An Automated, Open-Source Calibration System for Peroxyacyl Nitrates

Harry Hu, Dmitry Ivanov, Hans D. Osthoff

Department of Chemistry, University of Calgary, 2500 University Drive N.W., Calgary, Alberta, Canada, T2N 1N4

Correspondence to: Harry Hu (harry.hu1@ucalgary.ca)

Abstract. A fully automated, open-source calibration system was developed to control the separation and quantification of peroxyacyl nitrates (PANs) using gas chromatography (GC). The system communicates with three data acquisition (DAQ) boards: Measurement Computing's USB-1408FS, USB-1608FS, and National Instruments' USB-6009. These DAQ boards controlled mass flow controllers, performed timed valve switching, and acquired analog signals. The software replaces proprietary LabVIEW code while offering precise control of PAN injections. The system proved to be reliable and robust through multiple tests, including successful isolation of PAN from a mixture containing acryloyl peroxynitrate (APAN) and other atmospheric contaminants. A stress test showed stability and reliability under long term continuous operation. Additionally, the platform allows flexible configuration of flow and timing sequences, requires no licensing, and is portable across compatible Windows systems. Minor usability limitations were identified but did not impact analytical performance, allowing this system to be an effective and reproducible tool for laboratory research of PAN-type compounds.

1. Introduction

Peroxyacyl nitrates (PANs), with the general chemical formula $RC(O)O_2NO_2$, are a class of nitrogen containing organic peroxides synthesized in the troposphere through the photochemical oxidation of volatile organic compounds (VOCs) in the presence of nitrogen oxides ($NO_x \equiv NO + NO_2$). Elevated levels of PANs are usually observed in environments impacted by photochemical smog or biomass burning emissions, both of which indicate degraded air quality (Grosjean et al., 1993; Gaffney and Marley, 2021). In such environments, human exposure to PANs has been linked to unfavourable health effects, particularly eye discomfort and irritation (Altshuller, 1978). PANs are relevant in tropospheric photochemistry since they act as reservoirs for reactive nitrogen oxides. Unlike NO and NO_2, which react and deposit near their emission sources, PANs are thermally more stable under colder conditions and can persist for hours to days, allowing the transportation of NO_x. Once transported to warmer

regions, they thermally decompose, releasing NO_2 and peroxy radicals which enhance local ozone production (Roberts, 1990).

While acetyl peroxynitrate (also known as peroxyacetyl nitrate or PAN) is usually the most abundant, other PAN-type species such as propionyl peroxynitrate (PPN), acryloyl peroxynitrate (APAN), methacryloyl peroxynitrate (MPAN), i-butyryl peroxynitrate (PiBN), n-butyryl peroxynitrate (PnBN), benzoyl peroxynitrate (PBzN), and furoyl peroxynitrate (fur-PAN) have also been detected (Roberts et al., 2022).

A representative example of PAN formation is methacryloyl peroxynitrate, which is produced from methacrolein (2-methylprop-2-enal) (Mielke & Osthoff, 2012). The mechanism begins with abstraction of the aldehydic hydrogen by the hydroxyl radical, forming an acyl radical. This radical reacts with molecular oxygen to form a peroxyacyl radical, which then combines with NO_2 to produce MPAN. The reaction sequence is as follows:

$$CH_2=C(CH_3)CHO + OH \rightarrow CH_2=C(CH_3)CO + H_2O \quad (R1)$$

$$CH_2=C(CH_3)CO + O_2 \rightarrow CH_2=C(CH_3)C(O)O_2 \quad (R2)$$

$$CH_2=C(CH_3)C(O)O_2 + NO_2 \rightleftharpoons CH_2=C(CH_3)C(O)O_2NO_2 \quad (R3)$$

Mixing ratios of PANs in ambient air have been quantified using gas chromatography with electron capture detection (GC-ECD) (Darley et al., 1963; Flocke et al., 2005; Tokarek et al., 2014) and thermal dissociation chemical ionization mass spectrometry (TD-CIMS) (Slusher et al., 2004; Zheng et al., 2011; Mielke & Osthoff, 2012). These methods require calibration as instrumental response varies between compounds.

Synthesis of PANs for use as standards is limited by their instability. For example, diffusion sources often produce mixed or impure outputs, especially for compounds like APAN or MPAN which have a reactive double bond in the side chain. As a solution to this, PANs have been generated by the photolysis of a carbonyl compound in the presence of NO_x. Carbonyl compound precursors include ketones such as 2-propanone and 3-pentanone (Warneck & Zerbach, 1992; Furgeson et al., 2011; Rider et al., 2015; Pätz et al., 2002), aldehydes such as ethanal or propanal (Volz-Thomas et al., 2002), and acyl chlorides such as acryloyl and methacryloyl chloride (Veres and Roberts, 2015). A recent example of PAN formation using the photochemical source is APAN, which is produced from acrolein (prop-2-enal) (Gomez et al., 2025).

Separation of PAN-type compounds prior to detection is important for two reasons. First, the thermal dissociation cavity ring-down spectroscopy (TD-CRDS) instrument used for quantification is unable to distinguish between overlapping species within the same chemical family. It only measures the thermally dissociated product nitrogen dioxide, so without prior separation, the signals from different species overlap and cannot be resolved. Therefore, a preparatory-scale gas chromatography has been used beforehand to separate analytes prior to their quantification (Roberts et al., 2022).

National Instruments (NI) LabVIEW has served the Osthoff group as a software platform for instrument control and data acquisition (Tokarek et al., 2014). LabVIEW's graphical interface and integration with NI data acquisition (DAQ) devices (also referred to as "boards"), such as the USB-6008/6009 (Figure 1) and Measurement Computing Corporation's USB-1408FS/1608FS (Figure 2) have made it a convenient solution for acquiring data from analytical instruments. However, after NI's acquisition by Emerson in 2023 and changes of its licensing model to a subscription model, LabVIEW has become expensive to maintain for university laboratories.

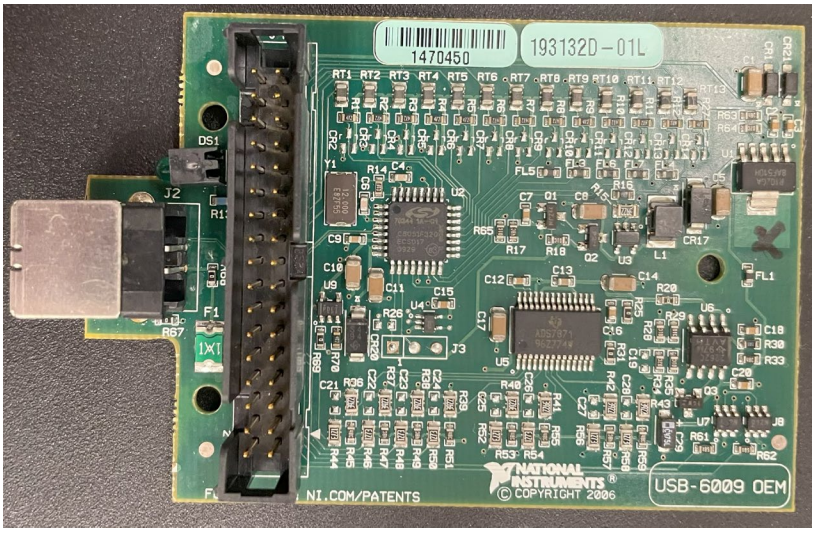

Figure 1. National Instruments USB-6009 OEM board used for data acquisition and control of actuators within LabVIEW environments.

Figure 2. The Measurement Computing USB-1408FS DAQ board. Commonly used in LabVIEW based setups for analog and digital control.

To address these limitations, open-source alternatives such as Python may be used. Python is an open-source programming language that is commonly used in science and engineering due to its simplicity, readability, and broad range of applications. This programming language is particularly well-suited to research environments, as it allows chemists to automate experiments, control hardware, process data, and generate graphical user interfaces (GUI) within a single environment.

Python can control laboratory instruments through libraries such as “mcculw”, which supports Measurement Computing devices (e.g., USB-1408FS or USB-1608FS), or “nidaqmx”, which communicates with National Instruments hardware (e.g., USB-6008 or USB-6009). These tools make it possible to replace all of LabVIEW’s functions, such as voltage measurements, signal timing, and valve control. Additionally, Python’s compatibility with data analysis and graph plotting libraries (e.g., “numpy”: for efficient computation, “matplotlib”: a data visualization library) makes it a powerful tool for researchers who value transparency and long-term sustainability in the lab.

In this work, we demonstrate that a custom-built calibration system programmed in Python provides a functional alternative to proprietary software such as National Instruments’ LabVIEW. The Python-based control system was implemented to actuate a 2-position valve for sample collection, allowing the separation of PAN-type compound

mixtures using a preparatory scale gas chromatography column. This setup that enables automation of valve switching and data acquisition without the licensing constraints associated with commercial software.

2. Methods

The main instrument used in this project was the GC-ECD, which was used for the separation and detection of PAN-type compounds (Tokarek et al., 2014). Although a TD-CRDS (Taha et al., 2018) is available in the lab and relevant for future work, it was not used in this project. However, the software developed here is compatible with TD-CRDS and can be adapted to automate its operation as well.

A Valco Instruments Company Inc. (VICI) two-position microelectric actuator was used to control switching in the GC-ECD (Figures 3A, 3B). This actuator rotates a multi-port valve between two fixed positions and can be triggered either manually or through a digital input/output port that is set to either on (5 V input) or off (0 V input). We controlled the actuator electronically by sending an electrical signal from the DAQ board to the actuator's input pins. Specifically, sending 0 V to pin 5 activated position A, while sending 0 V to pin 6 activated position B (Figure 4). The actuator itself was powered through a 24 V power supply.

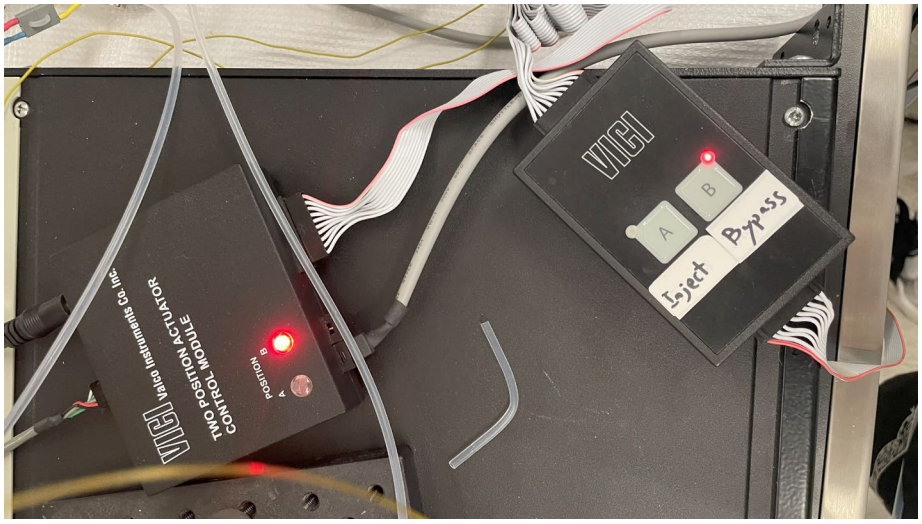

Figure 3A. A VICI two-position actuator control module used to switch flow between the bypass and separation lines. The illuminated indicator corresponds to the active flow path.

