Developing software for controlling an ultracold atomic apparatus

A thesis submitted in partial fulfillment of the requirement
for the degree of Bachelor of Science in
Physics from the College of William and Mary in Virginia,

by

Kerry Wang

Advisor: Prof. Seth Aubin

Prof. David Armstrong

Williamsburg, Virginia
May 19 2022

Contents

Acknowledgments

I would like to thank my research advisor Dr. Seth Aubin for his assistance and guidance throughout the course of this project. Without his help, this project would not have been possible. I would also like to thank fellow members of the lab Stephen Rosene, Morgan Logsdon, and William Miyahira for their support and company through tranquil and trying times alike.

List of Figures

List of Tables

Abstract

Ultracold atomic systems have achieved unprecedented precision in control of atoms for quantum sensing, such as in atom interferometers. This quantum control opens up applications that require extremely precise measurements, e.g. gravity sensing. However, these ultracold systems require their own complex experimental control systems. In order to control our ultracold apparatus, more than 70 devices must be triggered with microsecond-level precision timing, which is currently handled by an ADwin sequencer. This sequencer receives commands from software running on a connected computer. This thesis describes work to expand and improve this sequencer. We modified the software to accommodate 40 analog channels, 64 digital channels, and 280 events per channel. We demonstrated the ability to extend the program using Python add-ons by adding channel simulation functionality. Additionally, we increased the timing precision and accuracy of the ADwin sequencer by replacing its internal clock with an external atomic clock signal. We successfully modified the existing sequencer program to reference an external 100 kHz trigger signal and decreased intrachannel jitter from 740 ns to 60 ns. We measured interchannel latency and jitter between different channels and confirmed that the latency did not exceed 15 ns and the jitter did not exceed 25 ns. We tested several wave generators and delay generators in our timing tests and concluded they did not have a significant impact on the ADwin's observed performance.

Chapter 1

Introduction

Bose-Einstein Condensates (BECs) were first experimentally created in 1995 [1] by a team at JILA (Boulder, CO), for which they were awarded the 2001 Nobel Prize in Physics. Since then, research into BECs and ultracold atoms has developed rapidly. One property of BECs of interest is that they can manifest quantum effects, such as wave-particle duality, on a macroscopic scale. These quantum effects have direct applications in fields such as quantum computing and atom interferometry. In particular, atomic waves are much more sensitive to gravity than light waves, due to their mass, and propagate at much lower velocities allowing longer integration times. Exploiting this fact, an atom interferometer can deliver unmatched precision in tasks such as measurements of local gravitational fields [2]. The Aubin lab conducts research into spin-dependent trapped atom interferometry using an ultracold atom apparatus, which creates BECs via a number of cooling steps. The cooling process begins by Doppler-cooling ^{87}Rb in a magneto-optical trap (MOT), and finishes with RF evaporative cooling in a micromagnetic trap on an atom chip. The MOT uses three pairs of counter-propagating lasers, one for each axis, as well as a pair of coils arranged in an anti-Helmholtz configuration to collect and cool over 5×10^8 rubidium atoms from room temperature to 30 μK with a cycle time of about 40 seconds [3]. The Aubin lab is interested in using tightly-grouped traces on small printed circuit

Figure 1.1: Photo of ADwin-Pro II sequencer.

boards, referred to as "atom chips", to create magnetic trapping potentials. In every step of the process, the numerous devices in play must all be coordinated down to the microsecond level. This is accomplished with the ADwin-Pro II sequencer (Jäger Computergesteuerte Messtechnik GmbH), which can be programmed using software developed in-house. A photo of the ADwin-Pro II is shown in Figure 1.1. However, as experiments grow in complexity, limitations in the present software and hardware are becoming apparent. An overview of the experimental setup is presented in Figure 1.2.

The ADwin-Pro II sequencer that the Aubin lab currently uses is capable of supporting up to 15 expansion cards, each of which can house either 8 analog (-10 to +10 V) or 32 digital (~3.5 V TTL) inputs or outputs. This project includes both software and hardware upgrades to this sequencer. Currently, the sequencer has 4 analog and 2 digital cards, for a combined total of 32 analog outputs and 64 digital outputs. However, due to software limitations, only 32 lines of each type can be controlled at the same time. Expanding the number of control lines is nontrivial, as these

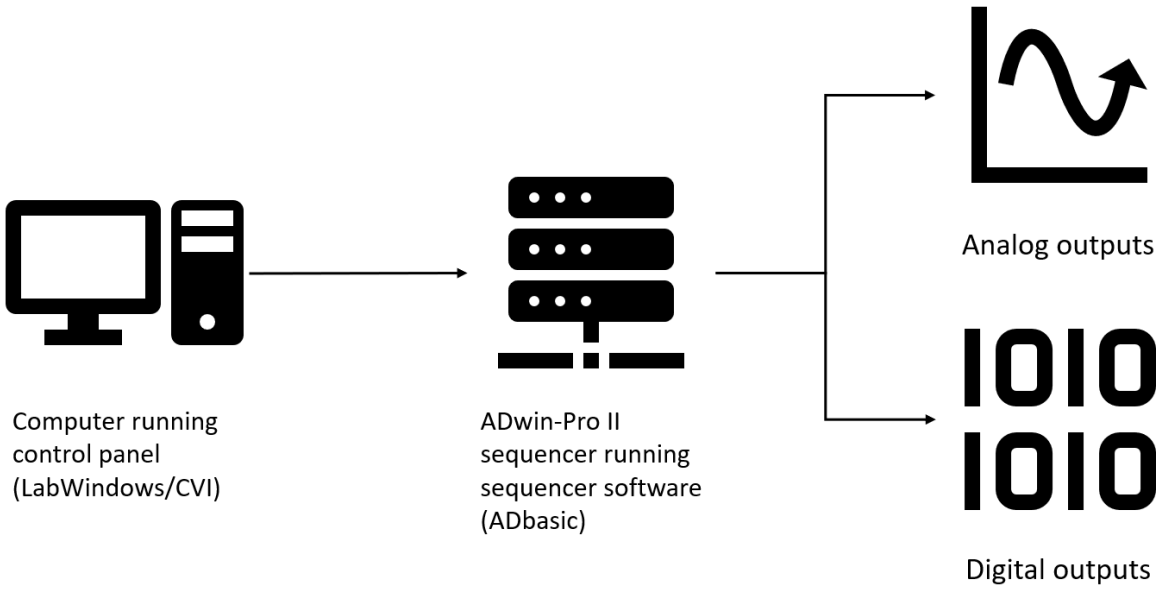

Figure 1.2: Overview diagram of the sequencer setup.

numbers were hardcoded when the program was originally written[1]. A screenshot of the original control software is shown in Figure 1.3. The most pressing objective of this project is to expand the software so the full capability of the sequencer can be utilized. This would also open the way for additional expansion cards in the future.

In addition, more features have been requested for the control software. One requested feature is channel simulation, where output voltages for channels are calculated and plotted on a graph without being sent to the ADwin. This can assist with data visualization and troubleshooting without the need to be connected to the lab apparatus. Implementing these features would make the software more useful for designing and running experiments.

On the hardware side, new experiments require increasingly precise timing. The ADwin-Pro II is equipped with a T11 processor module running at 300 MHz, however its timing precision and accuracy is inadequate for future atom interferometry

[1]The original software was developed by Dr. Stefan Myrskog in the Thywissen group at the University of Toronto. This software has been improved by a number of researchers.

experiments. Integrating the lab's atomic clock signal with the sequencer would allow increased timing precision without the need to invest in new equipment.

This thesis describes work on the objectives described above. Chapter 2 discusses the control software's codebase and our improvements. Chapter 3 presents our hardware modifications to the apparatus and their effects on the ADwin's timing performance. Finally, Chapter 4 concludes with a summary and brief outlook on possible future work.

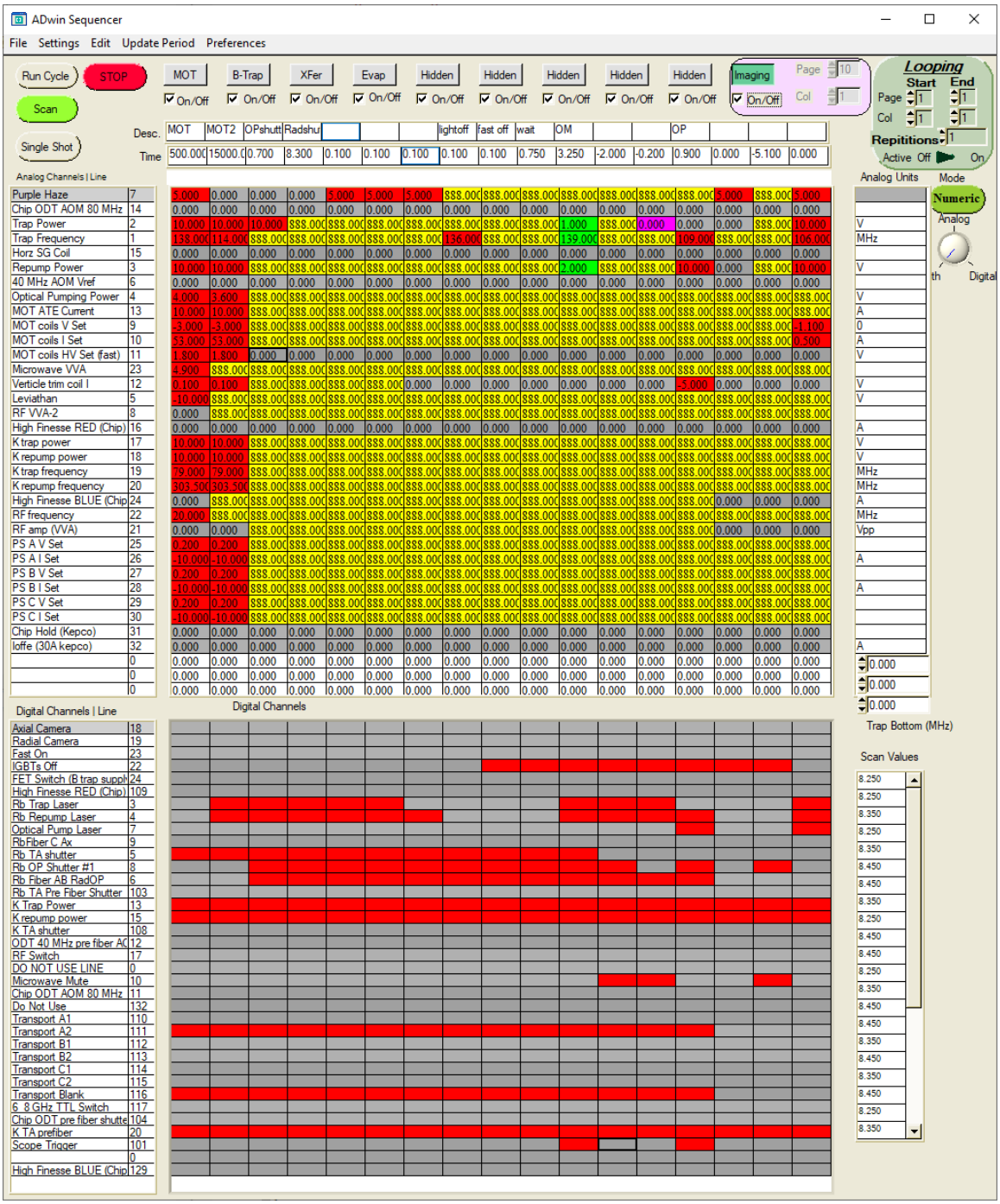

Figure 1.3: Screenshot of the GUI for the original control software. The main interface is split between analog channels above and digital channels below. Each row represents a single channel and each column represents a user-defined duration. Buttons at the top allow users to switch between pages, which are executed sequentially. Colors represent on/off state for digital channels and function type for analog channels.

