UI for Servicing General Purpose Data Acquisition Device
Summer 2021 Research
Nathan LoPresto
& Dr. Lucas J. Koerner

Introduction: When tackling the problem of real-time plotting and retention of ADC data, there is a host of instrumentation and software open and available for researchers. Each available third party product provides incredible depth and flexibility, making them all viable in the space of electrophysiology research. What is not present, however, is a product catered to a general purpose data acquisition device. Each product on the market, not to their fault, lacks the horizontal integration to cover the entire breadth of a general purpose data acquisition device. However, while borrowing pieces of open source software, gaining inspiration from the backend of third-party products, and a host of original code, our team was able to create a product successfully servicing electrophysiology researchers with an adaptable DAQ. To center the research and product planning, the design specifications and needs of the product were laid out as follows:

The product was designed around the process of electrophysiology research as it stands. Common practices, like data post-processing, real time peri-event oscilloscopes, and industry-standardized metadata files were all taken into account when designing the software. Research teams, in many cases, already have time-honored ways of conducting research, and the product couldn't interfere with that. To account for this, the design needed to have familiar file formats, easily usable UIs, and a product with the same features that researchers depend on. Additionally, the knobs and buttons on standard electrophysiology equipment needed to be mirrored as digital controls, giving researchers the adaptability they need to feel comfortable with a new product.

The product needed to comply with specific transfer speeds. While working with high speed analog to digital converters, like the AD7961, data is coming in for processing and retention at such a high speed, that every single choke point had to be accounted for and pre-planned before the eventual deployment of the product. With many of the ADCs running at 5MSPS at 16 bits, transfer speeds needed to exceed 10 MB/s, all while processing and saving data for post processing. With these speed specifications in mind, not only could hardware be selected, but also software that utilized burst reads in order for personal computers to have the power and reliability to replace more traditional methods of electrophysiology research.

The following sections will cover each product that the DAQ took inspiration from, its relationship to the electrophysiology community, and how a portion or design feature was implemented into the product, followed by the eventual integration of these parts into a working whole.

pClamp: The pClamp 11 Software Suite by Molecular Devices is the current industry standard for electrophysiology experiments. The software suite includes three major products, including the Clampex Software, the Clampfit Software and the AxoScope software (Molecular Devices LLC, 2016). Clampex Software is most relevant to the project, controlling data acquisition and real-time graphing. The software is used to measure any analog parameters passing through their own DigiData 1440a system, including measurements like end-plate currents, and fluorescence signals. From there, the data is graphed and saved as an Axon Binary File Format file (Molecular Devices LLC, 2013). Although most universal file reader packages can read ABF files, both the Clampex Software and pClamp Software give researchers the ability to read ABF files inside their own viewing window. Most importantly, the ABF files are useful for post-processing, and Clampex gives users the ability to pull ABF files straight from the viewer and use any third-party post-processing software that they want to use. Clampex includes countless other features, like high-speed mode, gap-free mode, etc. but it essentially works as a recording oscilloscope, and is recognized as the premier software in the electrophysiology community.

The pCLAMP suite was inspirational, but was too specific to their native catalog of hardware. While the DAQ runs off an OpalKelly XEM7310 board for data transfers, the software hinges on using the DigitData 1440 system. Instead of using their software, aspects of the design became inspiration for the DAQ. The data retention system was especially useful, and was the most influential in modelling the software after pCLAMP. The ability to save data into a hierarchical file format, both readable and useful for post processing, is exactly what researchers need. The project ended up using the HDF5 file format for post-processing and JSON for metadata, but the idea of using a binary encoded file format for post processing was directly adopted from pCLAMP's design. With a total of 4 ADCs, running at 5MSPS at 16/18 bit resolution, pCLAMP's file formatting and rapid read/write design was perfect for implementation into the project.

dPatch: The dPatch amplifier system by Sutter instruments is the leading design for patch clamp amplifiers. While most patch clamp amplifiers were designed over 20 years ago, the dPatch, designed through the 2010s, was able to capitalize on newer technologies like FPGA design and Arm Core processors. This allows the dPatch to run at 5 MHz at 22 bit resolution, which was unheard of until the advent of the dPatch (Sutter Instruments, 2018). To achieve this, dPatch Front Panel includes a trigger system, which is instrumental to reading data at high speeds. A temporary storage location, which could be as deep as 8MBs or as shallow as 1 bit, is continuously filled with data from a pipe, and a "trigger" is set at a certain depth to output "filled" or "not filled." This trigger tells the host computer when to read the buffer, instead of a continuous read. This can speed up the read rate, and make sure that the storage location is never full and dropping data.