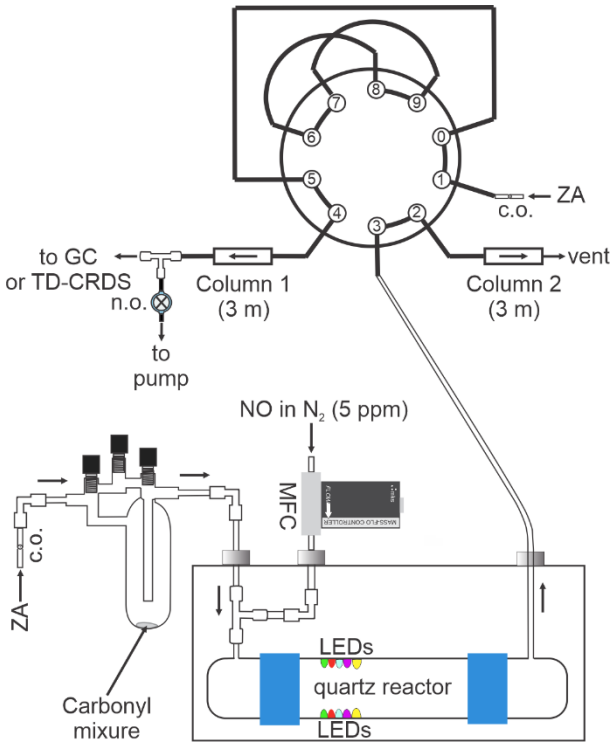

Figure 3B. Schematic of the GC-ECD experimental setup with a two-position VICI microelectric actuator for valve switching. The actuator alternates the valve between two positions, directing flow either through column 1 (to detector) or column 2 (to vent). Abbreviations: ZA: zero air; c.o.: critical orifice.

Digital Control of the Actuator

Digital Communication Protocol

Pins 1 and 2 provide ground and +5 volt outputs, respectively; pins 3 and 4 are TTL outputs for Position A and Position B, and are considered asserted at 0 volts and deasserted at 5 volts. (This is sometimes referred to as "negative true logic".) Pins 5 and 6 are digital inputs for switching to Position A and Position B. They can be driven either by 5 volt TTL/CMOS logic or by contact closure to ground (Pin 1). Isolated contact closure outputs are available at Pins 7 and 8 for Position A and Pins 9 and 10 for Position B. If there is a positioning error due to valve sticking, clamp ring slippage, etc., the output is set to "0" (all lines high for a negative true output).

Digital I/O Cable	
Pin #	**Signal Description**
1	Ground
2	+5 VDC
3	Position A output
4	Position B output
5	Position A input
6	Position B input
7	Position A relay contact output
8	Position A relay contact output
9	Position B relay contact output
10	Position B relay contact output

Figure 4. Digital input/output pin assignments for the VICI two-position microelectric actuator (VICI, 2009).

The VICI valve was an on/off switch between two flow paths (Figure 4): the bypass line and the separation loop. The separation loop directs flow to the instrument (either GC or TD-CRDS), while the bypass line vents to the fume hood. The GC also contains a second internal VICI valve which governs sample injection onto the column.

2.1. Measurement Computing's USB-1408FS

An automated, open-source software platform was written in Python (v3.13) using the PyCharm integrated development environment to control valve switching and perform real-time signal acquisition during chromatographic experiments. The program interfaces with a Measurement Computing USB-1408FS DAQ board via the "mcculw" library, allowing direct communication between both analog and digital inputs and outputs. The system was debugged and refined using generative AI to optimize both code structure and graphical interface behaviour.

The USB-1408FS can be used in experiments requiring signal acquisition and timed valve switching. It provides four single-ended analog input channels, two analog outputs, and sixteen digital input/output lines (Figure 5). In this application, analog input channel 0 was connected to the signal output of the GC-ECD, and digital output lines 0 and 1 on port A were connected to the VICI valve controller.

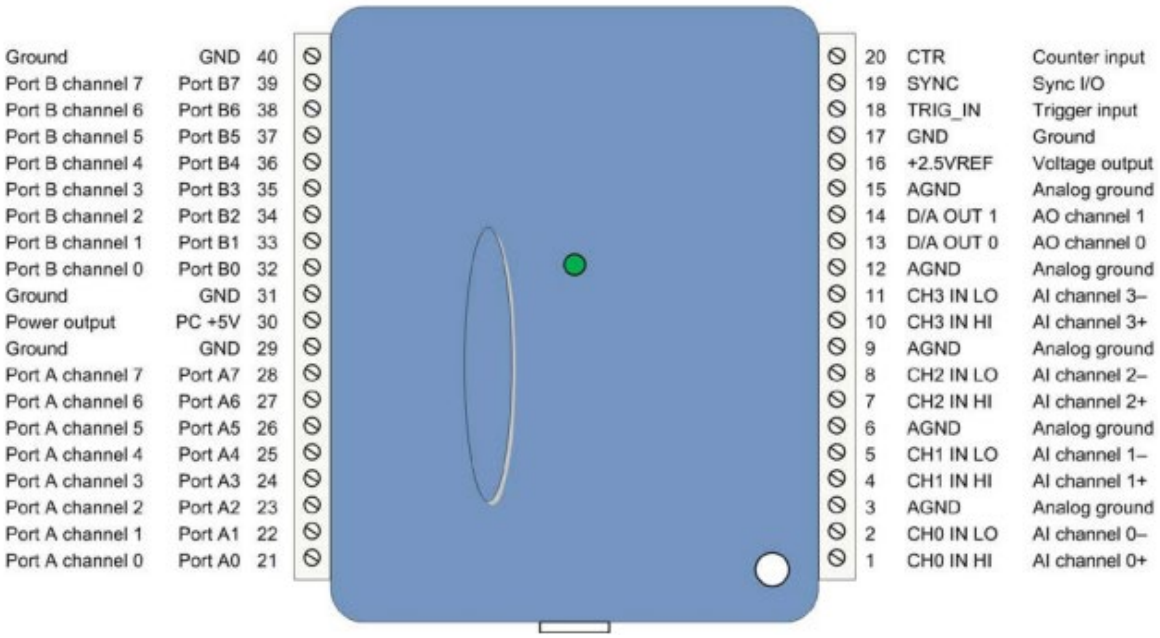

Figure 5. Pinout diagram of the Measurement Computing USB-1408FS DAQ board. (Measurement Computing Corporation, 2023a)

The software was modularized into five core components: valve control, data acquisition, GUI display, user configurable settings, and a main launcher script. This was designed to be modular, which allows the user to configure experimental parameters without the GUI in case an unforeseen issue arises. The GUI was developed using the "tkinter" package, a Python toolkit for building windows-based GUIs. Two tabs were provided: configuration and control. The configuration tab allows users to specify hardware channels (analog/digital outputs) and select which DAQ board to control, in case there is more than one DAQ board connected. The control tab displays the experiment progress in real time. Elements displayed consisted of elapsed time, signal voltage readings and graph, the valve switching schedule and manual override buttons to switch valves.

Two valves were controlled by the USB-1408FS DAQ board. Two digital output lines were used to operate valve positions A and B. These valves were toggled by changing the voltage on individual pins. Digital valve control was achieved by setting the appropriate output line "low" (0 V) which the VICI valve controller interprets as a signal to switch (Figure 6). For example, to activate position A, we set its port to "low" state to complete a circuit. To deactivate position A, we return the line to a "high" (5 V) state.

```python
# Function to switch to Position A (open Valve A, close Valve B)
def set_valve_position_a(self) -> None:
    if not self._hardware_available:
        print("Simulating valve A open")
        return

    try:
        ul.d_bit_out(self.board_num, DigitalPortType.FIRSTPORTA, 0, 0)  # Valve A ON
        ul.d_bit_out(self.board_num, DigitalPortType.FIRSTPORTA, 1, 1)  # Valve B OFF
    except Exception as e:
        print(f"Error setting valve A: {str(e)}")

# Function to switch to Position B (open Valve B, close Valve A)
def set_valve_position_b(self) -> None:
    if not self._hardware_available:
        print("Simulating valve B open")
        return

    try:
        ul.d_bit_out(self.board_num, DigitalPortType.FIRSTPORTA, 0, 1)  # Valve A OFF
        ul.d_bit_out(self.board_num, DigitalPortType.FIRSTPORTA, 1, 0)  # Valve B ON
    except Exception as e:
        print(f"Error setting valve B: {str(e)}")
```

Figure 6. Python functions used to control valve A and valve B through digital output lines on the USB-1408FS DAQ board. Each function selectively sets the appropriate digital lines to either a low or high state. set_valve_position_a() activates valve A while ensuring valve B remains deactivated. Conversely, set_valve_position_b() activates valve B while ensuring valve A remains deactivated. These operations prevent both valves from being activated at the same time and avoid interference with other unused digital lines.

Valve transitions could be scheduled through the GUI using a timing table (Figure 7). These time points were stored as a schedule and executed automatically relative to the experiment start time. Multiple swaps were allowed and configurable in case more than one compound needed to be separated.

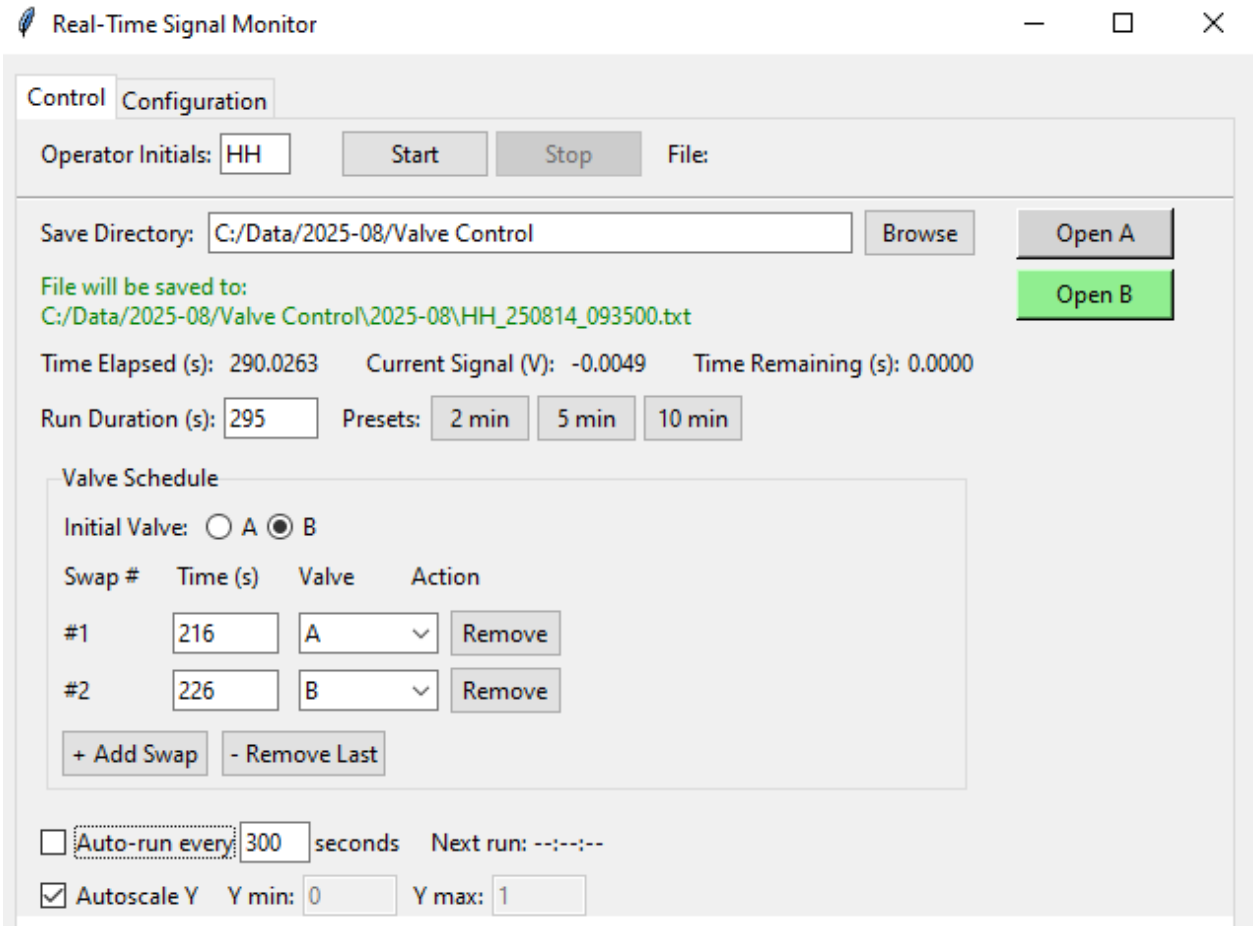

Figure 7. Custom Python-based GUI for scheduling valve transitions. The software allows timed switching between valve positions A and B, with configurable swap times and run durations. Execution timings were relative to the start of data acquisition.