Chapter 2

Software

This chapter describes the work done to overhaul the codebase and also serves as a reference for future work on the codebase.

2.1 Organization

The main part of the software project is the control software, a LabWindows/CVI project written in C89 with some C99 extensions enabled. In addition, the project also contains an ADbasic program that the control software uploads to the ADwin and a Python program that provides plotting functionality. The project consists of many files, with types and responsibilities as described below. The program begins in `main.c`, which initializes variables and dynamically creates GUI elements. The bulk of user interaction happens in `GUIDesign.c`, which is responsible for the main interface. Additional user interfaces (such as settings windows) are called by `GUIDesign.c` as required. When the user runs the panel, `GUIDesign.c` uses the `Adwin.c` library provided by the manufacturer to upload the `ADwin11.btl` bootloader and the `TransferDataExternalClock.TB1` program to the ADwin sequencer. It then uploads the panel data into the ADwin's memory and signals the bootloader to start the program. `TransferDataExternalClock.TB1` then reads the panel data from the ADwin's memory and begins execution. A diagram of this process is shown

in Figure 2.1.

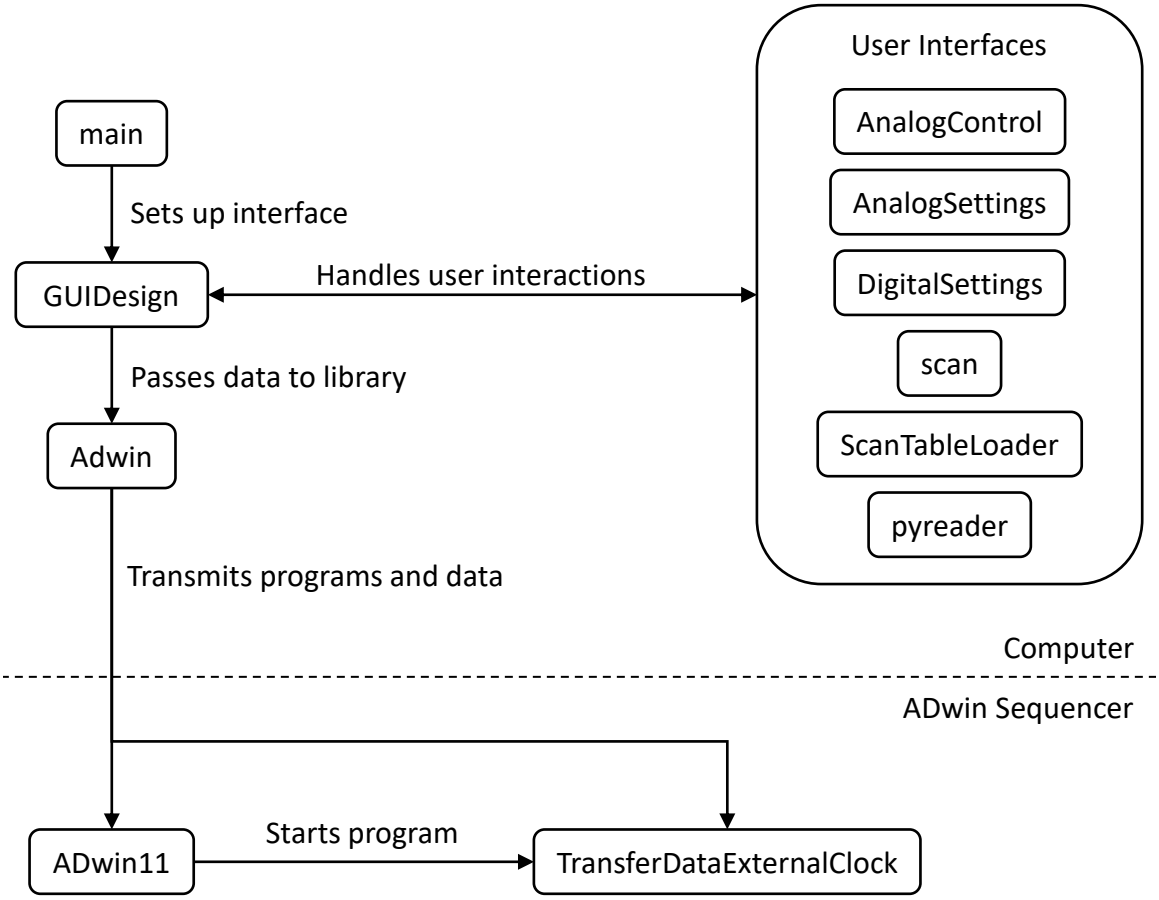

Figure 2.1: Diagram of program flow.

2.1.1 File types

***.c** Source code files. Contains the actual logic of the program.

***.h** Header files. Contains function and variable declarations so code from one source file can access code in another source file. Some of these are automatically generated by LabWindows.

***.uir** User interface resource files. Contains information on how to layout the user interface elements on the screen.

***.bas** ADbasic source code files. Contains instructions for the ADwin sequencer.

***.TB1** Compiled ADbasic binaries. To be loaded into the ADwin sequencer.

***.btl** ADwin bootloader. Small program that manages other ADwin programs.

***.py** Python scripts. Contains logic for Python programs.

2.1.2 File responsibilities

Adwin.* The library used to transmit programs and data from the control software to the ADwin sequencer. Provided by the manufacturer.

ADwin11.btl Bootloader for the ADwin's T11 processor. Provided by the manufacturer.

AnalogControl.* Interface for setting the type of analog signal to output in each cell.

AnalogSettings.* Interface for naming and setting channel number for each row in the analog table.

DigitalSettings.* Interface for naming and setting channel number for each row in the digital table.

GUIDesign.* The main interface. Responsible for the analog and digital tables, settings, saving/loading, and sending commands to the ADwin sequencer.

main.* Initial setup of variables, data structures, and runtime creation of user interface elements.

scan.* Interface for setting analog and time scans.

ScanTableLoader.* Interface for populating the scan values table.

TransferData.* ADwin program that receives data from the control software and manages the ADwin's outputs. Two versions are available, one using the internal clock and one referencing an external signal.

vars.h Special header file containing definitions of macros, structs, and global variables.

2.2 Version Control and Documentation

While the primary goal of this project is to upgrade the software to support the needs of current and future experiments, at the same time it is important to organize the code and maintain correct and up-to-date documentation so that future work on the software can avoid significant retreading of past work. Over the years, many different versions of the program were backed up by saving them under new folders. The folder names usually included the creation date and a short message regarding what changes were made, though in some cases work continued after the listed date. There are several issues with this approach. First, it is not immediately clear what the latest version of the software is. Second, it is difficult to see what changes have been made between versions and the reasons for the changes. Last, each time a new version is saved, an entirely new copy of the program is created, which quickly takes up storage. To solve these issues, the first task of the project was to establish a chronology of the versions and enter them into a Git repository.

Git [4] is a version control tool widely used in software development to keep track of files. A collection of files managed with Git is called a *repository*. As users make changes to files in the repository, they can *commit* those changes. Commits save

changes incrementally, i.e. each commit only keeps track of what changed between itself and the previous commit. Thus, using commits greatly reduce space requirements. In addition, each commit is labeled with a message and unique identifier, ensuring that each change is documented and allowing for rollbacks to specific previous versions, which can help with debugging. Each version of the software was committed into Git in chronological order, and as improvements were made they were regularly committed as well. Repositories can also be easily cloned and stored in multiple locations for data integrity. Commits to one repository can be pushed to the others, keeping all of them up to date. Should one repository suffer a loss of data, it can pull the commit history from another repository and rebuild itself[1]. We maintain three repositories: one on the computer's hard drive, one on the lab's network storage server, and one on the repository hosting service GitHub. The repository is publicly accessible on GitHub and can be found in Ref. [5].

Additionally, proper documentation of code is important to assist future work. Originally, documentation was accomplished by leaving comments in the code, however it remains difficult to get a broad overview of the responsibilities of each file. By rewriting comments to follow a specific format, Doxygen was used to generate documentation in HTML format. The generated documentation is indexed, searchable, and includes hyperlinks for easy lookup of function and struct definitions. Using GitHub Actions, this process was automated and triggers every time the GitHub repository is updated. The full documentation is also available on GitHub and can be found in Ref. [6].

[1]This comes from experience, as at one point the lab's storage server did suffer a loss of data while being updated. We restored the storage server's repository from the other two and did not lose any progress.

2.3 Automated Quality Control

At a combined total of over 4,500 lines of code, the codebase is sizable enough that it would take a significant amount of time to review every line and check for bugs and bad coding practices. This is especially important since the codebase has had many developers over its nearly two decades of existence. Assumptions made by legacy code may become invalid, causing parts of the program to fail over time. The most serious problems can be caught by the C compiler, which will warn and fail to compile on issues such as type mismatches and broken syntax. However, just because a program compiles does not mean it is correct.

To check for deeper issues, we used Cppcheck, a static code analyzer. In addition to simply checking for code validity, it also checks for valid but potentially dangerous code, such as accessing a variable after it is declared but before it is assigned a value or suspected misuse of the assignment(`=`) operator where the comparison(`==`) operator was intended[2]. Use of automated code analyzers helped fix dubious code before they manifested as bugs.

Lastly, code style is the subject of many a debate in the programming world. The C language ignores most whitespace characters, so spacing such as indentation and line breaks are up to individual preference[3]. In principle, one could remove every line break from the codebase to fit the entire program on a single line of code. This, while impressive, would also make for a spectacularly unreadable and unmaintainable codebase. Inconsistent styles may lead to confusion of whether a line of code is inside or outside a loop, or where one function definition ends and another begins. To solve this, we used clang-format, a code formatting tool, to automatically format our code

[2]Both `if(a=b)` and `if(a==b)` are valid statements in C, with dramatically different meanings. The latter checks if a and b are equal, while the former assigns the value of b to a and then checks if a is nonzero. This is a typo that even experienced programmers may overlook.