Although the DAQ is using an FPGA to customize the hardware instead of the dPatch amplifier system, it uses triggers for a faster continuous read of the ADCs. The ADCs in the

project, AD7961s, deliver 5MSPS at 16 bits and push the host computer to its limits, especially when using multiple nodes (Analog Devices). To achieve the speeds needed, the ADC-host communication uses transfer lengths of 64kB per read, triggered by a "half-full FIFO" signal. A FIFO, or First-in First-out system, holds data in buffers, which can be read by the host computer in larger chunks, increasing the transfer speed, and making sure each ADC output is read and collected. With the use of a OpalKelly 7310 board, the DAQ was able to use OpalKelly's FrontPanel API, which gives robust communication between Verilog endpoints and the Python terminal. Most importantly, the API gave a bridge between the verilog triggers and the host computer, allowing the team to focus on each respective portion of the product. Having a direct wire between the half-full verilog output and a host-computer trigger gave the host computer the ability to read bursts of ADC data at the right intervals. Additionally, the FrontPanel API made metadata retention easier to develop and more accurate than relying on researcher input. With the API, when the FPGA is configured, Board ID, device version, etc. are pulled straight from the board and into the JSON file. Researchers, with metadata, can look at larger trends in the data without having to worry about any independent variables like device or device version. From the dPatch amplifier system, the hardware/software integration design helped to design the product, and compensate for disparities in transfer speed.

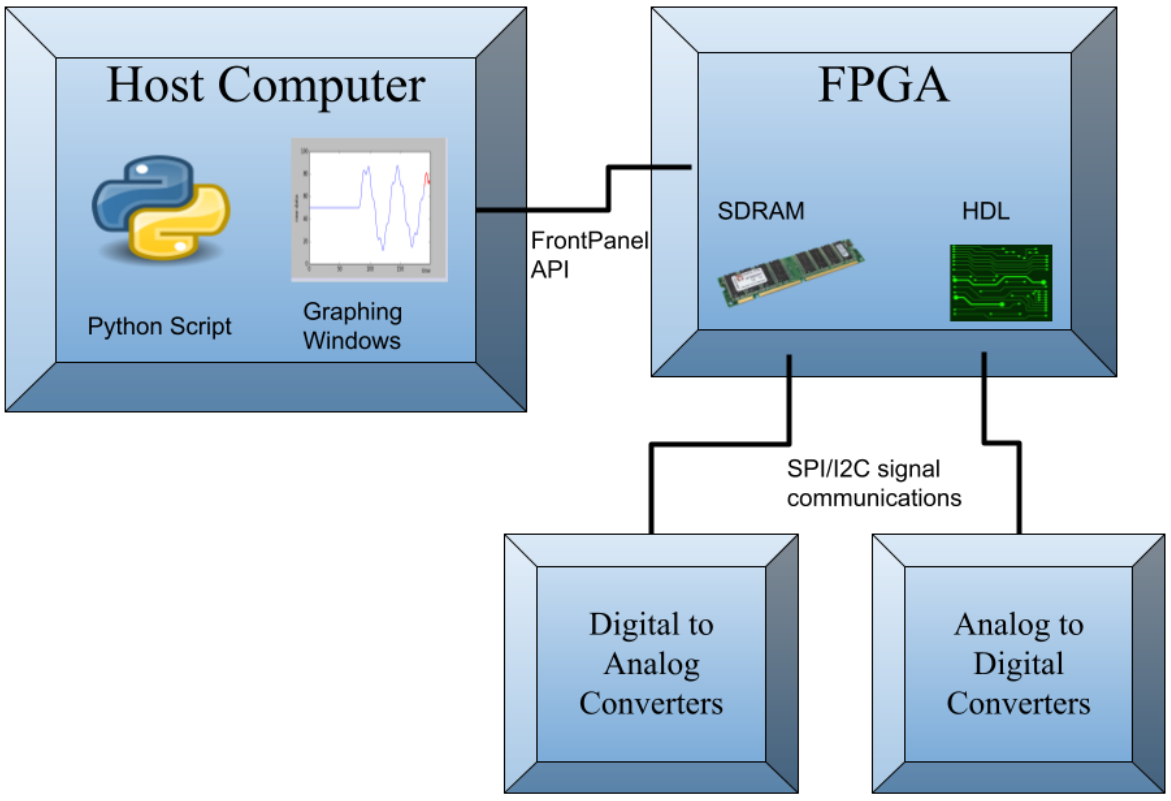
PyQt Package: The PyQt5 toolkit is one of the most powerful and popular cross-platform GUI libraries for Python. Created by RiverBank Computing, it contains a whole host of components, including their QtCore application settings, QtGui GUI software and their pyqtgraphing software, which borrows from Matplotlib as its workhorse (What Is PyQt?). Although other software packages like TKinter offer robust UI packages, the PyQt library was most useful due to its native graphing software. Other languages may be better for UI development, like JavaScript, but due to the intertwined nature of the FrontPanel Python API and the user inputs from the UI, the entire program needed to be written in Python, which limited graphing software options. Additionally, the PyQt package offers code flexibility, which allows the developer to design around the concept of Object Oriented design. This computer science concept gives the developer greater control over the creation and implementation of individual instances of "objects." Having each channel as a single "graphing object" helped to allocate channel addresses and downsampling factors for each individual ADC, and eventually gave the terminal commands better control over these attributes.