If the DAQ board or the VICI valve controller was not detected, the system entered a simulation mode, and the valve commands were printed to console rather than physically executed. This allowed offline testing and debugging.

Voltage signals were sampled from an analog channel at 10 kHz. Data were collected in blocks of 100 samples, and the average voltage of each block was used for display and storage.

Run durations were user-configurable, where the user enters the desired run duration in seconds as seen in Fig. 7. Preset buttons for 2-, 5-, or 10-minute runs also exist as these run durations were commonly used in the group. An auto-run mode feature was introduced which allowed repeated data collection at fixed intervals. This is intended

for unattended operation during long term experiments. To ensure consistent organization of data files and alignment with time-based protocols, each run begins precisely at a time that is an exact multiple of the run duration. For instance, if the run duration is set to 300 s (5 min), data collection would begin at timestamps such as 13:05:00, 13:10:00, 13:15:00, etc. The software continuously calculates the time remaining until the next scheduled run and starts it automatically at the appropriate moment.

After an experimental run, all data is recorded to a tab delimited text file with time values expressed in seconds since January 1st, 1904. Although an unusual time format, this epoch format is commonly used in scientific computing. In our case, Igor Pro uses January 1st, 1904, as a reference date for time calculations (Figure 8). The data file is named using the group's convention "UI_YYMMDD_HHMMSS" where "UI" represents user's initials, YYMMDD represents the current date (year/month/day), and HHMMSS represents the current time (HH:MM:SS, 24-hour format).

```
# Generate filename for this run
initials = settings.operator_initials.upper()
stamp = datetime.now().strftime("%y%m%d_%H%M%S")
self.current_filename = f"{initials}_{stamp}"


# File I/O
def writeData(self, data: list[tuple[float, float]]) -> None:
    if not data or not self.filename:
        return

    with open(self.filename, "w", encoding="utf-8") as file:
        for epoch, v in data:
            #format: "[time since Jan 1st 1904] TAB [Signal up to 4 decimal places]"
            file.write(f"{epoch:.4f}\t{v:.4f}\n")
    print(f"Saved {len(data)} rows to {self.filename}")
```

Figure 8. Python script that automatically generates filenames using operator's initials and timestamp in the format YYMMDD_HHMMSS (top). Python function writeData() used to save experimental results to a tab delimited text file (bottom).

This software enables chemists to run time-resolved chromatography experiments on Measurement Computing's USB-1408FS DAQ board with precise valve control, real-time signal monitoring, and automated data logging, without relying on proprietary software such as LabVIEW. The open-source design, GUI accessibility, and the fallback simulation mode make it suitable for both research and teaching laboratories.

2.2. Measurement Computing's USB-1608FS

A second version of the software was developed for compatibility with Measurement Computing USB-1608FS DAQ board. Although similar in function to the USB-1408FS, the USB-1608FS features a different pinout and analog input configuration compared to the USB-1408FS (Figure 9). This required slight modifications to the software due to how analog input signals were handled and which ports were used to control the valves. The user interface and experimental procedure remained the same, but switching to the USB-1608FS required internal reassignment of digital lines.

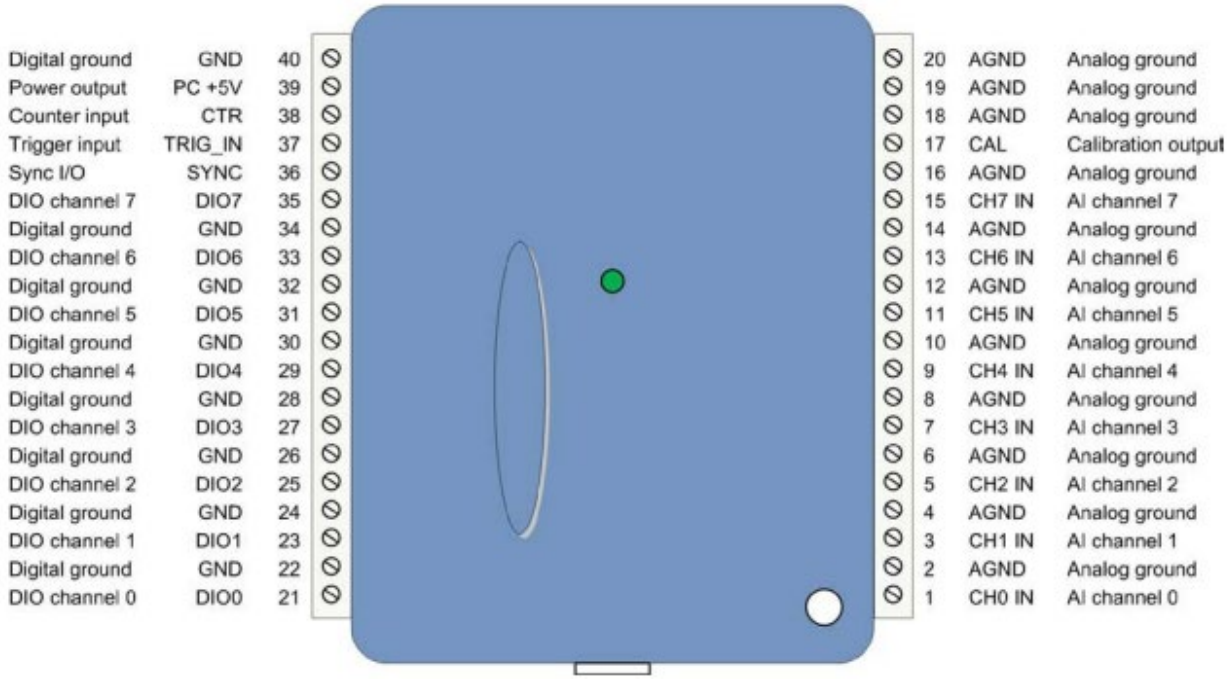

Figure 9. Pinout diagram of the Measurement Computing USB-1608FS DAQ board. (Measurement Computing Corporation, 2023b)

The board was connected to the GC-ECD using the same principle as the USB-1408FS, but with modifications to account for the USB-1608FS pin layout (Figure 10). Analog channel 0 was connected to the analog signal output from the electron capture detector, just like the USB-1408FS. Valve control was achieved through digital output lines DIO3 and DIO1 on the DAQ board. These lines were wired to the GC's internal VICI valve controller where it switches the columns.

```python
def _set_valve(self, a_state: int, b_state: int):
    """Set both valve control lines to specified states"""
    if not self._hardware_available:
        return

    try:
        # Set Position A control (DIO3)
        ul.d_bit_out(self.board_num, DigitalPortType.AUXPORT, 3, a_state)
        # Set Position B control (DIO1)
        ul.d_bit_out(self.board_num, DigitalPortType.AUXPORT, 1, b_state)
    except Exception as e:
        print(f"Error setting valve states: {str(e)}")
```

Figure 10. Python function _set_valve() used to control valve states via digital outputs on the USB-1608FS DAQ board. Same logic as for the USB-1408FS, but different lines.

Despite having different ports and channels, the GUI, experimental procedure, and software logic remained the same. All other features, including run scheduling, auto-run mode, and data storage conventions, were kept identical between versions. The modular software design allowed both boards to be supported simultaneously with no adjustments. This ensured that users could operate either DAQ board interchangeably within the same experimental set up, without requiring reconfiguration or changes to their established GC-ECD procedures.

2.3. National Instruments' USB-6008 and USB-6009

The Osthoff group uses an in-house-designed mass flow and valve control boxes powered by a National Instruments USB-6008 or USB-6009. Analog output from the DAQ is used to set and read supplied voltages to mass flow controllers (MFCs), while digital outputs are used to toggle up to four extra valves in the lab.

For this project, MFCs were used to control the flow of NO gases into the PAN generator and the instrument system. Each MFC has a specific range of flow rates, either 30, 15, 5 or 1 standard liters per minute (SLPM), or 500, 100, 20, 10 standard cubic centimeters per minute (SCCM). Normalization allows experiments to be compared throughout different temperatures and pressures. Each MFC has a linear response between the input voltage and its corresponding flow rate. For example, a 30 SLPM MFC delivers its maximum flow (30 SLPM) when supplied with 5 V. Supplying 2.5 V to that same device produced half the flow (15 SLPM). Similarly, a 500 SCCM MFC would output 250 SCCM when provided 2.5 V (Figure 11).