[3]This property of C has been used (and abused) to create text art that are also valid programs. See https://www.ioccc.org/2015/burton/prog.c for one particularly impressive example.

in a consistent style. clang-format supports a number of styles corresponding to major style guides; we chose Google's style guide for its strictness and completeness. The Google style guide can be found in Reference [7].

2.4 Software Improvements

A significant amount of work was spent on removing broken or obsolete code and rewriting it to make the code more readable while maintaining its functionality. Between the start of this project and the most recent commit, 5,103 lines of code were added while 18,502 lines were modified or deleted[4].

One major obstacle was the way the original program handled repeated elements. For example, to change the page, clicking the first button called a function to change to the first page, the second button called a separate but nearly identical function to change to the second page, and so on. In this way, many parameters of the software were hardcoded, and expansion was nontrivial. This was solved by code deduplication. Instead of having separate functions to perform nearly identical tasks, they were replaced by a single function that is aware of its context. For example, all the page-changing buttons now call a single function, which detects which button was clicked and changes to that corresponding page. This example is illustrated in Figure 2.2. In addition to deduplicating code and making it easier to read, this approach also makes the code easy to expand, as increasing the number of elements (buttons, table rows/columns, etc.) no longer increases the amount of code.

However, the layout of user interface elements was still hardcoded into the `.uir` files. Increasing the number of pages would require manually editing the resource

[4] https://github.com/tecnd/adwinGUI/compare/8c313b5...adwingui-v1.0.0 While the actual commit for the start of this project is `fe56ca9`, a later commit, `8c313b5`, was chosen to make this comparison due to the fact that a related but now retired project was also included with the original files, along with a full copy of its manufacturer documentation and sample code. That project's removal greatly inflates the numbers — up to 5,129 additions and 767,476 deletions!

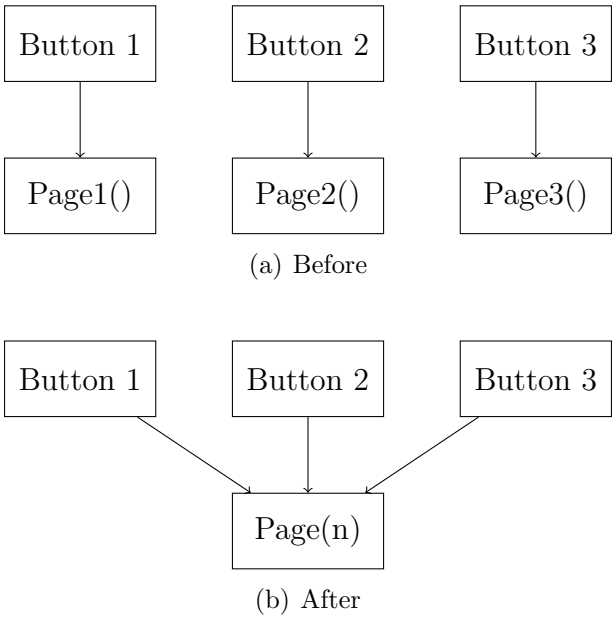

Figure 2.2: An example of code deduplication.

file to add another button. Fortunately, LabWindows allows for modification of the user interface during runtime. This means that elements can be created and moved through code. Leveraging this ability, the existing layout was removed and instead a variable amount of elements are created and positioned when the program starts. Following all of these changes, the number of pages, analog channels, digital channels, and number of columns per page can all be changed by altering the corresponding compiler define, with no code required. A summary of original and improved capabilities is listed in Table 2.1, and a screenshot of the new interface can be found in Figure 2.3. Our chosen values reflect our expectations for future expansion; the parameters can be further increased to support any configuration of the ADwin-Pro II sequencer.

2.4.1 Verification

Before changes are committed to the codebase, we must verify that the changes do not introduce errors into the program. The first lines of defense are the compiler and static

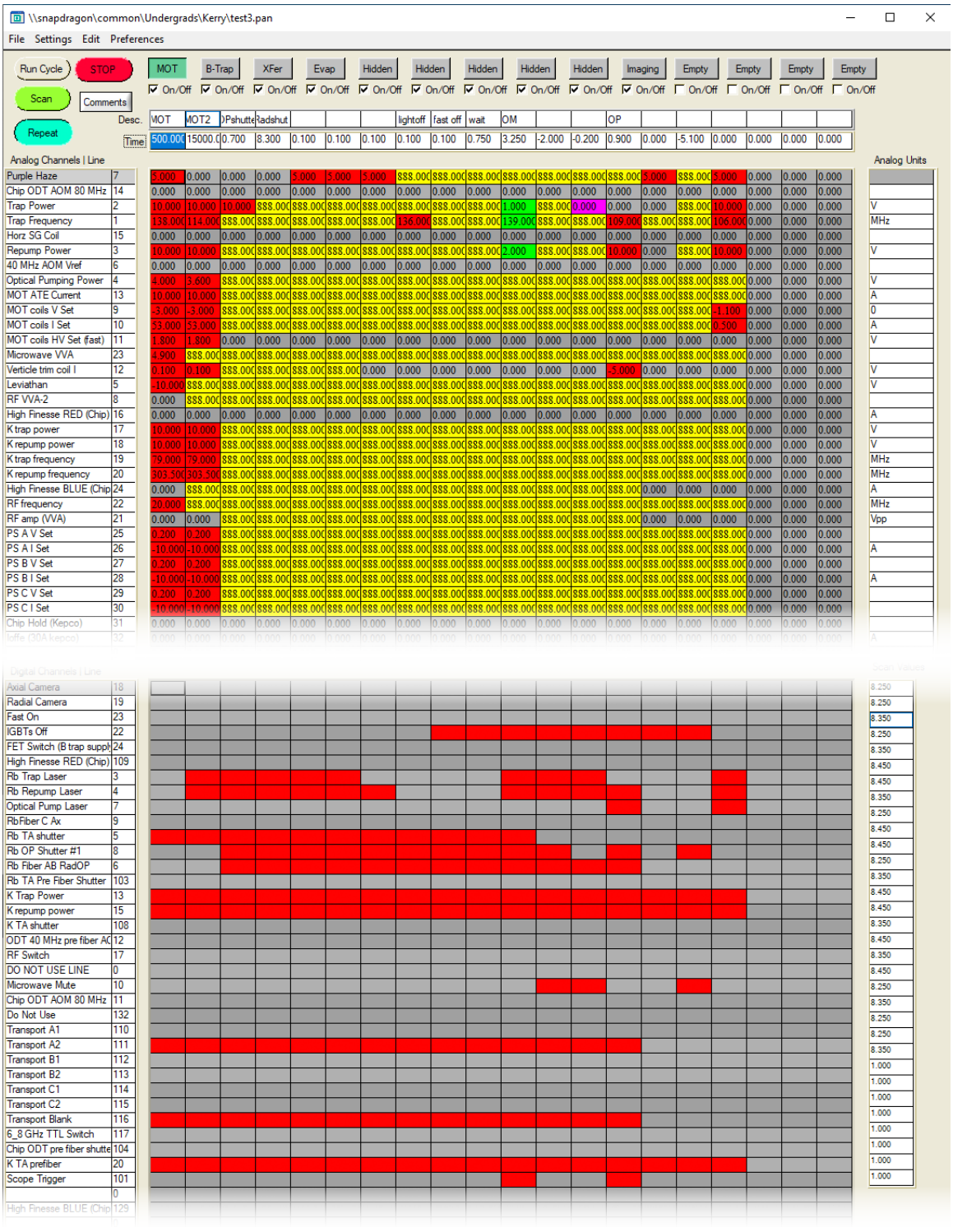

Figure 2.3: Screenshot of new control software. The original layout has been mostly preserved. Unused and obsolete elements were removed to make space for more pages and columns. The figure has been truncated to fit on the page.

	Original	Improved
Analog channels	32	40*
Digital channels	32	64
Pages	10	14
Columns per page	17	20
Total events per channel	170	280

Table 2.1: Comparison between capabilities of original versus improved software.
* While the software can control up to 40 analog channels, only 32 channels are currently available in hardware.

code analyzer, as discussed in Section 2.3. Afterwards, to ensure that modifications to the software did not break existing functionality, we perform a simple test of opening and closing a shutter. At major milestones, we fully verify the functionality of the software by loading and running a procedure to generate a BEC. After these tests pass, changes are committed and pushed to the repositories, which also triggers Doxygen to generate new documentation. Figure 2.4 shows a flowchart summarizing the development workflow.

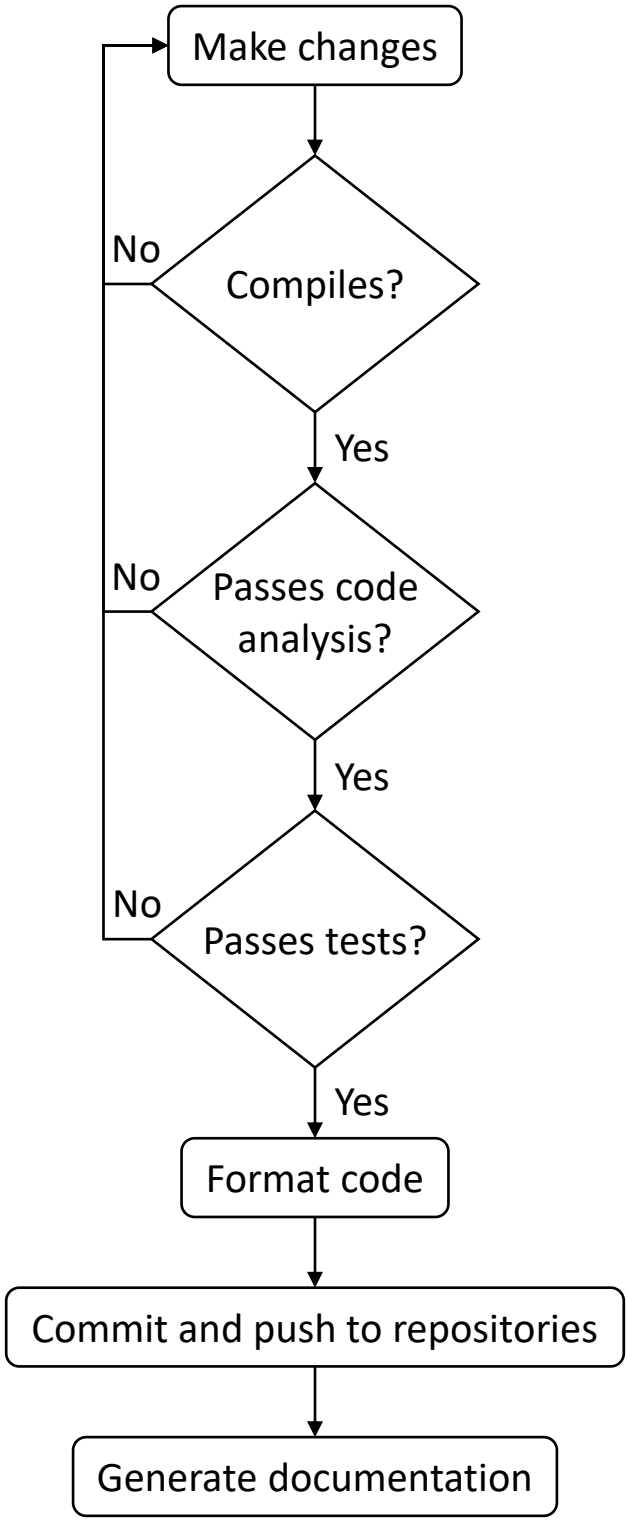

Figure 2.4: Diagram of development workflow.

