_spin_purification

Table D.19: The globals in the group "K MT Decompress"

Name (units)	Value	Name (units)	Value
k_optical_pumping (Bool)	s030_k_optical_pumping	k_optical_pumping_end_repump_ duration (s)	60e-6
k_optical_pumping_amplitude (Arb)	489	k_optical_pumping_frequency (MHz)	88.8
k_optical_pumping_bias_x (A)	0	k_optical_pumping_initial_ repump_duration (s)	0
k_optical_pumping_bias_y (A)	1	k_optical_pumping_repump_ amplitude (Arb)	700
k_optical_pumping_bias_z (A)	0.07	k_optical_pumping_repump_ frequency (MHz)	107.1
k_optical_pumping_delay (s)	1.3e-3	k_optical_pumping_time (s)	150e-6

Table D.20: The globals in the group "K Optical Pumping"

Name (units)	Value	Name (units)	Value
k_raman_global_amp (Arb)	1023	k_raman_red_freq (MHz)	65
k_raman_global_freq (MHz)	100	k_raman_transfer_after_time (s)	0
k_raman_red_amp (Arb)	673		

Table D.21: The globals in the group “K Raman - Common”

Name (units)	Value	Name (units)	Value
k_raman_22_to_10 (Bool)	s0950_raman_transfer and s095k1_transfer_22_to_10	k_raman_22_to_10_blue_freq (MHz)	k_raman_22_to_10_level_separation_freq/2.-k_raman_red_freq
k_raman_22_to_10_bias_x (A)	0	k_raman_22_to_10_duration (s)	400e-6
k_raman_22_to_10_bias_y (A)	0	k_raman_22_to_10_field_transfer_time	0e-3
k_raman_22_to_10_bias_z (A)	0.5	k_raman_22_to_10_level_separation_freq (MHz)	254+4.801
k_raman_22_to_10_blue_amp (Arb)	440	k_raman_22_to_10_red_amp (Arb)	673

Table D.22: The globals in the group “K Raman |2,2> to |1,0>”

Name (units)	Value	Name (units)	Value
k_blow_away_22 (Bool)	s095k2_blow_away_22	k_transfer_freq_step (Hz/sample)	k_transfer_range/k_transfer_num_samples*1e6
k_rf_amp (arb)	.99	k_transfer_num_samples (samples)	k_transfer_time*k_transfer_sample_rate
k_to_11 (Bool)	s095k1_transfer_22_to_11	k_transfer_pi_pulse_time (s)	0
k_transfer__adiabatic_passage (Bool)	not k_transfer__pi_pulse	k_transfer_range (MHz)	0.02
k_transfer__pi_pulse (Bool)	False	k_transfer_rate (MHz/s)	0.05
k_transfer_bias_y (A)	0	k_transfer_sample_rate (Hz)	3e4
k_transfer_bias_z (A)	0.9	k_transfer_time (s)	1.0*k_transfer_range/k_transfer_rate
k_transfer_centre_freq (MHz)	14.04	k_transfer_trunc (detuned_ MHz)	0

Table D.23: The globals in the group “K to |1,1>”

Name (units)	Value	Name (units)	Value
k_22_10_rf1_amp (Arb)	.99	k_22_10_transfer_num_samples (samples)	k_22_10_transfer_time*k_22_10_transfer_sample_rate
k_22_10_rf2_amp (Arb)	.99	k_22_10_transfer_range (MHz)	0.3
k_22_10_rf2_freq (MHz)	31	k_22_10_transfer_rate (MHz/s)	6
k_22_10_transfer_bias_y (A)	0	k_22_10_transfer_sample_rate (Hz)	3e4
k_22_10_transfer_bias_z (A)	30/7.514	k_22_10_transfer_time (s)	1.0*k_22_10_transfer_range/k_22_10_transfer_rate
k_22_10_transfer_centre_freq (MHz)	11+1.7	k_22_10_transfer_trunc (detuned_MHz)	0
k_22_10_transfer_freq_step (Hz/sample)	k_22_10_transfer_range/k_22_10_transfer_num_samples*1e6	k_22_to_10 (Bool)	s095k1_transfer_22_to_10

Table D.24: The globals in the group “K |2,2> to |1,0>”

Name (units)	Value	Name (units)	Value
diole_kick_amplitude (MHz)	0.5	kick_for_slosh (Bool)	False
dipole_kick_duration (s)	0.05	kick_for_slosh_current (A)	0.2
dipole_kick_slosh_time (s)	0		

Table D.25: The globals in the group “Kicked Slosh measurement”

Name (units)	Value	Name (units)	Value
k_central_MOT_load_time (s)	20	k_source_current_east (A)	4.33
k_offset_frequency (MHz)	1.7	k_source_current_max (A)	21.5
k_push_amplitude (Arb.)	160	k_source_current_sum (A)	k_source_current_east+k_source_current_west+k_source_current_top+k_source_current_bottom
k_push_frequency (MHz)	98.3	k_source_current_top (A)	5.7
k_push_repump_amplitude (Arb)	890	k_source_current_west (A)	5.1
k_push_repump_frequency (MHz)	105	k_trap_load_amplitude (Arb.)	750
k_repump_load_amplitude (Arb.)	1023	k_trap_load_frequency (MHz)	81.7
k_repump_load_frequency (MHz)	102.3	use_k (Bool)	s01_use_k
k_source_current_bottom (A)	4.6		

Table D.26: The globals in the group "MOT - K"

Name (units)	Value	Name (units)	Value
rb_central_MOT_load_time (seconds)	0.3	rb_push_duration (seconds)	0.6e-3
rb_central_MOT_repump_amplitude (Arb)	700	rb_push_frequency (MHz)	95
rb_central_MOT_repump_frequency (MHz)	80	rb_source_MOT_load_time (seconds)	12e-3
rb_central_MOT_trap_amplitude (Arb)	700	rb_source_current (A)	7.3
rb_central_MOT_trap_frequency (MHz)	79	rb_source_repump_amplitude (Arb)	750
rb_laser_offset (MHz)	0	rb_source_repump_frequency (MHz)	85.3
rb_op_mot_displacer_amplitude (Arb)	0	rb_source_trap_amplitude (Arb)	670
rb_op_mot_displacer_frequency (MHz)	84	rb_source_trap_frequency (MHz)	84
rb_push_amplitude (Arb)	0.7	use_rb (Bool)	s01_use_rb

Table D.27: The globals in the group "MOT - Rb"

Name (units)	Value	Name (units)	Value
MT_compress (Bool)	s05_MT_compress	central_MT_compressed_bias_y (A)	0.22
MT_compressed_hold_time (s)	0.5 if microwave_evaporation else 1	central_MT_compressed_bias_z (A)	0.14
MT_compression_time (s)	0.5	central_MT_compressed_quad (A)	135
central_MT_compressed_bias_x (A)	0.32		

Table D.28: The globals in the group "MT Compress"

Name (units)	Value	Name (units)	Value
MT_decompress (Bool)	s07_MT_decompress	blow_away_rb_time (s)	(mw_evap_stop-blow_away_mw_end)/blow_away_rb_rate
MT_decompress_evap (Bool)	s07_MT_decompress_evap	blow_away_rb_use_light (Bool)	False
MT_decompression_evap_time (s)	0.4	central_MT_decompressed_bias_x (A)	central_MT_compressed_bias_x
MT_decompression_hold_time (s)	50e-3	central_MT_decompressed_bias_y (A)	central_MT_compressed_bias_y
MT_decompression_time (seconds)	MT_decompression_evap_time*2	central_MT_decompressed_bias_z (A)	central_MT_compressed_bias_z
blow_away_mw_end (MHz)	-20	central_MT_decompressed_quad (A)	10
blow_away_rb (Bool)	s070_blow_away_rb	mt_extra_decompress_hold_microwaves_on (Bool)	True
blow_away_rb_rate (MHz/s)	14	mw_evap_decompress_stop (MHz)	mw_evap_stop

Table D.29: The globals in the group "MT Decompress"

Name (units)	Value	Name (units)	Value
mw_evap_rotated_amp (arb)	0.3	mw_evap_rotated_time (s)	0.5
mw_evap_rotated_end (MHz)	7	mw_evap_rotated_time_constant (s)	0.4
mw_evap_rotated_start (MHz)	10	rotated_microwave_evaporation (Bool)	False

Table D.30: The globals in the group "MW evap rotated"

Name (units)	Value	Name (units)	Value
bottom_feshbach_coils_saturation (V)	10	central_quad_field_calibration (G/cm)	0.8
bottom_feshbach_coils_shift (A)	0	central_quad_saturation (V)	10
bottom_feshbach_coils_slope (A/V)	2*feshbach_coils_slope	central_quad_shift (A)	-1.991611E-01
central_bias_x_coil_saturation (V)	2.88	central_quad_slope (A/V)	15.82738
central_bias_x_coil_shift (A)	-1.648352E-04	feshbach_coils_saturation (V)	10
central_bias_x_coil_slope (A/V)	0.5231099	feshbach_coils_shift (A)	0
central_bias_y_coil_saturation (V)	4	feshbach_coils_slope (A/V)	2.08
central_bias_y_coil_shift (A)	-1.143956E-02	k_2D_MOT_coil_slope (A/V)	0.9077
central_bias_y_coil_slope (A/V)	0.5280549	rb_source_MOT_saturation (V)	10
central_bias_z_coil_saturation (V)	9.23	rb_source_MOT_shift (A)	-0.0771
central_bias_z_coil_shift (A)	3.368e-15	rb_source_MOT_slope (A/V)	1.4392
central_bias_z_coil_slope (A/V)	0.52		

Table D.31: The globals in the group "Magneatos"

Name (units)	Value	Name (units)	Value
central_MT_capture_bias_x (A)	0	central_MT_capture_quad (A)	37
central_MT_capture_bias_y (A)	0	magnetic_trap (Bool)	s04_magnetic_trap
central_MT_capture_bias_z (A)	0	magnetic_trap_hold_time (s)	0 if MT_compress else 1

Table D.32: The globals in the group "Magnetic Trap"

Name (units)	Value	Name (units)	Value
IQ_amp (Arb)	0.3	mw_evap_mid (MHz detuned)	40
evap_rate (MHz/s)	14	mw_evap_points (points)	1000
evaporated_MT_hold_time (s)	0	mw_evap_start (MHz detuned)	53
microwave_LO_freq (MHz)	7034	mw_evap_stop (MHz detuned)	3.4
microwave_evaporation (Bool)	s06_microwave_evaporation	mw_evap_time (s)	(mw_evap_mid-mw_evap_stop)/evap_rate
mw_evap_initial_rate (MHz/s)	10	mw_evap_time_constant (s)	1.0
mw_evap_initial_time (s)	(mw_evap_start-mw_evap_mid)/mw_evap_initial_rate	mw_evap_type	'linear'

Table D.33: The globals in the group "Microwave Evaporation"

Name (units)	Value	Name (units)	Value
transport_aom_A (W)	1.969	transport_lens_current_cal (A/V)	0.0193
transport_aom_c (W)	1.901	transport_lens_fit_a (a)	-0.821729108941288
transport_aom_f (Arb)	0.527	transport_lens_fit_b (b)	-15.390108776081
transport_aom_phase (rad)	3.262	transport_lens_fit_c (c)	0.628028016961235
transport_lens_I_max (A)	0.2		

Table D.34: The globals in the group "OptotuneLens"

Name (units)	Value	Name (units)	Value
PGC_bias_x (A)	0	rb_pgc_repump_amplitude (Arb)	600
PGC_bias_y (A)	0.1	rb_pgc_repump_frequency (MHz)	80
PGC_bias_z (A)	0.07	rb_pgc_time (s)	4.5e-3
PGC_quad (A)	0	rb_pgc_time_constant (s)	0.5e-3
pgc_field_turn_time (s)	1e-3	rb_pgc_trap_amplitude (Arb.)	600
rb_PGC (Bool)	s02_rb_PGC	rb_pgc_trap_frequency (MHz)	58

Table D.35: The globals in the group "PGC"

Name (units)	Value	Name (units)	Value
parametric_heating (Bool)	False	trap_heating_frequency (Hz)	170
trap_heating_amplitude (fraction)	0.1	trap_heating_time (s)	0.5

Table D.36: The globals in the group "Parametric Heating"

Name (units)	Value	Name (units)	Value
post_mw_transfer_evaporation (Bool)	True	post_post_mw_transfer_evaporation_hold_time (s)	0
post_mw_transfer_evaporation_duration (s)	4	square_cell_post_mw_evaporation_end_amplitude_2D (arb)	0.17
post_mw_transfer_evaporation_time_constant (s)	1	square_cell_post_mw_evaporation_end_amplitude_cross (arb)	0.265
post_mw_transfer_evaporation_trunc (fraction)	1	square_cell_post_mw_evaporation_end_amplitude_transport (arb)	48

Table D.37: The globals in the group "Post mw transfer evap"

Name (units)	Value	Name (units)	Value
rfblaster_0_delay (s)	880e-6		

Table D.38: The globals in the group "RFBlasters"

Name (units)	Value	Name (units)	Value
cross_trap_off_ramp_duration (s)	0.35	sine_ramp_off_cross_trap (Bool)	False
flat_trap_hold_time (s)	0		

Table D.39: The globals in the group "Ramp off cross"

Name (units)	Value	Name (units)	Value
rb_MOT_compress_time (s)	0.1	rb_central_MOT_compress_current (A)	18.6
rb_central_MOT_compress_bias_x (A)	0	rb_compressed_MOT_power_ramp_time (s)	1e-3
rb_central_MOT_compress_bias_y (A)	0.0001	rb_compressed_MOT_trap_amplitude (arb)	600
rb_central_MOT_compress_bias_z (A)	0.165		

Table D.40: The globals in the group "Rb CMOT"

Name (units)	Value	Name (units)	Value
rb_central_MOT_load_bias_x (A)	0	rb_central_MOT_load_bias_z (A)	0.5
rb_central_MOT_load_bias_y (A)	0.5	rb_central_MOT_load_current (A)	15

Table D.41: The globals in the group "Rb MOT - Central"

Name (units)	Value	Name (units)	Value
op_mw_transfer_centre_freq (MHz)	1.3	rb_optical_pumping_aom_warm_frequency (MHz)	70
op_mw_transfer_range (MHz)	0.2	rb_optical_pumping_end_repump_duration (s)	1.5e-4
op_mw_transfer_rate (MHz/s)	100	rb_optical_pumping_frequency (MHz)	81.5
optical_pumping_bias_x (A)	0.4	rb_optical_pumping_initial_repump_delay (s)	0
optical_pumping_bias_y (A)	1.0	rb_optical_pumping_initial_repump_duration (s)	0
optical_pumping_bias_z (A)	0.095	rb_optical_pumping_repump_amplitude (Arb.)	500
rb_optical_pumping	s03_rb_optical_pumping	rb_optical_pumping_repump_frequency (MHz)	75.2
rb_optical_pumping_amplitude (Arb.)	250	rb_optical_pumping_shutter_time (s)	1.2e-3
rb_optical_pumping_aom_warm_amplitude (Arb.)	700	rb_optical_pumping_time (s)	200e-6

Table D.42: The globals in the group “Rb Optical Pumping”

Name (units)	Value	Name (units)	Value
rotated_evaporation (Bool)	False	rotated_evaporation_time_constant_cross (s)	1.5
rotated_evaporation_duration (s)	1	rotated_trap_cross_beam_evaporation_amplitude (arb)	0.000000001
rotated_evaporation_hold_time (s)	0.35	rotated_trap_transport_beam_evaporation_amplitude (arb)	0.000000001
rotated_evaporation_time_constant (s)	1.5	square_cell_rotated_evaporation_end_amplitude_2D (arb)	0.2

Table D.43: The globals in the group “Rotated evaporation”

Name (units)	Value	Name (units)	Value
sorensen_high_voltage (V)	0.95	sorensen_low_voltage (V)	0.55

Table D.44: The globals in the group “Settings”

Name (units)	Value	Name (units)	Value
evaporated_square_cell_hold_time (s)	0	square_cell_evaporation_end_amplitude_cross (fraction)	0.4
square_cell_evaporation (Bool)	True	square_cell_evaporation_end_amplitude_transport (hardware)	134
square_cell_evaporation_duration (s)	4.5	square_cell_evaporation_time_constant (s)	4
square_cell_evaporation_end_amplitude_2D (arb)	0.24	square_cell_evaporation_transport_time_constant (s)	2.2
square_cell_evaporation_end_amplitude_2D_analog (V)	3	square_evap_trunc (fraction)	1

Table D.45: The globals in the group “Square Cell Evaporation”

Name (units)	Value	Name (units)	Value
cross_beam_frequency (MHz)	99	square_cell_hold_time	0
square_cell_cross_beam_amplitude (arb)	0.99	square_cell_transport_beam_cross_amplitude (hardware)	990
square_cell_cross_beam_turn_on_time (s)	0.01	square_cell_transport_beam_cross_time_constant (s)	1.5
square_cell_cross_delay (s)	0	square_cell_transport_beam_ramp_gradient (hardware/s)	800
square_cell_crossed_dipole (Bool)	True	square_cell_transport_beam_ramp_type ('exp' or 'linear' or 'None')	'linear'

Table D.46: The globals in the group "Square cell cross trap"

Name (units)	Value	Name (units)	Value
square_cell_microwave_11_to_22 (Bool)	True	square_cell_mw_transfer_range (MHz)	0.2
square_cell_mw_transfer_centre_freq (MHz)	1.46	square_cell_mw_transfer_rate (MHz/s)	150

Table D.47: The globals in the group "Square cell microwave to 22"

Name (units)	Value	Name (units)	Value
square_cell_quad_current (A)	16.85	square_cell_quad_trap_amplitude_cross (arb)	square_cell_post_mw_evaporation_end_amplitude_cross
square_cell_quad_end_bias_current (A)	1.6	square_cell_quad_trap_amplitude_transport (arb)	square_cell_post_mw_evaporation_end_amplitude_transport
square_cell_quad_side_bias_current (A)	5.037	square_cell_quad_trap_hold_time (seconds)	0
square_cell_quad_trap (Bool)	True	square_cell_quad_turn_on_time (seconds)	0.35
square_cell_quad_trap_amplitude_2D (arb)	square_cell_post_mw_evaporation_end_amplitude_2D		

Table D.48: The globals in the group "Square cell quad trap"

Name (units)	Value	Name (units)	Value
fluoro_image_MOT (Bool)	False	s0950_raman_transfer (Bool)	False
s01_use_k (Bool)	True	s0951_transfer_to_2_1 (Bool)	False
s01_use_rb (Bool)	True	s0952_blow_away_22 (Bool)	False
s021_k_compressed_MOT (Bool)	True	s0952_transfer_to_1_0 (Bool)	False
s022_k_compressed_MOT_cooling (Bool)	True	s095k1_transfer_22_to_10 (Bool)	False
s02_rb_PGC (Bool)	True	s095k1_transfer_22_to_11 (Bool)	False
s030_k_optical_pumping (Bool)	True	s095k2_blow_away_22 (Bool)	False
s031_k_initial_MT (Bool)	True	s09_dipole_trap_evaporation (Bool)	False
s032_k_spin_purification (Bool)	True	s10_transport_to_square_cell (Bool)	False
s03_rb_optical_pumping (Bool)	True	s11__absorption_image_both_species (Bool)	s11__absorption_image_k and s11__absorption_image_rb
s04_magnetic_trap (Bool)	True	s11__absorption_image_k (Bool)	True
s05_MT_compress (Bool)	True	s11__absorption_image_rb (Bool)	False
s06_microwave_evaporation (Bool)	True	s11__central_absorption_image (Bool)	True
s070_blow_away_rb (Bool)	False	s11__square_cell_bottom_imaging (Bool)	False
s07_MT_decompress (Bool)	True	s11__square_cell_side_imaging (Bool)	False
s07_MT_decompress_evap (Bool)	True	s11_central_image_k_fluoro (Bool)	False
s08_load_dipole_trap (Bool)	True	s11_central_image_k_recapture (Bool)	False
s0900_load_pure_dipole (Bool)	True	s11_central_image_rb_fluoro (Bool)	False
s0950_microwave_transfer (Bool)	False	verbose (Bool)	True

Table D.49: The globals in the group “Stages”

Name (units)	Value	Name (units)	Value
flat_trap_aom_frequency (MHz)	99	rotated_trap_transport_beam_amplitude (arb)	50
rotate_down (Bool)	False	stepper_down_T (s)	0.075
rotate_up (Bool)	False	stepper_up_T (s)	0.075
rotated_trap_cross_beam_amplitude (arb)	0.4	trap_2d (Bool)	True
rotated_trap_hold_time (s)	0.1	vortex_gyroscope_end_amplitude (arb)	0.4

Table D.50: The globals in the group “Stepper”

Name (units)	Value	Name (units)	Value
amplitude_heating (Bool)	False	transport_evap_amplitude_ramp_type ("sine" or "exp" or "linear")	'linear'
cycle (-)	0	transport_evap_amplitude_ramp_types (list)	('exp', 'sine', 'linear')
lens_warm_current (A)	0.0	transport_evap_before_transport (Bool)	False
return_trip (Bool)	False	transport_evap_before_transport_amp (arb)	650
sine4_transport (Bool)	False	transport_evap_dipole_trap_evaporation_time_constant	2
sine_transport (Bool)	True	transport_pre_ramp_time (s)	0.5
spline_transport (Bool)	False	transport_quad_current (A)	0
square_cell_bias_field (A)	0	transport_quad_field_time (s)	0.05
square_cell_transport_amplitude (fraction)	pure_dipole_amplitude	transport_quad_peak_fraction	0.5
transport_break_hold_time (s)	0	transport_quad_turnon_distance (fraction)	0.25
transport_break_position (distance)	0.10	transport_recompress (Bool)	False
transport_break_sine_ramp (Bool)	False	transport_recompress_amp_ramp_time (s)	1
transport_dipole_return_amplitude (arb)	900	transport_time (sec)	2.2
transport_distance (fraction)	0.134	transport_to_square_cell (Bool)	s10_transport_to_square_cell

Table D.51: The globals in the group "Transport - Square Cell"

Name (units)	Value	Name (units)	Value
vortex_spoon (Bool)	False	vortex_spoon_amplitude (arb)	500
vortex_spoon_A_end (V)	0.545	vortex_spoon_ramp_time (s)	0.1
vortex_spoon_A_initial (V)	0.485	vortex_spoon_wait_time (s)	400e-3
vortex_spoon_B_initial (V)	-0.6		

Table D.52: The globals in the group "Vortex spoon"

Name (units)	Value	Name (units)	Value
central_dipole_levitation_bias_x (A)	0.3	central_dipole_levitation_switch_time (s)	50e-3
central_dipole_levitation_bias_y (A)	0.35	central_dipole_levitation_time (s)	150e-3
central_dipole_levitation_bias_z (A)	0.012	levitate_central_dipole (Bool)	False
central_dipole_levitation_quad (A)	16		

Table D.53: The globals in the group “central dipole levitation”

Name (units)	Value	Name (units)	Value
defocus_for_2D (Bool)	False	defocused_amplitude (arb)	300
defocus_time (s)	2	defocused_position (A)	0.129

Table D.54: The globals in the group “defocus 2D”

Name (units)	Value	Name (units)	Value
final_2D_amplitude (arb)	0.18	rotated_hybrid_evaporation_duration (s)	3
rotated_BEC_hold_time (s)	0.5	rotated_hybrid_evaporation_time_constant (s)	1
rotated_hybrid_evaporation (Bool)	False	rotated_hybrid_evaporation_trunc (fraction)	1

Table D.55: The globals in the group “rotated hybrid evap”

Name (units)	Value	Name (units)	Value
squeeze_2D (Bool)	False	squeeze_2D_time (s)	0.5
squeeze_2D_amplitude (arb)	0.1	squeezed_2D_hold_time (s)	0.5

Table D.56: The globals in the group “squeeze 2D”

Name (units)	Value	Name (units)	Value
MT_kill_amp (arb)	0.25	hologram_exposure_time (s)	4.5
MT_kill_time (s)	0.01	kill_MT_for_alignment (Bool)	False

Table D.57: The globals in the group “test”