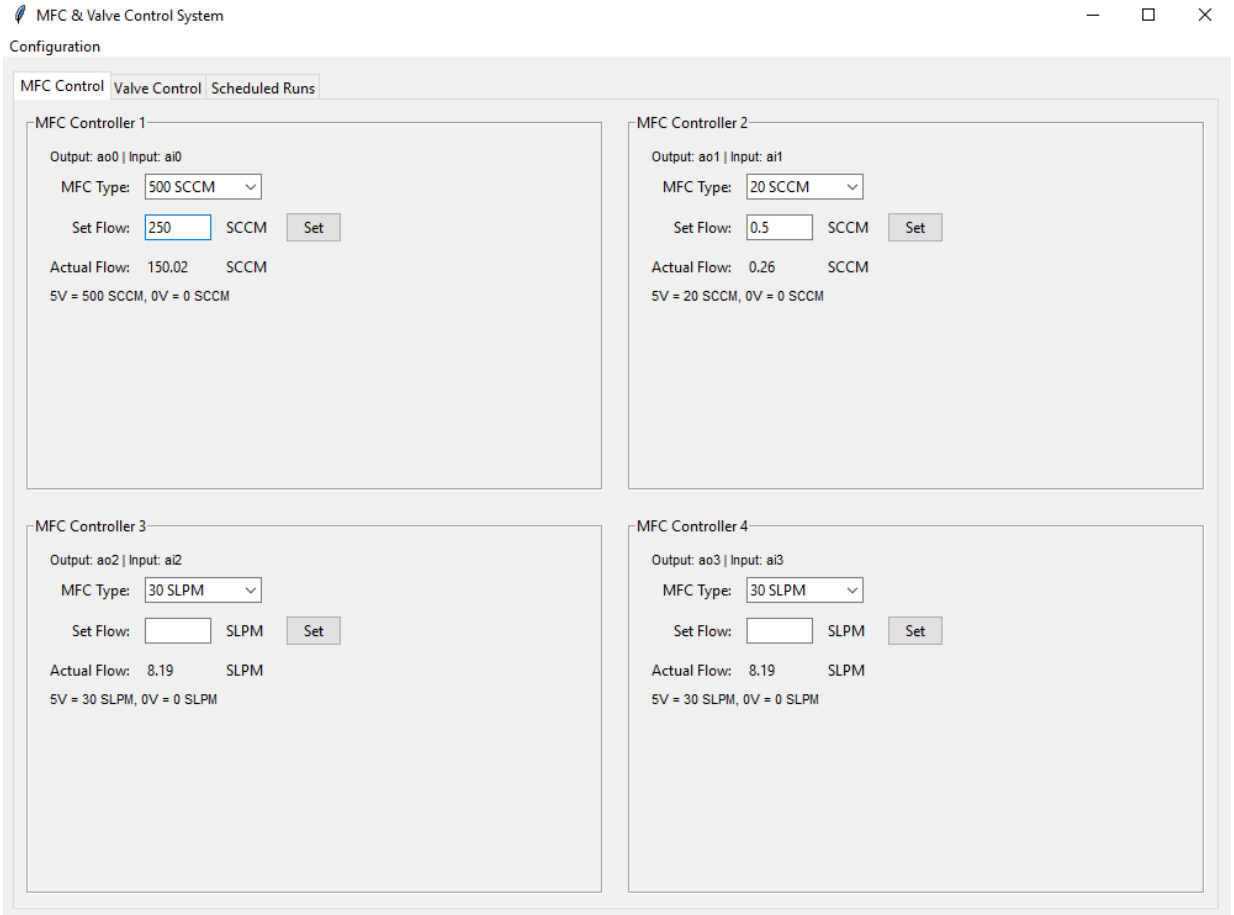

Figure 11. The MFC control user interface. Each panel corresponds to one MFC channel, where the user can select the MFC type, input a target flow rate, and monitor the actual flow. The system accepts SLPM or SCCM depending on the selected MFC and linearly maps the analog voltage to the maximum flow rate of the device. The bottom of each panel displays flow rate to voltage conversions for reference.

Valve toggling was handled by digital output lines, where each line was assigned to a valve. Similarly to both Measurement Computing's DAQ boards described previously, the valves connected to the NI USB-6009 could be scheduled to switch states at user-defined times. A GUI allowed users to input swap times relative to the experiment start time, enabling precise control over gas routing sequences (Figures 12, 13). The implementation of time based digital valve control provided a consistent and automated method to coordinate gas delivery and separation across multiple instruments.

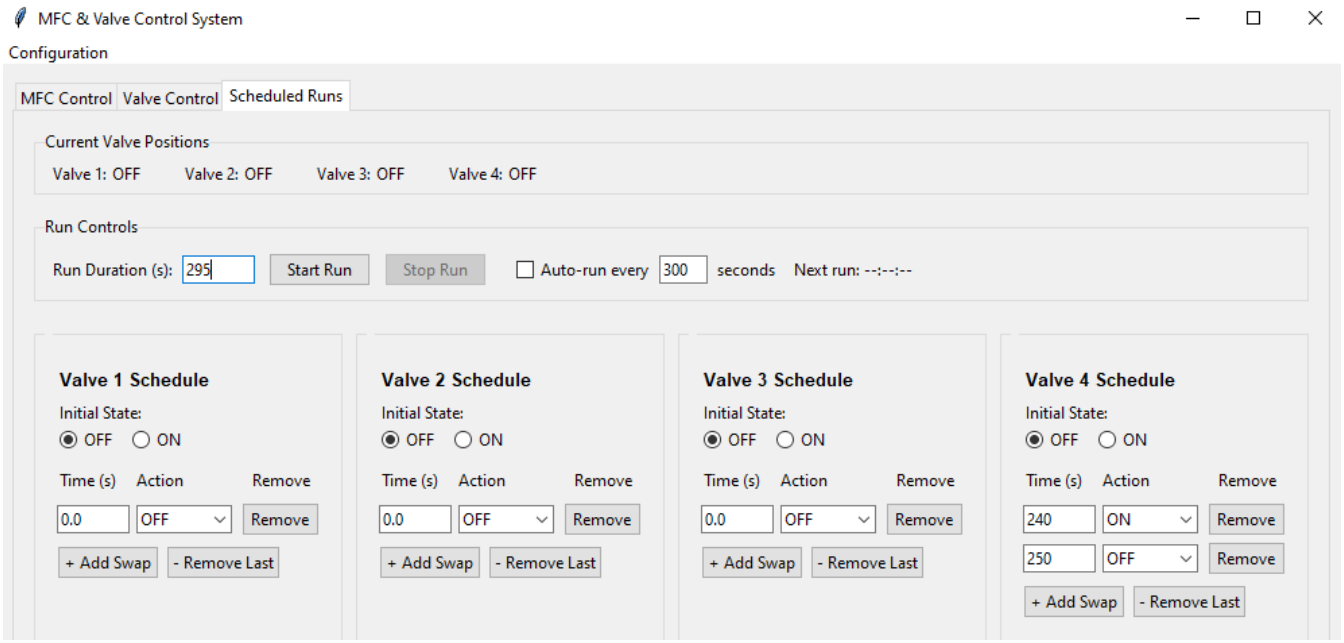

Figure 12. The "Scheduled Runs" tab within the MFC and Valve Control System GUI. Each valve can be individually scheduled to swap states (ON/OFF) at specified times. The run duration and swap times are relative to the experiment start time. An auto-run mode allows repetition of runs for unattended operation.

```
def cancel_schedules(self):
    """Cancel all scheduled jobs for this valve"""
    for job_id in self.swap_job_ids:
        self.master.after_cancel(job_id)
    self.swap_job_ids = []

def set_valve_state(self, state):
    """Set valve state and track current state"""
    self.current_state = state
    self.daq.write_digital(self.port_line, state)

def turn_off(self):
    """Turn off this valve"""
    self.current_state = False
    self.daq.write_digital(self.port_line, False)
```

Figure 13. Example implementation of valve control in Python. Functions allow cancellation of scheduled events (cancel_schedules), setting a valve state (set_valve_state), and shut down (turn_off). These functions are implemented with the DAQ digital output lines to enable automated toggling of valves during experiments.

3. Results and Discussion

3.1. System Testing and Performance

The Python program was tested across multiple days and configurations to assess its performance in instrumental control and separating PAN-type compounds from mixtures using the setup shown in Fig. 3B.

On July 28th, a successful injection of a PAN mixture into the GC using the external valve was demonstrated (Fig. 14). The external valve regulated how much of the mixture entered the GC column. This setup was sufficient to direct gas pulses into the column and provided sharp chromatographic peaks. During this testing phase, one computer tested the external valve swaps while another computer was controlling the interior GC valves and the MFCs. Valve synchronization testing revealed a consistent offset of 21 s between system clocks of the two computers used during this phase. This was compensated by delaying one system's swap, achieving synchronized injection.

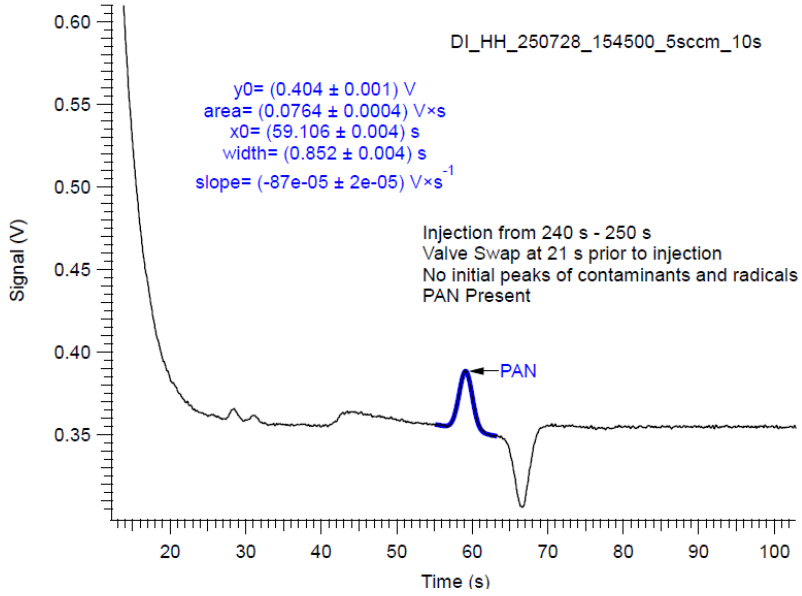

Figure 14. Chromatogram shows successful PAN injection using the external valve. The injection occurred between 240-250 seconds, with a 21 second valve swap delay used to compensate for clock offsets between two computers controlling the system.

By July 31st, complete software control of the GC system (both external and internal valves) was achieved through Python code using both USB-1408FS and USB-1608FS (Figure 15). This milestone marked a full transition away from LabVIEW for the GC side of the setup. The Python-based system enabled automated and precise injection controls and bypass switching.