2.5 Python

The requirement for fixed-size data structures in C makes working with variable-sized data difficult. While we have found workarounds so far, such as overestimating size requirements when allocating arrays and storing their actual size separately, this approach can only get us so far. In order to implement more features geared towards data analysis, like channel simulation and plotting, we must choose a more suitable language for the task. Moving away from LabWindows also reduces our reliance on proprietary software licences. Python is a favorite in the data science industry for its flexibility in data structures, and has a diverse open-source community. We chose DearPyGUI as our user interface library for its simplicity of use.

One potential approach was to completely rewrite the program in Python. This would allow us to fully take advantage of Python's convenience features such as variable-length strings as well as fully decoupling from LabWindows. We trialed this approach, however we deemed that replicating every feature of the LabWindows program in Python was nontrivial and would take a significant amount of time. A screenshot of a prototype is shown in Figure 2.5.

To avoid duplication of effort in creating a replacement from the ground up, we opted to use Python in a supplementary role. When prompted, the main LabWindows program encodes and writes its current state into a text file. Any Python add-on programs can then parse the text file and use the data in whatever way it wishes. This allows us to leverage Python's strengths for new features while not throwing away code that already works. This approach also allows for development of add-ons without needing knowledge of how the LabWindows program works, significantly lowering the barrier of entry. We used this method to add digital channel plotting as a Python add-on, a screenshot of which can be found in Figure 2.6.

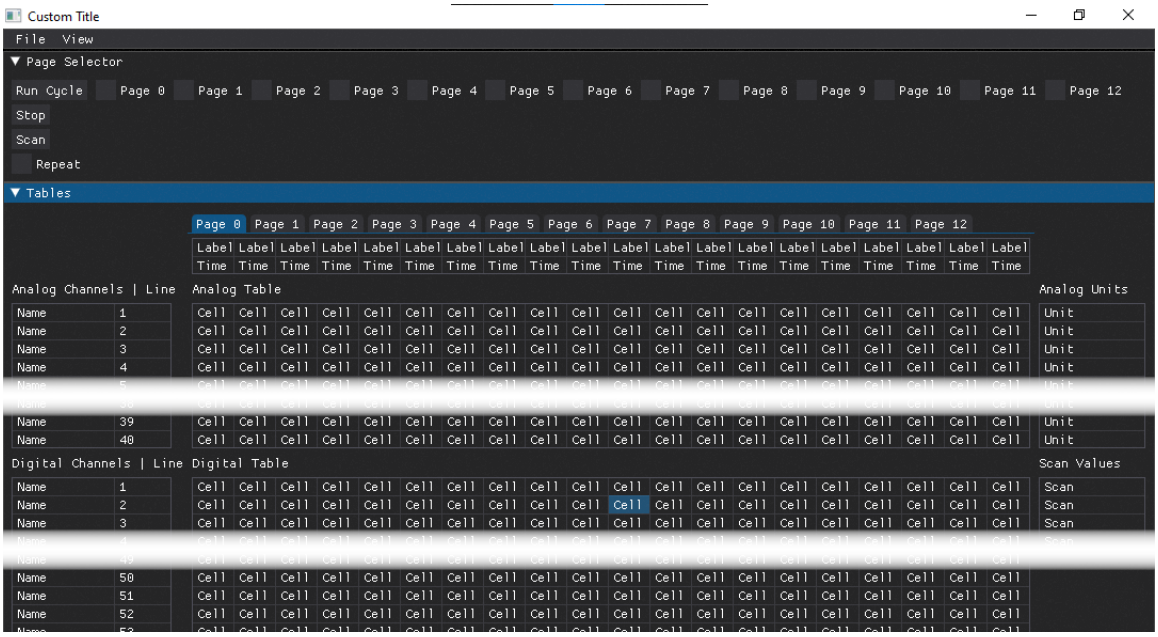

Figure 2.5: Screenshot of Python control software. Figure has been truncated to fit.

2.6 Save Conversion

Due to the way the program saves and loads experiments, as development progressed there came a point where save files generated before a certain version could not be loaded in newer versions. The Aubin lab has many saved experiments created with old versions of the software, and while it is possible to manually recreate old experiments in the new software, a save converter to make the old save files compatible with the new software is desirable.

2.6.1 The save process

Saving an experiment from the control software results in the creation of two files: a `.pan` file and a `.arr` file. Both are required to successfully load a saved experiment. Normally, LabWindows provides a built-in function to save and load the state of a program to and from a file. However, this only partially satisfies our needs. The

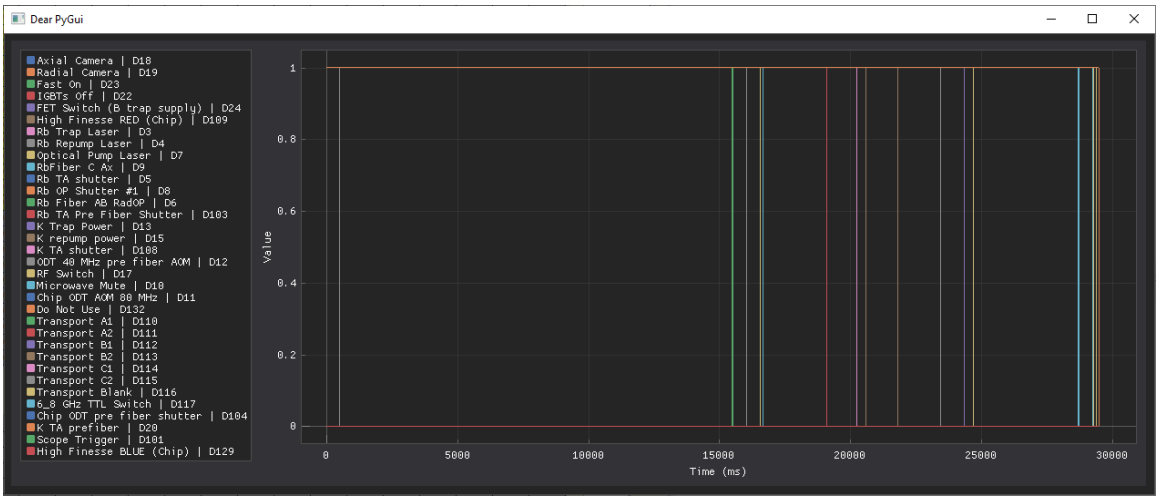

Figure 2.6: Screenshot of external channel simulation.

LabWindows save function simply records the value of every user interface component in a proprietary `.pan` file format, and the load function restores those values.

For simple values, such as the names of the pages and the list of what pages are active, this works fine. However, for more complex data structures such as the analog and digital channel tables, the user interface components only serve as a way to interface with the underlying data arrays. The built-in save and load functions do not work for these values. To compensate, the software writes a second file: the `.arr` file, containing the binary representations of the arrays. Since we know the size of the arrays and the order they were written to the file, we can simply read the contents directly back into memory to load the arrays. The former assumes the contents of the panel do not change, while the latter assumes the size of the arrays do not change. So when we switched to generating panel components at runtime and increased the number of columns, channels, and pages, both of these processes broke down.

2.6.2 Building a save converter

To build a tool that can make old saves usable again, we first need something that can read the old saves. We rolled back through the repository's commit history until we found the final version before save compatibility was lost. We then used this as a base by creating a *branch*: commits only have one parent, but can have many children. By creating a branch, we essentially keep the history of the codebase up to the point of divergence but allow us to take it in a different direction. Figure 2.7 shows a diagram of a branched repository.

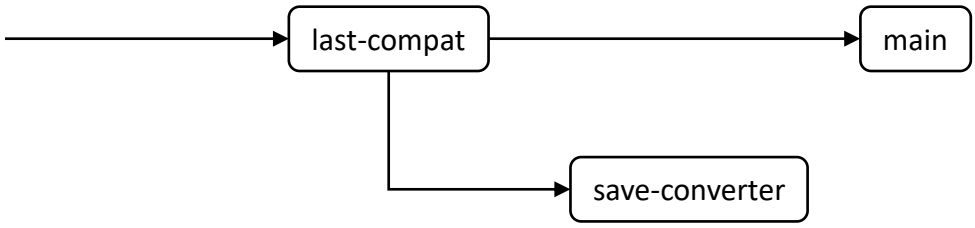

Figure 2.7: Diagram of a repository branch. `last-compat` represents the last commit that is compatible with the old save format, `main` represents the main branch of development, and `save-converter` represents the save converter's branch of development.

On this branch, we reduced the program to a minimum file loader. Everything not needed to load a save file was removed. We then needed to have it output files that the new versions on the main branch could read. For the `.arr` file, since we already know what sizes of arrays and in what order the program expects, we could create the same sized arrays, copy over the array data, and write them in the expected order. However, the `.pan` file is a proprietary format and required additional work.

2.6.3 The template injection method

We suspected that the `.pan` file was a binary file format. This was confirmed by opening the file in a hex editor and looking for saved label and page names. In addition, we noted that as long as all page names and labels had eight characters or less, the sizes of the `.pan` files remained the same, down to the byte[5]. This suggested that the value of each component was being assigned an address in the file to read and write to. Since the loading function has to restore each component to its saved value, the way these address were assigned must be deterministic, i.e. a component will receive the same address every time. We used this to our advantage by creating a "template" save file using the new software with every label set to an eight-character marker, for example "DESC0102" for the description label on the first page for the second column. We then noted the address of each saved value. Finally, to create a compatible `.pan` file, we would make a copy of the template, seek to the relevant addresses, and inject the corresponding values from the loaded old save, overwriting the marker. Figure 2.8 showcases a comparison of an injected save file with the template. The software would then dutifully load the altered labels from each address into its corresponding component. Using this method, we were able to reverse-engineer the `.pan` format to convert old saves into a format compatible with the new software. This method should remain viable should the panel layout change and break save compatibility again in the future.

[5]Having any string longer than eight characters caused another eight bytes to be allocated to it, shifting every subsequent location by eight bytes.