D.2 Experiment logic

Here we show an 'unsanitised' copy of common experiment logic used to produce ultracold atoms for several different experiments in the Monash K-Rb lab. This experiment logic is paired with the set of globals introduced in §D.1. The connection table shown is only a subset of lab connection table, as it does not include devices and I/O used only in the 'science' stage of an experiment (for which we have not included the experiment logic). A representation of the full lab connection table can be found in appendix D.3.

```
from __future__ import division

import cPickle

from labscript import *
from labscriptlib.krb.switchablecoildriver import *
from labscriptlib.krb.measurements import monitor_coils
from labscript_utils.unitconversions import *
from labscriptlib.krb import transport_lens_ramp

from labscript_devices.PulseBlaster import PulseBlaster
from labscript_devices.NI_PCIe_6363 import NI_PCIe_6363
from labscript_devices.NI_PCI_6733 import NI_PCI_6733
from labscript_devices.NovaTechDDS9M import NovaTechDDS9M
from labscript_devices.PhaseMatrixQuickSyn import PhaseMatrixQuickSyn, QuickSynDDS
from labscript_devices.RFBlaster import RFBlaster
from labscript_devices.Camera import Camera
from labscript_devices.ZaberStageController import ZaberStageController, ZaberStageTLS28M

rb_hyperfine = 6834.6826109043 * MHz
k_hyperfine = 254.0 * MHz


if rotate_up:
    try:
        pkl_file = open('C:\user_scripts\labscriptlib\krb\stepper_ramp.pkl', 'rb')
        stepper_ts = array(cPickle.load(pkl_file))
        pkl_file.close()
        print "Loaded stepper motor ramp"
    except:
       print "Failed loading stepper motor ramp points, please run stepper_ramp.py to generate the pickle"

PulseBlaster(name='pulseblaster_0', board_number=1)
ClockLine(name='pulseblaster_0_ni_clock', pseudoclock=pulseblaster_0.pseudoclock, connection='flag 0')
ClockLine(name='pulseblaster_0_novatech_clock', pseudoclock=pulseblaster_0.pseudoclock, connection='flag 1')

NI_PCIe_6363(name='ni_pcie_6363_0',parent_device=pulseblaster_0_ni_clock, clock_terminal='/ni_pcie_6363_0/PFI0'
                                , MAX_name='ni_pcie_6363_0',acquisition_rate=1e3)
NI_PCI_6733 (name='ni_pci_6733_0', parent_device=pulseblaster_0_ni_clock, clock_terminal='/ni_pcie_6363_0/PFI0'
                                , MAX_name='ni_pci_6733_0')

PulseBlaster(name='pulseblaster_1', board_number=0, trigger_device=ni_pcie_6363_0,trigger_connection="port0/line22"
                                )
ClockLine(name='pulseblaster_1_clock', pseudoclock=pulseblaster_1.pseudoclock, connection='flag 0')


PulseBlaster(name='pulseblaster_3', board_number=3, trigger_device=ni_pcie_6363_0,trigger_connection="port0/line3")

NovaTechDDS9M(name='novatechdds9m_0', parent_device=pulseblaster_0_novatech_clock, com_port='com8')
NovaTechDDS9M(name='novatechdds9m_1', parent_device=pulseblaster_0_novatech_clock, com_port='com9')
NovaTechDDS9M(name='novatechdds9m_2', parent_device=pulseblaster_0_novatech_clock, com_port='com10')
NovaTechDDS9M(name='novatechdds9m_3', parent_device=pulseblaster_1_clock, com_port='com11')



if microwave_evaporation:
    PhaseMatrixQuickSyn(name='phasematrix_0', com_port="COM32")

    RFBlaster('rfblaster_0','130.194.171.221',trigger_device=ni_pcie_6363_0,trigger_connection="port0/line23")

    QuickSynDDS( name = 'microwaves', parent_device=phasematrix_0,    connection='dds 0')
    DDS(    'quad_mod_I',   rfblaster_0.direct_outputs,    'dds 0')
    DDS(    'quad_mod_Q',   rfblaster_0.direct_outputs,    'dds 1')

if microwave_transfer:
    RFBlaster('rfblaster_1','130.194.171.223',trigger_device=ni_pcie_6363_0,trigger_connection="port0/line10")
    DDS(    'potassium_rf',   rfblaster_1.direct_outputs,    'dds 0')
    DDS(    'potassium_rf_2',   rfblaster_1.direct_outputs,    'dds 1')

DDS(          name='rb_imaging_push_aom',              parent_device=pulseblaster_0.direct_outputs,    connection='dds
                                0')
DDS(          name='rb_optical_pumping_aom',                    parent_device=pulseblaster_0.direct_outputs,
                                connection='dds 1')
```

```
69
70  DDS(        name='green_sheet_aom',                  parent_device=pulseblaster_1.direct_outputs,   connection='dds
                                0')
71
72
73  DDS(        name='k_raman_blue_80',                   parent_device=pulseblaster_3.direct_outputs,   connection='
                                dds 0')
74
75  DDS(        name='rb_central_MOT_trap_aom',           parent_device=novatechdds9m_0,  connection='channel 0',
                                digital_gate={'device':ni_pcie_6363_0,'connection':'port0/line5'})
76  DDS(        name='rb_central_MOT_repump_aom',         parent_device=novatechdds9m_0,  connection='channel 1',
                                digital_gate={'device':pulseblaster_0.direct_outputs,'connection':'flag 3'})
77  StaticDDS(   name='rb_source_MOT_repump_aom',         parent_device=novatechdds9m_0,  connection='channel 2',
                                digital_gate={'device':pulseblaster_0.direct_outputs,'connection':'flag 4'})
78  StaticDDS(   name='rb_source_MOT_trap_aom',           parent_device=novatechdds9m_0,  connection='channel 3',
                                digital_gate={'device':ni_pcie_6363_0,'connection':'port0/line6'})
79  DigitalOut(   name='novatechdds9m_0_table_enable',    parent_device=pulseblaster_0.direct_outputs,   connection='
                                flag 6')
80
81  DDS(        name='k_MOT_trap_aom',                    parent_device=novatechdds9m_1,  connection='channel 0',
                                digital_gate={'device':ni_pcie_6363_0,'connection':'port0/line7'})
82  DDS(        name='k_imaging_push_trap_aom',           parent_device=novatechdds9m_1,  connection='channel 1',
                                digital_gate={'device':pulseblaster_0.direct_outputs,'connection':'flag 8'})
83  StaticDDS(   name='k_lock_aom',                       parent_device=novatechdds9m_1,  connection='channel 2',
                                digital_gate={'device':pulseblaster_0.direct_outputs,'connection':'flag 9'})
84  StaticDDS(   name='rb_lock_aom',                      parent_device=novatechdds9m_1,  connection='channel 3')
85  DigitalOut(   name='novatechdds9m_1_table_enable',    parent_device=pulseblaster_0.direct_outputs,   connection='
                                flag 11')
86
87  DDS(        name='k_MOT_repump_aom',                  parent_device=novatechdds9m_2,  connection='channel 0',
                                digital_gate={'device':ni_pcie_6363_0,'connection':'port0/line0'})
88  DDS(        name='k_imaging_push_repump_aom',         parent_device=novatechdds9m_2,  connection='channel 1',
                                digital_gate={'device':ni_pcie_6363_0,'connection':'port0/line1'})
89  StaticDDS(   name='k_raman_shift_aom',                parent_device=novatechdds9m_2,  connection='channel 2')
90  StaticDDS(   name='k_raman_red_80',                   parent_device=novatechdds9m_2,  connection='channel 3',
                                digital_gate={'device':pulseblaster_3.direct_outputs,'connection':'flag 0'})
91  DigitalOut(   name='novatechdds9m_2_table_enable',    parent_device=ni_pcie_6363_0,   connection='port0/line4')
92
93  DDS(        name='dipole_trap_1_aom',                 parent_device=novatechdds9m_3,  connection='channel 0',
                                digital_gate={'device':pulseblaster_1.direct_outputs,'connection':'flag 2'},
94              amp_conv_class=SineAom,
95              amp_conv_params={
96                 "A": transport_aom_A,
97                 "f": transport_aom_f,
98                 "phase": transport_aom_phase,
99                 "c": transport_aom_c
100             })
101
102 DDS(        name='dipole_trap_2_aom',                 parent_device=novatechdds9m_3,  connection='channel 1',
                                digital_gate={'device':pulseblaster_1.direct_outputs,'connection':'flag 3'},
103             amp_conv_class=SineAom,
104             amp_conv_params={
105                "A": transport_aom_A,
106                "f": transport_aom_f,
107                "phase": transport_aom_phase,
108                "c": transport_aom_c
109             })
110
111 DigitalOut(   name='novatechdds9m_3_table_enable',    parent_device=pulseblaster_1.direct_outputs,   connection='
                                flag 1')
112
113 DigitalOut(   name="microwave_switch",                parent_device=pulseblaster_0.direct_outputs,   connection='flag 2'
                                )
114 DigitalOut(   name='sorensen_voltage_control',       parent_device=pulseblaster_0.direct_outputs,   connection='flag 7'
                                )
115
116 Shutter(      name='central_MOT_imaging_shutter',     parent_device=ni_pcie_6363_0,   connection='port0/line8',
                                delay=(2.83e-3,2.45e-3)) #open_by, close_from     #Sh_0_1
117 Shutter(      name='k_push_shutter',                  parent_device=ni_pcie_6363_0,   connection='port0/line9', delay=
                                (2.54e-3,2.35e-3)) #open, close           #Sh_0_2
118 Shutter(      name='k_source_MOT_shutter',            parent_device=ni_pcie_6363_0,   connection='port0/line11',
                                delay=(2.37e-3,2.54e-3)) #open, close           #Sh_0_4
119 Shutter(      name='rb_source_MOT_shutter',           parent_device=ni_pcie_6363_0,   connection='port0/line12',
                                delay=(2.49e-3,2.44e-3)) #open, close           #Sh_1_1
120 Shutter(      name='rb_optical_pumping_repump_shutter', parent_device=ni_pcie_6363_0,   connection='port0/line13',
                                 delay=(2.59e-3,1.74e-3)) #open_from, close_from    #Sh_1_2
121 Shutter(      name='rb_optical_pumping_shutter',      parent_device=pulseblaster_0.direct_outputs,   connection='flag
                                5', delay=(2.73e-3,3.25e-3)) #open_from, close_from    #Sh_1_3
122 Shutter(      name='science_bottom_imaging_shutter',  parent_device=ni_pcie_6363_0,   connection='port0/line14',
                                delay=(3.16e-3,1.74e-3)) #open_by, close_from     #Sh_1_4
123
124 DigitalOut(   name='stepper_trigger',     parent_device=ni_pcie_6363_0,   connection='port0/line17')
125 DigitalOut(   name='central_bias_z_coil_polarity',     parent_device=ni_pcie_6363_0,   connection='port0/line18')
126 DigitalOut(   name='central_bias_y_coil_polarity',     parent_device=ni_pcie_6363_0,   connection='port0/line19')
127 DigitalOut(   name='stepper_direction',     parent_device=ni_pcie_6363_0,   connection='port0/line20')
128
129 DigitalOut(   name='dipole_1_dump_flipper',     parent_device=ni_pcie_6363_0,   connection='port0/line21')
130
```

```
Shutter(      name='central_MOT_shutter',              parent_device=ni_pcie_6363_0,    connection='port0/line28',
                                delay=(3.11e-3,2.19e-3))  #open_by, close_from     #Sh_2_1
Shutter(      name='bragg_raman_shutter',     parent_device=ni_pcie_6363_0,    connection='port0/line29', delay=(3.07e
                                -3,2.16e-3))  #open_by, close_from     #Sh_2_2
Shutter(      name='rb_push_shutter',              parent_device=ni_pcie_6363_0,    connection='port0/line31', delay=
                                (2.78e-3,2.17e-3))  #open, close          #Sh_2_4


AnalogOut(    name='feshbach_coils',          parent_device=ni_pcie_6363_0,    connection='ao0',
            unit_conversion_class=UnidirectionalCoilDriver,
            unit_conversion_parameters={
                "slope": feshbach_coils_slope,
                "shift": feshbach_coils_shift,
                "saturation": feshbach_coils_saturation
            })
AnalogOut(    name='rb_source_MOT_coils',          parent_device=ni_pcie_6363_0,    connection='ao2',
            unit_conversion_class=UnidirectionalCoilDriver,
            unit_conversion_parameters={
                "slope": rb_source_MOT_slope,
                "shift": rb_source_MOT_shift,
                "saturation": rb_source_MOT_saturation
            })
AnalogOut(    name='central_Bq',                      parent_device=ni_pci_6733_0,     connection='ao0',
            unit_conversion_class=UnidirectionalCoilDriver,
            unit_conversion_parameters={
                "slope": central_quad_slope,
                "shift": central_quad_shift,
                "saturation": central_quad_saturation
            })
AnalogOut(    name='central_bias_x_coil',              parent_device=ni_pci_6733_0,     connection='ao1',
            unit_conversion_class=UnidirectionalCoilDriver,
            unit_conversion_parameters={
                "slope": central_bias_x_coil_slope,
                "shift": central_bias_x_coil_shift,
                "saturation": central_bias_x_coil_saturation
            })
AnalogOut(    name='central_bias_y_coil',              parent_device=ni_pci_6733_0,     connection='ao2',
            unit_conversion_class=UnidirectionalCoilDriver,
            unit_conversion_parameters={
                "slope": central_bias_y_coil_slope,
                "shift": central_bias_y_coil_shift,
                "saturation": central_bias_y_coil_saturation
            })
AnalogOut(    name='central_bias_z_coil',              parent_device=ni_pci_6733_0,     connection='ao3',
            unit_conversion_class=UnidirectionalCoilDriver,
            unit_conversion_parameters={
                "slope": central_bias_z_coil_slope,
                "shift": central_bias_z_coil_shift,
                "saturation": central_bias_z_coil_saturation
            })
AnalogOut(    name='k_source_MOT_west_coil',           parent_device=ni_pci_6733_0,     connection='ao4',
                                unit_conversion_class=UnidirectionalCoilDriver, unit_conversion_parameters={"slope":
                                k_2D_MOT_coil_slope})
AnalogOut(    name='k_source_MOT_top_coil',            parent_device=ni_pci_6733_0,     connection='ao5',
                                unit_conversion_class=UnidirectionalCoilDriver, unit_conversion_parameters={"slope":
                                k_2D_MOT_coil_slope})
AnalogOut(    name='k_source_MOT_east_coil',           parent_device=ni_pci_6733_0,     connection='ao6',
                                unit_conversion_class=UnidirectionalCoilDriver, unit_conversion_parameters={"slope":
                                k_2D_MOT_coil_slope})
AnalogOut(    name='k_source_MOT_bottom_coil',         parent_device=ni_pci_6733_0,     connection='ao7',
                                unit_conversion_class=UnidirectionalCoilDriver, unit_conversion_parameters={"slope":
                                k_2D_MOT_coil_slope})


AnalogIn(     'central_Bq_control_monitor',          ni_pcie_6363_0,        'ai16')
AnalogIn(     'central_Bq_monitor',                ni_pcie_6363_0,        'ai17')
AnalogIn(     'central_bias_y_control_monitor',      ni_pcie_6363_0,        'ai18')
AnalogIn(     'central_bias_y_monitor',            ni_pcie_6363_0,        'ai19')
AnalogIn(     'central_bias_x_control_monitor',      ni_pcie_6363_0,        'ai20')
AnalogIn(     'central_bias_x_monitor',            ni_pcie_6363_0,        'ai21')
AnalogIn(     'central_bias_z_control_monitor',      ni_pcie_6363_0,        'ai22')
AnalogIn(     'central_bias_z_monitor',            ni_pcie_6363_0,        'ai23')

coil_dictionary = {"central_Bq":     central_Bq_monitor,
                "central_bias_x": central_bias_x_monitor,
                "central_bias_y": central_bias_y_monitor,
                "central_bias_z": central_bias_z_monitor,

                "central_bias_x_control": central_bias_x_control_monitor,
                "central_bias_y_control": central_bias_y_control_monitor,
                "central_bias_z_control": central_bias_z_control_monitor,
                "central_Bq_control": central_Bq_control_monitor,
               }


if central_image_k_recapture:
    red_camera_exposure_time = k_recapture_imaging_time
elif central_image_k_fluoro:
    red_camera_exposure_time = k_fluoro_imaging_time
```

```
209 elif central_image_rb_fluoro:
210     red_camera_exposure_time = rb_fluoro_imaging_time
211 
212 if verbose: print "Red camera exposure time will be %.9f"%red_camera_exposure_time
213 
214 if central_imaging or central_image_k_recapture or central_image_k_fluoro or central_image_rb_fluoro or
                                    square_cell_side_imaging:
215     Camera(         name='central_MOT_camera',              parent_device=ni_pcie_6363_0,     connection='port0/line16',
                                    BIAS_port=42518, serial_number="111C00D1BE", SDK="Photonfocus",
216                 effective_pixel_size=5.916e-6, exposure_time=red_camera_exposure_time, orientation='side')
217 
218 if square_cell_bottom_imaging:
219     Camera(         name='science_bottom_camera',               parent_device=ni_pcie_6363_0,     connection='port0/line26
                                    ', BIAS_port=42521, serial_number="14C2", SDK="Andor2",
220             effective_pixel_size=4.5e-6, exposure_time=iXon_exposure_time, orientation='bottom')
221 
222 
223 ######## END CONNECTION TABLE  ########
224 
225 ## Work out when to trigger the rf blaster so that it is roughly at the end of the MOT loads
226 
227 if use_k and use_rb:
228     total_load_time = k_central_MOT_load_time+rb_central_MOT_load_time
229 elif use_k:
230     total_load_time = k_central_MOT_load_time
231 elif use_rb:
232     total_load_time = rb_central_MOT_load_time
233 else:
234     raise LabscriptError("You must use at least one of Potassium or Rubidium, otherwise you're doing nothing!")
235 total_load_time += 100e-3
236 
237 if total_load_time > 2:
238     total_load_time -= 1.9
239 if microwave_evaporation:
240     rfblaster_0.set_initial_trigger_time(total_load_time)
241     if verbose: print "rfblaster_0 triggered at t=%.9f"%(total_load_time)
242 
243 
244 sorensen_critical_current = 65.0
245 
246 # Define our time variable. Zero is a good time to start the experiment!
247 t = 0
248 start()
249 
250 ######### Enable Novatechs ###########
251 novatechdds9m_0_table_enable.go_high(t)
252 novatechdds9m_1_table_enable.go_high(t)
253 novatechdds9m_2_table_enable.go_high(t)
254 novatechdds9m_3_table_enable.go_high(novatechdds9m_3_table_enable.t0)
255 
256 ######## Maintain DDS output of the laser locks #######
257 k_lock_aom.setfreq(87*MHz)
258 k_lock_aom.setamp(0.63)
259 
260 rb_lock_aom.setfreq(47*MHz)
261 rb_lock_aom.setamp(800./1024)
262 
263 ################################################################################
264 #
265 # setup switchable coil drivers for the central bias fields
266 #
267 #   This code manages the combination of a coil driver and a polarity reversing
268 #   relay. When the current is requested to be negative by the user, this code
269 #   automatically activates the relay to swap the current direction, and can do
270 #   this part way throug a ramp, at the appropiate time.
271 #
272 # x switchable coil driver
273 #
274 # switchablecoildriver('central_bias_x_coil_driver',central_bias_x_coil,central_bias_x_coil_polarity)
275 central_bias_x_coil_driver = central_bias_x_coil
276 #
277 # y switchable coild driver
278 #
279 switchablecoildriver('central_bias_y_coil_driver',central_bias_y_coil,central_bias_y_coil_polarity)
280 # central_bias_y_coil_driver = central_bias_y_coil
281 #
282 # z switchable coil driver
283 #
284 switchablecoildriver('central_bias_z_coil_driver',central_bias_z_coil,central_bias_z_coil_polarity)
285 # central_bias_z_coil_driver = central_bias_z_coil
286 #
287 ################################################################################
288 
289 # set Sorenson voltage to "low" mode
290 sorensen_voltage_control.go_low(t)
291 
292 # turn the dipole trap on (probably important to do it now for temperature stability)
293 dipole_trap_1_aom.setfreq(dipole_trap_1_aom.t0,dipole_trap_aom_frequency*MHz)
294 dipole_trap_1_aom.setamp(dipole_trap_1_aom.t0, dipole_trap_power, units="hardware")
```

```
295 dipole_trap_2_aom.setfreq(dipole_trap_1_aom.t0,dipole_trap_aom_frequency*MHz)
296 dipole_trap_2_aom.setamp(dipole_trap_1_aom.t0, dipole_trap_power, units="hardware")
297
298
299 # Keep the optical pumping AOM running, otherwise it will program a zero and run out of amplitude registers!
300 rb_optical_pumping_aom.setfreq(t,rb_optical_pumping_aom_warm_frequency*MHz)
301 rb_optical_pumping_aom.setamp(t,rb_optical_pumping_aom_warm_amplitude/1023.)
302 rb_optical_pumping_aom.enable(t)
303
304
305
306 central_MOT_shutter.close(t)
307 t+=100e-3
308
309
310 # Raman beam setup
311 k_raman_shift_aom.setfreq(k_raman_global_freq*MHz)
312 k_raman_shift_aom.setamp(k_raman_global_amp/1023.)
313
314 k_raman_red_80.setfreq(k_raman_red_freq*MHz)
315 k_raman_red_80.setamp(k_raman_red_amp/1023.)
316 k_raman_red_80.disable(t)
317 k_raman_blue_80.setfreq(t, 80*MHz)
318 k_raman_blue_80.setamp(t, .43)
319 k_raman_blue_80.disable(t)
320
321
322 central_MOT_shutter.open(t)
323 ######## BEGIN MOT LOAD ############
324
325 if use_k:
326     central_Bq.constant(t, k_central_MOT_load_current, "A")
327     central_bias_x_coil_driver.constant(t, k_central_MOT_load_bias_x, "A")
328     central_bias_y_coil_driver.constant(t, k_central_MOT_load_bias_y, "A")
329     central_bias_z_coil_driver.constant(t, k_central_MOT_load_bias_z, "A")
330
331     k_source_MOT_shutter.open(t)
332     # k_central_MOT_shutter.open(t)
333
334     k_MOT_trap_aom.enable(t)
335     k_MOT_trap_aom.setfreq(t, k_trap_load_frequency * MHz)
336     k_MOT_trap_aom.setamp(t, k_trap_load_amplitude / 1023.0)
337
338     k_MOT_repump_aom.enable(t)
339     k_MOT_repump_aom.setfreq(t,k_repump_load_frequency * MHz)
340     k_MOT_repump_aom.setamp(t,k_repump_load_amplitude / 1023.0)
341
342     # Enable push light
343     k_push_shutter.open(t)
344
345     k_imaging_push_trap_aom.enable(t)
346     k_imaging_push_trap_aom.setfreq(t, k_push_frequency * MHz)
347     k_imaging_push_trap_aom.setamp(t, k_push_amplitude / 1023.0)
348
349     k_imaging_push_repump_aom.enable(t)
350     k_imaging_push_repump_aom.setfreq(t, k_push_repump_frequency * MHz)
351     k_imaging_push_repump_aom.setamp(t, k_push_repump_amplitude / 1023.0)
352
353
354     # prepare source MOT fields
355     k_source_MOT_west_coil.constant(t,k_source_current_west, "A")
356     k_source_MOT_east_coil.constant(t, k_source_current_east, "A")
357     k_source_MOT_top_coil.constant(t, k_source_current_top, "A")
358     k_source_MOT_bottom_coil.constant(t, k_source_current_bottom, "A")
359
360     # K load happens now!
361     t+=k_central_MOT_load_time
362
363     # Then turn everything off after the load:
364     # Turn off source coils
365     k_source_MOT_west_coil.constant(t,0, "A")
366     k_source_MOT_east_coil.constant(t, 0, "A")
367     k_source_MOT_top_coil.constant(t, 0, "A")
368     k_source_MOT_bottom_coil.constant(t, 0, "A")
369
370     # Turn off source light (leave AOMs on, they give light to the central MOT!)
371     k_source_MOT_shutter.close(t)
372
373     # Turn off push beam
374     k_push_shutter.close(t)
375     k_imaging_push_trap_aom.disable(t)
376     k_imaging_push_repump_aom.disable(t)
377     ### end K load ###
378     if k_compressed_MOT:
379         ### begin K compressed MOT
380         central_Bq.ramp(t,k_compressed_MOT_time, k_central_MOT_load_current,k_compressed_MOT_current, 1e3, "A"
                                    )
381         central_bias_x_coil_driver.ramp(t,k_compressed_MOT_time, k_central_MOT_load_bias_x,
                                        k_compressed_MOT_bias_x, 1e3, "A")
```

```
382         central_bias_y_coil_driver.ramp(t,k_compressed_MOT_time, k_central_MOT_load_bias_y,
                                    k_compressed_MOT_bias_y, 1e3, "A")
383         central_bias_z_coil_driver.ramp(t,k_compressed_MOT_time, k_central_MOT_load_bias_z,
                                    k_compressed_MOT_bias_z, 1e3, "A")
384
385         k_MOT_trap_aom.frequency.ramp(t, k_compressed_MOT_time,k_trap_load_frequency * MHz,
                                    k_trap_compressed_MOT_frequency * MHz,1e3)
386         k_MOT_trap_aom.amplitude.ramp(t, k_compressed_MOT_time, k_trap_load_amplitude / 1023.0,
                                    k_trap_compressed_MOT_amplitude / 1023.0,1e3)
387
388         k_MOT_repump_aom.frequency.ramp(t, k_compressed_MOT_time,k_repump_load_frequency * MHz,
                                    k_repump_compressed_MOT_frequency * MHz,1e3)
389         k_MOT_repump_aom.amplitude.ramp(t, k_compressed_MOT_time, k_repump_load_amplitude / 1023.0,
                                    k_repump_compressed_MOT_amplitude / 1023.0,1e3)
390
391         t+= k_compressed_MOT_time
392         t+= k_compressed_MOT_hold_time
393
394         if k_compressed_MOT_cooling:
395             k_MOT_trap_aom.frequency.ramp(t, k_compressed_MOT_cooling_time,k_trap_compressed_MOT_frequency *
                                    MHz,k_trap_compressed_MOT_cooling_frequency * MHz,1e3)
396             k_MOT_trap_aom.amplitude.ramp(t, k_compressed_MOT_cooling_time, k_trap_compressed_MOT_amplitude
                                    / 1023.0,k_trap_compressed_MOT_cooling_amplitude / 1023.0,1e3)
397
398             k_MOT_repump_aom.frequency.ramp(t, k_compressed_MOT_cooling_time,
                                    k_repump_compressed_MOT_frequency * MHz,
                                    k_repump_compressed_MOT_cooling_frequency * MHz,1e3)
399             k_MOT_repump_aom.amplitude.ramp(t, k_compressed_MOT_cooling_time,
                                    k_repump_compressed_MOT_amplitude / 1023.0,
                                    k_repump_compressed_MOT_cooling_amplitude / 1023.0,1e3)
400
401             t+= k_compressed_MOT_cooling_time
402             k_MOT_trap_aom.disable(t)
403             k_MOT_repump_aom.disable(t)
404
405     # # ## K Optical pumping
406     if k_optical_pumping:
407         central_Bq.constant(t, 0, "A")
408         # central_Bq.constant(t, -10)
409
410         central_bias_x_coil_driver.constant(t, k_optical_pumping_bias_x, "A")
411         central_bias_y_coil_driver.constant(t, k_optical_pumping_bias_y, "A")
412         central_bias_z_coil_driver.constant(t, k_optical_pumping_bias_z, "A")
413
414         # Prepare shutters
415         central_MOT_imaging_shutter.open(t)
416
417         # Wait some time to make sure fields have changed and shutter is open
418         t += k_optical_pumping_delay
419
420
421         #set the AOM frequencies - start with only repump on
422         k_imaging_push_trap_aom.setfreq(t, k_optical_pumping_frequency*MHz)
423         k_imaging_push_trap_aom.setamp(t, k_optical_pumping_amplitude/1023.)
424         k_imaging_push_trap_aom.disable(t)
425
426         k_imaging_push_repump_aom.setfreq(t, k_optical_pumping_repump_frequency*MHz)
427         k_imaging_push_repump_aom.setamp(t, k_optical_pumping_repump_amplitude/1023.)
428         k_imaging_push_repump_aom.enable(t)
429
430         # wait repump initial time
431         t+=k_optical_pumping_initial_repump_duration
432
433         # then turn on optical pumping light
434         k_imaging_push_trap_aom.enable(t)
435
436         # optically pump for the appropriate time
437         t+= k_optical_pumping_time
438
439         # then turn off OP light
440         k_imaging_push_trap_aom.disable(t)
441
442         # leave repump on for a bit longer then turn off
443         t+=k_optical_pumping_end_repump_duration
444         k_imaging_push_repump_aom.disable(t)
445         central_MOT_imaging_shutter.close(t+10e-3)
446
447     if k_initial_MT:
448         if k_MT_compressed_quad > sorensen_critical_current:
449             sorensen_voltage_control.go_high(t)
450         central_Bq.ramp(t,k_MT_compression_time,k_compressed_MOT_current,k_MT_compressed_quad,1e3,"A")
451         central_bias_x_coil_driver.ramp(t, k_MT_compression_time, k_compressed_MOT_bias_x,
                                    k_MT_compressed_bias_x, 10e3, "A")
452         central_bias_y_coil_driver.ramp(t, k_MT_compression_time, k_compressed_MOT_bias_y,
                                    k_MT_compressed_bias_y, 10e3, "A")
453         central_bias_z_coil_driver.ramp(t, k_MT_compression_time, k_compressed_MOT_bias_z,
                                    k_MT_compressed_bias_z, 10e3, "A")
454         t+=k_MT_compression_time
455
```

```
456         t+=k_MT_hold_time
457
458         if k_spin_purification:
459             # decompress K MT for spin purification
460             central_Bq.sine_ramp(t,k_MT_decompression_time,k_MT_compressed_quad,k_MT_decompressed_quad,1e3,"A
                                    ")
461
462             #Move the bias fields so that they are ready for the Rb MOT next too
463             central_bias_x_coil_driver.ramp(t, k_MT_decompression_time, k_MT_compressed_bias_x,
                                        rb_central_MOT_load_bias_x, 1e3, "A")
464             central_bias_y_coil_driver.ramp(t, k_MT_decompression_time, k_MT_compressed_bias_y,
                                        rb_central_MOT_load_bias_y, 1e3, "A")
465             central_bias_z_coil_driver.ramp(t, k_MT_decompression_time, k_MT_compressed_bias_z,
                                        rb_central_MOT_load_bias_z, 1e3, "A")
466
467             if k_MT_decompressed_quad < sorensen_critical_current:
468                 #linear:
469                 t_switch = ((sorensen_critical_current - k_MT_compressed_quad)/(k_MT_decompressed_quad-
                                        k_MT_compressed_quad))*k_MT_decompression_time
470                 #sine:
471                 #   t_switch = (2.*MT_decompression_time/pi)*arcsin(sqrt((sorensen_critical_current-central_MT_compressed_quad)/(
                                        central_MT_decompressed_quad-central_MT_compressed_quad)))
472                 sorensen_voltage_control.go_low(t+t_switch)
473                 t+=k_MT_decompression_time
474
475
476             t += k_MT_decompressed_hold_time
477
478
479 #### Now load some Rb ####
480 if use_rb:
481
482     if verbose: print "Beginning Rb load at t=%.9f"%t
483
484     # Prepare source MOT light
485     rb_source_MOT_trap_aom.enable(t)
486     rb_source_MOT_trap_aom.setfreq(rb_source_trap_frequency * MHz)
487     rb_source_MOT_trap_aom.setamp(rb_source_trap_amplitude / 1023.0)
488
489     rb_source_MOT_repump_aom.enable(t)
490     rb_source_MOT_repump_aom.setfreq(rb_source_repump_frequency * MHz)
491     rb_source_MOT_repump_aom.setamp(rb_source_repump_amplitude / 1023.0)
492
493     rb_source_MOT_shutter.open(t)
494
495
496     # Prepare central MOT light
497     rb_central_MOT_trap_aom.enable(t)
498     rb_central_MOT_trap_aom.setfreq(t,rb_central_MOT_trap_frequency * MHz)
499     rb_central_MOT_trap_aom.setamp(t,rb_central_MOT_trap_amplitude / 1023.0)
500
501     rb_central_MOT_repump_aom.enable(t)
502     rb_central_MOT_repump_aom.setfreq(t,rb_central_MOT_repump_frequency * MHz)
503     rb_central_MOT_repump_aom.setamp(t,rb_central_MOT_repump_amplitude / 1023.0)
504
505     # Prepare push light (but leave it off for now)
506     rb_imaging_push_aom.disable(t)
507     rb_imaging_push_aom.setfreq(t,rb_push_frequency * MHz)
508     rb_imaging_push_aom.setamp(t,rb_push_amplitude)
509     rb_push_shutter.open(t)
510
511
512     #turn on the source MOT coils
513     rb_source_MOT_coils.constant(t,rb_source_current,"A")
514
515     # central MOT coils
516     central_Bq.constant(t,rb_central_MOT_load_current,"A")
517     central_bias_x_coil_driver.constant(t,rb_central_MOT_load_bias_x,"A")
518     central_bias_y_coil_driver.constant(t,rb_central_MOT_load_bias_y,"A")
519     central_bias_z_coil_driver.constant(t,rb_central_MOT_load_bias_z,"A")
520
521
522     # now the source MOT should be on, along with the central MOT
523     # we begin the pushing sequence:
524     if verbose: print "Starting Rb MOT load at t = %s"%t
525     rb_load_t = t
526     while rb_load_t + rb_source_MOT_load_time + rb_push_duration < rb_central_MOT_load_time + t:
527         # let the source MOT load
528         rb_load_t += rb_source_MOT_load_time
529
530         # turn off trap light to repump while we push, and turn on push beam
531         rb_source_MOT_trap_aom.disable(rb_load_t)
532         rb_imaging_push_aom.enable(rb_load_t)
533
534         #wait for the push duration
535         rb_load_t += rb_push_duration
536
537         #turn the push off and the MOT back on
538         rb_source_MOT_trap_aom.enable(rb_load_t)
```

```
539         rb__imaging__push__aom.disable(rb__load__t)
540
541
542     #load is now complete
543     #turn off source
544     rb__source__MOT__trap__aom.disable(rb__load__t)
545     rb__source__MOT__repump__aom.disable(rb__load__t)
546     rb__source__MOT__shutter.close(rb__load__t)
547
548     rb__push__shutter.close(rb__load__t)
549
550     rb__source__MOT__coils.constant(rb__load__t,0,"A")
551     t = rb__load__t
552     ### end Rb load ###
553     if verbose: print "Finishing Rb MOT load at t = %s"%t
554
555     # Rb CMOT
556     central__Bq.sine__ramp(t,rb__MOT__compress__time,rb__central__MOT__load__current, rb__central__MOT__compress__current
                                    , 1e3, "A")
557     central__bias__x__coil__driver.sine__ramp(t, rb__MOT__compress__time, rb__central__MOT__load__bias__x,
                                    rb__central__MOT__compress__bias__x, 1e3, "A")
558     central__bias__y__coil__driver.sine__ramp(t, rb__MOT__compress__time, rb__central__MOT__load__bias__y,
                                    rb__central__MOT__compress__bias__y, 1e3, "A")
559     central__bias__z__coil__driver.sine__ramp(t, rb__MOT__compress__time, rb__central__MOT__load__bias__z,
                                    rb__central__MOT__compress__bias__z, 1e3, "A")
560
561     t+=rb__MOT__compress__time
562
563     t+=rb__central__MOT__trap__aom.amplitude.ramp(t, rb__compressed__MOT__power__ramp__time,
                                    rb__central__MOT__trap__amplitude/1023., rb__compressed__MOT__trap__amplitude/1023.,
                                    1e3)
564
565     if verbose: print "Rb MOT compression finished at t = %s"%t
566
567     ##### Begin PGC ####
568
569     # Prepare fields ##
570     if rb__PGC:
571         # turn off lasers
572         rb__central__MOT__trap__aom.disable(t)
573         rb__central__MOT__repump__aom.disable(t)
574
575         # set the fields
576         if verbose: print "pgc fields start ramping at t=%.9f"%t
577         pgc__field__ramp__rate = 5e4
578
579         central__Bq.sine__ramp(t,pgc__field__turn__time, rb__central__MOT__compress__current, PGC__quad,
                                    pgc__field__ramp__rate, "A")
580         central__bias__x__coil__driver.sine__ramp(t, pgc__field__turn__time, rb__central__MOT__compress__bias__x, PGC__bias__x,
                                    pgc__field__ramp__rate, "A")
581         central__bias__y__coil__driver.sine__ramp(t, pgc__field__turn__time, rb__central__MOT__compress__bias__y, PGC__bias__y,