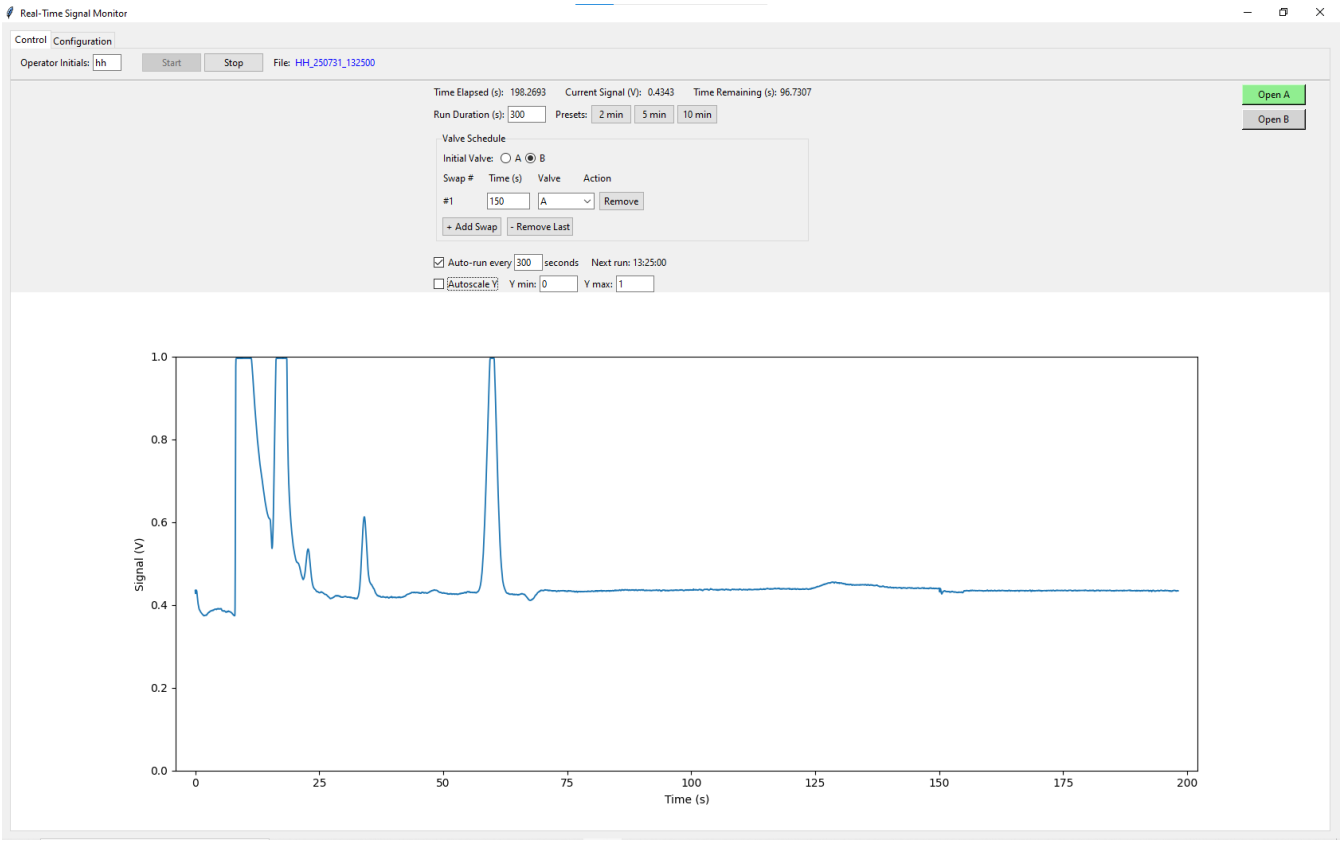

Figure 15. Screenshot taken on July 31st, 2025 (file: HH_250731_132500) showing the first successful chromatographic peaks from fully Python controlled GC injections and valve switching.

Between July 31st and August 1st, a 24 hour stress test was conducted to determine the system's robustness under extended operation (Figure 16). Blank injections were repeatedly scheduled via the Python control system using the auto-run feature. These tests confirmed that the software could maintain accurate timing, perform valve toggling, and log data continuously without error.

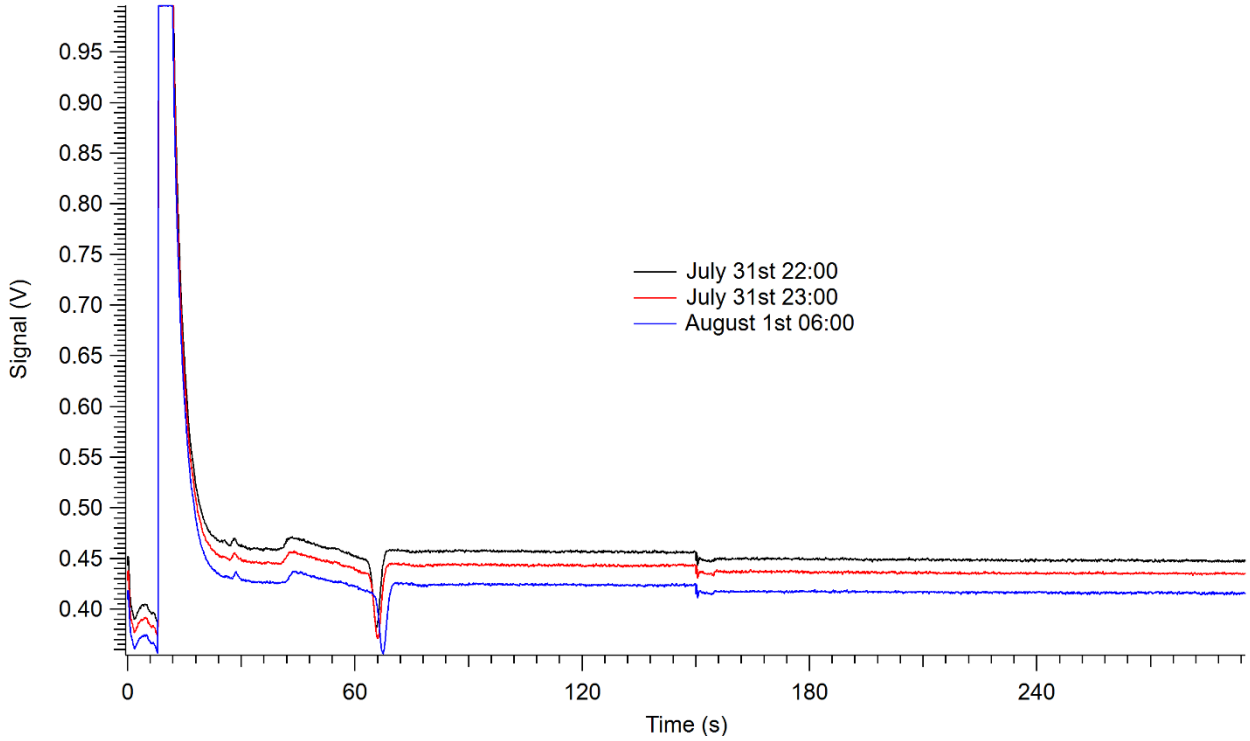

Figure 16. Overlay of three arbitrary blank runs taken during a 24 hour continuous stress test between July 31st and August 1st, 2025. Blank runs were triggered at regular intervals using the auto-run feature of the program. The internal GC valve swap was triggered at 150 s of each run. The consistent shape and timing of the valve triggering across all chromatograms confirm software stability.

As the physical setup of the instrument was completed prior to my involvement in the project, the specific functions and controls of each valve had to be discovered through a combination of system testing and signal response patterns. During further experimentation, it was determined that the GC system incorporates three valves arranged in series, each responsible for a unique phase of the injection and separation process. In chronological order, the first valve is a VICI actuator responsible for the amount of PANs that enters the pre-column. This valve is controlled by the USB-1408FS. The second valve, a needle valve between the pre-column and the GC, is toggled by a digital output line on the NI USB-6009 DAQ board. The third valve, the internal GC injection valve, is controlled by the USB-1608FS with another VICI actuator.

The needle valve, controlled by the NI DAQ, plays a role in separating individual PAN-type compounds. By precisely controlling the timing of when gases pass from the pre-column into the GC, it determines which species

are injected. For example, PAN can be isolated from contaminants by changing the needle valve injection timing as seen in Fig. 17. Additionally, if PAN and APAN are in the same mixture, they can be isolated in the chromatogram (Figure 18). This discovery refined the injection scheduling logic and contributed to the reliable and reproducible separation of PAN-type compounds.

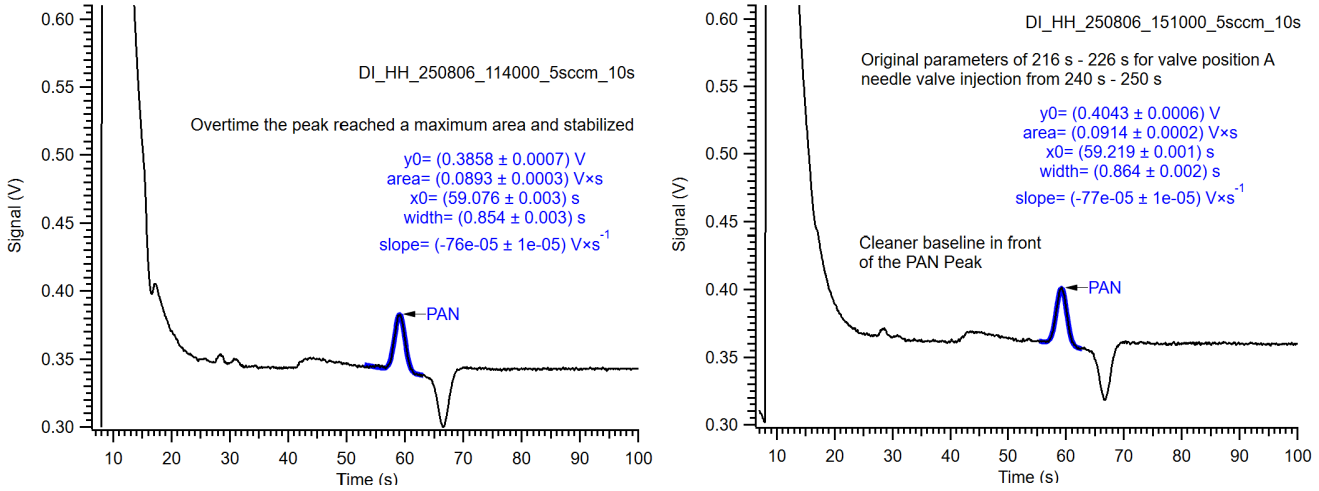

Figure 17. Comparison of two chromatograms showing the effect of needle valve timing on PAN peak isolation.
Left: Chromatogram showing a PAN peak in a system with no needle valve interference.
Right: Chromatogram showing the same PAN peak affected by the needle valve swapping.

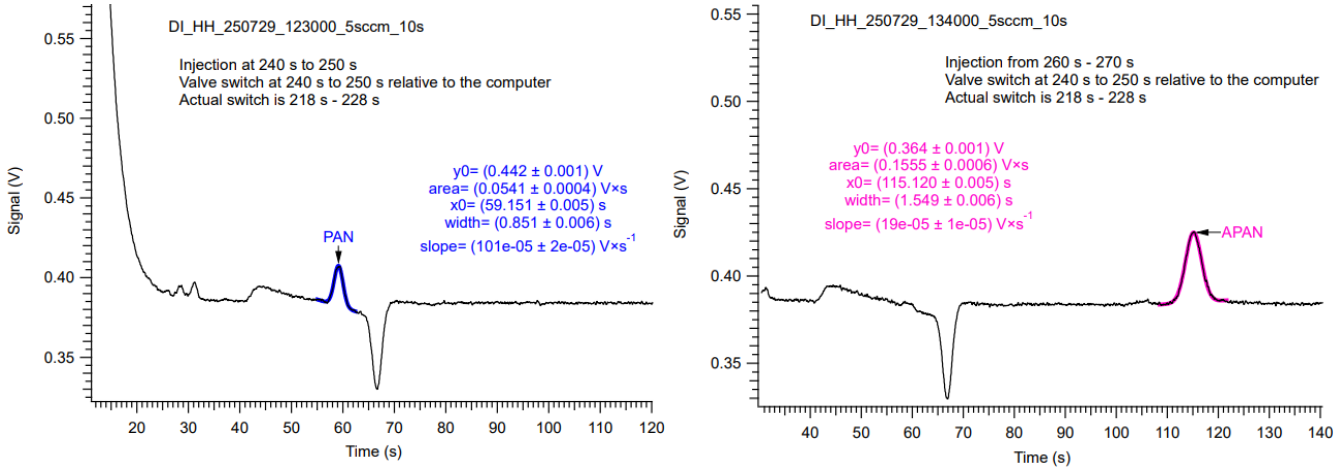

Figure 18. Comparison of two chromatograms showing the effect of needle valve timing on PAN peak isolation.
Left: Chromatogram showing the isolation of PAN from a mixture of PAN and APAN.
Right: Chromatogram showing the isolation of APAN from a mixture of PAN and APAN.

3.2. Limitations

A limitation of the current implementation is its dependency on specific operating systems and hardware environments. The software could only be successfully run on 64-bit Windows 10 or Windows 11 machines. Compilation on a 32-bit Windows 10 system was unsuccessful since several versions of the “matplotlib” library failed to compile due to unresolved Microsoft Visual Studio C/C++ build errors.