After choosing this package, the design was fairly straightforward. The UI centers around a single runnable Python script, which allows the user to input the ADCs to graph, the downsampling factor for each channel, and the waveforms written to the SDRAM. From these user-defined values, the main script begins writing the waveforms to the SDRAM. This on-board memory allows the HDL to read waveforms without using another ADC, and gives the user complete control over the DAC output waveforms. Next, the script creates each channel and subsequent graphing window for each ADC channel. With addresses and downsampling factors tied to each channel, on each update of the plot, the PyQt graphing window updates with a less

data-heavy version of the current channel, and saves the entire data set into an array. This array, when the filemaker is called, is turned into a binary-encoded format and used as the dataset for the HDF5 file.

While the graphing windows are running, the script also allows the user to use the Python terminal. In this terminal, each function defined in the script can be called and executed at any time, tied to each graphing window object. This allows the user to slow down sampling speed, shift values, and pause each channel without having to pre-determine the experiment before execution of the script. Additionally, the Python terminal is useful for saving data. In contrast to a "save and exit" command on termination of the script, calling a "save" command in the terminal allows the user to choose which channels to save, when to save them and allows the user to save two distinct data sets in one run of the program.

Integration:



The goal, when integrating a host of third party hardware and software into a coherent product, was to avoid "Frankensteining" parts together, and instead, create a hierarchical design for the project before writing a single line of code. In order to achieve this, the communication between Python scripts and HDL relied on Python dictionaries filled with key values. A Python dictionary allows you to pair key values, like hexadecimal addresses, with a key word instead of an index

value. With the registers stored by name into their respective chip, the script was able to specify the chip, register and command all in one human-readable line of code. This helped to design across the entire repository all with a couple source files, containing the chips and register maps associated with the hardware. With this system, the team was able to work on each respective part of the project without worrying about memorizing chips and addresses to read/write to. Additionally, every integration obstacle between third party products would be more manageable in a semantically consistent environment.

The first major integration obstacle was the glaring disparity between the FPGA speed and the speed of the host computer. While working with AD7961s, the host computer is sampling significantly faster than it could read, convert and update the plot. To account for this, FIFOs are used to hold data before being dumped into the host computer. The host computer is not necessarily limited by the amount of data it can read at one time, but instead bounded by the time between each reading. By reading the ADC pipe into a FIFO, 64 kbs deep, each reading of the pipe could be time effective. In the verilog code, where the FIFO is created and lives, a half-full signal, tied to the depth of the FIFO, tells the host computer when to take another snapshot of the staged data. If the FIFO filled before a read, data would be lost, and if it was read empty, reading from the FIFO would be wasteful. With snapshots of several thousands samples, each reading can be dumped into the HDF5 file all at once, and downsampled for the graphing windows. This buffer between an incredibly fast FPGA and the host computer was the key component in solving the speed disparity.

The second major integration obstacle was the disparity between the speed of the ADC data coming into the host computer (accounted for in the first integration obstacle) and the speed of the PyQt graphing software. Although the PyQt graphing package updates significantly faster than Matplotlib, or other Python packages, it still cannot compete with AD7961s or any other ADC. For playback and post-processing, downsampling anything going straight into the HDF5 file wasn't an option, but the RAM-heavy graphing software doesn't need every point to be shown. So, to account for this integration obstacle, a downsampling factor for each individual ADC was added into the researcher-inputted source file. When the researcher chooses any downsampling factor higher than 1, the amount of data sent to the PyQt widget is trimmed into a smaller set, while still representative of the original set. This allows the host computer to allocate more RAM to data retention, while still giving a human-readable graphing window centered around general trends, rather than showing every individual data point.

Timing Conclusions: Although not directly relating to the integration of third party software into the UI, the transfer speed testing that was done in concurrence with the development of the product is important to discuss. When testing the read/write speed through the FIFOs, disparities in transfer speed became apparent.