                                    pgc__field__ramp__rate, "A")
582         central__bias__z__coil__driver.sine__ramp(t, pgc__field__turn__time, rb__central__MOT__compress__bias__z, PGC__bias__z,
                                    pgc__field__ramp__rate, "A")
583
584         t += pgc__field__turn__time
585         if verbose: print "pgc fields ready at t =%.9f"%t
586
587         ## Set frequencies and turn on light.
588
589         rb__central__MOT__trap__aom.enable(t)
590         rb__central__MOT__trap__aom.setfreq(t, rb__pgc__trap__frequency*MHz)
591         rb__central__MOT__trap__aom.setamp(t, rb__pgc__trap__amplitude/1023.)
592
593         rb__central__MOT__repump__aom.enable(t)
594         rb__central__MOT__repump__aom.setfreq(t, rb__pgc__repump__frequency*MHz)
595         rb__central__MOT__repump__aom.setamp(t, rb__pgc__repump__amplitude/1023.)
596
597         # PGC happens
598         t+= rb__pgc__time
599         # Turn off light after we're done!
600         rb__central__MOT__trap__aom.disable(t)
601         rb__central__MOT__repump__aom.disable(t)
602         if verbose: print "Rb PGC finished at t=%.9f"%(t)
603
604     if load__dipole__trap and magnetic__trap:
605         dipole__1__dump__flipper.go__high(t)
606
607         dipole__trap__1__aom.setfreq(t,dipole__trap__aom__frequency*MHz)
608         dipole__trap__1__aom.setamp(t, dipole__trap__power, units="hardware")
609         dipole__trap__1__aom.enable(t+0.5)
610
611         dipole__trap__2__aom.setfreq(t,dipole__trap__aom__frequency*MHz)
612         dipole__trap__2__aom.setamp(t, dipole__trap__power, units="hardware")
613         dipole__trap__2__aom.enable(t+0.5)
614
615     #### Rb Optical Pumping ####
616     if rb__optical__pumping:
617         # Close the central shutter. This is important, as we don't want repump light going into the central MOT
```

```
618         # But we do want MOT repump on, which goes into the optical pumping repump beam
619         central_MOT_shutter.close(t)
620
621         # turn off all the MOT light (in case we didn't do PGC)
622         rb_central_MOT_trap_aom.disable(t)
623         rb_central_MOT_repump_aom.disable(t)
624
625         # prepare optical pumping light (leave OP aom off for now)
626         rb_optical_pumping_shutter.open(t)
627
628
629         # open repump shutters and set frequencies, but leave light off until we've changed the fields and the shutters are completely open
630         rb_optical_pumping_repump_shutter.open(t)
631         rb_central_MOT_repump_aom.setfreq(t, rb_optical_pumping_repump_frequency * MHz)
632         rb_central_MOT_repump_aom.setamp(t,rb_optical_pumping_repump_amplitude / 1023.0)
633
634
635         # prepare magnetic fields
636         central_Bq.constant(t, 0, "A")
637         central_bias_x_coil_driver.constant(t, optical_pumping_bias_x, "A")
638         central_bias_y_coil_driver.constant(t, optical_pumping_bias_y, "A")
639         central_bias_z_coil_driver.constant(t, optical_pumping_bias_z, "A")
640
641
642
643         # Wait for the fields to have changed and the shutters to have opened/closed
644         t += rb_optical_pumping_shutter_time
645
646         #Start of actual optical pumping stuff! Begin with some repump
647
648
649         rb_optical_pumping_aom.setfreq(t,rb_optical_pumping_frequency*MHz)
650         rb_optical_pumping_aom.setamp(t,rb_optical_pumping_amplitude/1023.)
651         rb_optical_pumping_aom.enable(t)
652         # After the optical pumping repump delay turn the repump to the full optical pumping settings and turn on OP light
653
654         rb_central_MOT_repump_aom.enable(t+rb_optical_pumping_initial_repump_delay)
655         assert rb_optical_pumping_initial_repump_delay < rb_optical_pumping_time, 'you cannot delay repump longer
                                    than the total optical pumping time'
656
657
658         # optically pump for rb_optical_pumping_time
659         t+=rb_optical_pumping_time
660
661         # then turn off optical pumping light but leave repump on for now
662         rb_optical_pumping_aom.disable(t)
663
664         # wait for rb_optical_pumping_end_repump_duration
665         t += rb_optical_pumping_end_repump_duration
666         # then finally turn off optical pumping repump
667         rb_central_MOT_repump_aom.disable(t)
668
669         if verbose: print 'Optical pumping done, shutter closed at t=%.9f'%(t)
670         rb_optical_pumping_repump_shutter.close(t)
671         rb_optical_pumping_shutter.close(t)
672
673         # change optical pumping aom to our "warming" settings to keep it warm for the next run
674         rb_optical_pumping_aom.setfreq(t,rb_optical_pumping_aom_warm_frequency*MHz)
675         rb_optical_pumping_aom.setamp(t,rb_optical_pumping_aom_warm_amplitude/1023.)
676         rb_optical_pumping_aom.enable(t+5e-3)
677
678     if magnetic_trap:
679         # set the fields to our capture values
680         central_Bq.constant(t, central_MT_capture_quad, "A")
681         central_bias_x_coil_driver.constant(t, central_MT_capture_bias_x, "A")
682         central_bias_y_coil_driver.constant(t, central_MT_capture_bias_y, "A")
683         central_bias_z_coil_driver.constant(t, central_MT_capture_bias_z, "A")
684         t+= magnetic_trap_hold_time
685 if MT_compress:
686     if central_MT_compressed_quad > sorensen_critical_current:
687         sorensen_voltage_control.go_high(t)
688     if use_rb:
689         central_Bq.ramp(t,MT_compression_time,central_MT_capture_quad,central_MT_compressed_quad,1e3,"A")
690         central_bias_x_coil_driver.ramp(t, MT_compression_time, central_MT_capture_bias_x,
                                    central_MT_compressed_bias_x, 1e3, "A")
691         central_bias_y_coil_driver.ramp(t, MT_compression_time, central_MT_capture_bias_y,
                                    central_MT_compressed_bias_y, 1e3, "A")
692         central_bias_z_coil_driver.ramp(t, MT_compression_time, central_MT_capture_bias_z,
                                    central_MT_compressed_bias_z, 1e3, "A")
693     else:
694         central_Bq.ramp(t,MT_compression_time,k_MT_decompressed_quad,central_MT_compressed_quad,1e3,"A")
695         central_bias_x_coil_driver.ramp(t, MT_compression_time, rb_central_MOT_load_bias_x,
                                    central_MT_compressed_bias_x, 1e3, "A")
696         central_bias_y_coil_driver.ramp(t, MT_compression_time, rb_central_MOT_load_bias_y,
                                    central_MT_compressed_bias_y, 1e3, "A")
697         central_bias_z_coil_driver.ramp(t, MT_compression_time, rb_central_MOT_load_bias_z,
                                    central_MT_compressed_bias_z, 1e3, "A")
698
699
```

```
700
701     t+=MT_compression_time
702     if verbose: print "MT compression finishes at t=%.9f"%t
703
704     t+= MT_compressed_hold_time
705     if verbose: print "Held in compressed MT until t=%.9f"%t
706
707
708 if magnetic_trap and MT_compress and microwave_evaporation:
709
710     if verbose: print "Microwave evaporation starts at t=%.9f"%t
711     microwave_switch.go_high(t)
712     quad_mod_I.setphase(t,-90)
713     quad_mod_Q.setphase(t,0)
714     quad_mod_I.setamp(t,IQ_amp)
715     quad_mod_Q.setamp(t,IQ_amp)
716
717     ###
718     # Calculated in runmanager, but FYI:
719     #   mw_evap_time = (mw_evap_start-mw_evap_stop)/evap_rate
720     ###
721
722     if mw_evap_type == 'linear':
723         quad_mod_I.frequency.ramp(t, mw_evap_initial_time, microwave_LO_freq * MHz - rb_hyperfine - mw_evap_start
                                      *MHz, microwave_LO_freq * MHz - rb_hyperfine - mw_evap_mid*MHz, (1.0*
                                      mw_evap_points)/mw_evap_time)
724         t += quad_mod_Q.frequency.ramp(t, mw_evap_initial_time, microwave_LO_freq * MHz - rb_hyperfine -
                                      mw_evap_start*MHz, microwave_LO_freq * MHz - rb_hyperfine - mw_evap_mid*MHz, (
                                      1.0*mw_evap_points)/mw_evap_time)
725         quad_mod_I.frequency.ramp(t, mw_evap_time, microwave_LO_freq * MHz - rb_hyperfine - mw_evap_mid*MHz,
                                      microwave_LO_freq * MHz - rb_hyperfine - mw_evap_stop*MHz, (1.0*mw_evap_points)/
                                      mw_evap_time)
726         t += quad_mod_Q.frequency.ramp(t, mw_evap_time, microwave_LO_freq * MHz - rb_hyperfine - mw_evap_mid*
                                      MHz, microwave_LO_freq * MHz - rb_hyperfine - mw_evap_stop*MHz, (1.0*
                                      mw_evap_points)/mw_evap_time)
727     elif mw_evap_type == 'exp':
728         quad_mod_I.frequency.exp_ramp_t(t, mw_evap_time, microwave_LO_freq * MHz - rb_hyperfine - mw_evap_start
                                      *MHz, microwave_LO_freq * MHz - rb_hyperfine - mw_evap_stop*MHz,
                                      mw_evap_time_constant, (1.0*mw_evap_points)/mw_evap_time)
729         t += quad_mod_Q.frequency.exp_ramp_t(t, mw_evap_time, microwave_LO_freq * MHz - rb_hyperfine -
                                      mw_evap_start*MHz, microwave_LO_freq * MHz - rb_hyperfine - mw_evap_stop*MHz,
                                      mw_evap_time_constant, (1.0*mw_evap_points)/mw_evap_time)
730     else:
731         raise LabscriptError('You must set mw_evap_type to "linear" or "exp" (it is currently %s)'%str(mw_evap_type))
732
733     if verbose: print "Microwave evaporation finishes at t=%.9f"%t
734     microwave_switch.go_low(t)
735     quad_mod_I.setamp(t,0)
736     quad_mod_Q.setamp(t,0)
737
738     if blow_away_rb:
739         if verbose: print "Blowing away Rb atoms at t=%.9f"%t
740
741         if blow_away_rb_use_light:
742             rb_central_MOT_trap_aom.setfreq(t, rb_imaging_frequency * MHz)
743             rb_central_MOT_trap_aom.setamp(t, rb_imaging_amplitude)
744             central_MOT_shutter.open(t)
745             rb_central_MOT_trap_aom.enable(t)
746             t+=blow_away_rb_time
747             rb_central_MOT_trap_aom.disable(t)
748             central_MOT_shutter.close(t)
749         else:
750             quad_mod_I.frequency.ramp(t, blow_away_rb_time, microwave_LO_freq * MHz - rb_hyperfine - mw_evap_stop
                                      *MHz, microwave_LO_freq * MHz - rb_hyperfine - blow_away_mw_end*MHz, (1.0*1000)
                                      /blow_away_rb_time)
751             quad_mod_Q.frequency.ramp(t, blow_away_rb_time, microwave_LO_freq * MHz - rb_hyperfine -
                                      mw_evap_stop*MHz, microwave_LO_freq * MHz - rb_hyperfine - blow_away_mw_end*
                                      MHz, (1.0*1000)/blow_away_rb_time)
752             t += blow_away_rb_time
753
754 # If MT_decompress is enabled (requires MT & MT Compress & microwave_evaporation but keep at top level to avoid cascading if statements)
755 if magnetic_trap and MT_compress and MT_decompress:
756
757
758     if verbose: print "Magnetic trap decompression starts at t=%.9f"%t
759     central_Bq.sine_ramp(t,MT_decompression_time,central_MT_compressed_quad,central_MT_decompressed_quad,1e3,"
                                      A")
760     central_bias_x_coil_driver.sine_ramp(t, MT_decompression_time, central_MT_compressed_bias_x,
                                      central_MT_decompressed_bias_x, 1e3, "A")
761     central_bias_y_coil_driver.sine_ramp(t, MT_decompression_time, central_MT_compressed_bias_y,
                                      central_MT_decompressed_bias_y, 1e3, "A")
762     central_bias_z_coil_driver.sine_ramp(t, MT_decompression_time, central_MT_compressed_bias_z,
                                      central_MT_decompressed_bias_z, 1e3, "A")
763
764
765     if central_MT_decompressed_quad < sorensen_critical_current:
766         # Calculate when we should switch the quadrupole coil voltage
767         #linear:
```

```
768         #  t_switch = ((sorensen_critical_current - central_MT_compressed_quad)/(central_MT_decompressed_quad-central_MT_compressed_quad))*
                                    MT_decompression_time
769         #sine:
770         t_switch = (2.*MT_decompression_time/pi)*arcsin(sqrt((sorensen_critical_current-central_MT_compressed_quad)/
                                    (central_MT_decompressed_quad-central_MT_compressed_quad)))
771         sorensen_voltage_control.go_low(t+t_switch)
772         if verbose: print 'sorenson PSU voltage switched to low at t=%.9f'%(t+t_switch)
773
774     if MT_decompress_evap:
775         if MT_decompression_evap_time > MT_decompression_time:
776             raise Exception("MT_decompression_evap_time should be <= MT_decompression_time")
777         print rfblaster_0.trigger(t,1e-3)
778         t += 1e-3
779         if verbose: print "Microwave evaporation (during Magnetic trap decompression) ramp starts at t=%.9f"%t
780         microwave_switch.go_high(t)
781         quad_mod_I.setphase(t,-90)
782         quad_mod_Q.setphase(t,0)
783         quad_mod_I.setamp(t,IQ_amp)
784         quad_mod_Q.setamp(t,IQ_amp)
785         quad_mod_I.frequency.ramp(t, MT_decompression_evap_time, microwave_LO_freq * MHz - rb_hyperfine -
                                    mw_evap_stop*MHz, microwave_LO_freq * MHz - rb_hyperfine -
                                    mw_evap_decompress_stop*MHz, 1e3)
786         quad_mod_Q.frequency.ramp(t, MT_decompression_evap_time, microwave_LO_freq * MHz - rb_hyperfine -
                                    mw_evap_stop*MHz, microwave_LO_freq * MHz - rb_hyperfine -
                                    mw_evap_decompress_stop*MHz, 1e3)
787         if verbose: print "Microwave evaporation (during Magnetic trap decompression) ramp ends at t=%.9f"%(t+
                                    MT_decompression_evap_time)
788
789         if mt_extra_decompress_hold_microwaves_on:
790             microwave_switch.go_low(t+MT_decompression_time)
791             quad_mod_I.setamp(t+MT_decompression_time,0)
792             quad_mod_Q.setamp(t+MT_decompression_time,0)
793             if verbose: print "Microwaves (during Magnetic trap decompression) turned off at t=%.9f"%(t+
                                    MT_decompression_time)
794         else:
795             microwave_switch.go_low(t+MT_decompression_evap_time)
796             quad_mod_I.setamp(t+MT_decompression_evap_time,0)
797             quad_mod_Q.setamp(t+MT_decompression_evap_time,0)
798             if verbose: print "Microwaves (during Magnetic trap decompression) turned off at t=%.9f"%(t+
                                    MT_decompression_evap_time)
799
800
801     t += MT_decompression_time
802     if verbose: print "Magnetic trap decompression finishes at t=%.9f"%t
803
804     t+=MT_decompression_hold_time
805     if verbose: print "Held in decompressed trap until t=%.9f"%t
806
807
808 if magnetic_trap and MT_compress and microwave_evaporation and MT_decompress and load_dipole_trap and
                                    load_pure_dipole_trap:
809     central_bias_z_coil_driver.sine_ramp(t, MT_load_pure_dipole_decompress_time,central_MT_decompressed_bias_z,
                                    pure_dipole_central_bias_z,1e3,"A")
810     central_bias_x_coil_driver.sine_ramp(t,MT_load_pure_dipole_decompress_time,central_MT_decompressed_bias_x,
                                    pure_dipole_central_bias_x,1e3,"A")
811     central_bias_y_coil_driver.sine_ramp(t,MT_load_pure_dipole_decompress_time,central_MT_decompressed_bias_y,
                                    pure_dipole_central_bias_y,1e3,"A")
812
813     t += central_Bq.sine_ramp(t, MT_load_pure_dipole_decompress_time,central_MT_decompressed_quad,0,1e3,"A")
814     central_bias_z_coil_driver.sine_ramp(t, pure_dipole_hold_time,pure_dipole_central_bias_z,0.001,1e3,"A")
815     t += pure_dipole_hold_time
816
817 if magnetic_trap and MT_compress and microwave_evaporation and MT_decompress and load_dipole_trap and
                                    load_pure_dipole_trap:
818
819     if s0950_raman_transfer:
820
821         if k_raman_22_to_10:
822
823             k_raman_blue_80.setfreq(t, k_raman_22_to_10_blue_freq*MHz)
824             k_raman_blue_80.setamp(t, k_raman_22_to_10_blue_amp/1023.)
825             central_bias_x_coil_driver.sine_ramp(t, k_raman_22_to_10_field_transfer_time,pure_dipole_central_bias_x,
                                    k_raman_22_to_10_bias_x,1e5,"A")
826             central_bias_y_coil_driver.sine_ramp(t, k_raman_22_to_10_field_transfer_time,pure_dipole_central_bias_y,
                                    k_raman_22_to_10_bias_y,1e5,"A")
827             t+=central_bias_z_coil_driver.sine_ramp(t, k_raman_22_to_10_field_transfer_time,
                                    pure_dipole_central_bias_z,k_raman_22_to_10_bias_z,1e5,"A")
828
829             print "K Raman pulse (|2,2> -> |1,0>) of duration %.6f at T=%.9F"%(k_raman_22_to_10_duration, t)
830             k_raman_red_80.enable(t)
831             k_raman_blue_80.enable(t)
832
833             # t+= k_raman_22_to_10_duration
834             for i in range(2000):
835                 k_raman_blue_80.setfreq(t, k_raman_22_to_10_blue_freq*MHz-0.01*MHz+0.02/2000*MHz)
836                 t+= k_raman_22_to_10_duration/2000.
837             # t += k_raman_blue_80.frequency.ramp(t, k_raman_22_to_10_duration, k_raman_22_to_10_blue_freq*MHz-0.05*MHz,
                                    k_raman_22_to_10_blue_freq*MHz+0.05*MHz, 2000)
838
```

```
839             k_raman_blue_80.disable(t)
840             k_raman_red_80.disable(t)
841
842             central_bias_x_coil_driver.sine_ramp(t, k_raman_22_to_10_field_transfer_time,k_raman_22_to_10_bias_x,
                                pure_dipole_central_bias_x,1e5,"A")
843             central_bias_y_coil_driver.sine_ramp(t, k_raman_22_to_10_field_transfer_time,k_raman_22_to_10_bias_y,
                                pure_dipole_central_bias_y,1e5,"A")
844             t+=central_bias_z_coil_driver.sine_ramp(t, k_raman_22_to_10_field_transfer_time,
                                k_raman_22_to_10_bias_z,pure_dipole_central_bias_z,1e5,"A")
845
846         t+= k_raman_transfer_after_time
847
848
849
850     if microwave_transfer:
851         central_bias_x_coil_driver.sine_ramp(t, mw_transfer_field_time,pure_dipole_central_bias_x,mw_transfer_bias_x
                                ,1e5,"A")
852         central_bias_y_coil_driver.sine_ramp(t, mw_transfer_field_time,pure_dipole_central_bias_y,mw_transfer_bias_y
                                ,1e5,"A")
853         t+=central_bias_z_coil_driver.sine_ramp(t, mw_transfer_field_time,pure_dipole_central_bias_z,
                                mw_transfer_bias_z,1e5,"A")
854
855         if k_to_11:
856             central_bias_y_coil_driver.sine_ramp(t, mw_transfer_field_time,mw_transfer_bias_y,k_transfer_bias_y,1e5,"A
                                ")
857             t+=central_bias_z_coil_driver.sine_ramp(t, mw_transfer_field_time,mw_transfer_bias_z,k_transfer_bias_z,
                                1e5,"A")
858             rfblaster_1.trigger(t-10e-3,1e-3)
859             potassium_rf.setamp(t,k_rf_amp)
860
861             if k_transfer___adiabatic_passage:
862                 k_transfer_start = k_transfer_centre_freq - 0.5* k_transfer_range
863                 k_transfer_stop = k_transfer_centre_freq + 0.5* k_transfer_range
864                 # k_transfer_time = 1.0*k_transfer_range/k_transfer_rate
865                 if k_transfer_trunc is not False:
866                     k_transfer_time = (2.0*((k_transfer_centre_freq+k_transfer_trunc)-(k_transfer_centre_freq - 0.5*
                                k_transfer_range))/k_transfer_range)*k_transfer_time
867                     k_transfer_stop = k_transfer_centre_freq+k_transfer_trunc
868
869                 if verbose: print "k tranfer |1,1> time is %.6f"%k_transfer_time
870                 print "Sweeping potassium rf detuning from %.6f MHz to %.6f MHz in %.9f seconds AT T=%.9F"%(
                                k_transfer_start,k_transfer_stop,k_transfer_time,t)
871                 potassium_rf.frequency.ramp(t, k_transfer_time, k_hyperfine + k_transfer_start*MHz, k_hyperfine +
                                k_transfer_stop*MHz, k_transfer_sample_rate)#4e4/mw_transfer_time)
872
873                 t+=k_transfer_time
874             elif k_transfer___pi_pulse:
875                 if verbose: print "k transfer |1,1> time is %.6f"%k_transfer_time
876                 print "Pi pulse at %.6f MHz for %.9f seconds long T=%.9F"%(k_transfer_centre_freq,
                                k_transfer_pi_pulse_time,t)
877                 potassium_rf.frequency.constant(t, k_transfer_centre_freq*MHz+k_hyperfine)
878                 t+=k_transfer_pi_pulse_time
879             else:
880                 raise LabscriptError('You must choose a type of state transfer (adiabatic passage or Pi pulse) for K transfer to
                                |1,1>')
881             potassium_rf.setamp(t,0)
882
883         elif k_22_to_10:
884             central_bias_y_coil_driver.sine_ramp(t, mw_transfer_field_time,mw_transfer_bias_y,
                                k_22_10_transfer_bias_y,1e5,"A")
885             t+=central_bias_z_coil_driver.sine_ramp(t, mw_transfer_field_time,mw_transfer_bias_z,
                                k_22_10_transfer_bias_z,1e5,"A")
886             rfblaster_1.trigger(t-10e-3,1e-3)
887             potassium_rf.setamp(t,k_22_10_rf1_amp)
888             potassium_rf_2.setamp(t,k_22_10_rf2_amp)
889
890             k_transfer_start = k_22_10_transfer_centre_freq - 0.5* k_22_10_transfer_range
891             k_transfer_stop = k_22_10_transfer_centre_freq + 0.5* k_22_10_transfer_range
892             if k_22_10_transfer_trunc is not False:
893                 k_22_10_transfer_time = (2.0*((k_22_10_transfer_centre_freq+k_transfer_trunc)-(
                                k_22_10_transfer_centre_freq - 0.5* k_22_10_transfer_range))/
                                k_22_10_transfer_range)*k_22_10_transfer_time
894                 k_transfer_stop = k_22_10_transfer_centre_freq+k_22_10_transfer_trunc
895
896             if verbose: print "k transfer from |2,2> to |1,0> time is %.6f"%k_transfer_time
897             print "Sweeping potassium rf detuning from %.6f MHz to %.6f MHz in %.9f seconds AT T=%.9F"%(
                                k_transfer_start,k_transfer_stop,k_22_10_transfer_time,t)
898             potassium_rf.frequency.ramp(t, k_22_10_transfer_time, k_hyperfine + k_transfer_start*MHz, k_hyperfine +
                                k_transfer_stop*MHz, k_22_10_transfer_sample_rate)#4e4/mw_transfer_time)
899             potassium_rf_2.setfreq(t, k_22_10_rf2_freq*MHz)
900
901             t+=k_22_10_transfer_time
902             potassium_rf.setamp(t,0)
903             potassium_rf_2.setamp(t,0)
904
905         if k_blow_away_22:
906             central_bias_x_coil_driver.sine_ramp(t, mw_transfer_field_time,mw_transfer_bias_x,central_imaging_bias_x,
                                1e5,"A")
907             if k_to_11:
```

```
908             central_bias_y_coil_driver.sine_ramp(t, mw_transfer_field_time,k_transfer_bias_y,central_imaging_bias_y,
                                1e5,"A")
909             t+=central_bias_z_coil_driver.sine_ramp(t, mw_transfer_field_time,k_transfer_bias_z,
                                central_imaging_bias_z,1e5,"A")
910         elif k_22_to_10:
911             central_bias_y_coil_driver.sine_ramp(t, mw_transfer_field_time,k_22_10_transfer_bias_y,
                                central_imaging_bias_y,1e5,"A")
912             t+=central_bias_z_coil_driver.sine_ramp(t, mw_transfer_field_time,k_22_10_transfer_bias_z,
                                central_imaging_bias_z,1e5,"A")
913         else:
914             central_bias_y_coil_driver.sine_ramp(t, mw_transfer_field_time,mw_transfer_bias_y,
                                central_imaging_bias_y,1e5,"A")
915             t+=central_bias_z_coil_driver.sine_ramp(t, mw_transfer_field_time,mw_transfer_bias_z,
                                central_imaging_bias_z,1e5,"A")
916         central_MOT_imaging_shutter.open(t)
917         k_imaging_push_trap_aom.enable(t)
918         k_imaging_push_trap_aom.setfreq(t, k_imaging_frequency * MHz)
919         k_imaging_push_trap_aom.setamp(t, k_imaging_amplitude / 1023.0)
920         t+= 5e-3
921         k_imaging_push_trap_aom.disable(t)
922         central_MOT_imaging_shutter.close(t)
923
924         central_bias_x_coil_driver.sine_ramp(t, mw_transfer_field_time,central_imaging_bias_x,mw_transfer_bias_x,
                                1e5,"A")
925         central_bias_y_coil_driver.sine_ramp(t, mw_transfer_field_time,central_imaging_bias_y,mw_transfer_bias_y,
                                1e5,"A")
926         t+=central_bias_z_coil_driver.sine_ramp(t, mw_transfer_field_time,central_imaging_bias_z,
                                mw_transfer_bias_z,1e5,"A")
927
928     else:
929         if k_to_11:
930             central_bias_y_coil_driver.sine_ramp(t, mw_transfer_field_time,k_transfer_bias_y,mw_transfer_bias_y,1e5
                                ,"A")
931             t+=central_bias_z_coil_driver.sine_ramp(t, mw_transfer_field_time,k_transfer_bias_z,mw_transfer_bias_z
                                ,1e5,"A")
932         elif k_22_to_10:
933             central_bias_y_coil_driver.sine_ramp(t, mw_transfer_field_time,k_22_10_transfer_bias_y,
                                mw_transfer_bias_y,1e5,"A")
934             t+=central_bias_z_coil_driver.sine_ramp(t, mw_transfer_field_time,k_22_10_transfer_bias_z,
                                mw_transfer_bias_z,1e5,"A")
935
936
937     central_bias_x_coil_driver.sine_ramp(t, mw_transfer_field_time,mw_transfer_bias_x,pure_dipole_central_bias_x
                            ,1e5,"A")
938     central_bias_y_coil_driver.sine_ramp(t, mw_transfer_field_time,mw_transfer_bias_y,pure_dipole_central_bias_y
                            ,1e5,"A")
939     t+=central_bias_z_coil_driver.sine_ramp(t, mw_transfer_field_time,mw_transfer_bias_z,
                            pure_dipole_central_bias_z,1e5,"A")
940
941     t+= mw_transfer_after_time
942
943 ##### Potassium central imaging section ####
944
945 ### K absorption imaging
946 if central_imaging and absorption_image_k:
947     # Turn the MOT light off
948     if not k_imaging_MOT_repump:
949         central_MOT_shutter.close(t)
950
951     k_MOT_trap_aom.disable(t)
952     k_MOT_repump_aom.disable(t)
953
954     rb_central_MOT_trap_aom.disable(t)
955
956     rb_central_MOT_repump_aom.disable(t)
957
958     # Open the imaging shutter and prepare the imaging light
959     central_MOT_imaging_shutter.open(t)
960     k_imaging_push_trap_aom.setfreq(t, k_imaging_frequency * MHz)
961     k_imaging_push_trap_aom.setamp(t, k_imaging_amplitude / 1023.0)
962
963     if k_imaging_MOT_repump:
964         k_MOT_repump_aom.setfreq(t-10e-3,k_imaging_repump_frequency * MHz)
965         k_MOT_repump_aom.setamp(t-10e-3,k_imaging_repump_amplitude / 1023.0)
966         central_MOT_shutter.open(t-10e-3)
967     elif k_imaging_repump:
968         k_imaging_push_repump_aom.setfreq(t, k_imaging_repump_frequency * MHz)
969         k_imaging_push_repump_aom.setamp(t, k_imaging_repump_amplitude / 1023.0)
970
971     # Turn off the dipole trap.
972     # This may happen some time during the magnetic component drop
973     # if we want to get rid of the magnetically trapped part
974
975     # We don't want it staying on any longer than the drop time though!
976     dipole_delay = min(dipole_image_delay, drop_time)
977     if verbose: print "Disable dipole trap for imaging at t=%.6f"%(t+dipole_delay)
978     dipole_trap_1_aom.disable(t+dipole_delay)
979     dipole_trap_2_aom.disable(t+dipole_delay)
980
```

```
981     # Turn the magnetic trap/MOT field off. We use a "negative" voltage on the
982     # magneato to make it switch faster. Ensure that sorenson is on low voltage now!
983     central_Bq.constant(t, -10)
984     sorensen_voltage_control.go_low(t)
985
986     # Set up imaging bias fields
987     central_bias_x_coil_driver.constant(t, central_imaging_bias_x, "A")
988     central_bias_y_coil_driver.constant(t, central_imaging_bias_y, "A")
989     central_bias_z_coil_driver.constant(t, central_imaging_bias_z, "A")
990
991
992     # Start exposing the camera while the atoms are falling
993     t += drop_time-0.5*(red_camera_exposure_time-imaging_pulse_time)
994     central_MOT_camera.expose('absorption_k41', t, 'atoms')
995
996     t += 0.5*(red_camera_exposure_time-imaging_pulse_time)
997
998     # atoms have now fallen for drop_time
999     # flash the light on
1000
1001    k_imaging_push_trap_aom.enable(t)
1002
1003    if k_imaging_MOT_repump:
1004        k_MOT_repump_aom.enable(t-.5e-3)
1005
1006    elif k_imaging_repump:
1007        k_imaging_push_repump_aom.enable(t)
1008
1009    t += imaging_pulse_time
1010
1011    # Then turn it off again
1012    k_imaging_push_trap_aom.disable(t)
1013    k_imaging_push_repump_aom.disable(t)
1014    k_MOT_repump_aom.disable(t)
1015
1016    # Wait for the exposure to finish!
1017    t += 0.5*(red_camera_exposure_time-imaging_pulse_time)
1018
1019    # Wait the interframe time before taking the flat field image
1020    t += interframe_time
1021
1022    # Flat field exposure -- do the same thing, but now there shouldn't be any atoms left!
1023    central_MOT_camera.expose('absorption_k41', t, 'flat')
1024    t += 0.5*(red_camera_exposure_time-imaging_pulse_time)
1025
1026    k_imaging_push_trap_aom.enable(t)
1027    if k_imaging_MOT_repump:
1028        k_MOT_repump_aom.enable(t)
1029    elif k_imaging_repump:
1030        k_imaging_push_repump_aom.enable(t)
1031
1032    t += imaging_pulse_time
1033
1034    k_imaging_push_trap_aom.disable(t)
1035    k_imaging_push_repump_aom.disable(t)
1036    k_MOT_repump_aom.disable(t)
1037
1038    # wait for the exposure to finish
1039    t += 0.5*(red_camera_exposure_time-imaging_pulse_time)
1040
1041    # Again wait the interframe time before taking the dark image
1042    t += interframe_time
1043
1044    # Dark field exposure
1045    central_MOT_camera.expose('absorption_k41', t, 'dark')
1046    t += red_camera_exposure_time
1047
1048    ## and we're done with this image, continue on to final cleanup before ending experiment
1049
1050 ## K recapture imaging
1051 elif central_image_k_recapture:
1052    # Open the shutter we'll need to get them ready
1053    central_MOT_shutter.open(t)
1054
1055    # If there is a non-zero drop time, then turn off everything and let the cloud expand, before recapturing
1056    if drop_time:
1057        sorensen_voltage_control.go_low(t)
1058        central_Bq.constant(t, 0, "A")
1059        # Lets leave the bias fields how they were, then switch them to the recapture MOT fields when it's recapture time
1060        k_MOT_trap_aom.disable(t)
1061        k_MOT_repump_aom.disable(t)
1062        rb_central_MOT_trap_aom.disable(t)
1063
1064        rb_central_MOT_repump_aom.disable(t)
1065        t+=drop_time
1066    else:
1067        rb_central_MOT_trap_aom.disable(t)
1068
1069        rb_central_MOT_repump_aom.disable(t)
```

```
1070
1071        # Now turn on our recapture MOT
1072        k_MOT_trap_aom.enable(t)
1073        k_MOT_trap_aom.setfreq(t, k_trap_recapture_imaging_frequency * MHz)
1074        k_MOT_trap_aom.setamp(t, k_trap_recapture_imaging_amplitude / 1023.0)
1075
1076        k_MOT_repump_aom.enable(t)
1077        k_MOT_repump_aom.setfreq(t,  k_repump_recapture_imaging_frequency * MHz)
1078        k_MOT_repump_aom.setamp(t, k_repump_recapture_imaging_amplitude / 1023.0)
1079
1080        sorensen_voltage_control.go_low(t)
1081        central_Bq.constant(t, k_recapture_imaging_quad, "A")
1082        central_bias_x_coil_driver.constant(t, 0, "A")
1083        central_bias_y_coil_driver.constant(t, 0, "A")
1084        central_bias_z_coil_driver.constant(t, 0, "A")
1085
1086
1087        # Wait some time to recapture the atoms
1088        t += k_recapture_imaging_MOT_load_time
1089
1090        # Then expose the camera
1091        central_MOT_camera.expose('fluorescence_k41', t, 'atoms')
1092
1093        # Wait until the image has been taken
1094        t += red_camera_exposure_time
1095
1096        ## and we're done with this image, continue on to final cleanup before ending experiment
1097
1098
1099    ## K fluorescence imaging
1100    elif central_image_k_fluoro:
1101        # Turn off all the MOTs/traps, but leave shutters needed for K open!
1102        central_MOT_shutter.open(t)
1103
1104        k_MOT_trap_aom.disable(t)
1105        k_MOT_repump_aom.disable(t)
1106
1107        rb_central_MOT_trap_aom.disable(t)
1108
1109        rb_central_MOT_repump_aom.disable(t)
1110
1111        # Turn off the dipole trap.
1112        # This may happen some time during the magnetic component drop
1113        # if we want to get rid of the magnetically trapped part
1114
1115        # We don't want it staying on any longer than the drop time though!
1116        dipole_delay = max(dipole_image_delay, drop_time)
1117        if verbose: print "Disable dipole trap for imaging at t=%.6f"%(t+dipole_delay)
1118
1119        # Turn the magnetic trap/MOT field off. We use a "negative" voltage on the
1120        # magneato to make it switch faster. Ensure that sorenson is on low voltage now!
1121        central_Bq.constant(t, -10)
1122        sorensen_voltage_control.go_low(t)
1123
1124        # Set up imaging bias fields
1125        central_bias_x_coil_driver.constant(t, central_imaging_bias_x, "A")
1126        central_bias_y_coil_driver.constant(t, central_imaging_bias_y, "A")
1127        central_bias_z_coil_driver.constant(t, central_imaging_bias_z, "A")
1128
1129        k_MOT_trap_aom.setfreq(t, k_trap_fluoro_frequency * MHz)
1130        k_MOT_trap_aom.setamp(t, k_trap_fluoro_amplitude / 1023.0)
1131        k_MOT_repump_aom.setfreq(t, k_repump_fluoro_frequency * MHz)
1132        k_MOT_repump_aom.setamp(t, k_repump_fluoro_amplitude / 1023.0)
1133        t += drop_time
1134        central_MOT_camera.expose('fluorescence_k41', t, 'atoms')
1135        k_MOT_trap_aom.enable(t)
1136        k_MOT_repump_aom.enable(t)
1137        t += red_camera_exposure_time
1138        k_MOT_trap_aom.disable(t)
1139        k_MOT_repump_aom.disable(t)
1140        t += interframe_time
1141        k_MOT_trap_aom.enable(t)
1142        k_MOT_repump_aom.enable(t)
1143        central_MOT_camera.expose('fluorescence_k41', t, 'dark')
1144        k_MOT_trap_aom.disable(t)
1145        k_MOT_repump_aom.disable(t)
1146        t += red_camera_exposure_time
1147    ##### END Central K imaging #####
1148
1149    ##### Rubidium central imaging section ####
1150
1151    elif central_imaging and absorption_image_rb:
1152        # Turn off all the MOTs/traps
1153        rb_central_MOT_trap_aom.disable(t)
1154        rb_central_MOT_repump_aom.disable(t)
1155        if rb_imaging_repump:
1156            central_MOT_shutter.open(t)
1157            rb_central_MOT_repump_aom.setfreq(t, rb_imaging_repump_frequency*MHz)
1158            rb_central_MOT_repump_aom.setamp(t, rb_imaging_repump_amplitude/1023.)
```