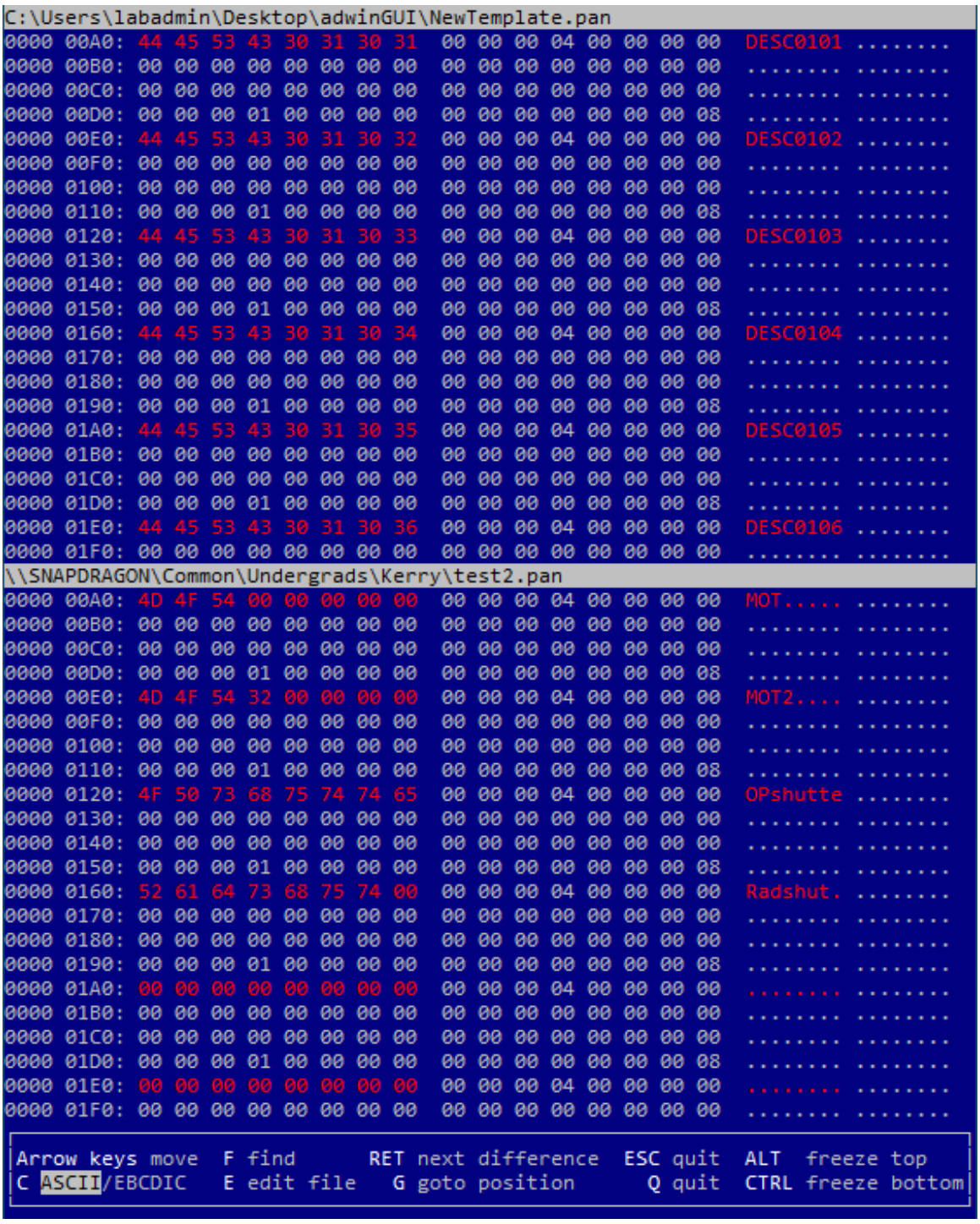

Figure 2.8: Comparison of injected save file (bottom) and the template (top) in a hex editor. Differing bytes are highlighted in red.

Chapter 3

Hardware

This chapter presents a comparison of the ADwin sequencer's timing properties with and without the use of an external reference clock. We discuss the motivation behind adding an external reference clock to the ADwin sequencer and experiments to detect improvements in the sequencer's timing properties. Our results are summarized at the end of the chapter.

3.1 Motivation and Concept

As experiments grow in complexity, and in particular for those involving time domain quantum interference, it becomes crucial that we have precise control over when events happen. While the ADwin sequencer is sufficient for millisecond-accurate signaling in experiments lasting on the order of several seconds, we are interested in microsecond- and even sub-microsecond-level timekeeping that is stable over multiple hours. The Aubin lab already has devices capable of keeping time to the accuracy we require, but none can control as many devices as the ADwin. The most straightforward solution is to have the ADwin sequencer reference a high-accuracy external clock signal instead of its internal clock.

While there is no direct way to replace the internal clock, the ADwin is equipped with an external trigger input. The ADwin supports an alternate execution mode

where the internal clock is ignored and code instead executes on an external trigger signal. We modified the sequencer code so that instead of relying on the internal clock to keep time, it directly counts the number of external trigger signals. A diagram comparing the two modes is shown in Figure 3.1. We then connected a function generator (Siglent SDG5122 Waveform Generator) to the lab's 10 MHz reference clock (Stanford Research Systems PRS10 Rubidium Standard) and generated a 100 kHz square wave as the trigger signal. At 100 kHz, the modified ADwin should be capable of updating its outputs as fast as once every 10 μs. We verified this by generating a 50 kHz square wave from the ADwin. Figure 3.2 shows a screenshot from the oscilloscope used to confirm the cycle time.

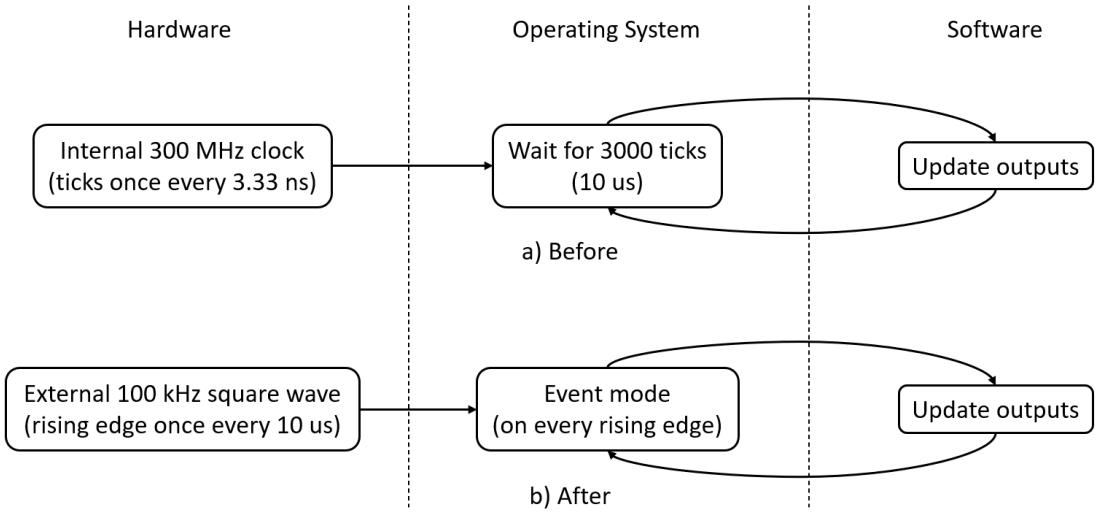

Figure 3.1: Flow diagram of ADwin system state a) using the internal clock and b) after modifications to use an external signal.

3.2 Metrics

We judge the ADwin's timing properties based on several metrics. First, the ADwin must keep consistent time. That is, every second should be the same length. For

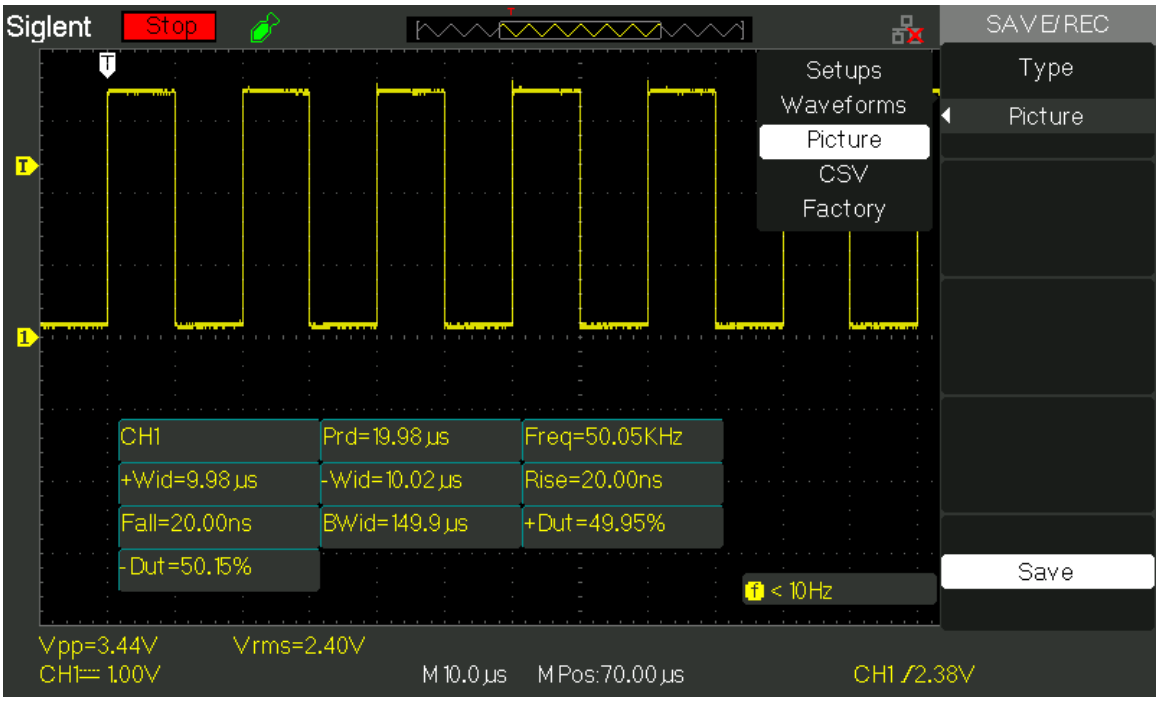

Figure 3.2: Oscilloscope screenshot confirming 10 μs cycle time for the ADwin sequencer triggered by the external clock.

example, if the ADwin and a known good clock are both set to output a pulse one second after receiving a trigger signal, any differences between the two will cause the pulses to arrive at slightly different times. We call the time delay between ideally simultaneous events *latency*. Over many trials, the latency between the two pulses should remain constant. However, no clock is perfect, and the latency will always vary from trial to trial. We call this variation *intrachannel jitter*. With low intrachannel jitter, the ADwin can be synchronized to the known good clock via a constant scale factor. Figure 3.3 shows an example of intrachannel jitter.