First, both write and read speeds, when the size of the transfer was enlarged, increased, and would lead to a more efficient design. When the read or write transfer size was set to the

minimum (1024 bytes), the transfer speed was only about 100 MB/s, while the maximum transfer size (16kB) would produce transfer speeds topping 300 MB/s. What is important to note, additionally, is that the host computer can only do one read command at a time, and in reality, the multiple channels being read at the same time are actually being held in a queue, and cycled through at the predetermined sample speed. With the host computer's sampling process and the speed increase with FIFO depth in mind, the final design for sampling 4 AD7961s fell into place. Instead of giving each and every AD7961 its own channel and making 4 times the number of calls to "read," instead, the designs relied on a 16kB deep FIFO with each quarter of the data reserved for each chip. This way, instead of making reads for each chip, we could instead increase the size of the FIFO and read all 4 channels at the same time. For example, if each read of the AD7961 was 4kB deep, it would take 8µs for all four of the processes to finish, while with a single FIFO, 16kB deep calling every chip, it would only take 5µs. This increase in speed, over 50%, means that more data can be captured, and that the host computer can spend more energy on other processes.

Complimenting the transfer size speed tests were tests regarding blockread/blockwrite downtines. When calculating the speed that data was going in and out of the FIFO, timestamps were only allotted to the start and the end of the transfer. While this gave insightful information about the transfer sizes that were needed, it doesn't account for the gaps between transfers. Above, when deciding between a burst of smaller reads compared to one larger read, it was assumed that the time between the transfers would be negligible, and that the large disparity in speed between the two transfer sizes would account for this smaller difference in downtime. However, deciding between 4 full-depth reads, and 1 read, accounting for all AD7961s, would make this testing necessary. While Python does not allow for a perfect test, an estimation could be made timing the entire process, and subtracting the elapsed time during the transfers. For example, in one test, the total elapsed time was 8.37 seconds for all of the reads and writes, while the total elapsed time of the transfers themselves was 7.76 seconds. From this, and multiple tests of the same nature, the total time in setup for each transfer was about 7.24%, equalling about .37µs per call. This amount of time, compounding over millions of samples, educated the decision to make every FIFO read of the AD7961s to be in a single sample, rather than compound 4 times the amount of setup intervals over the course of the entire experiment.

The final speed test to complement the UI design was testing the speed of the graphing windows from the PyQt graphing library. Keep in mind, while running these tests, the speed update speed of the graphing windows will depend on the OS, host computer, background processes etc. These widgets, while chosen for being faster than many other graphing packages, still run significantly slower than the ADC channels can pull. While the ADCs are sampling at several million samples per second, the graphing windows can only show around 200 updates per second, and far less with multiple graphing windows shown at once. An update could either be a single point added to the graph, or multiple points (when updating multiple points, the speed was comparable to adding each point individually). The software can support about 9 channels maximum, but at that point, the windows are only showing about 40 updates per second. With

this relationship in mind, the downsampling factor for the graphing windows could take shape. For every 25,000 samples being saved and appended to the HDF5, only one set needs to be added to the graphing window. Any more points, and they would be discarded by the graphing software anyways. This testing allowed both the data retention and graphing windows to be run at maximal speed.

Project Conclusion/Personal Note: While the purpose of the project, designing a UI to facilitate a general purpose operational amplifier, was successful, there is a considerable amount of work outside my own power to bring this product to researchers hands. For instance, Dr. Koerner, in the upcoming year, is planning to design tests, in correspondence with the biology department, to show the DAQ's effectiveness in the lab. Past just creating a reliable and accurate device, it was also my job to make something that was open-source and customizable to the needs of any user. With this comes a responsibility on my part to make sure that my work is well documented and commented on in a way that can be passed onto another researcher or faculty member. Through this paper, the commented Python code and my continued communication with Dr. Koerner and the University, I hope that the work I have done this summer will be a resource for the future of electrophysiology instrumentation.

Works Cited

Analog Devices. “AD7961.” *Datasheet and Product Info | Analog Devices*, www.analog.com/en/products/ad7961.html#product-overview.

Molecular Devices LLC. “Axon Binary File Format User Guide.” Feb. 2013.

Molecular Devices LLC. *PCLAMP 10 Data Acquisition and Analysis Software*. Jan. 2016, mdc.custhelp.com/euf/assets/content/pCLAMP10_User_Guide.pdf.

Sutter Instruments. *DPatch Electrophysiology Amplifier*. 2018, www.sutter.com/AMPLIFIERS/dpa.html.

“What Is PyQt?” *Riverbank Computing | Introduction*, riverbankcomputing.com/software/pyqt/intro.