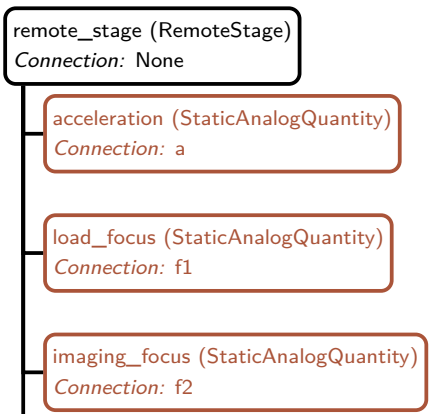
```
1159        rb_central_MOT_repump_aom.enable(t + drop_time - min(drop_time, 2e-3))
1160        if verbose: print "Repump light turned on for imaging at t=%.9f"%(t + drop_time - min(drop_time, 2e-3))
1161    else:
1162        central_MOT_shutter.close(t)
1163
1164    # Turn off the dipole trap.
1165    # This may happen some time during the magnetic component drop
1166    # if we want to get rid of the magnetically trapped part
1167
1168    # We don't want it staying on any longer than the drop time though!
1169    dipole_delay = min(dipole_image_delay, drop_time)
1170    if verbose: print "Disable dipole trap for imaging at t=%.6f"%(t+dipole_delay)
1171
1172    if side_imaging_expand_in_beam_1:
1173        dipole_trap_1_aom.setamp(t,side_imaging_expanded_power,"hardware")
1174        dipole_trap_1_aom.disable(t+drop_time-0.2e-3)
1175        dipole_trap_2_aom.disable(t)
1176    elif side_imaging_expand_in_beam_2:
1177        dipole_trap_1_aom.disable(t)
1178        dipole_trap_2_aom.setamp(t,side_imaging_expanded_power,"hardware")
1179        dipole_trap_2_aom.disable(t+drop_time-0.2e-3)
1180    else:
1181        dipole_trap_1_aom.disable(t+dipole_delay)
1182        dipole_trap_2_aom.disable(t+dipole_delay)
1183
1184    # Open the shutter and prepare the imaging light
1185    central_MOT_imaging_shutter.open(t)
1186    rb_imaging_push_aom.setfreq(t, rb_imaging_frequency * MHz)
1187    rb_imaging_push_aom.setamp(t, rb_imaging_amplitude)
1188
1189    # Turn the magnetic trap/MOT field off. We use a "negative" voltage on the
1190    # magneato to make it switch faster. Ensure that sorenson is on low voltage now!
1191    central_Bq.constant(t, -10)
1192    sorensen_voltage_control.go_low(t)
1193
1194    # Set up imaging bias fields
1195    central_bias_x_coil_driver.constant(t, central_imaging_bias_x, "A")
1196    central_bias_y_coil_driver.constant(t, central_imaging_bias_y, "A")
1197    central_bias_z_coil_driver.constant(t, central_imaging_bias_z, "A")
1198
1199    # If we're doing Stern-Gerlach imaging, we'll pulse the quad field on and off again during the drop time
1200    if SG_imaging:
1201        central_Bq.constant(t + 6e-3, SG_current, "A")
1202        central_Bq.constant(t + 12e-3, -10)
1203
1204
1205    # Atoms exposure
1206
1207    # Start exposing camera during tof
1208    t += drop_time-0.5*(red_camera_exposure_time-imaging_pulse_time)
1209
1210    central_MOT_camera.expose('absorption_rb87', t, 'atoms')
1211
1212    # now wait until drop_time before flashing on imaging light
1213    t += 0.5*(red_camera_exposure_time-imaging_pulse_time)
1214    rb_imaging_push_aom.enable(t)
1215    if verbose: print "Image taken at t = %.9f"%t
1216    t += imaging_pulse_time
1217    rb_imaging_push_aom.disable(t)
1218
1219    # Now wait until camera has finished exposing
1220    t += 0.5*(red_camera_exposure_time-imaging_pulse_time)
1221
1222    # Flat field exposure
1223    t+= interframe_time
1224
1225    central_MOT_camera.expose('absorption_rb87', t, 'flat')
1226
1227    t += 0.5*(red_camera_exposure_time-imaging_pulse_time)
1228
1229    rb_imaging_push_aom.enable(t)
1230    t += imaging_pulse_time
1231    rb_imaging_push_aom.disable(t)
1232    # and finish exposing camera:
1233    t += 0.5*(red_camera_exposure_time-imaging_pulse_time)
1234
1235    # Dark field exposure
1236    t += interframe_time
1237    central_MOT_camera.expose('absorption_rb87', t, 'dark')
1238    t += red_camera_exposure_time
1239
1240 elif central_image_rb_fluoro:
1241    # Turn off all the MOTs/traps, but leave shutters needed for Rb open!
1242    central_MOT_shutter.open(t)
1243    k_MOT_trap_aom.disable(t)
1244    k_MOT_repump_aom.disable(t)
1245    rb_central_MOT_trap_aom.disable(t)
1246    rb_central_MOT_repump_aom.disable(t)
1247    if rb_imaging_repump:
```

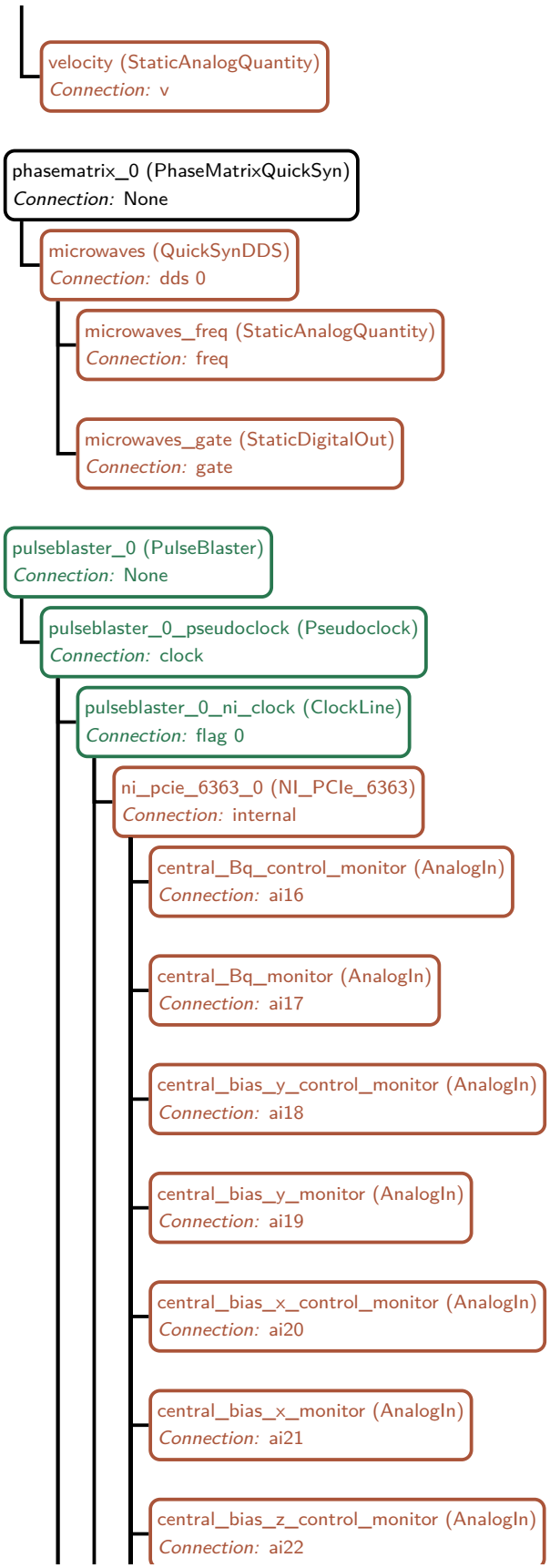
```
1248            central_MOT_shutter.open(t)
1249            rb_central_MOT_repump_aom.enable(t + drop_time - min(drop_time, 2e-3))
1250        else:
1251            pass
1252
1253        # Turn off the dipole trap.
1254        # This may happen some time during the magnetic component drop
1255        # if we want to get rid of the magnetically trapped part
1256
1257        # Turn the magnetic trap/MOT field off. We use a "negative" voltage on the
1258        # magneato to make it switch faster. Ensure that sorenson is on low voltage now!
1259        central_Bq.constant(t, -10)
1260        sorensen_voltage_control.go_low(t)
1261
1262        # Set up imaging bias fields
1263        central_bias_x_coil_driver.constant(t, central_imaging_bias_x, "A")
1264        central_bias_y_coil_driver.constant(t, 0,"A") #central_imaging_bias_y, "A")
1265        central_bias_z_coil_driver.constant(t, central_imaging_bias_z, "A")
1266
1267        rb_central_MOT_trap_aom.setfreq(t, rb_fluoro_frequency * MHz)
1268        rb_central_MOT_trap_aom.setamp(t, rb_fluoro_amplitude / 1023.0)
1269        t += drop_time
1270        central_MOT_camera.expose('fluorescence_rb87', t, 'atoms')
1271        rb_central_MOT_trap_aom.enable(t)
1272        t += rb_fluoro_imaging_time
1273        rb_central_MOT_trap_aom.disable(t)
1274        t += interframe_time
1275        rb_central_MOT_trap_aom.enable(t)
1276        central_MOT_camera.expose('fluorescence_rb87', t, 'dark')
1277        rb_central_MOT_trap_aom.disable(t)
1278        t += red_camera_exposure_time
1279
1280    ### END Rb Central imaging ###
1281    ######## END CENTRAL IMAGING #########
1282
1283    ######################### END OF EXPERIMENT, NOW SET SOME SENSIBLE DEFAULTS
                                    ##################################
1284    if verbose: print "Experiment over, start cleanup at t = %s"%t
1285    t+=100e-3
1286    # close the imaging shutter
1287    central_MOT_imaging_shutter.close(t)
1288    science_bottom_imaging_shutter.close(t)
1289    # Turn the central coils on for a MOT, but leave the bias coils off (they might get a bit warm)
1290    sorensen_voltage_control.go_low(t)
1291    central_Bq.constant(t, rb_central_MOT_load_current, "A")
1292
1293    central_bias_x_coil_driver.constant(t, 0, "A")
1294    central_bias_y_coil_driver.constant(t, 0, "A")
1295    central_bias_z_coil_driver.constant(t, 0, "A")
1296
1297    feshbach_coils.constant(t,0)
1298
1299    # Turn the source MOT coils on
1300    k_source_MOT_west_coil.constant(t,k_source_current_west, "A")
1301    k_source_MOT_east_coil.constant(t, k_source_current_east, "A")
1302    k_source_MOT_top_coil.constant(t, k_source_current_top, "A")
1303    k_source_MOT_bottom_coil.constant(t, k_source_current_bottom, "A")
1304
1305    rb_source_MOT_coils.constant(t,rb_source_current,"A")
1306
1307    # Turn the source MOT lights on
1308    k_source_MOT_shutter.open(t)
1309
1310    k_MOT_trap_aom.enable(t)
1311    k_MOT_trap_aom.setfreq(t,k_trap_load_frequency * MHz)
1312    k_MOT_trap_aom.setamp(t,k_trap_load_amplitude / 1023.0)
1313
1314    k_MOT_repump_aom.enable(t)
1315    k_MOT_repump_aom.setfreq(t,k_repump_load_frequency * MHz)
1316    k_MOT_repump_aom.setamp(t,k_repump_load_amplitude / 1023.0)
1317
1318    rb_source_MOT_trap_aom.enable(t)
1319    rb_source_MOT_repump_aom.enable(t)
1320    rb_source_MOT_shutter.open(t)
1321
1322    # Set the push beams up
1323    k_imaging_push_trap_aom.setfreq(t, k_push_frequency * MHz)
1324    k_imaging_push_trap_aom.setamp(t, k_push_amplitude / 1023.0)
1325    k_imaging_push_trap_aom.enable(t)
1326
1327    k_imaging_push_repump_aom.setfreq(t, k_push_repump_frequency * MHz)
1328    k_imaging_push_repump_aom.setamp(t, k_push_repump_amplitude / 1023.0)
1329    k_imaging_push_repump_aom.enable(t)
1330    k_push_shutter.open(t)
1331
1332    rb_imaging_push_aom.setfreq(t,rb_push_frequency * MHz)
1333    rb_imaging_push_aom.setamp(t,rb_push_amplitude)
1334
1335    # turn on central MOT light
```