Second, the ADwin must be consistent between its channels. For example, if two different channels are set to switch from low to high at the same time, the latency between one channel going high and the other should be consistent and minimal, ideally on the order of nanoseconds. For the same reasons as above, the latency will

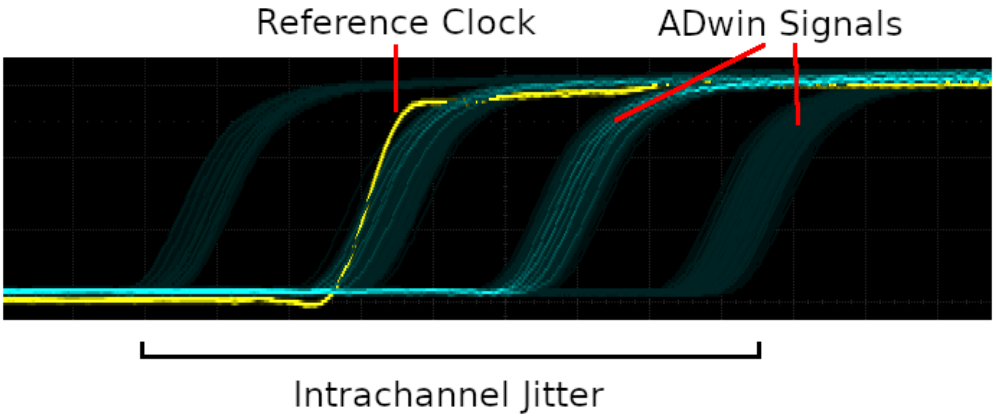

Figure 3.3: Example of intrachannel jitter. The oscilloscope is triggered on the reference clock (yellow). Traces were set to persist, showing the jitter between the reference clock and the ADwin (blue).

vary from trial to trial. We call this variation *interchannel jitter*. High interchannel latency would indicate the inability to coordinate between devices and will cause inconsistent experimental results. We are also interested if these values depend on the choice of channels.

3.3 Experiments

We conducted three experiments to measure the ADwin's timing properties. First, we measured the ADwin's intrachannel jitter by comparing it to the lab's 10 MHz rubidium standard. We measured the average intrachannel latency for various experiment durations and evaluated if there was a linear relationship between latency and duration. Next, we characterized the long-term jitter behavior by recording the latency over several hours. We analyzed interchannel latencies and jitters for a variety of channel pairs. Last, we investigated the impact of new equipment on the apparatus.

3.3.1 Intrachannel jitter

In this experiment, we measured the intrachannel latency of the ADwin by comparing it against a trusted reference clock. We calculated the intrachannel jitter by conducting multiple trials and taking the difference of the maximum and minimum latencies. We collected data at multiple timescales from $t = 0.1$ ms to 10 s. The number of trials performed at each timescale can be found in Table 3.1.

Timescale	Number of trials
≤ 1 s	100
2 s	50
5 s	25
10 s	20

Table 3.1: Number of trials performed at each timescale

We used a Stanford Research Systems DG535 delay generator clocked by the lab's 10 MHz rubidium standard as our reference. The DG535 was set up to wait for a trigger signal, wait for a set time t, and output a pulse. The ADwin was configured to output two signals. First, it would output a pulse at the start of the experiment to trigger the delay generator. Then the ADwin would also wait and output a signal at time t. The ADwin and the delay generator were both connected to a Siglent SDS1104X-E oscilloscope to measure and record the latency between the two signals. To avoid the second ADwin signal triggering the delay generator again, each ADwin signal was sent on its own channel. A diagram of the experiment setup is shown in Figure 3.4 with the timing diagram shown in Figure 3.5. We conducted the experiment twice, once with the original software using the ADwin's internal clock and once with our modifications using the external trigger signal. The ADwin trigger signal was a 100 kHz square wave generated by a Siglent SDG5122 waveform generator also connected to the rubidium standard.

We plotted the collected data against time (see Figure 3.6) and found both trials

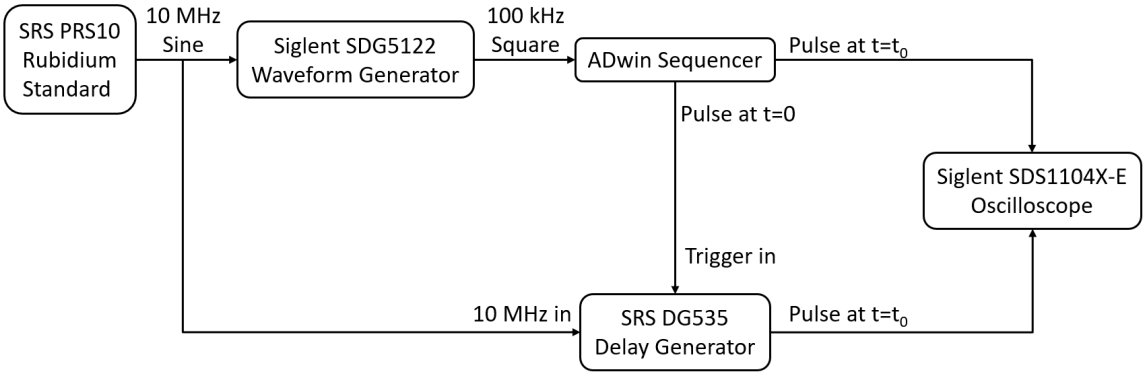

Figure 3.4: Diagram of the experimental setup to measure intrachannel jitter.

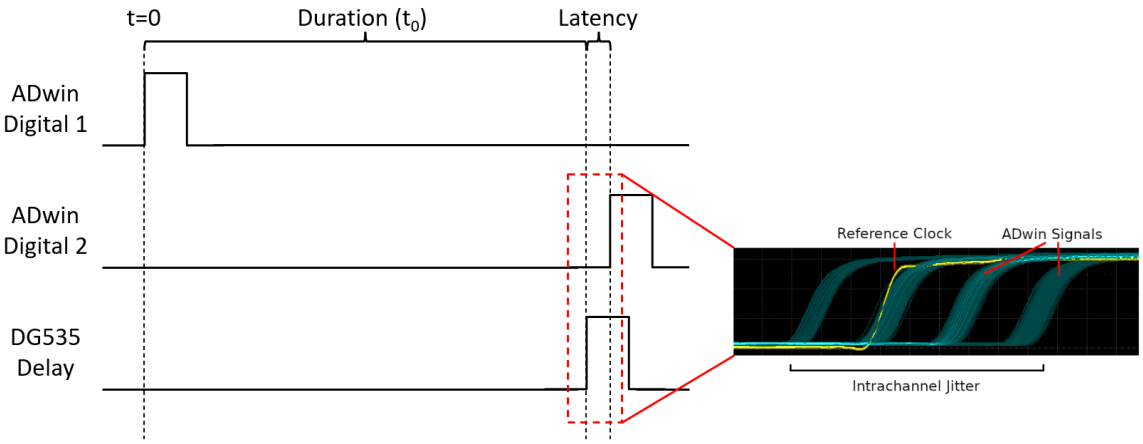

Figure 3.5: Timing diagram of the experiment to measure intrachannel jitter.

to be linear, indicating that the clocks were operating normally. We then performed a linear regression to find the clock drift relative to the lab's reference clock. Our results show that the original configuration exhibits a clock timing difference of 5.895 ppm compared to the reference, or around 500 ms per day, and an intrachannel jitter of less than 150 ns. Our modifications greatly improve on this with a drift of 0.08939 ppb compared to the reference, or 7.723 μs per day, and an intrachannel jitter of less than 40 ns.

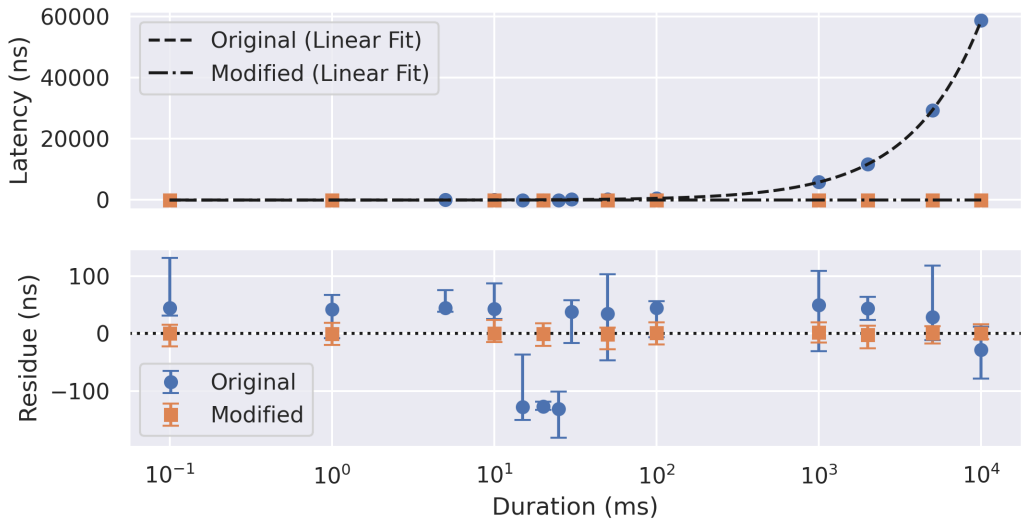

Figure 3.6: Plot and residues of the measured time offset between the ADwin and DG535. The markers represent the means, while the error bars represent the maximums and minimums. The fit equation for the original and modified trials are $y = 5.895t - 1.546 \times 10^2$ and $y = 8.939 \times 10^{-5}t - 1.121 \times 10^2$, respectively. While the fit lines are linear, they appear curved in the plot due to the logarithmic scaling.

3.3.2 Long-term intrachannel jitter

As a follow-up experiment, we measured the ADwin's long-term clock stability by repeating trials at timescale $t = 10$ s for multiple hours. We collected 1039 ($\sim$ 3 hours) and 697 ($\sim$ 2 hours) samples on two different days using the ADwin's internal clock. For comparison, we also recorded 435 ($\sim$ 72 minutes) samples using the external trigger. We hypothesized that over long periods of time, the ADwin's internal clock may wander with respect to the reference clock, resulting in greater interchannel jitter than short-term tests may reveal.

As can be seen from Figure 3.7, the ADwin's internal clock displays a tendency to wander over long periods. While our previous results suggest that the internal clock has a intrachannel jitter of under 150 ns, these longer measurements have ranges of 690 ns and 740 ns. Meanwhile, the measurement using the external trigger did

not demonstrate any wandering behavior and has a range of 60 ns. More detailed summary statistics are shown in Table 3.2.