Attempts to run the program on a Raspberry Pi running Raspbian were also unsuccessful. Within the Raspbian operating environment, the Python libraries "mcculw" and "nidaqmx" do not exist. Measurement Computing provides an alternative library, “uldaq” for the USB-1408/1608FS DAQ boards which is compatible with Linux operating systems. With sufficient time and development effort, refactoring to interface with “uldaq” is a feasible path in the future.

Currently, the code does not support simultaneous communication with multiple DAQ boards. Each board must have its own instance therefore, synchronized control across multiple DAQ boards may be more difficult.

3.3. Conclusion

This work demonstrates that an open-source Python based software can fully replace LabVIEW in operating a GC setup for PAN-type compound analysis. It successfully controlled all components, including flow controllers, valve toggles, and injections. Three DAQ boards were used: Measurement Computing’s USB-1408FS, USB-1608FS, and National Instruments’ USB-6009. Testing confirmed reliable performance, with features such as synchronized injections, long run time stability, and precise control of compound separation through valve timing.

Compared to the legacy LabVIEW system, the Python code offered better versatility with user defined swap sequences, and full functionality without licensing constraints. Results include effective separation of PAN from APAN and contaminants, and supported full automation of the GC system. Minor issues, such as overlapping valve indicator lights and a text field bug in the USB-1408FS interface, were identified but did not affect core performance. This system provides a reliable, cost-effective platform for PAN calibration and GC separation workflows.

Code Availability

All code and documentation are uploaded to a GitHub repository (https://github.com/illiteratewaffle/Analytical-Data-Acquisition).

Measurement Computing's USB-1408FS

DataAcquisition.py

```
# DataAcquisition.py
import os

import numpy as np
from mcculw import ul
from mcculw.enums import ULRange
from mcculw.ul import ULError
import ctypes as ct
import threading, queue, time
from datetime import datetime

from Settings import settings


class DataAcquisition:
    blockSize = 100
    samplingFrequency = 10_000  # Hz

    def __init__(self):
        # Board-specific settings (updated to use settings)
        self.board_num = settings.ai_board_number
        self.channel = settings.ai_channel
        self.ai_range = ULRange.BIP20VOLTS
        self._hardware_available = False
```

```
        try:
            # Try to access the board to verify it exists
            ul.get_board_name(self.board_num)
            self._hardware_available = True
        except ULError:
            print(f"Warning: Analog input board {self.board_num} not found. Running in simulation mode.")
            self._hardware_available = False

        # Pre-allocate one-block buffer
        self._buf = (ct.c_uint16 * self.blockSize)()

        # Run bookkeeping
        self.data: list[tuple[float, float]] = []

        # initial file name
        self.filename = None

        # Threading helpers
        self._queue: queue.Queue | None = None
        self._thread: threading.Thread | None = None
        self._running = threading.Event()

    def set_filename(self, filename):
        self.filename = filename

    # Public control surface
    def attach_queue(self, q: queue.Queue) -> None:
        """GUI supplies a queue to receive (t_rel, volts)."""
        self._queue = q
```

```
    def start(self) -> None:
        if self._thread and self._thread.is_alive():
            return
        self._running.set()
        self._thread = threading.Thread(target=self._worker, daemon=True)
        self._thread.start()

    def stop(self, join_timeout: float = 1.0) -> None:
        self._running.clear()
        if self._thread:
            self._thread.join(timeout=join_timeout)
            self._thread = None
        self.writeData(self.data)  # auto-save
        self.data = []  # clear for next run

    # Background worker — runs in its own thread
    def _worker(self) -> None:
        t0 = time.perf_counter()
        while self._running.is_set():
            volts = self.getSignalData()
            if volts is None:
                continue  # skip bad scan, keep running

            t_rel = time.perf_counter() - t0
            epoch1904 = self.getTimeData()
            self.recordData(epoch1904, volts)

            if self._queue:
                try:
                    self._queue.put_nowait((t_rel, volts))
                except queue.Full:  # drop oldest if GUI lags
```

```
                _ = self._queue.get_nowait()
                self._queue.put_nowait((t_rel, volts))
        # a_in_scan blocks ≈ blockSize/samplingFrequency
        # so no extra sleep is needed

    # Low-level helpers
    def getSignalData(self) -> float | None:
        if not self._hardware_available:
            # Simulate a sine wave when hardware isn't available
            return 2.5 + 2.5 * np.sin(time.perf_counter() * 2 * np.pi * 0.1)

        try:
            ul.a_in_scan(self.board_num,
                         self.channel, self.channel,
                         self.blockSize, self.samplingFrequency,
                         self.ai_range, self._buf, 0)

            counts = np.ctypeslib.as_array(self._buf).mean()
            return float(ul.to_eng_units(self.board_num, self.ai_range, int(counts)))
        except ULError as e:
            print("UL error:", e.errorcode, e.message)
            return None

    def getTimeData(self) -> float:
        return (datetime.now() - datetime(1904, 1, 1)).total_seconds()

    def recordData(self, epoch1904: float, volts: float) -> None:
        self.data.append((epoch1904, volts))

    # File I/O. Writes data to file, saves file to computer
    def writeData(self, data: list[tuple[float, float]]) -> None:
```

```
        if not data or not self.filename:
            return

        try:
            # Get directory path and ensure it exists
            dir_path = os.path.dirname(self.filename)
            if dir_path:  # Only try to create if there is a directory component
                os.makedirs(dir_path, exist_ok=True)

            # Write the data file
            with open(self.filename, "w", encoding="utf-8") as f:
                for epoch, v in data:
                    # Format: "[time since Jan 1st 1904] TAB [Signal up to 4 decimal places]"
                    f.write(f"{epoch:.4f}\t{v:.4f}\n")

            print(f"Successfully saved {len(data)} rows to {self.filename}")

        except PermissionError as e:
            print(f"Error: Permission denied when writing to {self.filename}: {str(e)}")
        except OSError as e:
            print(f"Error writing to {self.filename}: {str(e)}")
        except Exception as e:
            print(f"Unexpected error saving data: {str(e)}")
```

Display.py

- ```
  import time
  import queue
  import tkinter as tk
  from tkinter import ttk, messagebox, filedialog
  import os
  ```
  ```

```
from datetime import datetime, timedelta
import math

import matplotlib

matplotlib.use("TkAgg")
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import matplotlib.pyplot as plt

from DataAcquisition import DataAcquisition
from Valves import Valves
from Settings import settings


class Display:
    """Real-time GUI with variable valve-swap schedules, X/Y autoscaling, and auto-run."""

    OPEN_CLR = "#90EE90"
    CLOSED_CLR = "#D3D3D3"

    #
    # ------------------------------------------------------------------------
    #  Construction
    #
    # ------------------------------------------------------------------------
    def __init__(self, root: tk.Tk):
        self.root = root
        self.root.title("Real-Time Signal Monitor")

        # Create notebook for tabs
        self.notebook = ttk.Notebook(self.root)
```

```
        self.notebook.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

        # Create control tab
        self.control_tab = ttk.Frame(self.notebook)
        self.notebook.add(self.control_tab, text="Control")

        # Create config tab
        self.config_tab = ttk.Frame(self.notebook)
        self.notebook.add(self.config_tab, text="Configuration")

        self.daq = DataAcquisition()
        self.dataQueue = queue.Queue()
        self.daq.attach_queue(self.dataQueue)

        self.blockMS = max(
            1,
            round(1000 * self.daq.blockSize / self.daq.samplingFrequency),
        )

        # Run-state
        self.recording = False
        self.maxDuration = settings.effective_run_duration
        self.currentValve = "A"
        self.swap_job_ids = []  # List to store all swap job IDs

        # Auto-run state
        self.auto_run_job = None
        self.next_run_time = None

        # Initialize valves after settings
        self.valves = Valves()
```

```
        if not hasattr(self.daq, '_hardware_available') or not self.daq._hardware_available:
            messagebox.showwarning("Hardware Not Found",
                                   f"Analog input board {settings.ai_board_number} not found. Running in simulation mode.")

        if not hasattr(self.valves, '_hardware_available') or not self.valves._hardware_available:
            messagebox.showwarning("Hardware Not Found",
                                   f"Digital I/O board {settings.dio_board_number} not found. Valve controls will be simulated.")

        # Build GUI
        self._build_widgets()

        self.jobId = self.root.after(self.blockMS, self.updateLoop)
        self.root.protocol("WM_DELETE_WINDOW", self.closeWindow)

        # Start auto-run scheduler if enabled
        if settings.auto_run:
            self._start_auto_run_scheduler()

    #
    # GUI layout
    #
    def _build_widgets(self):
        # Build configuration tab first
        self._build_config_tab()

        # Build control tab
        self._build_control_tab()
```

```
580
        def _build_config_tab(self):
            """Build the configuration tab"""
            config_frm = ttk.LabelFrame(self.config_tab, text="Board Configuration", padding=10)
            config_frm.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)
585
            # AI Board
            ai_frm = ttk.Frame(config_frm)
            ai_frm.pack(fill=tk.X, pady=5)
            ttk.Label(ai_frm, text="AI Board Number:").pack(side=tk.LEFT, padx=(0, 10))
590         self.ai_board_var = tk.IntVar(value=settings.ai_board_number)
            ai_board_spin = ttk.Spinbox(ai_frm, from_=0, to=15, width=5,
                                        textvariable=self.ai_board_var)
            ai_board_spin.pack(side=tk.LEFT)

595         # DIO Board
            dio_frm = ttk.Frame(config_frm)
            dio_frm.pack(fill=tk.X, pady=5)
            ttk.Label(dio_frm, text="DIO Board Number:").pack(side=tk.LEFT, padx=(0, 10))
            self.dio_board_var = tk.IntVar(value=settings.dio_board_number)
600         dio_board_spin = ttk.Spinbox(dio_frm, from_=0, to=15, width=5,
                                         textvariable=self.dio_board_var)
            dio_board_spin.pack(side=tk.LEFT)

            # AI Channel
605         chan_frm = ttk.Frame(config_frm)
            chan_frm.pack(fill=tk.X, pady=5)
            ttk.Label(chan_frm, text="AI Channel:").pack(side=tk.LEFT, padx=(0, 10))
            self.ai_channel_var = tk.IntVar(value=settings.ai_channel)
            ai_channel_spin = ttk.Spinbox(chan_frm, from_=0, to=15, width=5,
610                                       textvariable=self.ai_channel_var)
```

```
        ai_channel_spin.pack(side=tk.LEFT)

        # Apply Button
        btn_frm = ttk.Frame(config_frm)
        btn_frm.pack(fill=tk.X, pady=(20, 5))
        apply_btn = ttk.Button(btn_frm, text="Apply Configuration",
                               command=self._apply_config)
        apply_btn.pack(pady=10)

        # Status message
        self.config_status = tk.StringVar(value="")
        ttk.Label(config_frm, textvariable=self.config_status, foreground="blue").pack()

    def _build_control_tab(self):
        """Build the main control tab"""
        # Top bar
        top = ttk.Frame(self.control_tab, padding=(10, 5))
        top.pack(fill=tk.X)

        ttk.Label(top, text="Operator Initials:").grid(row=0, column=0, sticky="w")
        self.initialsVar = tk.StringVar()
        ttk.Entry(top, width=5, textvariable=self.initialsVar).grid(
            row=0,
            column=1,
            padx=(2, 15),
        )

        self.startBtn = ttk.Button(top, text="Start", command=self.startRecording)
        self.startBtn.grid(row=0, column=2, padx=(10, 2))

        self.stopBtn = ttk.Button(top, text="Stop", command=self.stopRecording, state="disabled")
```