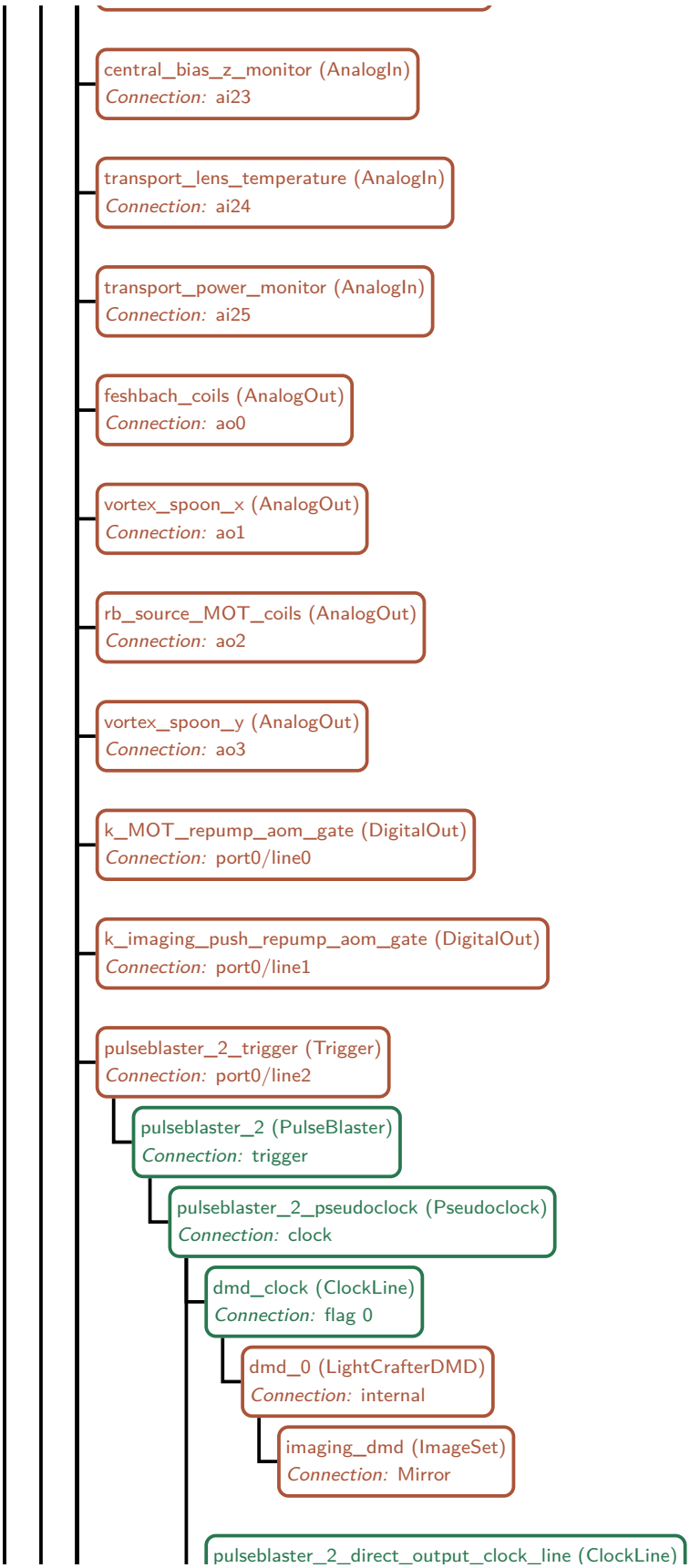
```
1336 central_MOT_shutter.open(t)
1337
1338 rb_central_MOT_trap_aom.enable(t)
1339 rb_central_MOT_trap_aom.setfreq(t,rb_central_MOT_trap_frequency * MHz)
1340 rb_central_MOT_trap_aom.setamp(t,rb_central_MOT_trap_amplitude / 1023.0)
1341
1342 rb_central_MOT_repump_aom.enable(t)
1343 rb_central_MOT_repump_aom.setfreq(t,rb_central_MOT_repump_frequency * MHz)
1344 rb_central_MOT_repump_aom.setamp(t,rb_central_MOT_repump_amplitude / 1023.0)
1345
1346 if trap_2d and transport_to_square_cell:
1347     flat_trap_aom.setamp(t,0)
1348
1349 if rotate_up and not rotate_down:
1350     stepper_direction.go_low(t)
1351     t+=0.1
1352     rotate_down_ts = 0.1 * stepper_ts
1353     for tt in rotate_down_ts:
1354         stepper_trigger.go_high(t+tt)
1355         stepper_trigger.go_low(t+tt+2.05e-6)
1356     t+= rotate_down_ts[-1]
1357
1358
1359 dipole_trap_1_aom.disable(t)
1360 dipole_trap_2_aom.disable(t)
1361
1362 t+=0.5
1363 dipole_1_dump_flipper.go_low(t)
1364
1365
1366 # turn off Feshbach coils
1367 feshbach_coils.constant(t,0,"A")
1368 t += 500e-6
1369 novatechdds9m_0_table_enable.go_low(t)
1370 novatechdds9m_1_table_enable.go_low(t)
1371 novatechdds9m_2_table_enable.go_low(t)
1372 novatechdds9m_3_table_enable.go_low(t)
1373
1374 # setup analog acquisitions for coil current monitoring
1375 # now that we know how long the experiment will be!
1376 monitor_coils(t,coil_dictionary)
1377
1378 stop(t + 200e-6)
```

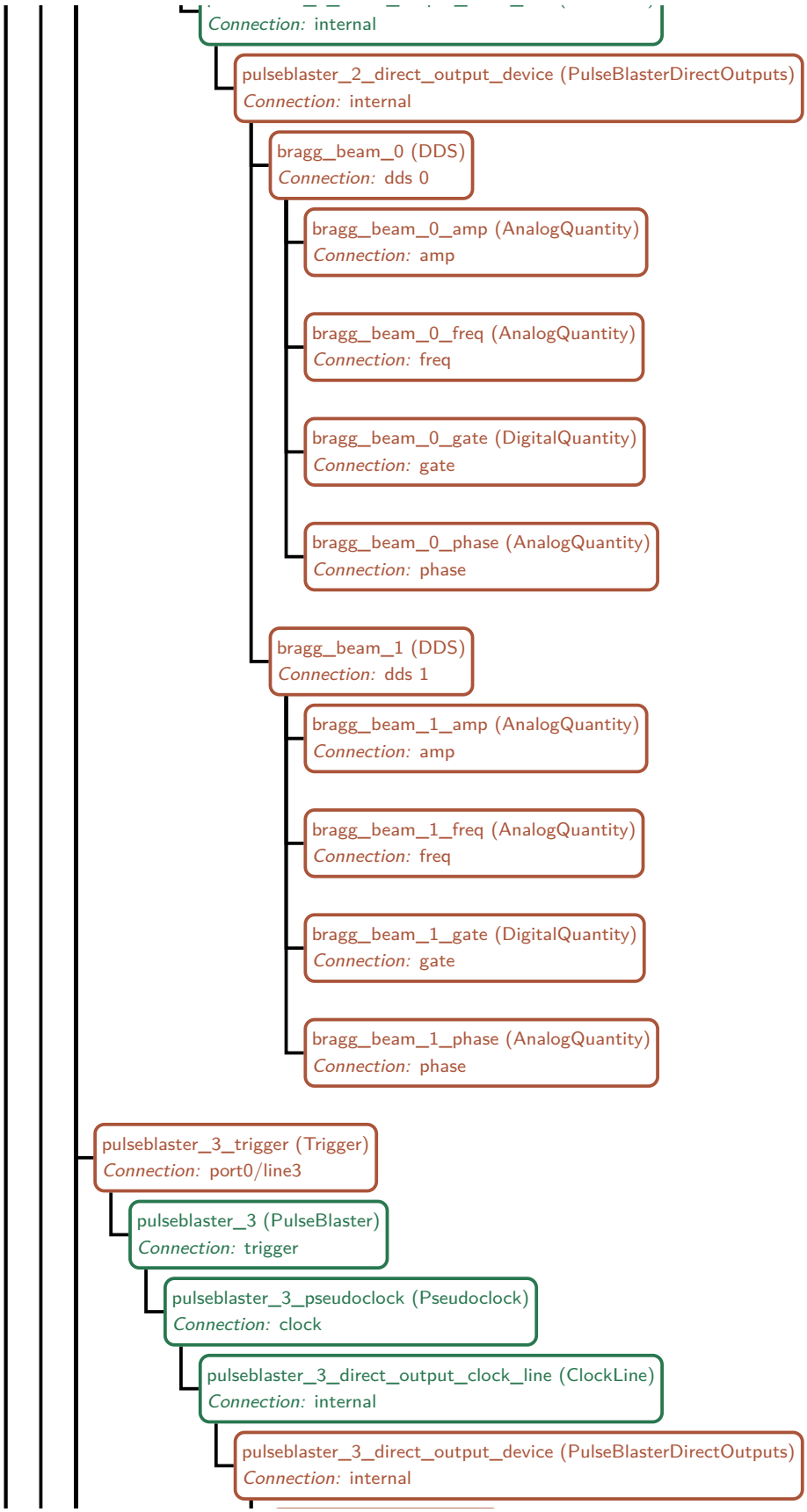
D.3 Device hierarchy

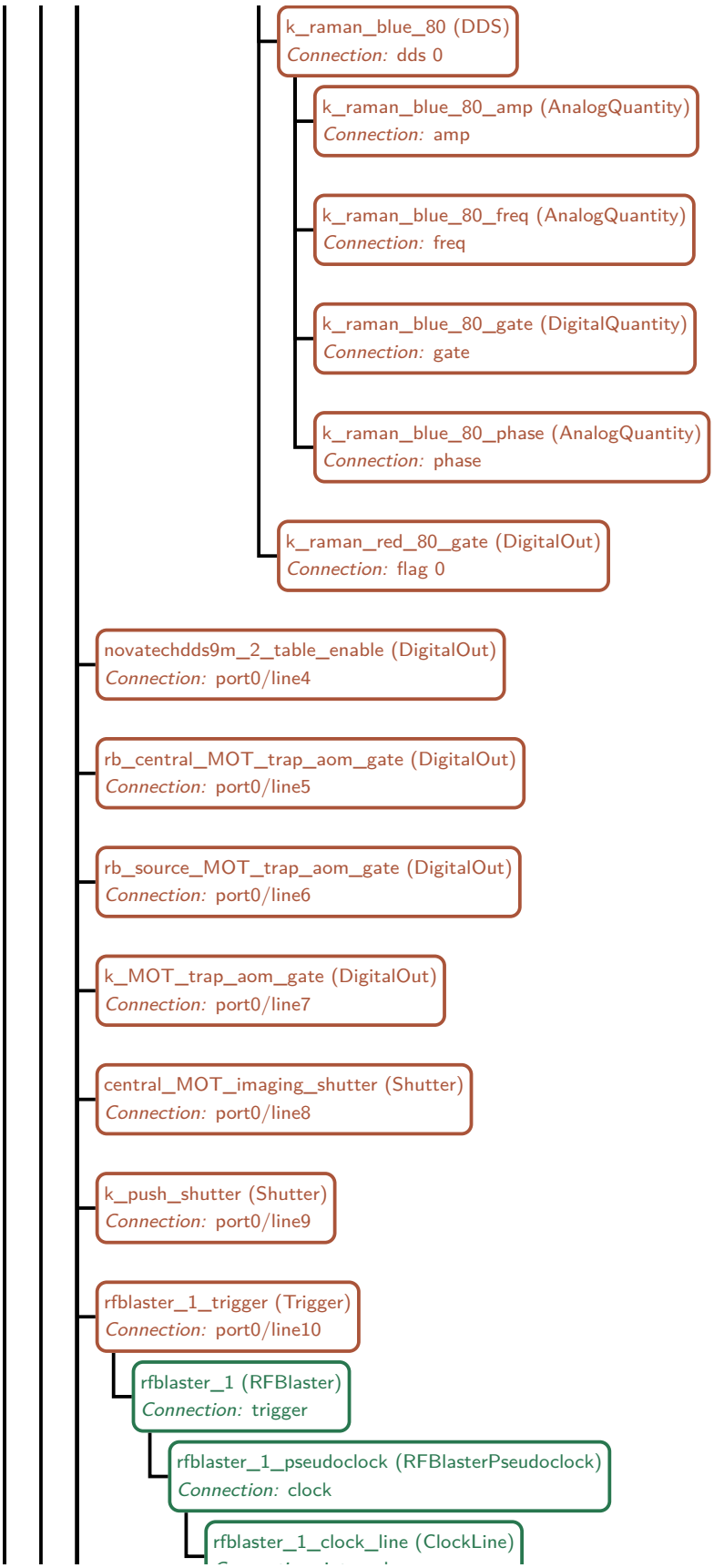
Here we show the device and channel hierarchy for the K-Rb lab. This contains all the necessary devices and I/O to run both the dual-species experiment detailed in §8.1 and the single-species experiment studying vortex dynamics detailed in §8.2. Note that the `RemoteStage` and `PhaseMatrixQuickSyn` devices only contain 'static' outputs that are updated once prior to the start of the experiment (and never during the experiment). As such, these devices are separated from the remainder of the device hierarchy, as they do not rely on the master pseudoclock (`pulseblaster_0`).

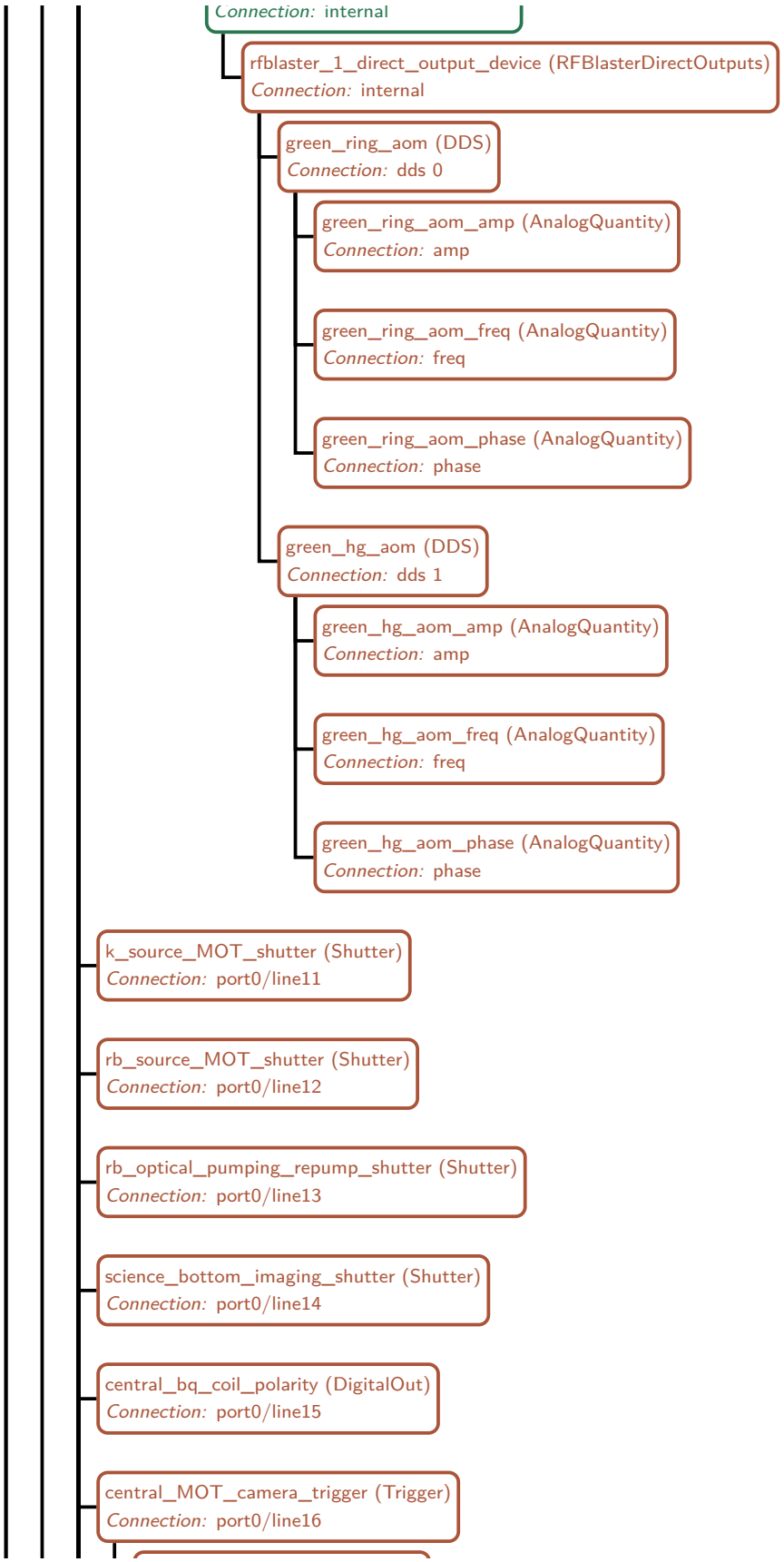


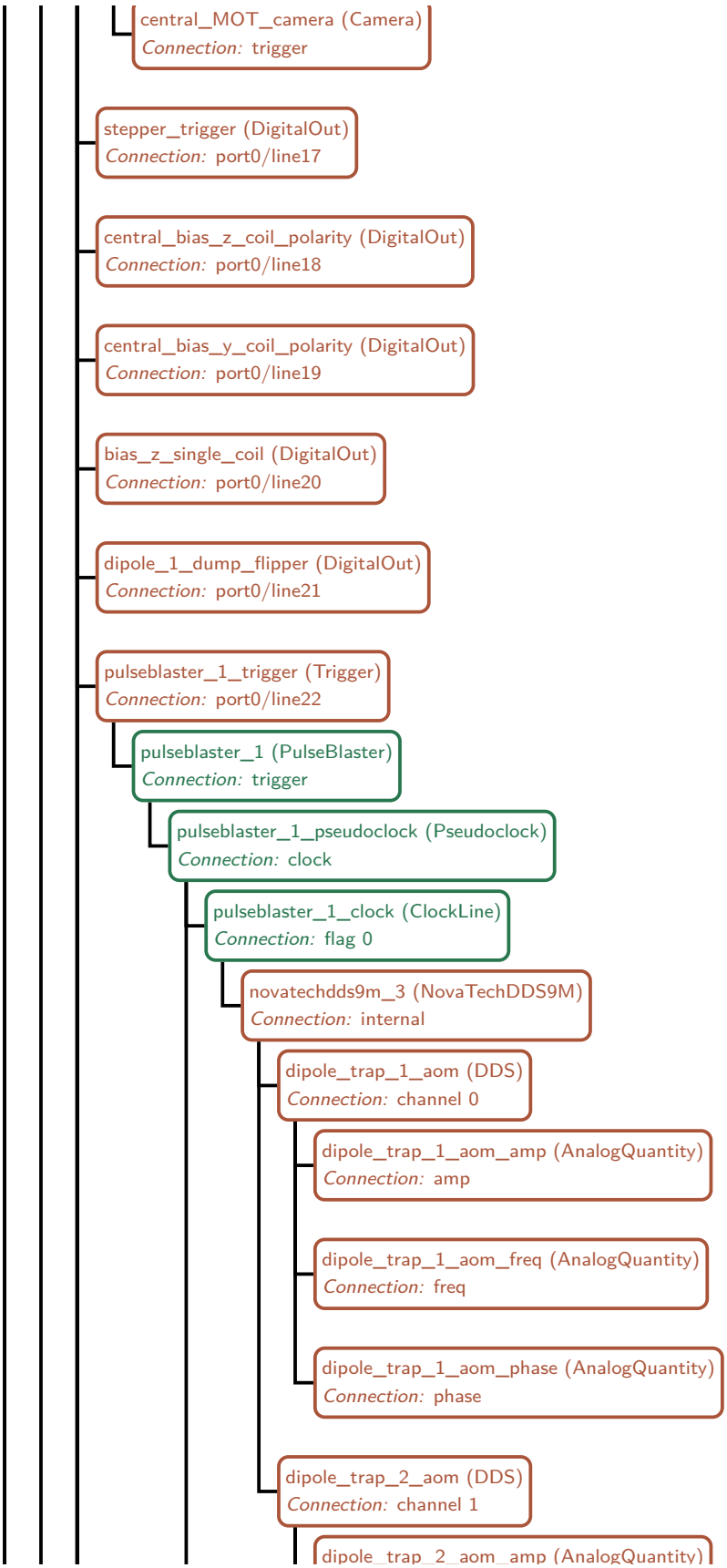

velocity (StaticAnalogQuantity)
Connection: v
phasematrix_0 (PhaseMatrixQuickSyn)
Connection: None
microwaves (QuickSynDDS)
Connection: dds 0
microwaves_freq (StaticAnalogQuantity)
Connection: freq
microwaves_gate (StaticDigitalOut)
Connection: gate
pulseblaster_0 (PulseBlaster)
Connection: None
pulseblaster_0_pseudoclock (Pseudoclock)
Connection: clock
pulseblaster_0_ni_clock (ClockLine)
Connection: flag 0
ni_pcie_6363_0 (NI_PCIe_6363)
Connection: internal
central_Bq_control_monitor (AnalogIn)
Connection: ai16
central_Bq_monitor (AnalogIn)
Connection: ai17
central_bias_y_control_monitor (AnalogIn)
Connection: ai18
central_bias_y_monitor (AnalogIn)
Connection: ai19
central_bias_x_control_monitor (AnalogIn)
Connection: ai20
central_bias_x_monitor (AnalogIn)
Connection: ai21
central_bias_z_control_monitor (AnalogIn)
Connection: ai22

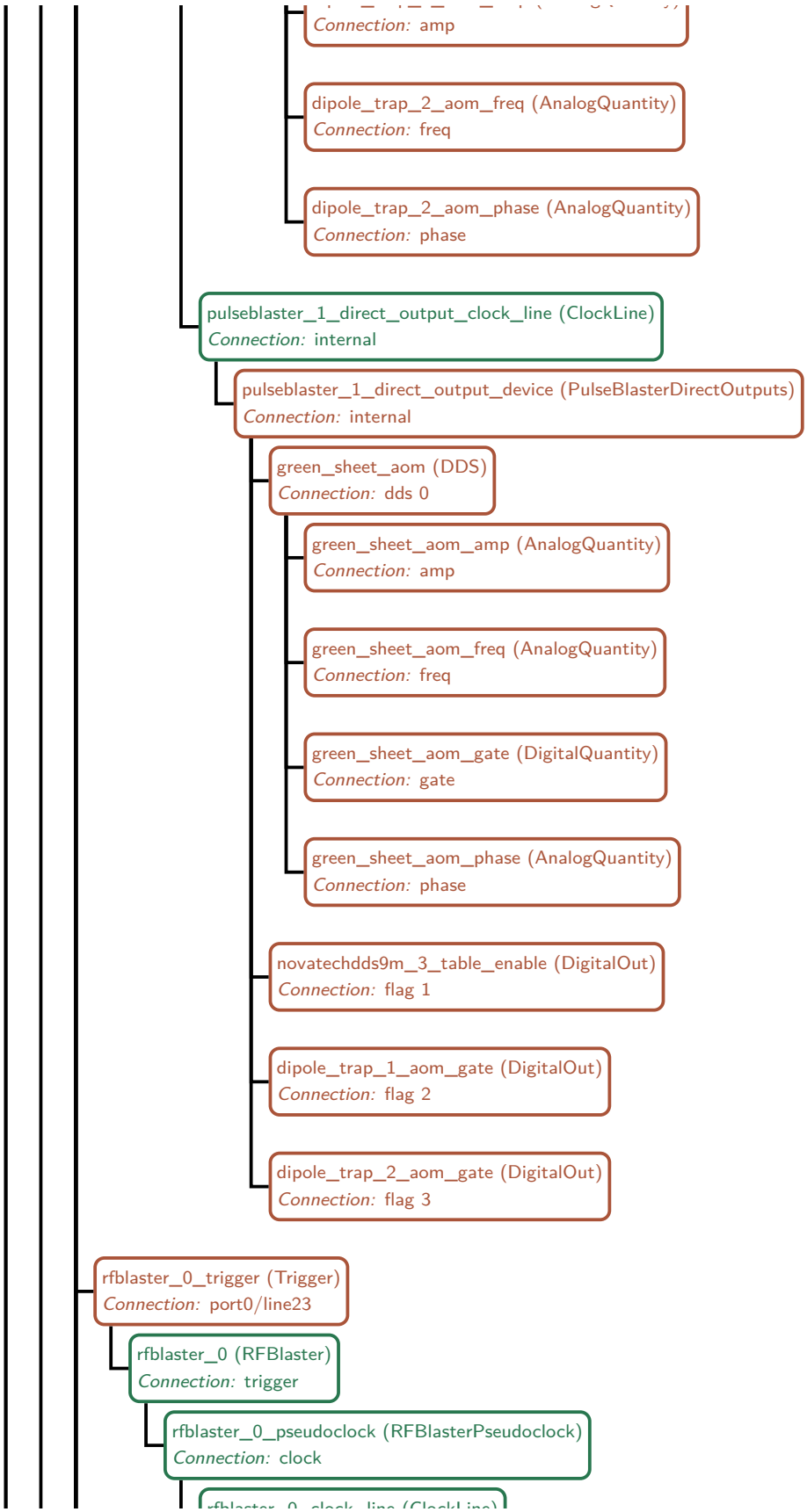

central_bias_z_monitor (AnalogIn)
Connection: ai23
transport_lens_temperature (AnalogIn)
Connection: ai24
transport_power_monitor (AnalogIn)
Connection: ai25
feshbach_coils (AnalogOut)
Connection: ao0
vortex_spoon_x (AnalogOut)
Connection: ao1
rb_source_MOT_coils (AnalogOut)
Connection: ao2
vortex_spoon_y (AnalogOut)
Connection: ao3
k_MOT_repump_aom_gate (DigitalOut)
Connection: port0/line0
k_imaging_push_repump_aom_gate (DigitalOut)
Connection: port0/line1
pulseblaster_2_trigger (Trigger)
Connection: port0/line2
pulseblaster_2 (PulseBlaster)
Connection: trigger
pulseblaster_2_pseudoclock (Pseudoclock)
Connection: clock
dmd_clock (ClockLine)
Connection: flag 0
dmd_0 (LightCrafterDMD)
Connection: internal
imaging_dmd (ImageSet)
Connection: Mirror
pulseblaster_2_direct_output_clock_line (ClockLine)