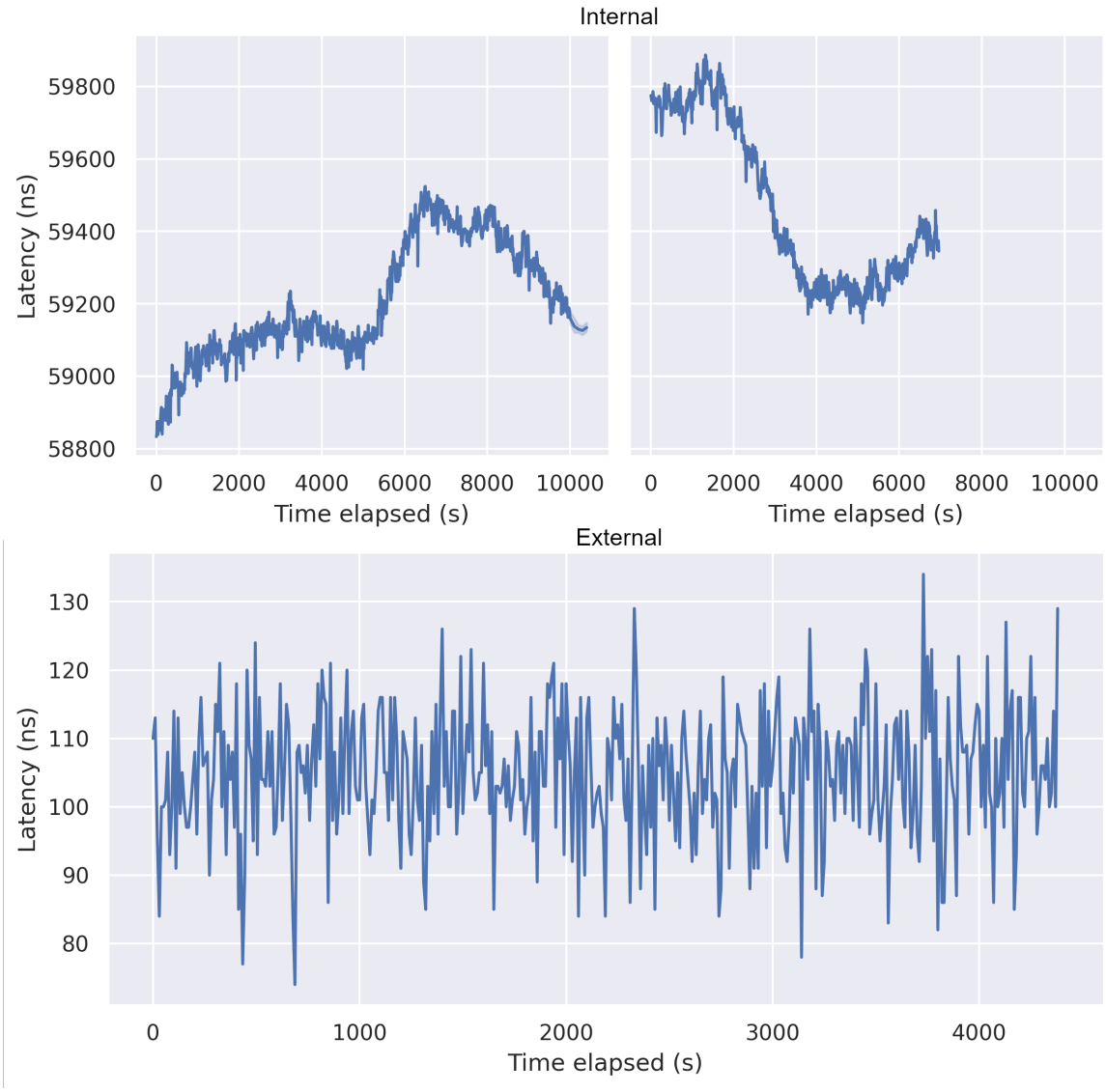

Figure 3.7: Long-term intrachannel jitter from a three-hour (top left) and a two-hour (top right) measurement using the internal clock and a 72-minute measurement (bottom) using the external trigger.

Clock	Samples	Mean	Maximum	Minimum	Range	Std. Deviation
Internal	1039	59.202 μs	59.524 μs	58.833 μs	690 ns	158 ns
Internal	697	59.470 μs	59.887 μs	59.147 μs	740 ns	225 ns
External	435	104.90 ns	134 ns	74 ns	60 ns	9.5 ns

Table 3.2: Summary statistics for long-term measurements of intrachannel time stability.

3.3.3 Interchannel jitter

To test the interchannel jitter, we connected two channels from the ADwin to the oscilloscope and set them to simultaneously switch from low to high. We then measured the latency between the two signals and calculated the jitter. Our current ADwin configuration has two digital cards, each controlling a bank of 32 channels, as well as many analog outputs; we tested the latency between digital channels on the same bank, digital channels on different banks, and between analog and digital cards. In addition, each bank houses 4 independent circuits, each responsible for a group of 8 channels, so we also tested between digital channels from the same circuit and from different circuits. A diagram of the internal organization of the ADwin's digital channels is shown in Figure 3.8. Each test was repeated for 500 trials, and the results are plotted in Figure 3.9. For all but one test one of the signals consistently arrived before the other; for those we calculated the latency as $t_{after} - t_{before}$ to get strictly positive values. However, when testing between the digital banks, there was no such consistency. In this case, we calculated the latency as $t_{bank1} - t_{bank2}$, which results in negative values when the signal from bank 1 arrives earlier than the signal from bank 2. We then calculated the interchannel jitter as the difference between the maximum and minimum recorded latencies. A plot of the interchannel jitters can be found in Figure 3.10. To quickly check if different pairs of channels from the same circuit had different latencies, 10 samples were taken from channel 32 to each other

channel from the same circuit for each bank. The results are shown in Figure 3.11.

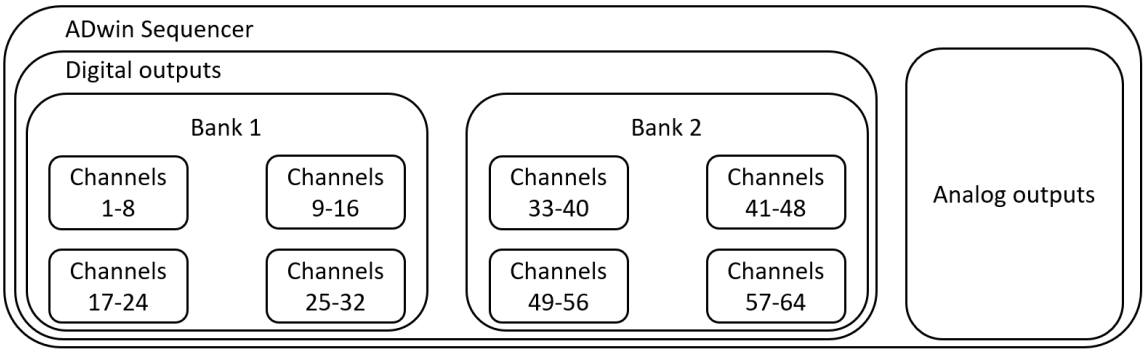

Figure 3.8: Diagram of the internal layout of the ADwin's digital channels.

Our results show that while the choice of channels does have an effect on inter-channel latency and jitter, for strictly digital channels the effect is not significant at the microsecond-level. The latency between channels on the same bank is under 5 ns and the jitter is even less. There is only a small penalty of a few nanoseconds at most for channels from different circuits on the same bank. Across banks, the penalty is more severe but absolute latency is under 15 ns and jitter is under 25 ns. While the jitter between analog and digital signals is comfortably below 40 ns, we cannot recommend using analog channels for digital signalling due to its 500 ns latency.

3.3.4 New equipment

Last, we investigated if the jitter could be further reduced by improving the external clock. The Siglent SDG5122 datasheet claims a square wave jitter of $\leq$ 200 ps [8], while the Agilent 33521A claims a square wave jitter of $<$ 40 ps [9]. The Siglent SDG6022X is a waveform generator that can also act as a delay generator. The DG535 has an 85 ns trigger delay [10], while the SDG6022X does not. We ran four experiments, one for each possible combination of waveform and delay generator. For each, we measured the intrachannel latency of the ADwin and looked for changes in

the jitter. The results are plotted in Figure 3.12.

From our results, we conclude that there are no significant benefits to be gained from upgrading the waveform generator. While the DG535 has a latency floor at 85 ns due to its trigger delay, it retains the same amount of jitter as the SDG6022X. This signifies that the jitter is likely due to the ADwin itself rather than being from the delay generator. An interesting result is the discretization of latency values independent of the combination of external equipment used. This behavior suggests that the discretization is coming from the ADwin itself.

3.4 Discussion

With these results, we conclude that our modifications to the ADwin successfully enabled it to use an external clock signal. Using the lab's reference signal instead of the ADwin's internal clock reduced intrachannel jitter by an order of magnitude, from hundreds down to tens of nanoseconds. Interchannel latencies between digital channels remained low, under 15 ns for our worst result, and jitter did not exceed 40 ns, indicating that the modified ADwin is capable of microsecond and sub-microsecond timing tasks.