```
        self.stopBtn.grid(row=0, column=3)

        ttk.Label(top, text="File:").grid(row=0, column=4, padx=(10, 2))
        self.filenameVar = tk.StringVar(value="")
        self.filenameLabel = ttk.Label(top, textvariable=self.filenameVar, foreground="blue")
        self.filenameLabel.grid(row=0, column=5, sticky="w")

        ttk.Separator(self.control_tab, orient="horizontal").pack(fill=tk.X, pady=4)

        # Main frame
        main = ttk.Frame(self.control_tab)
        main.pack(fill=tk.X, padx=10)

        info = ttk.Frame(main)
        info.pack(side=tk.LEFT, expand=True)

        # New: Save Directory selection
        dir_frm = ttk.Frame(info)
        dir_frm.grid(row=0, column=0, columnspan=6, sticky="we", pady=(0, 5))

        ttk.Label(dir_frm, text="Save Directory:").pack(side=tk.LEFT, padx=(0, 5))
        self.save_dir_var = tk.StringVar(value=settings.save_directory)
        dir_entry = ttk.Entry(dir_frm, textvariable=self.save_dir_var, width=40)
        dir_entry.pack(side=tk.LEFT, fill=tk.X, expand=True)
        ttk.Button(
            dir_frm,
            text="Browse",
            width=8,
            command=self._select_directory
        ).pack(side=tk.LEFT, padx=(5, 0))
```

```
        # New: File path display
        self.file_path_var = tk.StringVar(value="")
        ttk.Label(
            info,
            textvariable=self.file_path_var,
            foreground="green",
            wraplength=500
        ).grid(row=1, column=0, columnspan=6, sticky="w", pady=(0, 5))

        # Live readouts
        ttk.Label(info, text="Time Elapsed (s):").grid(row=2, column=0, sticky="w")
        self.timeVar = tk.StringVar(value="0.0000")
        ttk.Label(info, textvariable=self.timeVar).grid(row=2, column=1, padx=(4, 20))

        ttk.Label(info, text="Current Signal (V):").grid(row=2, column=2, sticky="w")
        self.signalVar = tk.StringVar(value="0.0000")
        ttk.Label(info, textvariable=self.signalVar).grid(row=2, column=3, padx=(4, 20))

        ttk.Label(info, text="Time Remaining (s):").grid(row=2, column=4, sticky="w")
        self.remainingVar = tk.StringVar(value="0.0000")
        ttk.Label(info, textvariable=self.remainingVar).grid(row=2, column=5)

        # Run-duration row
        dur = ttk.Frame(info)
        dur.grid(row=3, column=0, columnspan=6, sticky="w", pady=(6, 0))

        ttk.Label(dur, text="Run Duration (s):").pack(side=tk.LEFT, padx=(0, 2))
        self.durationVar = tk.StringVar(value=str(settings.run_duration))
        self.durationEntry = ttk.Entry(dur, width=7, textvariable=self.durationVar)
        self.durationEntry.pack(side=tk.LEFT)
        self.durationEntry.bind("<FocusOut>", self._update_duration)
```

```
        self.durationEntry.bind("<Return>", self._update_duration)

        ttk.Label(dur, text="Presets:").pack(side=tk.LEFT, padx=(10, 2))
        preset_btns = [
            ("2 min", 120),
            ("5 min", 300),
            ("10 min", 600)
        ]
        for text, sec in preset_btns:
            ttk.Button(
                dur,
                text=text,
                width=7,
                command=lambda s=sec: self._set_duration(s)
            ).pack(side=tk.LEFT, padx=(0, 2))

        # Valve schedule controls
        valve_schedule_frm = ttk.LabelFrame(info, text="Valve Schedule")
        valve_schedule_frm.grid(row=4, column=0, columnspan=6, sticky="we", pady=(10, 5), padx=5)

        # Initial valve
        initial_frm = ttk.Frame(valve_schedule_frm)
        initial_frm.pack(fill=tk.X, padx=5, pady=(5, 0))
        ttk.Label(initial_frm, text="Initial Valve:").pack(side=tk.LEFT, padx=(0, 5))
        self.initialValveVar = tk.StringVar(value="A")
        ttk.Radiobutton(initial_frm, text="A", variable=self.initialValveVar,
    value="A").pack(side=tk.LEFT)
        ttk.Radiobutton(initial_frm, text="B", variable=self.initialValveVar,
    value="B").pack(side=tk.LEFT)

        # Valve swap table
```

```
        swap_frm = ttk.Frame(valve_schedule_frm)
        swap_frm.pack(fill=tk.X, padx=5, pady=5)

        # Table header
        header = ttk.Frame(swap_frm)
        header.pack(fill=tk.X, pady=(0, 5))
        ttk.Label(header, text="Swap #", width=8).pack(side=tk.LEFT)
        ttk.Label(header, text="Time (s)", width=8).pack(side=tk.LEFT, padx=5)
        ttk.Label(header, text="Valve", width=8).pack(side=tk.LEFT, padx=5)
        ttk.Label(header, text="Action", width=8).pack(side=tk.LEFT)

        # Container for swap rows
        self.swap_rows_frame = ttk.Frame(swap_frm)
        self.swap_rows_frame.pack(fill=tk.X)

        # Add/remove controls
        ctrl_frm = ttk.Frame(valve_schedule_frm)
        ctrl_frm.pack(fill=tk.X, padx=5, pady=(0, 5))
        ttk.Button(ctrl_frm, text="+ Add Swap", command=self._add_swap_row).pack(side=tk.LEFT)
        ttk.Button(ctrl_frm, text="- Remove Last", command=self._remove_last_swap).pack(side=tk.LEFT,
    padx=5)

        # Populate with existing schedule
        self.swap_vars = []
        for time, valve in settings.valve_schedule:
            self._add_swap_row(time, valve)

        # Auto-run row
        auto_run_f = ttk.Frame(info)
        auto_run_f.grid(row=5, column=0, columnspan=6, sticky="w", pady=(10, 0))

```

```
        self.autoRunVar = tk.BooleanVar(value=settings.auto_run)
        ttk.Checkbutton(
            auto_run_f,
            text="Auto-run every",
            variable=self.autoRunVar,
            command=self._toggle_auto_run
        ).pack(side=tk.LEFT)

        # Interval entry
        self.autoIntVar = tk.StringVar(value=str(settings.auto_run_interval))
        self.autoIntEntry = ttk.Entry(auto_run_f, width=5, textvariable=self.autoIntVar)
        self.autoIntEntry.pack(side=tk.LEFT)
        self.autoIntEntry.bind("<FocusOut>", self._update_auto_interval)
        self.autoIntEntry.bind("<Return>", self._update_auto_interval)

        ttk.Label(auto_run_f, text="seconds").pack(side=tk.LEFT, padx=(0, 2))

        # Next run display
        self.nextRunVar = tk.StringVar(value="Next run: --:--:--")
        ttk.Label(auto_run_f, textvariable=self.nextRunVar).pack(side=tk.LEFT, padx=(10, 0))

        # Y-axis controls
        yctrl = ttk.Frame(info)
        yctrl.grid(row=6, column=0, columnspan=6, sticky="w", pady=(6, 0))

        self.autoscaleVar = tk.BooleanVar(value=True)
        ttk.Checkbutton(
            yctrl,
            text="Autoscale Y",
            variable=self.autoscaleVar,
            command=self._toggleAutoscale,
```

```
        ).pack(side=tk.LEFT)

        ttk.Label(yctrl, text="Y min:").pack(side=tk.LEFT, padx=(10, 2))
        self.yMinVar = tk.StringVar(value="0")
        self.yMinEntry = ttk.Entry(yctrl, width=7, textvariable=self.yMinVar, state="disabled")
        self.yMinEntry.pack(side=tk.LEFT)

        ttk.Label(yctrl, text="Y max:").pack(side=tk.LEFT, padx=(6, 2))
        self.yMaxVar = tk.StringVar(value="1")
        self.yMaxEntry = ttk.Entry(yctrl, width=7, textvariable=self.yMaxVar, state="disabled")
        self.yMaxEntry.pack(side=tk.LEFT)

        # Manual valve buttons
        valve_f = ttk.Frame(main, padding=(20, 0))
        valve_f.pack(side=tk.RIGHT, anchor="ne")

        self.buttonA = tk.Button(valve_f, text="Open A", width=10, bg=self.CLOSED_CLR,
    command=self.toggleValveA)
        self.buttonA.pack(pady=(0, 5))

        self.buttonB = tk.Button(valve_f, text="Open B", width=10, bg=self.CLOSED_CLR,
    command=self.toggleValveB)
        self.buttonB.pack()

        # Matplotlib figure
        self.fig, self.ax = plt.subplots(figsize=(6, 4))
        self.ax.set_xlabel("Time (s)")
        self.ax.set_ylabel("Signal (V)")

        self.line, = self.ax.plot([], [], lw=1.3)
```

```
        FigureCanvasTkAgg(self.fig, master=self.control_tab).get_tk_widget().pack(fill=tk.BOTH,
    expand=True)

        self.xData = []
        self.yData = []

    #
    # Configuration methods
    #

    def _apply_config(self):
        """Apply new configuration settings"""
        try:
            if self.recording:
                messagebox.showerror("Error", "Cannot change configuration while recording")
                return

            # Validate inputs
            ai_board = int(self.ai_board_var.get())
            dio_board = int(self.dio_board_var.get())
            ai_channel = int(self.ai_channel_var.get())

            if not (0 <= ai_board <= 15):
                raise ValueError("AI board number must be 0-15")
            if not (0 <= dio_board <= 15):
                raise ValueError("DIO board number must be 0-15")
            if not (0 <= ai_channel <= 15):
                raise ValueError("AI channel must be 0-15")

            # Update settings
```

```
            settings.ai_board_number = ai_board
            settings.dio_board_number = dio_board
            settings.ai_channel = ai_channel

            # Reinitialize hardware
            self.valves = Valves()
            self.daq = DataAcquisition()
            self.daq.attach_queue(self.dataQueue)

            self.config_status.set("Configuration updated successfully")
        except ValueError as e:
            messagebox.showerror("Error", f"Invalid configuration: {str(e)}")

    #
    ------------------------------------------------------------------------------
    #  Valve schedule management
    #
    ------------------------------------------------------------------------------

    def _add_swap_row(self, time_val: float = 0.0, valve_val: str = "B"):
        """Add a new row to the valve schedule table"""
        row = ttk.Frame(self.swap_rows_frame)
        row.pack(fill=tk.X, pady=2)

        # Swap number
        swap_num = len(self.swap_vars) + 1
        ttk.Label(row, text=f"#{swap_num}", width=8).pack(side=tk.LEFT)

        # Time entry
        time_var = tk.StringVar(value=str(time_val))
        time_ent = ttk.Entry(row, width=8, textvariable=time_var)
        time_ent.pack(side=tk.LEFT, padx=5)
```