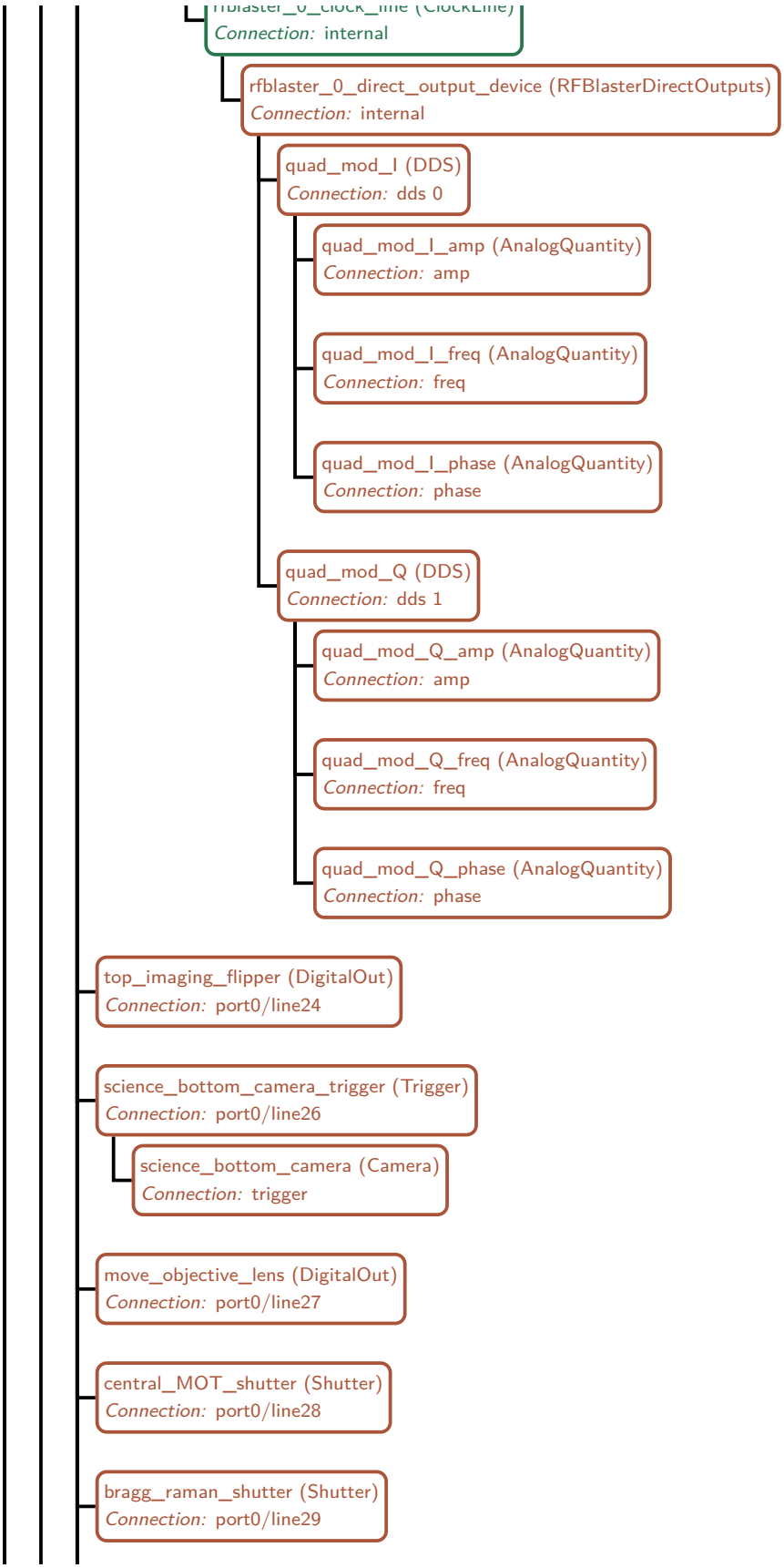

Connection: internal
pulseblaster_2_direct_output_device (PulseBlasterDirectOutputs)
Connection: internal
bragg_beam_0 (DDS)
Connection: dds 0
bragg_beam_0_amp (AnalogQuantity)
Connection: amp
bragg_beam_0_freq (AnalogQuantity)
Connection: freq
bragg_beam_0_gate (DigitalQuantity)
Connection: gate
bragg_beam_0_phase (AnalogQuantity)
Connection: phase
bragg_beam_1 (DDS)
Connection: dds 1
bragg_beam_1_amp (AnalogQuantity)
Connection: amp
bragg_beam_1_freq (AnalogQuantity)
Connection: freq
bragg_beam_1_gate (DigitalQuantity)
Connection: gate
bragg_beam_1_phase (AnalogQuantity)
Connection: phase
pulseblaster_3_trigger (Trigger)
Connection: port0/line3
pulseblaster_3 (PulseBlaster)
Connection: trigger
pulseblaster_3_pseudoclock (Pseudoclock)
Connection: clock
pulseblaster_3_direct_output_clock_line (ClockLine)
Connection: internal
pulseblaster_3_direct_output_device (PulseBlasterDirectOutputs)
Connection: internal

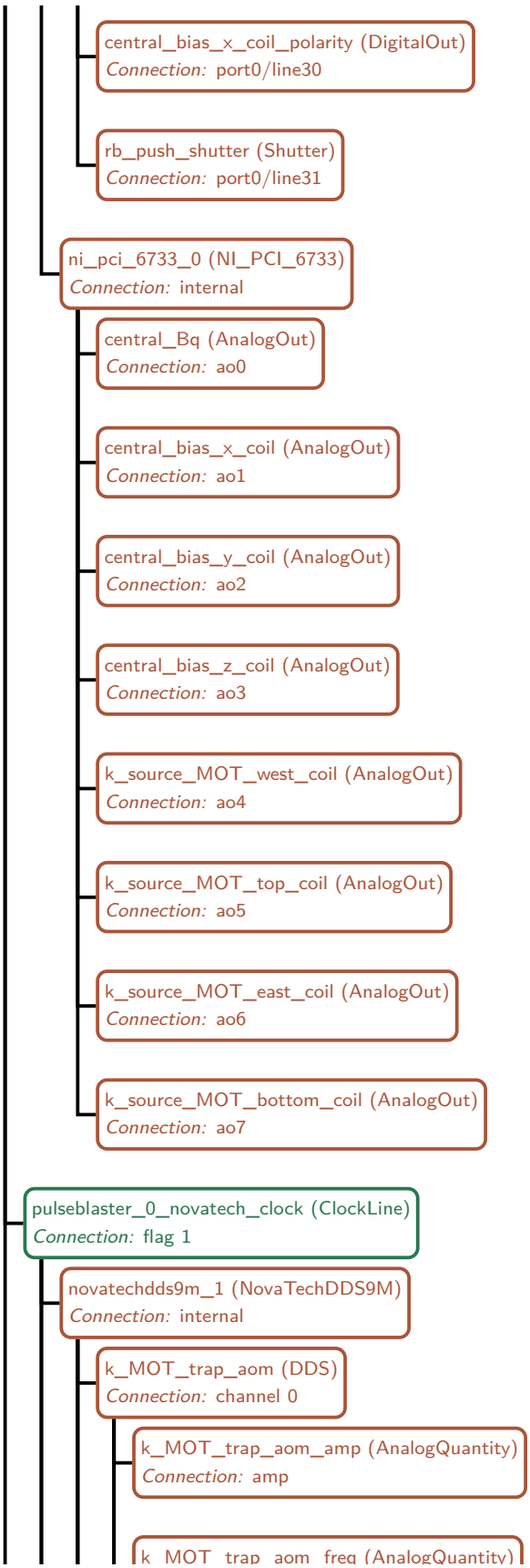

k_raman_blue_80 (DDS)
Connection: dds 0
k_raman_blue_80_amp (AnalogQuantity)
Connection: amp
k_raman_blue_80_freq (AnalogQuantity)
Connection: freq
k_raman_blue_80_gate (DigitalQuantity)
Connection: gate
k_raman_blue_80_phase (AnalogQuantity)
Connection: phase
k_raman_red_80_gate (DigitalOut)
Connection: flag 0
novatechdds9m_2_table_enable (DigitalOut)
Connection: port0/line4
rb_central_MOT_trap_aom_gate (DigitalOut)
Connection: port0/line5
rb_source_MOT_trap_aom_gate (DigitalOut)
Connection: port0/line6
k_MOT_trap_aom_gate (DigitalOut)
Connection: port0/line7
central_MOT_imaging_shutter (Shutter)
Connection: port0/line8
k_push_shutter (Shutter)
Connection: port0/line9
rfblaster_1_trigger (Trigger)
Connection: port0/line10
rfblaster_1 (RFBlaster)
Connection: trigger
rfblaster_1_pseudoclock (RFBlasterPseudoclock)
Connection: clock
rfblaster_1_clock_line (ClockLine)

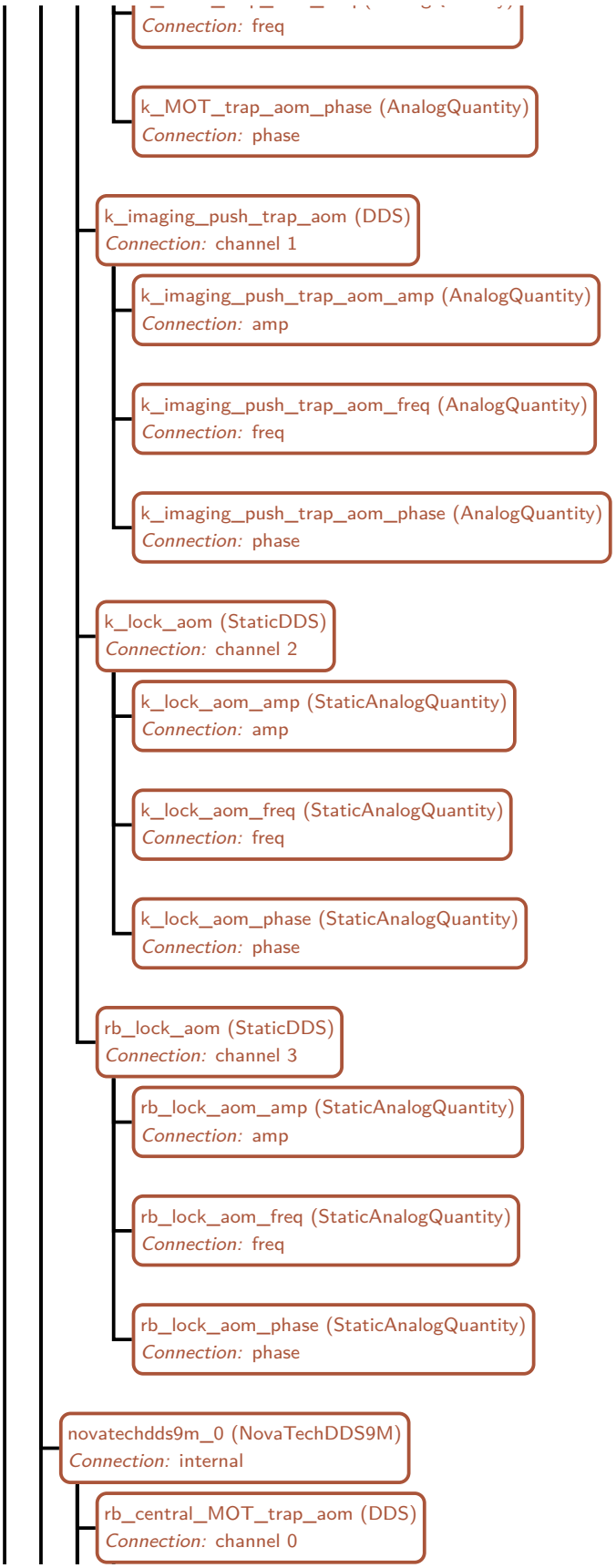

Connection: internal
rfblaster_1_direct_output_device (RFBlasterDirectOutputs)
Connection: internal
green_ring_aom (DDS)
Connection: dds 0
green_ring_aom_amp (AnalogQuantity)
Connection: amp
green_ring_aom_freq (AnalogQuantity)
Connection: freq
green_ring_aom_phase (AnalogQuantity)
Connection: phase
green_hg_aom (DDS)
Connection: dds 1
green_hg_aom_amp (AnalogQuantity)
Connection: amp
green_hg_aom_freq (AnalogQuantity)
Connection: freq
green_hg_aom_phase (AnalogQuantity)
Connection: phase
k_source_MOT_shutter (Shutter)
Connection: port0/line11
rb_source_MOT_shutter (Shutter)
Connection: port0/line12
rb_optical_pumping_repump_shutter (Shutter)
Connection: port0/line13
science_bottom_imaging_shutter (Shutter)
Connection: port0/line14
central_bq_coil_polarity (DigitalOut)
Connection: port0/line15
central_MOT_camera_trigger (Trigger)
Connection: port0/line16


central_MOT_camera (Camera)
Connection: trigger
stepper_trigger (DigitalOut)
Connection: port0/line17
central_bias_z_coil_polarity (DigitalOut)
Connection: port0/line18
central_bias_y_coil_polarity (DigitalOut)
Connection: port0/line19
bias_z_single_coil (DigitalOut)
Connection: port0/line20
dipole_1_dump_flipper (DigitalOut)
Connection: port0/line21
pulseblaster_1_trigger (Trigger)
Connection: port0/line22
pulseblaster_1 (PulseBlaster)
Connection: trigger
pulseblaster_1_pseudoclock (Pseudoclock)
Connection: clock
pulseblaster_1_clock (ClockLine)
Connection: flag 0
novatechdds9m_3 (NovaTechDDS9M)
Connection: internal
dipole_trap_1_aom (DDS)
Connection: channel 0
dipole_trap_1_aom_amp (AnalogQuantity)
Connection: amp
dipole_trap_1_aom_freq (AnalogQuantity)
Connection: freq
dipole_trap_1_aom_phase (AnalogQuantity)
Connection: phase
dipole_trap_2_aom (DDS)
Connection: channel 1
dipole_trap_2_aom_amp (AnalogQuantity)

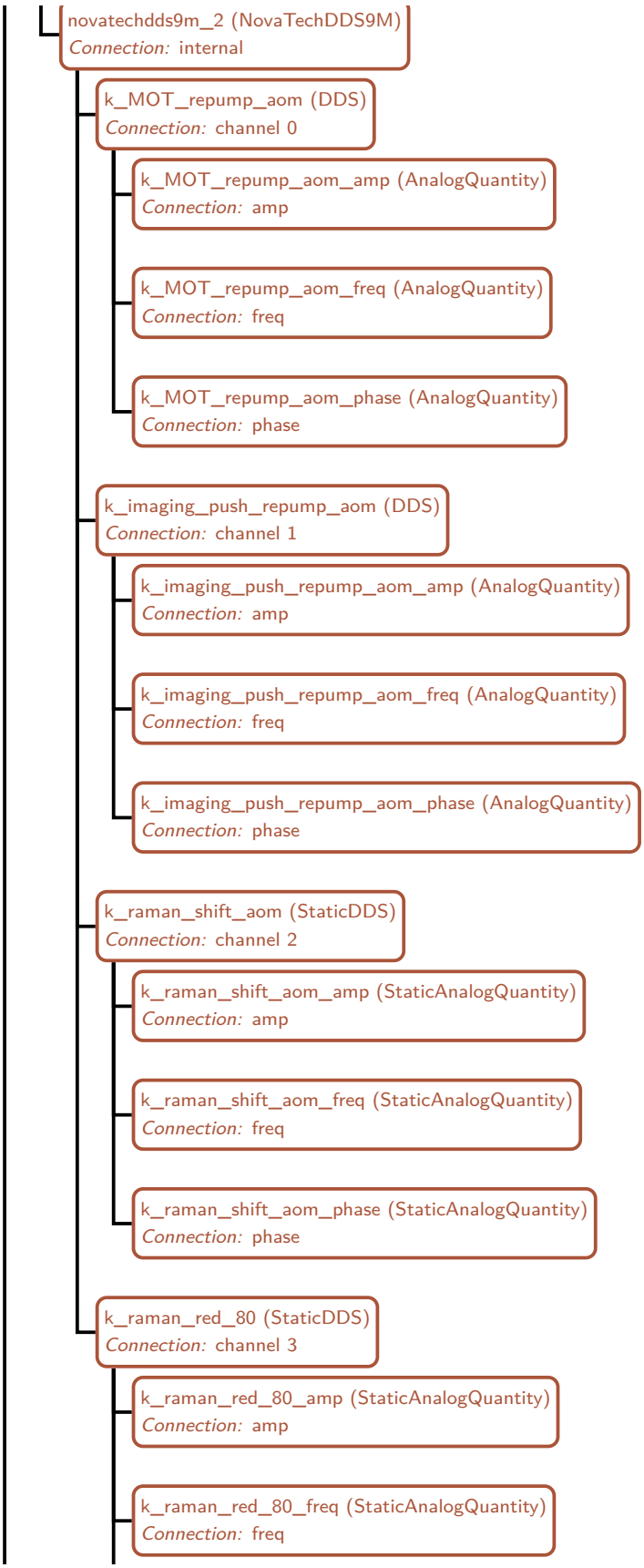

Connection: amp
dipole_trap_2_aom_freq (AnalogQuantity)
Connection: freq
dipole_trap_2_aom_phase (AnalogQuantity)
Connection: phase
pulseblaster_1_direct_output_clock_line (ClockLine)
Connection: internal
pulseblaster_1_direct_output_device (PulseBlasterDirectOutputs)
Connection: internal
green_sheet_aom (DDS)
Connection: dds 0
green_sheet_aom_amp (AnalogQuantity)
Connection: amp
green_sheet_aom_freq (AnalogQuantity)
Connection: freq
green_sheet_aom_gate (DigitalQuantity)
Connection: gate
green_sheet_aom_phase (AnalogQuantity)
Connection: phase
novatechdds9m_3_table_enable (DigitalOut)
Connection: flag 1
dipole_trap_1_aom_gate (DigitalOut)
Connection: flag 2
dipole_trap_2_aom_gate (DigitalOut)
Connection: flag 3
rfblaster_0_trigger (Trigger)
Connection: port0/line23
rfblaster_0 (RFBlaster)
Connection: trigger
rfblaster_0_pseudoclock (RFBlasterPseudoclock)
Connection: clock

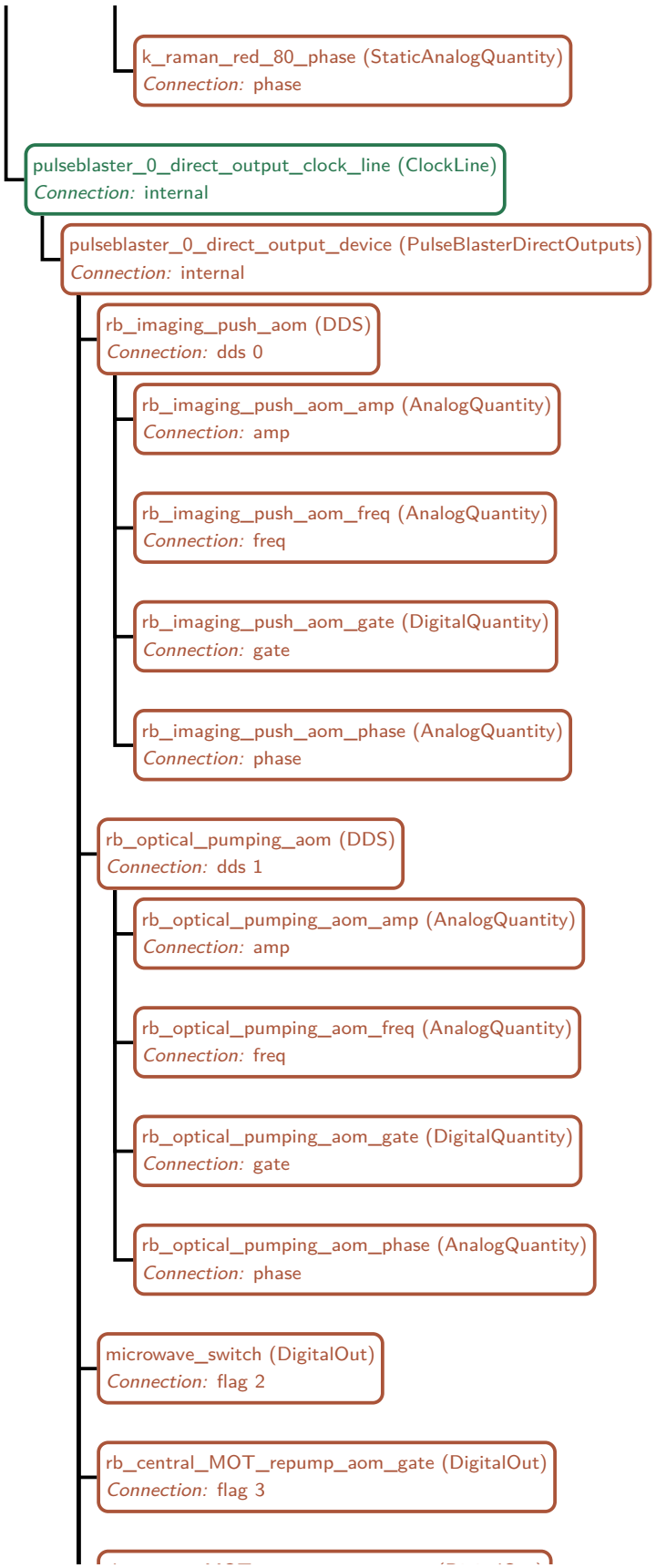

Connection: internal
rfblaster_0_direct_output_device (RFBlasterDirectOutputs)
Connection: internal
quad_mod_I (DDS)
Connection: dds 0
quad_mod_I_amp (AnalogQuantity)
Connection: amp
quad_mod_I_freq (AnalogQuantity)
Connection: freq
quad_mod_I_phase (AnalogQuantity)
Connection: phase
quad_mod_Q (DDS)
Connection: dds 1
quad_mod_Q_amp (AnalogQuantity)
Connection: amp
quad_mod_Q_freq (AnalogQuantity)
Connection: freq
quad_mod_Q_phase (AnalogQuantity)
Connection: phase
top_imaging_flipper (DigitalOut)
Connection: port0/line24
science_bottom_camera_trigger (Trigger)
Connection: port0/line26
science_bottom_camera (Camera)
Connection: trigger
move_objective_lens (DigitalOut)
Connection: port0/line27
central_MOT_shutter (Shutter)
Connection: port0/line28
bragg_raman_shutter (Shutter)
Connection: port0/line29

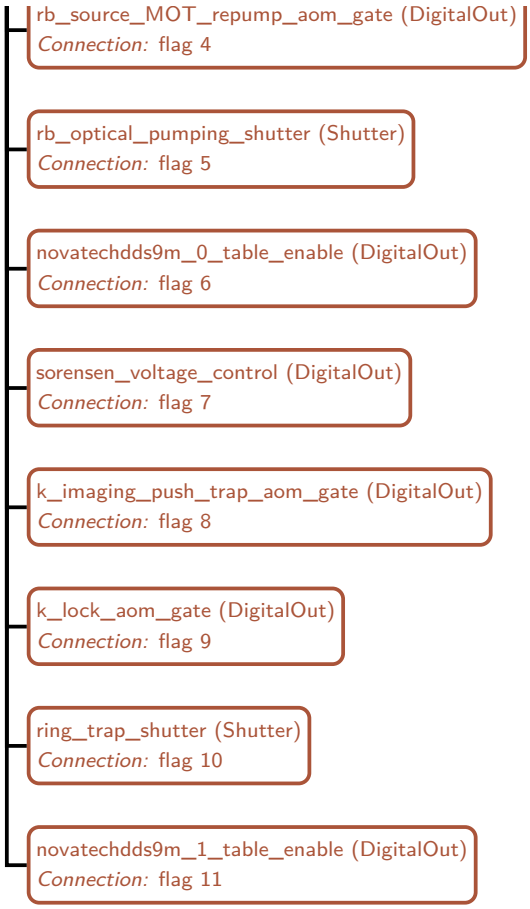

central_bias_x_coil_polarity (DigitalOut)
Connection: port0/line30
rb_push_shutter (Shutter)
Connection: port0/line31
ni_pci_6733_0 (NI_PCI_6733)
Connection: internal
central_Bq (AnalogOut)
Connection: ao0
central_bias_x_coil (AnalogOut)
Connection: ao1
central_bias_y_coil (AnalogOut)
Connection: ao2
central_bias_z_coil (AnalogOut)
Connection: ao3
k_source_MOT_west_coil (AnalogOut)
Connection: ao4
k_source_MOT_top_coil (AnalogOut)
Connection: ao5
k_source_MOT_east_coil (AnalogOut)
Connection: ao6
k_source_MOT_bottom_coil (AnalogOut)
Connection: ao7
pulseblaster_0_novatech_clock (ClockLine)
Connection: flag 1
novatechdds9m_1 (NovaTechDDS9M)
Connection: internal
k_MOT_trap_aom (DDS)
Connection: channel 0
k_MOT_trap_aom_amp (AnalogQuantity)
Connection: amp


Connection: freq
k_MOT_trap_aom_phase (AnalogQuantity)
Connection: phase
k_imaging_push_trap_aom (DDS)
Connection: channel 1
k_imaging_push_trap_aom_amp (AnalogQuantity)
Connection: amp
k_imaging_push_trap_aom_freq (AnalogQuantity)
Connection: freq
k_imaging_push_trap_aom_phase (AnalogQuantity)
Connection: phase
k_lock_aom (StaticDDS)
Connection: channel 2
k_lock_aom_amp (StaticAnalogQuantity)
Connection: amp
k_lock_aom_freq (StaticAnalogQuantity)
Connection: freq
k_lock_aom_phase (StaticAnalogQuantity)
Connection: phase
rb_lock_aom (StaticDDS)
Connection: channel 3
rb_lock_aom_amp (StaticAnalogQuantity)
Connection: amp
rb_lock_aom_freq (StaticAnalogQuantity)
Connection: freq
rb_lock_aom_phase (StaticAnalogQuantity)
Connection: phase
novatechdds9m_0 (NovaTechDDS9M)
Connection: internal
rb_central_MOT_trap_aom (DDS)
Connection: channel 0


rb_central_MOT_trap_aom_amp (AnalogQuantity)
Connection: amp
rb_central_MOT_trap_aom_freq (AnalogQuantity)
Connection: freq
rb_central_MOT_trap_aom_phase (AnalogQuantity)
Connection: phase
rb_central_MOT_repump_aom (DDS)
Connection: channel 1
rb_central_MOT_repump_aom_amp (AnalogQuantity)
Connection: amp
rb_central_MOT_repump_aom_freq (AnalogQuantity)
Connection: freq
rb_central_MOT_repump_aom_phase (AnalogQuantity)
Connection: phase
rb_source_MOT_repump_aom (StaticDDS)
Connection: channel 2
rb_source_MOT_repump_aom_amp (StaticAnalogQuantity)
Connection: amp
rb_source_MOT_repump_aom_freq (StaticAnalogQuantity)
Connection: freq
rb_source_MOT_repump_aom_phase (StaticAnalogQuantity)
Connection: phase
rb_source_MOT_trap_aom (StaticDDS)
Connection: channel 3
rb_source_MOT_trap_aom_amp (StaticAnalogQuantity)
Connection: amp
rb_source_MOT_trap_aom_freq (StaticAnalogQuantity)
Connection: freq
rb_source_MOT_trap_aom_phase (StaticAnalogQuantity)
Connection: phase


novatechdds9m_2 (NovaTechDDS9M)
Connection: internal
k_MOT_repump_aom (DDS)
Connection: channel 0
k_MOT_repump_aom_amp (AnalogQuantity)
Connection: amp
k_MOT_repump_aom_freq (AnalogQuantity)
Connection: freq
k_MOT_repump_aom_phase (AnalogQuantity)
Connection: phase
k_imaging_push_repump_aom (DDS)
Connection: channel 1
k_imaging_push_repump_aom_amp (AnalogQuantity)
Connection: amp
k_imaging_push_repump_aom_freq (AnalogQuantity)
Connection: freq
k_imaging_push_repump_aom_phase (AnalogQuantity)
Connection: phase
k_raman_shift_aom (StaticDDS)
Connection: channel 2
k_raman_shift_aom_amp (StaticAnalogQuantity)
Connection: amp
k_raman_shift_aom_freq (StaticAnalogQuantity)
Connection: freq
k_raman_shift_aom_phase (StaticAnalogQuantity)
Connection: phase
k_raman_red_80 (StaticDDS)
Connection: channel 3
k_raman_red_80_amp (StaticAnalogQuantity)
Connection: amp
k_raman_red_80_freq (StaticAnalogQuantity)
Connection: freq


k_raman_red_80_phase (StaticAnalogQuantity)
Connection: phase
pulseblaster_0_direct_output_clock_line (ClockLine)
Connection: internal
pulseblaster_0_direct_output_device (PulseBlasterDirectOutputs)
Connection: internal
rb_imaging_push_aom (DDS)
Connection: dds 0
rb_imaging_push_aom_amp (AnalogQuantity)
Connection: amp
rb_imaging_push_aom_freq (AnalogQuantity)
Connection: freq
rb_imaging_push_aom_gate (DigitalQuantity)
Connection: gate
rb_imaging_push_aom_phase (AnalogQuantity)
Connection: phase
rb_optical_pumping_aom (DDS)
Connection: dds 1
rb_optical_pumping_aom_amp (AnalogQuantity)
Connection: amp
rb_optical_pumping_aom_freq (AnalogQuantity)
Connection: freq
rb_optical_pumping_aom_gate (DigitalQuantity)
Connection: gate
rb_optical_pumping_aom_phase (AnalogQuantity)
Connection: phase
microwave_switch (DigitalOut)
Connection: flag 2
rb_central_MOT_repump_aom_gate (DigitalOut)
Connection: flag 3


rb_source_MOT_repump_aom_gate (DigitalOut)
Connection: flag 4
rb_optical_pumping_shutter (Shutter)
Connection: flag 5
novatechdds9m_0_table_enable (DigitalOut)
Connection: flag 6
sorensen_voltage_control (DigitalOut)
Connection: flag 7
k_imaging_push_trap_aom_gate (DigitalOut)
Connection: flag 8
k_lock_aom_gate (DigitalOut)
Connection: flag 9
ring_trap_shutter (Shutter)
Connection: flag 10
novatechdds9m_1_table_enable (DigitalOut)
Connection: flag 11