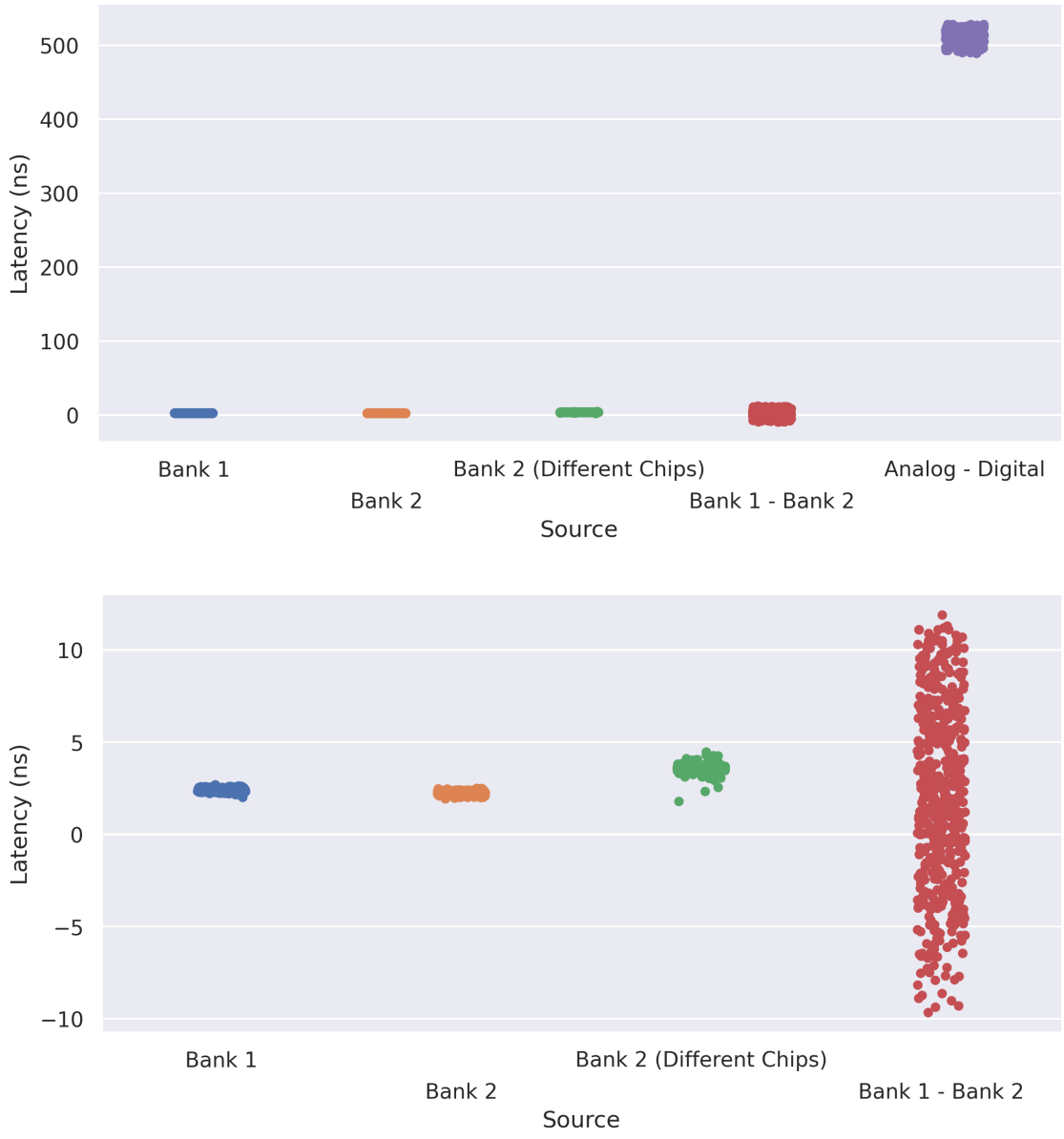

Figure 3.9: Plot of the latencies between different channels (top) and a zoomed in version excluding Analog-Digital (bottom).

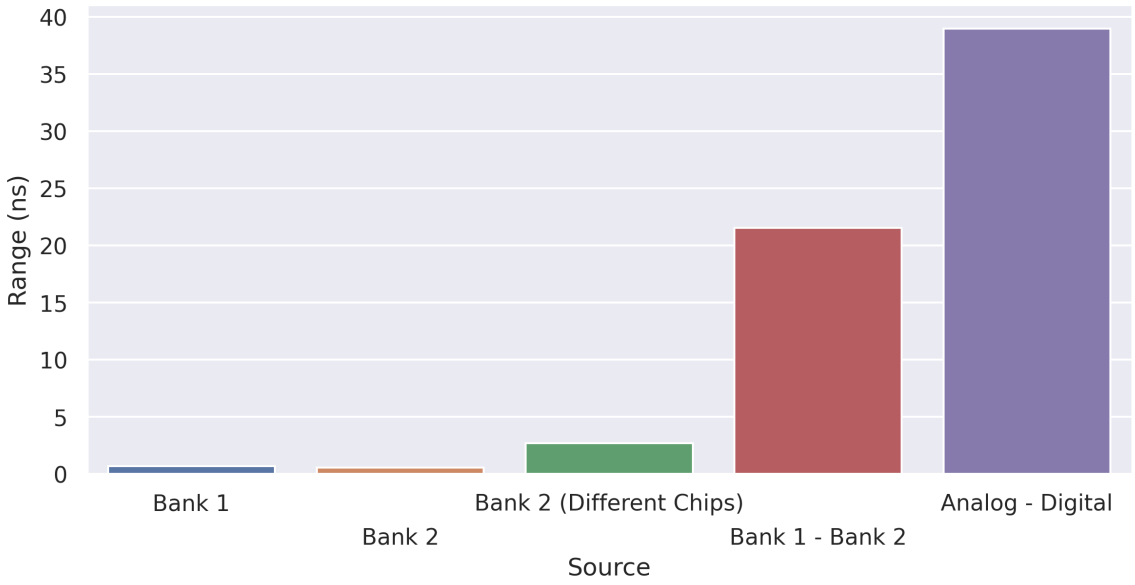

Figure 3.10: Plot of the jitters between different channels.

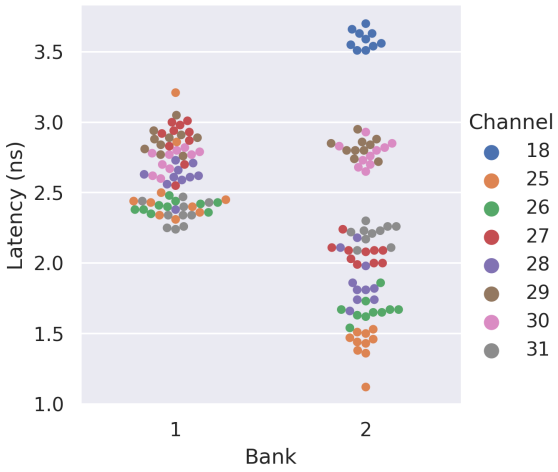

Figure 3.11: Plot of latencies between channel 32 from each bank and a spectrum of channels on the same bank. Channel 18 on bank 2 is managed by a different chip, but no other channels on bank 1 were free so no corresponding measurement could be made.

Figure 3.12: Plot of intrachannel latencies for combinations of waveform and delay generators.

Chapter 4

Summary and Outlook

4.1 Summary

Over the course of this project, we successfully upgraded the control software to support 40 analog and 64 digital channels, and the code can easily be modified to expand further. We also cleaned up and organized the codebase and wrote documentation to aid in future work on the control software. We established a framework for creating add-on programs and created a channel plotter add-on in Python as a proof of concept.

We also integrated an atomic clock with the ADwin sequencer, which decreased intrachannel jitter from 740 nanoseconds to 60 nanoseconds. We characterized the interchannel jitter between many channel sources and determined that the interchannel jitter between digital channels was acceptable for microsecond-precision timing.

4.2 Future Work

While we achieved most of our stated objectives, there remain many possibilities for future work. First, the channel plotting Python add-on can be expanded to also plot analog channels. The ADwin control software can be expanded to trigger other lab devices in concert with the ADwin. Last, work can be done to isolate and remove

the source of latency discretization from the ADwin and measure if the intrachannel jitter improves.

Appendix A

adwinGUI

Developer documentation can be found at https://tecnd.github.io/adwinGUI/[1].

A.1 User Guide

A.1.1 Menu Bar

- File
  - Load Parameters - Loads panel from a .pan/.arr pair. Both files must have the same name and be in the same directory.
  - Save Parameters - Saves panel to a .pan/.arr pair.
  - Export Channel Config - Deprecated. Writes analog and digital channel settings to a text file.
  - Export Panel - Writes analog and digital cell values to a text file for use in an external program. Skips columns with negative time. A column with a time of zero marks the end of a page.
  - Export Panel and Launch Python - Writes analog and digital cell values to a temporary text file and opens it in the Python plotter.
  - Exit - Exits the program.
- Settings

[1] <https://tecnd.github.io/adwinGUI/>

  - Analog Settings - Makes changes to analog lines, including channel number, name, units, and voltage limits. Must click Allow Changes to enable editing and Set Changes for each line.
  - Digital Settings - Makes changes to digital lines, including channel number and name. Channels 1-32 refer to channels 1-32 on the first digital I/O card (labeled 1-32), and channels 101-132 refer to channels 1-32 on the second digital I/O card (labeled 33-64). Must click Allow Changes to enable editing and Set Changes for each line.
  - Reboot ADwin - Reuploads the ADwin program and bootloader to the ADwin.
  - Clear Panel - Resets all cells to zero.
  - Scan Setup - Opens the scan setup panel.
- Edit
  - Note: All of these require a cell in the time table to be selected.
  - Insert Column - Inserts an empty column at the selected position. The last column on the page will be lost.
  - Delete Column - Deletes the column at the selected position and moves the following columns to the left.
  - Copy Column - Copies the column at the selected position.
  - Paste Column - Pastes the column to the selected position.
- Preferences
  - Use Compression - When generating ADwin tables, replace consecutive zeros with a negative integer representing the number of zeros. Greatly reduces the final size of the ADwin tables, leave on unless you have good reason not to.
  - Use Simple Timing - When calculating analog outputs, ignores the timescale

parameter and treats the full cell as the timescale. Leave on unless you have good reason not to.

A.1.2 Page Buttons

- Right-click to rename the page.

A.1.3 Analog and Digital Tables

- Double-click to change the cell's value. On the analog table, this opens a settings window. On the digital table, this toggles the cell.
- Right-click → Copy - Copies the value of the highlighted cell.
- Right-click → Paste - Pastes the copied value into all highlighted cells.

A.1.4 Scan Values Table

- Right-click → Load Values to generate scan values.

A.1.5 Usage Notes

- For whatever reason, redrawing the GUI takes significant time while repeat mode is active. This results in delays between cycles. This can be mitigated by minimizing the interface to skip redrawing the GUI.
- On Windows, use Win+Shift+S to quickly take screengrabs.

References

[1] M. H. Anderson et al. "Observation of Bose-Einstein Condensation in a Dilute Atomic Vapor". In: *Science* 269.5221 (1995), pp. 198–201. DOI: 10.1126/science.269.5221.198. URL: https://www.science.org/doi/abs/10.1126/science.269.5221.198.

[2] Susannah M. Dickerson et al. "Multiaxis Inertial Sensing with Long-Time Point Source Atom Interferometry". In: *Phys. Rev. Lett.* 111 (8 Aug. 2013), p. 083001. DOI: 10.1103/PhysRevLett.111.083001. URL: https://link.aps.org/doi/10.1103/PhysRevLett.111.083001.

[3] M. K. Ivory et al. "Atom chip apparatus for experiments with ultracold rubidium and potassium gases". In: *Review of Scientific Instruments* 85.4 (2014), p. 043102. DOI: 10.1063/1.4869781. URL: https://doi.org/10.1063/1.4869781.

[4] Linus Torvalds et al. *Git*. Version 2.36.1. May 5, 2022. URL: https://git-scm.com/.

[5] Kerry Wang and Seth Aubin. *adwinGUI*. URL: https://github.com/tecnd/adwinGUI.

[6] Kerry Wang. *adwinGUI Documentation*. URL: https://tecnd.github.io/adwinGUI/index.html.

[7] *Google C++ Style Guide*. Google. Oct. 2021. URL: https://google.github.io/styleguide/cppguide.html.

[8] *Function/Arbitrary Waveform Generator SDG5000 Series*. DS02050-E03C. Siglent Technologies.

[9] *30 MHz Function/Arbitrary Waveform Generators*. 5990-5914EN. Keysight Technologies. June 2021.

[10] *Digital Delay/Pulse Generator*. DG535c. Stanford Research Systems.