Appendix E

y_vs_auto.py lyse analysis script

```
from __future__ import print_function
from lyse import *
from pylab import *
from analysislib.krb import aliases
import six
from analysislib.common.imshow_irreg import imshow_irreg
import matplotlib.cm as cm

def _ensure_str(s):
    """convert bytestrings and numpy strings to python strings"""
    return s.decode() if isinstance(s, bytes) else str(s)

def get_alias(module, alias, default):
    if isinstance(default, six.string_types) or isinstance(default, bytes):
        default = _ensure_str(default)
    if isinstance(alias, six.string_types) or isinstance(alias, bytes):
        alias = _ensure_str(alias)
        return getattr(module, alias, default)
    return default

# Get the dataframe from lyse
df = data(timeout=10)

# Get a list of all unique shot sequences
# Each sequence may contain multiple shots (depending on the parameter space
# scan)
sequences = [os.path.split(path)[1][0:15] for path in df['filepath']]
sequences_unique = intersect1d(sequences, sequences)

# Extract the number of sequences to analyse from the last shot received by
# lyse
no_seq = df["no_sequences_to_analyse"].values[-1]

# slice the DataFrame so that we only have the last N sequences
# where N is equal to no_seq above
df = df[[sequence in sequences_unique[-(no_seq):] for sequence in sequences]]
# Generate a plot title that contains the sequences we are plotting
plot_title = ''.join([x + ', ' for x in sequences_unique[-(no_seq):]]) + '\n\n'

# Pull out global information from the last shot
last_run = Run(df['filepath'][-1])
expansions = last_run.get_globals_expansion()
units = last_run.get_units()

# iterate over the expansion type of every global
# detect which global variables form the axes of our parameter space
independents = []
both_species = False
image_order_global_name = None
for global_name, expansion_type in expansions.items():
    # This global indicates we want to create separate plots for each species
    # of atom we image. It should not be used as an axis of the parameter
    # space.
    #
    # Note: 'central_image_order' is a historical global we keep here for
    # backwards compatibility with old shot files
    if global_name in ['image_order', 'central_image_order']:
        image_order_global_name = global_name
        both_species = True
        continue

    # This global is not to be used as an axis of the parameter space.
    # We use it to "trick" runmanager into generating multiple identical shots
```

```
64      # so that we can see the variation of measured quantities
65      elif global_name == 'repeats':
66          continue
67
68      # If the expansion type is a zip group (not "outer")
69      # Then we need to check if it is a zip group that we can automatically
70      # handle
71      if expansion_type != "outer":
72          # If the expansion type matches the global name, then it is an
73          # automatically created zip group, and we use this value as the axis
74          # of our parameter space as it is the root global of the parameter
75          # space
76          if expansion_type == global_name:
77              independents.append(global_name)
78
79          # If the zip group name matches another global with an expansion type,
80          # then we ignore it because we'll catch the root global and use that
81          # as the axis of the parameter space
82          elif expansion_type in expansions and expansions[expansion_type] == expansion_type:
83              pass
84
85          # If it isn't the two above cases, then it means we have a custom
86          # named zip group which we can't yet handle nicely as we don't know
87          # which global to use as the parameter space axis
88          else:
89              raise Exception(
90                  'This sequence contains globals in a zip group that was not automatically created by runmanager, please use a
                                    different plotting script')
91
92      # it's an outer product (not a zip group) then use it as an axis of our
93      # parameter space
94      else:
95          independents.append(global_name)
96
97  # We can only handle up to a 2D parameter space as plotting anything else is
98  # tricky
99  if len(independents) > 2:
100     print(independents)
101     raise Exception(
102         'There are %s orthogonal globals changing, but this script can only plot up to 2. Please try a universe with more
                                    spatial dimensions.' %
103         len(independents))
104
105
106 #
107 # Define a function that can take a DataFrame (or slice of a DataFrame)
108 # and create a 1D or 2D plot (depending on the dimension of the parameter
109 # space) for each measurement result specified in a list
110 #
111 #   df: The DataFrame
112 #
113 #   plot_measurement: The list of measurements (this determine the number of
114 #                   plots produced). This list should conatin either a string
115 #                   or a tuple that will be used to index the DataFrame.
116 #                   Strings will also be first checked to see if they exist
117 #                   as an attribute of the aliases module, and the value of
118 #                   the alias will be used as the DataFrame index instead
119 #
120 def y_vs_auto(df, plot_measurement, independents):
121     # if plot_measurement is only a single string, convert it to a one item
122     # long list
123     if type(plot_measurement) in (str, string_):
124             plot_measurement = [plot_measurement]
125
126     # if we are not performing a parameter space scan, then we likely have a
127     # single shot on repeat. So do a y_vs_shot style plot
128     if len(independents) == 0:
129         # Iterate over the list of plot measurements (the measurements to
130         # use on the y-axis of a plot)
131         for i in plot_measurement:
132             # Extract the y values from the dataFrame based on the current
133             # plot measurement we are iterating over
134             y = df[get_alias(aliases, i, i)]
135
136             # Create the figure
137             figure()
138
139             # plot the y values
140             plot(y.values, "-", linewidth=2)
141
142             # Add labels to the plot axes and a title
143             xlabel("Shot")
144             ylabel(_ensure_str(i))
145             title(plot_title)
146
147             # Print out the mean and standard deviation of the data
148             # This provides a useful measure of the stability of the measured
149             # quantity
150             print("Mean %s = %0.3e +/- %0.3e (%s)" %
```

```
151                 (__ensure__str(i), mean(y), std(y), std(y) / mean(y) * 100))
152
153     # if we are performing a 1D parameter space scan we do a y__vs__x style plot
154     elif len(independents) == 1:
155         # Define the x-axis of the plot to match the parameter space axis
156         x__axis = independents[0]
157         x = df[x__axis]
158
159         # Iterate over the list of plot measurements (the measurements to
160         # use on the y-axis of a plot)
161         for i in plot__measurement:
162             # Extract the y values from the dataFrame based on the current
163             # plot measurement we are iterating over
164             y = df[get__alias(aliases, i, i)]
165
166             # Create the figure
167             figure()
168
169             # Create a set of plot colours, one for each separate sequence
170             # we are plotting
171             colors = cm.rainbow(np.linspace(0, 1, no__seq + 1))
172             color__i = 0
173
174             # Group by the sequence name and plot each sequence with a
175             # different colour
176             for name, group in df.groupby(df[['sequence']].values[:]):
177                 # for name, group in
178                 # df.groupby('sequence',as__index=False):
179                 x__t = group[x__axis]
180                 y__t = group[get__alias(aliases, i, i)]
181                 print(type(name))
182                 scatter(x__t, y__t, color=colors[color__i], label=name.strftime("%Y%m%dT%H%M%S"))
183                 color__i += 1
184
185             # Add labels to the plot axes and a title
186             title(plot__title)
187             xlabel("%s (%s)" % (x__axis, units[x__axis]))
188             ylabel(__ensure__str(i))
189
190             # Calculate averages for points in the paramater space that
191             # were run more than once (aka repeat shots)
192             avg__x = []
193             avg__y = []
194             avg__u__y = []
195             # Slice the DataFrame based on x-axis (parameter space axis)
196             for name, group in df.groupby(x__axis):
197                 # if there is more than one shot for this point in
198                 # parameter space, then calculate the mean and
199                 # standard deviation and store everything in lists
200                 if len(group) > 1:
201                     avg__x.append(
202                         group[x__axis][0])
203                     y = group[get__alias(aliases, i, i)]
204                     avg__y.append(mean(y))
205                     avg__u__y.append(std(y))
206
207             # If there is a parameter space point that has more than one
208             # shot, then make an error bar plot as well (overlays on the
209             # existing plot)
210             if avg__x:
211                 errorbar(avg__x, avg__y, yerr=avg__u__y, c="red", fmt="o", label="mean +/- std")
212                 max__y = max(avg__y)
213                 max__index = avg__y.index(max__y)
214                 # Print out statistics for this plot
215                 #       max_at:   location in parameter space of maximum value
216                 #       value is: average y_value at this point with
217                 #                 standard deviation if more than one shot
218                 #                 was run for this point in parameter space
219                 print('%s max at: %s=%0.3e, value is %0.3e +/- %0.3e (%3.1f)' %
220                       (__ensure__str(i), x__axis, avg__x[max__index], avg__y[max__index],
221                        avg__u__y[max__index],
222                        avg__u__y[max__index] / avg__y[max__index] * 100))
223
224             # now that all plotting is complete, show the legend
225             legend(loc=0)
226
227             # Adust the x-axis limits so that all datapoints are visible rather
228             # than a couple being stuck right on the edge
229             if len(x) > 1:
230                 xlim(x.min() - (x.max() - x.min()) / 10.0,
231                      x.max() + (x.max() - x.min()) / 10.0)
232
233
234     # if two things are changing then we do an imshow
235     elif len(independents) == 2:
236         # Define the x-axis of the plot to match the first parameter space axis
237         x__axis = independents[0]
238         x = array(df[x__axis])
239         # Define the y-axis of the plot to match the second parameter space
```

```
240         # axis
241         y_axis = independents[1]
242         y = array(df[y_axis])
243 
244         # Iterate over the list of plot measurements (the measurements to
245         # use on the z-axis of a plot)
246         for i in plot_measurement:
247             # Extract the z values from the dataFrame based on the current
248             # plot measurement we are iterating over
249             z = array(df[get_alias(aliases, i, i)])
250 
251             # Create the figure
252             figure()
253 
254             # Make the 2D plot using a custom version of matplotlib.imshow
255             # which can handle irregularly spaced data in the x-y parameter
256             # space. Regions in (x,y) space are coloured based on the z-value
257             # of the nearest (x,y,z) point
258             imshow_irreg(x, y, z,
259                          method='nearest', aspect='auto', cmap="Blues")
260 
261             # Add labels to the plot axes and a title
262             xlabel("%s (%s)" % (x_axis, units[x_axis]))
263             ylabel("%s (%s)" % (y_axis, units[y_axis]))
264             title("%s\n%s" % (_ensure_str(i), plot_title[:-2]))
265             # add a colour bar to the 2D plot
266             colorbar()
267 
268             # Overlay a scatter plot on the 2D plot that shows the actual
269             # locations of the (x,y,z) values on the (x,y) plane within the
270             # coloured regions provided by imshow.
271             scatter(x, y)
272 
273             # print out the parameter space corrdinates which contain the
274             # maximum value of the measurement
275             z_arg_max = z.argmax()
276             print('%s max at: %s=%0.3e, %s=%0.3e' %
277                   (_ensure_str(i), x_axis, x[z_arg_max], y_axis, y[z_arg_max]))
278 
279     # print line to break up all print statements from this run from those
280     # of the next run
281     print('')
282 
283 
284 #
285 # The below code calls the above y_vs_auto function
286 #
287 
288 # If we are imaging both species, then group by the image type
289 if both_species:
290     # The global "image_order" contains either True or False for imaging
291     # rubidium and potassium respectively
292     # This results in slicing the DataFrame in 2
293     for key, subdf in df.groupby(image_order_global_name):
294         # We extract the list of measurements to plot on the y-axis of graphs
295         # from a runmanager global that is stored in the DataFrame
296         # The set of meaurements is dependent on the species
297         plot_measurement = subdf["plot_measurement_rb"].values[-1] if key else subdf["plot_measurement_k"].values[-1]
298         # Call y_vs_auto function
299         y_vs_auto(subdf, plot_measurement, independents)
300 
301 # If we are only imaging one species in our sequence
302 else:
303     # Check if we are doing a potassium absorption image and use the
304     # appropriate y-axis values
305     if 'absorption_image_k' in df.columns and df['absorption_image_k'].values[-1]:
306         y_vs_auto(df, df["plot_measurement_k"].values[-1], independents)
307     # Check if we are doing a rubidium absorption image and use the appropriate
308     # y-axis values
309     elif 'absorption_image_rb' in df.columns and df['absorption_image_rb'].values[-1]:
310         y_vs_auto(df, df["plot_measurement_rb"].values[-1], independents)
311     # otherwise we must be doing something else (like a fluorescence image)
312     # so use a separate set of y-axis values
313     else:
314         y_vs_auto(df, df["plot_measurement"].values[-1], independents)
```

References

[1] W. Bolton. *Programmable Logic Controllers*. Butterworth-Heinemann (1997). [p 2]

[2] J. J. García-Ripoll, P. Zoller, and J. I. Cirac. *Quantum information processing with cold atoms and trapped ions*. Journal of Physics B: Atomic, Molecular and Optical Physics **38**, S567 (2005). DOI: 10.1088/0953-4075/38/9/008. [p 4]

[3] P.-I. Schneider and A. Saenz. *Quantum computation with ultracold atoms in a driven optical lattice*. Physical Review A **85**, 050304 (2012). DOI: 10.1103/PhysRevA.85.050304. [p 4]

[4] N. Robins, P. Altin, J. Debs, and J. Close. *Atom lasers: Production, properties and prospects for precision inertial measurement*. Physics Reports **529**, 265 (2013). DOI: 10.1016/j.physrep.2013.03.006. [p 4]

[5] C. Gross and I. Bloch. *Quantum simulations with ultracold atoms in optical lattices*. Science **357**, 995 (2017). DOI: 10.1126/science.aal3837. [p 4]

[6] D. A. W. Hutchinson. *Ultracold atoms for simulation of many body quantum systems*. Journal of Physics: Conference Series **793**, 012009 (2017). DOI: 10.1088/1742-6596/793/1/012009. [p 4]

[7] S. Eckel, A. Kumar, T. Jacobson, I. B. Spielman, and G. K. Campbell. *A rapidly expanding Bose–Einstein condensate: An expanding universe in the lab*. Physical Review X **8**, 021021 (2018). DOI: 10.1103/PhysRevX.8.021021. [p 4]

[8] P. T. Starkey, C. J. Billington, S. P. Johnstone, M. Jasperse, K. Helmerson, L. D. Turner, and R. P. Anderson. *A scripted control system for autonomous hardware-timed experiments*. Review of Scientific Instruments **84**, 085111 (2013). DOI: 10.1063/1.4817213. [pp 4, 24, 64, 67, 80, 123, and 141]

[9] labscript_suite — Bitbucket. https://bitbucket.org/labscript_suite/, (2018). [pp 4 and 55]

[10] D. Robinson. The incredible growth of Python. https://stackoverflow.blog/2017/09/06/incredible-growth-python/, (2017). [p 5]

[11] D. Robinson. Why is Python growing so quickly? https://stackoverflow.blog/2017/09/14/python-growing-quickly/, (2017). [p 5]

[12] S. Donnellan, I. R. Hill, W. Bowden, and R. Hobson. *A scalable arbitrary waveform generator for atomic physics experiments based on field-programmable gate array technology.* Review of Scientific Instruments **90**, 043101 (2019). DOI: 10.1063/1.5051124. [p 6]

[13] Cold atom laboratory (CAL) - NASA jet propulsion laboratory. https://coldatomlab.jpl.nasa.gov/, (2018). [p 6]

[14] C. C. Bradley, C. A. Sackett, J. J. Tollett, and R. G. Hulet. *Evidence of Bose–Einstein condensation in an atomic gas with attractive interactions.* Physical Review Letters **75**, 1687 (1995). DOI: 10.1103/PhysRevLett.75.1687. [p 9]

[15] K. B. Davis, M. O. Mewes, M. R. Andrews, N. J. van Druten, D. S. Durfee, D. M. Kurn, and W. Ketterle. *Bose–Einstein condensation in a gas of sodium atoms.* Physical Review Letters **75**, 3969 (1995). DOI: 10.1103/PhysRevLett.75.3969. [pp 9 and 11]

[16] M. H. Anderson, J. R. Ensher, M. R. Matthews, C. E. Wieman, and E. A. Cornell. *Observation of Bose–Einstein condensation in a dilute atomic vapor.* Science **269**, 198 (1995). DOI: 10.1126/science.269.5221.198. [pp 9 and 11]

[17] G. Modugno, G. Ferrari, G. Roati, R. J. Brecha, A. Simoni, and M. Inguscio. *Bose–Einstein condensation of potassium atoms by sympathetic cooling.* Science **294**, 1320 (2001). DOI: 10.1126/science.1066687. [p 9]

[18] D. G. Fried, T. C. Killian, L. Willmann, D. Landhuis, S. C. Moss, D. Kleppner, and T. J. Greytak. *Bose–Einstein condensation of atomic hydrogen.* Physical Review Letters **81**, 3811 (1998). DOI: 10.1103/PhysRevLett.81.3811. [p 9]

[19] A. Robert, O. Sirjean, A. Browaeys, J. Poupard, S. Nowak, D. Boiron, C. I. Westbrook, and A. Aspect. *A Bose–Einstein condensate of metastable atoms.* Science (2001). DOI: 10.1126/science.1060622. [p 9]

[20] F. Pereira Dos Santos, J. Léonard, J. Wang, C. J. Barrelet, F. Perales, E. Rasel, C. S. Unnikrishnan, M. Leduc, and C. Cohen-Tannoudji. *Bose–Einstein condensation of metastable helium.* Physical Review Letters **86**, 3459 (2001). DOI: 10.1103/PhysRevLett.86.3459. [p 9]

[21] A. Griesmaier, J. Werner, S. Hensler, J. Stuhler, and T. Pfau. *Bose–Einstein condensation of chromium.* Physical Review Letters **94**, 160401 (2005). DOI: 10.1103/PhysRevLett.94.160401. [p 9]

[22] S. Stellmer, M. K. Tey, B. Huang, R. Grimm, and F. Schreck. *Bose–Einstein condensation of strontium.* Physical Review Letters **103**, 200401 (2009). DOI: 10.1103/PhysRevLett.103.200401. [p 9]

[23] Y. Takasu, K. Maki, K. Komori, T. Takano, K. Honda, M. Kumakura, T. Yabuzaki, and Y. Takahashi. *Spin-singlet Bose–Einstein condensation of two-electron atoms.* Physical Review Letters **91**, 040404 (2003). DOI: 10.1103/PhysRevLett.91.040404. [p 9]

[24] K. Aikawa, A. Frisch, M. Mark, S. Baier, A. Rietzler, R. Grimm, and F. Ferlaino. *Bose–Einstein condensation of erbium.* Physical Review Letters **108**, 210401 (2012). DOI: 10.1103/PhysRevLett.108.210401. [p 9]

[25] P. Zeeman. *The effect of magnetisation on the nature of light emitted by a substance.* Nature **55**, 347 (1897). DOI: 10.1038/055347a0. [p 9]

[26] C. J. Foot. *Atomic physics.* Oxford master series in physics. Oxford University Press (2005). [p 9]

[27] V. S. Bagnato, L. G. Marcassa, S. G. Miranda, S. R. Muniz, and A. L. de Oliveira. *Measuring the capture velocity of atoms in a magneto-optical trap as a function of laser intensity.* Physical Review A **62**, 013404 (2000). DOI: 10.1103/PhysRevA.62.013404. [p 10]

[28] S. R. Muniz, K. M. F. Magalhães, P. W. Courteille, M. A. Perez, L. G. Marcassa, and V. S. Bagnato. *Measurements of capture velocity in a magneto-optical trap for a broad range of light intensities.* Physical Review A **65**, 015402 (2001). DOI: 10.1103/PhysRevA.65.015402. [p 10]

[29] G. L. Gattobigio, T. Pohl, G. Labeyrie, and R. Kaiser. *Scaling laws for large magneto-optical traps.* Physica Scripta **81**, 025301 (2010). DOI: 10.1088/0031-8949/81/02/025301. [p 10]

[30] C. J. Myatt, N. R. Newbury, R. W. Ghrist, S. Loutzenhiser, and C. E. Wieman. *Multiply loaded magneto-optical trap.* Optics Letters **21**, 290 (1996). DOI: 10.1364/OL.21.000290. [p 10]

[31] T. B. Swanson, D. Asgeirsson, J. A. Behr, A. Gorelov, and D. Melconian. *Efficient transfer in a double magneto-optical trap system.* Journal of the Optical Society of America B **15**, 2641 (1998). DOI: 10.1364/JOSAB.15.002641. [p 10]

[32] M. Greiner, I. Bloch, T. W. Hänsch, and T. Esslinger. *Magnetic transport of trapped cold atoms over a large distance.* Physical Review A **63**, 031401 (2001). DOI: 10.1103/PhysRevA.63.031401. [p 10]

[33] W. D. Phillips and H. Metcalf. *Laser deceleration of an atomic beam.* Physical Review Letters **48**, 596 (1982). DOI: 10.1103/PhysRevLett.48.596. [p 10]

[34] S. C. Bell, M. Junker, M. Jasperse, L. D. Turner, Y.-J. Lin, I. B. Spielman, and R. E. Scholten. *A slow atom source using a collimated effusive oven and a single-layer variable pitch coil Zeeman slower.* Review of Scientific Instruments **81**, 013105 (2010). DOI: 10.1063/1.3276712. [p 10]

[35] L. M. Bennie. *Design and construction of a Zeeman slower for ^{85}Rb and ^{87}Rb.* 3rd year research project, Monash University (2010). [p 10]

[36] J. Dalibard and C. Cohen-Tannoudji. *Laser cooling below the Doppler limit by polarization gradients: simple theoretical models.* Journal of the Optical Society of America B **6**, 2023 (1989). DOI: 10.1364/JOSAB.6.002023. [p 10]

[37] W. Petrich, M. H. Anderson, J. R. Ensher, and E. A. Cornell. *Stable, tightly confining magnetic trap for evaporative cooling of neutral atoms.* Physical Review Letters **74**, 3352 (1995). DOI: 10.1103/PhysRevLett.74.3352. [p 11]

[38] T. Esslinger, I. Bloch, and T. W. Hänsch. *Bose–Einstein condensation in a quadrupole-Ioffe-configuration trap.* Physical Review A **58**, R2664 (1998). DOI: 10.1103/PhysRevA.58.R2664. [p 11]

[39] Y.-J. Lin, A. R. Perry, R. L. Compton, I. B. Spielman, and J. V. Porto. *Rapid production of ^{87}Rb Bose–Einstein condensates in a combined magnetic and optical potential.* Physical Review A **79**, 063631 (2009). DOI: 10.1103/PhysRevA.79.063631. [p 11]

[40] T. Meyrath, F. Schreck, J. Hanssen, C. Chuu, and M. Raizen. *A high frequency optical trap for atoms using Hermite–Gaussian beams.* Optics Express **13**, 2843 (2005). DOI: 10.1364/OPEX.13.002843. [p 11]

[41] S. Tempone-Wiltshire, S. Johnstone, and K. Helmerson. *High efficiency, low cost holographic optical elements for ultracold atom trapping.* Optics Express **25**, 296 (2017). DOI: 10.1364/OE.25.000296. [pp 11 and 163]

[42] A. L. Gaunt, T. F. Schmidutz, I. Gotlibovych, R. P. Smith, and Z. Hadzibabic. *Bose–Einstein condensation of atoms in a uniform potential.* Physical Review Letters **110**, 200406 (2013). DOI: 10.1103/PhysRevLett.110.200406. [p 11]

[43] I. Bloch. *Ultracold quantum gases in optical lattices.* Nature Physics **1**, 23 (2005). DOI: 10.1038/nphys138. [p 11]

[44] M. Greiner, O. Mandel, T. Esslinger, T. W. Hänsch, and I. Bloch. *Quantum phase transition from a superfluid to a Mott insulator in a gas of ultracold atoms.* Nature **415**, 39 (2002). DOI: 10.1038/415039a. [p 11]

[45] P. J. Torres, P. G. Kevrekidis, D. J. Frantzeskakis, R. Carretero-González, P. Schmelcher, and D. S. Hall. *Dynamics of vortex dipoles in confined Bose–Einstein condensates.* Physics Letters A **375**, 3044 (2011). DOI: 10.1016/j.physleta.2011.06.061. [p 12]

[46] R. Navarro, R. Carretero-González, P. J. Torres, P. G. Kevrekidis, D. J. Frantzeskakis, M. W. Ray, E. Altuntaş, and D. S. Hall. *Dynamics of a few corotating vortices in Bose–Einstein condensates.* Physical Review Letters **110**, 225301 (2013). DOI: 10.1103/PhysRevLett.110.225301. [p 12]

[47] T. W. Neely, E. C. Samson, A. S. Bradley, M. J. Davis, and B. P. Anderson. *Observation of vortex dipoles in an oblate Bose–Einstein condensate.* Physical Review Letters **104**, 160401 (2010). DOI: 10.1103/PhysRevLett.104.160401. [p 12]

[48] N. Navon, A. L. Gaunt, R. P. Smith, and Z. Hadzibabic. *Emergence of a turbulent cascade in a quantum gas.* Nature **539**, 72 (2016). DOI: 10.1038/nature20114. [p 12]

[49] T. L. Nicholson, S. L. Campbell, R. B. Hutson, G. E. Marti, B. J. Bloom, R. L. McNally, W. Zhang, M. D. Barrett, M. S. Safronova, G. F. Strouse, W. L. Tew, and J. Ye. *Systematic evaluation of an atomic clock at* 2×10^{-18} *total uncertainty.* Nature Communications **6**, 6896 (2015). DOI: 10.1038/ncomms7896. [p 12]

[50] A. A. Wood, L. M. Bennie, A. Duong, M. Jasperse, L. D. Turner, and R. P. Anderson. *Magnetic tensor gradiometry using ramsey interferometry of spinor condensates.* Physical Review A **92**, 053604 (2015). DOI: 10.1103/PhysRevA.92.053604. [p 12]

[51] K. S. Hardman, P. J. Everitt, G. D. McDonald, P. Manju, P. B. Wigley, M. A. Sooriyabandara, C. C. N. Kuhn, J. E. Debs, J. D. Close, and N. P. Robins. *Simultaneous precision gravimetry and magnetic gradiometry with a Bose–Einstein condensate: A high precision, quantum sensor.* Physical Review Letters **117**, 138501 (2016). DOI: 10.1103/PhysRevLett.117.138501. [p 12]

[52] A. A. Wood, L. D. Turner, and R. P. Anderson. *Measurement and extinction of vector light shifts using interferometry of spinor condensates.* Physical Review A **94**, 052503 (2016). DOI: 10.1103/PhysRevA.94.052503. [p 12]

[53] T. M. Rvachov, H. Son, A. T. Sommer, S. Ebadi, J. J. Park, M. W. Zwierlein, W. Ketterle, and A. O. Jamison. *Long-lived ultracold molecules with electric and magnetic dipole moments.* Physical Review Letters **119**, 143001 (2017). DOI: 10.1103/PhysRevLett.119.143001. [p 12]

[54] R. W. Speirs, A. J. McCulloch, B. M. Sparkes, and R. E. Scholten. *Identification of competing ionization processes in the generation of ultrafast electron bunches from cold-atom electron sources.* Physical Review A **95**, 053408 (2017). DOI: 10.1103/PhysRevA.95.053408. [p 12]

[55] A. J. McCulloch, R. W. Speirs, J. Grimmel, B. M. Sparkes, D. Comparat, and R. E. Scholten. *Field ionization of Rydberg atoms for high-brightness electron and ion beams.* Physical Review A **95**, 063845 (2017). DOI: 10.1103/PhysRevA.95.063845. [p 12]

[56] D. S. Hall, M. W. Ray, K. Tiurev, E. Ruokokoski, A. H. Gheorghe, and M. Möttönen. *Tying quantum knots.* Nature Physics **12**, 478 (2016). DOI: 10.1038/nphys3624. [p 12]

[57] W. Lee, A. H. Gheorghe, K. Tiurev, T. Ollikainen, M. Möttönen, and D. S. Hall. *Synthetic electromagnetic knot in a three-dimensional skyrmion.* Science Advances **4**, eaao3820 (2018). DOI: 10.1126/sciadv.aao3820. [p 12]

[58] N. Goldman, J. C. Budich, and P. Zoller. *Topological quantum matter with ultracold gases in optical lattices.* Nature Physics **12**, 639 (2016). DOI: 10.1038/nphys3803. [p 12]

[59] C. A. Regal, M. Greiner, and D. S. Jin. *Observation of resonance condensation of fermionic atom pairs.* Physical Review Letters **92**, 040403 (2004). DOI: 10.1103/PhysRevLett.92.040403. [p 12]

[60] T. Bourdel, L. Khaykovich, J. Cubizolles, J. Zhang, F. Chevy, M. Teichmann, L. Tarruell, S. J. J. M. F. Kokkelmans, and C. Salomon. *Experimental study of the BEC-BCS crossover region in lithium 6.* Physical Review Letters **93**, 050401 (2004). DOI: 10.1103/PhysRevLett.93.050401. [p 12]

[61] C. Chin, M. Bartenstein, A. Altmeyer, S. Riedl, S. Jochim, J. H. Denschlag, and R. Grimm. *Observation of the pairing gap in a strongly interacting Fermi gas.* Science **305**, 1128 (2004). DOI: 10.1126/science.1100818. [p 12]

[62] Z. Hadzibabic, P. Krüger, M. Cheneau, B. Battelier, and J. Dalibard. *Berezinskii–Kosterlitz–Thouless crossover in a trapped atomic gas.* Nature **441**, 1118 (2006). DOI: 10.1038/nature04851. [p 12]

[63] P. Cladé, C. Ryu, A. Ramanathan, K. Helmerson, and W. D. Phillips. *Observation of a 2D Bose gas: From thermal to quasicondensate to superfluid.* Physical Review Letters **102**, 170401 (2009). DOI: 10.1103/PhysRevLett.102.170401. [p 12]

[64] B. Allard, T. Plisson, M. Holzmann, G. Salomon, A. Aspect, P. Bouyer, and T. Bourdel. *Effect of disorder close to the superfluid transition in a two-dimensional Bose gas.* Physical Review A **85**, 033602 (2012). DOI: 10.1103/PhysRevA.85.033602. [p 12]

[65] M. Jasperse, M. J. Kewming, S. N. Fischer, P. Pakkiam, R. P. Anderson, and L. D. Turner. *Continuous faraday measurement of spin precession without light shifts.* Physical Review A **96**, 063402 (2017). DOI: 10.1103/PhysRevA.96.063402. [p 12]

[66] K. E. Wilson, Z. L. Newman, J. D. Lowney, and B. P. Anderson. *In situ imaging of vortices in Bose–Einstein condensates.* Physical Review A **91**, 023621 (2015). DOI: 10.1103/PhysRevA.91.023621. [p 12]

[67] S. W. Seo, B. Ko, J. H. Kim, and Y. Shin. *Observation of vortex-antivortex pairing in decaying 2D turbulence of a superfluid gas.* Scientific Reports **7**, 4587 (2017). DOI: 10.1038/s41598-017-04122-9. [pp 12, 165, and 166]

[68] S. Häusler, S. Nakajima, M. Lebrat, D. Husmann, S. Krinner, T. Esslinger, and J.-P. Brantut. *Scanning gate microscope for cold atomic gases.* Physical Review Letters **119**, 030403 (2017). DOI: 10.1103/PhysRevLett.119.030403. [p 12]

[69] T. A. Bell, J. A. P. Glidden, L. Humbert, M. W. J. Bromley, S. A. Haine, M. J. Davis, T. W. Neely, M. A. Baker, and H. Rubinsztein-Dunlop. *Bose–Einstein condensation in large time-averaged optical ring potentials.* New Journal of Physics **18**, 035003 (2016). DOI: 10.1088/1367-2630/18/3/035003. [p 12]

[70] T. A. Bell, G. Gauthier, T. W. Neely, H. Rubinsztein-Dunlop, M. J. Davis, and M. A. Baker. *Phase and micromotion of Bose–Einstein condensates in a time-averaged ring trap.* Physical Review A **98**, 013604 (2018). DOI: 10.1103/PhysRevA.98.013604. [p 12]

[71] M. R. Sturm, M. Schlosser, R. Walser, and G. Birkl. *Quantum simulators by design: Many-body physics in reconfigurable arrays of tunnel-coupled traps.* Physical Review A **95**, 063625 (2017). DOI: 10.1103/PhysRevA.95.063625. [p 12]

[72] Z. Hadzibabic, C. A. Stan, K. Dieckmann, S. Gupta, M. W. Zwierlein, A. Görlitz, and W. Ketterle. *Two-species mixture of quantum degenerate Bose and Fermi gases.* Physical Review Letters **88**, 160401 (2002). DOI: 10.1103/PhysRevLett.88.160401. [p 12]

[73] G. Modugno, M. Modugno, F. Riboli, G. Roati, and M. Inguscio. *Two atomic species superfluid.* Physical Review Letters **89**, 190404 (2002). DOI: 10.1103/PhysRevLett.89.190404. [p 12]

[74] S. B. Papp, J. M. Pino, and C. E. Wieman. *Tunable miscibility in a dual-species Bose–Einstein condensate.* Physical Review Letters **101**, 040402 (2008). DOI: 10.1103/PhysRevLett.101.040402. [p 12]

[75] D. J. McCarron, H. W. Cho, D. L. Jenkin, M. P. Köppinger, and S. L. Cornish. *Dual-species Bose–Einstein condensate of ^{87}Rb and ^{133}Cs.* Physical Review A **84**, 011603 (2011). DOI: 10.1103/PhysRevA.84.011603. [p 12]

[76] M. Taglieber, A.-C. Voigt, T. Aoki, T. W. Hänsch, and K. Dieckmann. *Quantum degenerate two-species Fermi–Fermi mixture coexisting with a Bose–Einstein condensate.* Physical Review Letters **100**, 010401 (2008). DOI: 10.1103/PhysRevLett.100.010401. [p 12]

[77] A. Keshet and W. Ketterle. *A distributed, graphical user interface based, computer control system for atomic physics experiments.* Review of Scientific Instruments **84**, 015105 (2013). DOI: 10.1063/1.4773536. [pp 16, 20, and 24]

[78] A. Keshet. *A next-generation apparatus for lithium optical lattice experiments.* PhD thesis, Massachusetts Institute of Technology (2012). [p 20]

[79] A. Keshet. Cicero word generator technical and user manual, (2009). [pp 21 and 23]

[80] *MATLAB version R2018a.* The Mathworks Inc., Natick, Massachusetts, USA (2018). [p 23]

[81] T. E. Oliphant. *A guide to NumPy.* Trelgol Publishing, USA (2006). [pp 23 and 73]

[82] C. J. Billington. PineBlaster. `https://bitbucket.org/labscript_suite/pineblaster/`, (2018). [pp 24 and 80]

[83] P. T. Starkey, C. J. Billington, and M. Jasperse. PineBlasterV2. `https://bitbucket.org/labscript_suite/pineblasterv2/`, (2018). [pp 24 and 80]

[84] P. Cladé. PyDAQmx : a Python interface to the national instruments DAQmx driver. `http://pythonhosted.org/PyDAQmx/`, (2018). [p 24]

[85] T. Meyrath and F. Schreck. Strontium BEC. `http://www.strontiumbec.com/indexControl.html`. [pp 25 and 70]

[86] P. E. Gaskell, J. J. Thorn, S. Alba, and D. A. Steck. *An open-source, extensible system for laboratory timing and control.* Review of Scientific Instruments **80**, 115103 (2009). DOI: 10.1063/1.3250825. [p 26]

[87] R. M. Gao. *ZOINKS and z.759 - the unfinished computer experimental control system.* QDG Lab, Physics dept., UBC (2005). [p 26]

[88] K. Ladouceur. *Experimental advances toward a compact dual-species laser cooling apparatus.* Masters thesis, The University of British Columbia (2008). DOI: 10.14288/1.0066703. [p 26]

[89] Quantum optics group at ETH zurich. https://www.quantumoptics.ethz.ch/. [p 27]

[90] Hall labs - amherst physics. https://halllab.sites.amherst.edu/. [p 27]

[91] T. Stöferle. *Exploring atomic quantum gases in optical lattices.* PhD thesis, ETH Zurich (2005). DOI: 10.3929/ethz-a-005068694. [p 27]

[92] R. Landig. *Quantum phases emerging from competing short- and long-range interactions in an optical lattice.* PhD thesis, ETH Zurich (2016). DOI: 10.3929/ethz-a-010665821. [p 27]

[93] S. F. Owen and D. S. Hall. *Fast line-based experiment timing system for LabVIEW.* Review of Scientific Instruments **75**, 259 (2003). DOI: 10.1063/1.1630833. [pp 28 and 70]

[94] S. Bourdeauducq, whitequark, R. Jördens, Y. Sionneau, enjoy-digital, cjbe, JBoulder, hartytp, D. Slichter, and mntng. *m-labs/artiq: 3.6.* (2018). DOI: 10.5281/zenodo.1205217. [p 28]

[95] Qcodes: Modular data acquisition framework. https://github.com/QCoDeS/Qcodes, (2018). [p 28]

[96] J. M. Binder, A. Stark, N. Tomek, J. Scheuer, F. Frank, K. D. Jahnke, C. Müller, S. Schmitt, M. H. Metsch, T. Unden, T. Gehring, A. Huck, U. L. Andersen, L. J. Rogers, and F. Jelezko. *Qudi: A modular python suite for experiment control and data processing.* SoftwareX **6**, 85 (2017). DOI: 10.1016/j.softx.2017.02.001. [p 28]

[97] L. R. Dalesio, M. Kraimer, and A. J. Kozubal. *EPICS architecture.* In *ICALEPCS*, ICALEPCS, KEK, Tsukuba, Japan (1991). [p 28]

[98] EPICS - experimental physics and industrial control system. https://epics.anl.gov/. [p 28]

[99] J. Appel, A. MacRae, and A. I. Lvovsky. *A versatile digital GHz phase lock for external cavity diode lasers.* Measurement Science and Technology **20**, 055302 (2009). DOI: 10.1088/0957-0233/20/5/055302. [p 32]

[100] S. P. Johnstone. *Quantum turbulence in a planar Bose–Einstein condensate.* PhD thesis, Monash University (2018). DOI: 10.26180/5c171b9ecdb9c. [pp 32, 33, 36, 38, 40, 41, 42, 43, 47, 52, 55, 137, 163, 164, 165, and 166]

[101] C. J. Hawthorn, K. P. Weber, and R. E. Scholten. *Littrow configuration tunable external cavity diode laser with fixed direction output beam.* Review of Scientific Instruments **72**, 4477 (2001). DOI: 10.1063/1.1419217. [p 33]

[102] S. D. Saliba and R. E. Scholten. *Linewidths below 100 kHz with external cavity diode lasers*. Appl. Opt. **48**, 6961 (2009). DOI: 10.1364/AO.48.006961. [p 33]

[103] MOGLabs. *External Cavity Diode Laser Controller*. MOG Laboratories Pty Ltd, Melbourne, Australia, 7.01 edition (2011). [p 33]

[104] D. A. Steck. Rubidium 87 D line data. http://steck.us/alkalidata/rubidium87numbers.pdf, (2015). [pp 33 and 38]

[105] T. W. Hänsch, I. S. Shahin, and A. L. Schawlow. *High-resolution saturation spectroscopy of the sodium D lines with a pulsed tunable dye laser*. Physical Review Letters **27**, 707 (1971). DOI: 10.1103/PhysRevLett.27.707. [p 33]

[106] M. Jasperse. *Faraday magnetic resonance imaging of Bose–Einstein condensates*. PhD thesis, Monash University (2015). DOI: 10.4225/03/58b64dbe466ce. [pp 33, 48, 55, 136, 158, and 185]

[107] Thorlabs - TPA780P20 780 nm tapered amplifier, 2 W, 10 nm BW, butterfly pkg, PM fiber, FC/APC. https://www.thorlabs.com/thorproduct.cfm?partnumber=TPA780P20. [p 35]

[108] E. A. Donley, T. P. Heavner, F. Levi, M. O. Tataw, and S. R. Jefferts. *Double-pass acousto-optic modulator system*. Review of Scientific Instruments **76**, 063112 (2005). DOI: 10.1063/1.1930095. [p 37]

[109] P. T. Starkey. *An optical, experiment control and data acquisition system for a BEC lab*. Honours thesis, Monash University (2010). [p 37]

[110] V. Negnevitsky and L. D. Turner. *Wideband laser locking to an atomic reference with modulation transfer spectroscopy*. Optics Express **21**, 3103 (2013). DOI: 10.1364/OE.21.003103. [p 37]

[111] T. Tiecke. Properties of potassium. http://www.tobiastiecke.nl/archive/PotassiumProperties.pdf, (2011). [p 38]

[112] Thorlabs - GCH25-75 heater for ø9, ø19, or ø25 mm x l75 mm ref. Cells. https://www.thorlabs.com/thorproduct.cfm?partnumber=GCH25-75. [p 39]

[113] R. S. Williamson and T. Walker. *Magneto-optical trapping and ultracold collisions of potassium atoms*. Journal of the Optical Society of America B **12**, 1393 (1995). DOI: 10.1364/JOSAB.12.001393. [p 40]

[114] M. Prevedelli, F. S. Cataliotti, E. A. Cornell, J. R. Ensher, C. Fort, L. Ricci, G. M. Tino, and M. Inguscio. *Trapping and cooling of potassium isotopes in a double-magneto-optical-trap apparatus*. Physical Review A **59**, 886 (1999). DOI: 10.1103/PhysRevA.59.886. [p 40]

[115] M. Landini, S. Roy, L. Carcagní, D. Trypogeorgos, M. Fattori, M. Inguscio, and G. Modugno. *Sub-Doppler laser cooling of potassium atoms*. Physical Review A **84**, 043432 (2011). DOI: 10.1103/PhysRevA.84.043432. [p 40]

[116] Small ion pumps 25S - gamma vacuum. http://www.gammavacuum.com/index.php/product?id=38, (2018). [p 42]

[117] TiTanium sublimation pumps TSP cartridges - gamma vacuum. http://www.gammavacuum.com/index.php/product?id=23. [p 42]

[118] Thorlabs - BE10M-B 10x optical beam expander, AR coated: 650 - 1050 nm. https://www.thorlabs.com/thorproduct.cfm?partnumber=BE10M-B. [p 44]

[119] *Galil RIO-47xxx user manual*. Galil Motion Control, Inc., Rocklin, California, USA, 1.0r edition (2016). [p 48]

[120] *170MHz Four Channel Signal Generator Module*. NovaTech Instruments, Lynnwood, WA, USA (17-Apr-2015). [pp 49 and 100]

[121] S. P. Johnstone. Implementation of the novatech DDS9m - the labscript suite. http://labscriptsuite.org/blog/implementation-of-the-novatech-dds9m/, (2014). [p 50]

[122] *PulseBlasterDDS™ Model DDS-II-300 USB Owner's Manual*. SpinCore Technologies, Inc., Gainesville, Florida, USA (2017). [pp 50 and 101]

[123] A. Wood. *Spinor Bose–Einstein condensates in magnetic field gradients*. PhD thesis, Monash University (2015). DOI: 10.4225/03/58b3ba6e69f9a. [pp 52 and 158]

[124] G. Varoquaux. *Agile computer control of a complex experiment*. Computing in Science Engineering **10**, 55 (2008). DOI: 10.1109/MCSE.2008.47. [p 55]

[125] C. J. Billington. *State-dependent forces in cold quantum gases*. PhD thesis, Monash University (2018). DOI: 10.26180/5bd68acaf0696. [p 55]

[126] P. Starkey. *Labscript suite archive*. (2019). DOI: 10.26180/5ca988ad25ab3. [p 55]

[127] S. P. Johnstone, A. J. Groszek, P. T. Starkey, C. J. Billington, T. P. Simula, and K. Helmerson. *Evolution of large-scale flow from turbulence in a two-dimensional superfluid*. Science **364**, 1267 (2019). DOI: 10.1126/science.aat5793. [pp 65, 117, 163, 167, 173, and 177]

[128] M. D. McIlroy, E. N. Pinson, and B. A. Tague. *UNIX time-sharing system: Forward*. Bell System Technical Journal **57**, 1899 (1978). [p 66]

[129] E. S. Raymond. *The Art of UNIX Programming*. Pearson Education (2003). [p 66]

[130] R. P. Anderson, L. D. Turner, C. J. Billington, P. T. Starkey, J. Morris, and E. Payne. rpanderson / analysislib-mloop — Bitbucket. https://bitbucket.org/rpanderson/analysislib-mloop/, (2019). [pp 67 and 141]

[131] G. van Rossum, B. Warsaw, and N. Coghlan. PEP 8 – style guide for Python code. https://www.python.org/dev/peps/pep-0008/, (05-Jul-2001). [p 73]

[132] 16.16. ctypes — A foreign function library for Python — Python 3.6.6 documentation. https://docs.python.org/3.6/library/ctypes.html, (2018). [p 73]

[133] D. Abrahams and R. W. Grosse-Kunstleve. Building hybrid systems with boost.Python - 1.67.0. https://www.boost.org/doc/libs/1_67_0/libs/python/doc/html/article.html, (2003). [p 73]

[134] The Python standard library — Python 3.6.6 documentation. https://docs.python.org/3.6/library/, (2008). [p 73]

[135] PyPI – the Python package index. https://pypi.org/. [p 73]

[136] pip — pip 18.0 documentation. https://pip.pypa.io/en/stable/. [p 73]

[137] P. T. Starkey, C. J. Billington, and J. Werkmann. qtutils. https://bitbucket.org/philipstarkey/qtutils/src/default/. [pp 73 and 76]

[138] C. J. Billington and P. T. Starkey. zprocess. https://bitbucket.org/cbillington/zprocess/. [p 73]

[139] The HDF Group. Hierarchical data format, version 5, (1997). [p 73]

[140] Andrew Collette. HDF5 for Python. https://www.h5py.org/, (2008). [p 75]

[141] A. Collette. *Python and HDF5*. O'Reilly (2013). [p 75]

[142] Ømq language bindings. http://zeromq.org/bindings:_start. [p 75]

[143] P. Hintjens. 28/reqrep · ZeroMQ RFC. https://rfc.zeromq.org/spec:28/REQREP/, (2013). [p 75]

[144] GTK bug 683246 – memory leak in GtkSpinner. https://bugzilla.gnome.org/show_bug.cgi?id=683246. [p 75]

[145] GTK bug 685959 – [win32] memory leak on every redraw of a widget. https://bugzilla.gnome.org/show_bug.cgi?id=685959. [p 75]

[146] The Qt Company. Qt. https://www.qt.io. [p 76]

[147] PySide. https://wiki.qt.io/PySide. [p 76]

[148] [pyside-205] QCoreApplication.postevent leaks memory - qt bug tracker. https://bugreports.qt.io/browse/PYSIDE-205. [p 76]

[149] Riverbank Software. PyQt. https://riverbankcomputing.com/software/pyqt/. [p 76]

[150] Qt for Python. http://wiki.qt.io/Qt_for_Python. [p 76]

[151] 29.3. builtins — built-in objects — Python 3.6.6 documentation. https://docs.python.org/3.6/library/builtins.html, (2018). [p 90]

[152] P. T. Starkey, C. J. Billington, and S. P. Johnstone. labscript_suite / labscript_utils / unitconversions / linear_coil_driver.py — Bitbucket. https://bitbucket.org/labscript_suite/labscript_utils/src/e520f3fe2cf87728b4a53adce5c96743e74c6f89/unitconversions/linear_coil_driver.py?at=default, (2018-08-28). [p 102]

[153] 2. Lexical analysis — Python 3.6.6 documentation. https://docs.python.org/3.6/reference/lexical_analysis.html#identifiers, (2018). [p 108]

[154] C. J. Billington. labscript_suite / labscript_utils / camera_server.py — Bitbucket. https://bitbucket.org/labscript_suite/labscript_utils/src/319ee193562b5e3e01a24593a0b81499ad839b47/camera_server.py?at=default, (2018). [p 137]

[155] pandas: powerful Python data analysis toolkit — pandas 0.23.4 documentation. http://pandas.pydata.org/pandas-docs/version/0.23.4/, (2018). [p 138]

[156] pandas.dataframe.groupby — pandas 0.23.4 documentation. https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.groupby.html, (2018). [p 140]

[157] P. B. Wigley, P. J. Everitt, A. van den Hengel, J. W. Bastian, M. A. Sooriyabandara, G. D. McDonald, K. S. Hardman, C. D. Quinlivan, P. Manju, C. C. N. Kuhn, I. R. Petersen, A. N. Luiten, J. J. Hope, N. P. Robins, and M. R. Hush. *Fast machine-learning online optimization of ultra-cold-atom experiments*. Scientific Reports **6**, 25890 (2016). DOI: 10.1038/srep25890. [p 141]

[158] M-LOOP — M-LOOP 2.1.0 documentation. https://m-loop.readthedocs.io/. [p 141]

[159] C. Winter. PEP 3129 – class decorators. https://www.python.org/dev/peps/pep-3129/, (7-May-2007). [p 143]

[160] P. T. Starkey. labscript_suite / labscript / issues / #49 - allow the set_passed_properties decorator to be used on labscript objects that do not have an entry in the device group — Bitbucket. https://bitbucket.org/labscript_suite/labscript/issues/49/allow-the-set_passed_properties-decorator, (2018). [p 147]

[161] P. T. Starkey. labscript_suite / labscript / issues / #9 - additional column in the connection table — Bitbucket. https://bitbucket.org/labscript_suite/labscript/issues/9/additional-column-in-the-connection-table, (2018). [p 147]

[162] I. B. Spielman. labscript_suite / labscript / commit / ec69e1bd3974 — Bitbucket. https://bitbucket.org/labscript_suite/labscript/commits/ec69e1b, (2014). [p 147]

[163] Adnaco-S1A: 5Gb/s over fiber optic expansion system - 2 PCI and 2 PCIe slots | adnaco technology. http://www.adnaco.com/products/s1a/. [p 157]

[164] Moving a Bose–Einstein condensate experiment. https://www.youtube.com/watch?v=sQL3B5uQWIY, (2013). [p 157]

[165] L. M. Starkey. *Precise engineering of the Bose–Einstein condensate wavefunction using magnetic resonance control*. PhD thesis, Monash University (2016). DOI: 10.4225/03/58b8bdc496809. [p 158]

[166] A. Burchianti, C. D'Errico, S. Rosi, A. Simoni, M. Modugno, C. Fort, and F. Minardi. *Dual-species Bose–Einstein condensate of ^{41}K and ^{87}Rb in a hybrid trap.* (2018). [p 162]

[167] A. Clark. Pillow — pillow (PIL fork). https://pillow.readthedocs.io/, (2018). [p 165]

[168] X. Li, M. Ke, B. Yan, and Y. Wang. *Reduction of interference fringes in absorption imaging of cold atom cloud using eigenface method.* Chinese Optics Letters **5**, 128 (2007). [p 166]

[169] C. F. Ockeloen, A. F. Tauschinsky, R. J. C. Spreeuw, and S. Whitlock. *Detection of small atom numbers through image processing.* Physical Review A **82**, 061606 (2010). DOI: 10.1103/PhysRevA.82.061606. [p 166]

[170] A. Rakonjac, A. L. Marchant, T. P. Billam, J. L. Helm, M. M. H. Yu, S. A. Gardiner, and S. L. Cornish. *Measuring the disorder of vortex lattices in a Bose–Einstein condensate.* Physical Review A **93**, 013607 (2016). DOI: 10.1103/PhysRevA.93.013607. [p 166]

[171] G. Bradski. *The OpenCV library.* Dr. Dobb's Journal of Software Tools (2000). [p 166]

[172] S. Johnstone, A. Groszek, P. Starkey, C. Billington, T. Simula, and K. Helmerson. *Data and analysis code for "evolution of large-scale flow from turbulence in a two-dimensional superfluid".* (2018). DOI: 10.26180/5bdf9e6d58d59. [p 169]

[173] W. Alt. *An objective lens for efficient fluorescence detection of single atoms.* Optik - International Journal for Light and Electron Optics **113**, 142 (2002). DOI: 10.1078/0030-4026-00133. [p 169]

[174] L. M. Bennie, P. T. Starkey, M. Jasperse, C. J. Billington, R. P. Anderson, and L. D. Turner. *A versatile high resolution objective for imaging quantum gases.* Optics Express **21**, 9011 (2013). DOI: 10.1364/OE.21.009011. [pp 169, 171, and 177]

[175] T-LS28M specifications - linear stages, motorized, ball bearing - zaber technologies. https://www.zaber.com/products/linear-stages/T-LS/details/T-LS28M, (2018). [p 171]

[176] *The JSON Data Interchange Format.* Standard ECMA-404 1st Edition / October 2013, ECMA (2013). [p 180]