```
890
            # Valve selection
            valve_var = tk.StringVar(value=valve_val)
            valve_cmb = ttk.Combobox(row, width=8, textvariable=valve_var, state="readonly")
            valve_cmb['values'] = ("A", "B")
895         valve_cmb.pack(side=tk.LEFT, padx=5)

            # Remove button
            remove_btn = ttk.Button(row, text="Remove", width=8,
                                    command=lambda r=row: self._remove_swap_row(r))
900         remove_btn.pack(side=tk.LEFT)

            # Store variables
            self.swap_vars.append((time_var, valve_var, row))

905     def _remove_swap_row(self, row):
            """Remove a specific row from the valve schedule"""
            # Find and remove the row from our list
            for i, (time_var, valve_var, row_widget) in enumerate(self.swap_vars):
                if row_widget == row:
910                 self.swap_vars.pop(i)
                    row.destroy()
                    break

            # Renumber remaining swaps
915         for i, (_, _, row_widget) in enumerate(self.swap_vars):
                swap_num_label = row_widget.winfo_children()[0]
                swap_num_label.config(text=f"#{i + 1}")

        def _remove_last_swap(self):
920         """Remove the last swap from the schedule"""
```

```
        if self.swap_vars:
            _, _, row = self.swap_vars.pop()
            row.destroy()

    def _get_valve_schedule(self) -> list[tuple[float, str]]:
        """Get the current valve schedule from the UI"""
        schedule = []
        for time_var, valve_var, _ in self.swap_vars:
            try:
                time_val = float(time_var.get())
                valve_val = valve_var.get()
                if time_val >= 0 and valve_val in ("A", "B"):
                    schedule.append((time_val, valve_val))
                else:
                    messagebox.showerror("Invalid Input",
                                         f"Invalid valve schedule: time={time_val}, valve={valve_val}")
            except ValueError:
                messagebox.showerror("Invalid Input", "Time must be a number")
        return sorted(schedule, key=lambda x: x[0])

    #
    # Directory selection
    #
    def _select_directory(self):
        """Open directory dialog and update save path"""
        dir_path = filedialog.askdirectory(
            initialdir=self.save_dir_var.get(),
            title="Select Save Directory"
        )
```

```
        if dir_path:
            self.save_dir_var.set(dir_path)
            settings.save_directory = dir_path

    #
    # Duration and interval synchronization
    #
    def _set_duration(self, seconds: float):
        """Set duration without affecting auto-run interval"""
        settings.run_duration = seconds
        self.durationVar.set(str(seconds))

    def _update_duration(self, event=None):
        """Update from GUI entry"""
        try:
            seconds = float(self.durationVar.get())
            settings.run_duration = seconds
        except ValueError:
            pass

    def _update_auto_interval(self, event=None):
        """Update auto-run interval from GUI"""
        try:
            interval = int(self.autoIntVar.get())
            settings.auto_run_interval = interval
        except ValueError:
            pass

    #
```

```
    # Auto-run methods
985 #
    # ------------------------------------------------------------------------

    def _toggle_auto_run(self):
        settings.auto_run = self.autoRunVar.get()
        if settings.auto_run:
990         # Update settings from GUI
            try:
                settings.auto_run_interval = int(self.autoIntVar.get())
            except ValueError:
                pass  # Keep previous value
995         self._start_auto_run_scheduler()
        elif self.auto_run_job:
            self.root.after_cancel(self.auto_run_job)
            self.auto_run_job = None
            self.next_run_time = None
1000        self.nextRunVar.set("Next run: --:--:--")

    def _calculate_next_run(self):
        """Calculate next run time at exact second interval."""
        now = datetime.now()
1005    interval_seconds = settings.auto_run_interval

        # Calculate next time that is multiple of interval seconds
        current_seconds = now.hour * 3600 + now.minute * 60 + now.second
        remainder = current_seconds % interval_seconds
1010
        if remainder == 0:
            # Already at interval, run immediately
            next_seconds = current_seconds
```

```
            else:
                next_seconds = current_seconds + interval_seconds - remainder

            # Convert seconds back to time
            next_hour = next_seconds // 3600
            next_minute = (next_seconds % 3600) // 60
            next_second = next_seconds % 60

            # Create next run time
            next_run = now.replace(hour=next_hour, minute=next_minute, second=next_second,
        microsecond=0)

            # Handle day rollover
            if next_hour >= 24:
                next_run = next_run + timedelta(days=1)

            return next_run

        def _start_auto_run_scheduler(self):
            if not settings.auto_run:
                return

            # Calculate next run time
            self.next_run_time = self._calculate_next_run()
            self.nextRunVar.set(f"Next run: {self.next_run_time.strftime('%H:%M:%S')}")

            # Calculate delay in milliseconds
            now = datetime.now()
            delay_ms = int((self.next_run_time - now).total_seconds() * 1000)

            # Schedule next run
```

```
        if self.auto_run_job:
            self.root.after_cancel(self.auto_run_job)
        self.auto_run_job = self.root.after(delay_ms, self._execute_auto_run)

    def _execute_auto_run(self):
        if not settings.auto_run:
            return

        # Start recording
        self.startRecording()

        # Schedule next run after current run completes
        self.auto_run_job = self.root.after(
            int(settings.auto_run_interval * 1000),  # Use full interval
            self._start_auto_run_scheduler
        )

    #
    ------------------------------------------------------------------------------

    #  Valve helpers
    #
    ------------------------------------------------------------------------------

    def _setValveState(self, valve: str):
        if valve == "A":
            self.valves.set_valve_position_a()
            self.buttonA.config(bg=self.OPEN_CLR)
            self.buttonB.config(bg=self.CLOSED_CLR)
        else:
            self.valves.set_valve_position_b()
            self.buttonB.config(bg=self.OPEN_CLR)
            self.buttonA.config(bg=self.CLOSED_CLR)
```

```
        self.currentValve = valve

    #
1080 ─────────────────────────────────────────────────────────────────────────
    #  Autoscale toggle
    #
─────────────────────────────────────────────────────────────────────────────

    def _toggleAutoscale(self):
1085    state = "disabled" if self.autoscaleVar.get() else "normal"
        self.yMinEntry.config(state=state)
        self.yMaxEntry.config(state=state)

    #
1090 ─────────────────────────────────────────────────────────────────────────
    #  Start / Stop
    #
─────────────────────────────────────────────────────────────────────────────

    def startRecording(self):
1095    if self.recording:
            return

        # Set operator initials from GUI
        settings.operator_initials = self.initialsVar.get().strip() or "NULL"
1100
        # Generate filename for this run
        initials = settings.operator_initials.upper()
        stamp = datetime.now().strftime("%y%m%d_%H%M%S")
        self.current_filename = f"{initials}_{stamp}"
1105
        # Generate directory path
```

```
            base_dir = self.save_dir_var.get()
            yyyy_mm = datetime.now().strftime("%Y-%m")
            full_dir = os.path.join(base_dir, yyyy_mm)

            # Create directory if needed
            try:
                os.makedirs(full_dir, exist_ok=True)
            except Exception as e:
                messagebox.showerror(
                    "Directory Error",
                    f"Could not create directory {full_dir}: {str(e)}"
                )
                return

            # Set filename with full path
            full_path = os.path.join(full_dir, self.current_filename + ".txt")
            self.daq.set_filename(full_path)
            self.file_path_var.set(f"File will be saved to:\n{full_path}")

            # Show filename in UI
            self.filenameVar.set(self.current_filename)

            # Use effective duration (with 5s buffer)
            self.maxDuration = settings.effective_run_duration

            # Get valve schedule from UI
            settings.valve_schedule = self._get_valve_schedule()

            # Set initial valve
            self.currentValve = self.initialValveVar.get()
            self._setValveState(self.currentValve)
```

```
            # Clear any existing swap jobs
            for job_id in self.swap_job_ids:
                self.root.after_cancel(job_id)
            self.swap_job_ids = []

            # Schedule all valve swaps
            for swap_time, valve_target in settings.valve_schedule:
                if swap_time > 0 and swap_time < self.maxDuration:
                    job_id = self.root.after(
                        int(swap_time * 1000),
                        lambda v=valve_target: self._setValveState(v)
                    )
                    self.swap_job_ids.append(job_id)

            self.xData.clear()
            self.yData.clear()
            self.line.set_data([], [])

            self.startTime = time.perf_counter()
            self.recording = True
            self.daq.start()

            self.startBtn.config(state="disabled")
            self.stopBtn.config(state="normal")

        def stopRecording(self):
            if not self.recording:
                return

            self.recording = False
```

```
        self.daq.stop()

        # Cancel all swap jobs
        for job_id in self.swap_job_ids:
            self.root.after_cancel(job_id)
        self.swap_job_ids = []

        self.startBtn.config(state="normal")
        self.stopBtn.config(state="disabled")

        # Clear filename after short delay to show it was saved
        self.root.after(2000, lambda: self.filenameVar.set(""))

    #
    # ---------------------------------------------------------------------
    #  Main update loop
    #
    # ---------------------------------------------------------------------

    def updateLoop(self):
        while not self.dataQueue.empty():
            t_rel, v = self.dataQueue.get_nowait()
            self.xData.append(t_rel)
            self.yData.append(v)

        if self.xData:
            elapsed = self.xData[-1]
            remaining = max(0.0, self.maxDuration - elapsed)

            self.timeVar.set(f"{elapsed:.4f}")
            self.remainingVar.set(f"{remaining:.4f}")
            self.signalVar.set(f"{self.yData[-1]:.4f}")
```

```
            self.line.set_data(self.xData, self.yData)

            xmin = self.xData[0]
            xmax = self.xData[-1]
            pad_x = max(1e-6, (xmax - xmin) * 0.02)
            self.ax.set_xlim(xmin - pad_x, xmax + pad_x)

            if self.autoscaleVar.get():
                ymin = min(self.yData)
                ymax = max(self.yData)
                pad_y = max(1e-6, (ymax - ymin) * 0.05)
                self.ax.set_ylim(ymin - pad_y, ymax + pad_y)
            else:
                lims = self._manualYLimits()
                if lims:
                    self.ax.set_ylim(lims)

            self.fig.canvas.draw_idle()

            if self.recording and elapsed >= self.maxDuration:
                self.stopRecording()

        self.jobId = self.root.after(self.blockMS, self.updateLoop)

    #
    # ------------------------------------------------------------------
    #  Utility helpers
    # ------------------------------------------------------------------
    def _manualYLimits(self):
```

```
            try:
                ymin = float(self.yMinVar.get())
                ymax = float(self.yMaxVar.get())
                if ymin >= ymax:
                    raise ValueError
                return ymin, ymax
            except ValueError:
                return None

    def _safe_float(self, tk_var: tk.StringVar, default: float) -> float:
        try:
            return float(tk_var.get())
        except ValueError:
            return default

    # Manual override buttons
    def toggleValveA(self):
        self._setValveState("A")

    def toggleValveB(self):
        self._setValveState("B")

    # Clean shutdown
    def closeWindow(self):
        if self.recording:
            self.stopRecording()

        if self.jobId:
            self.root.after_cancel(self.jobId)

        for job_id in self.swap_job_ids:
```

```
                if job_id:
                    self.root.after_cancel(job_id)

            if self.auto_run_job:
                self.root.after_cancel(self.auto_run_job)

            self.root.destroy()


        # Stand-alone entry point
        def main():
            root = tk.Tk()
            Display(root)
            root.mainloop()


        if __name__ == "__main__":
            main()
```

Settings.py

```
from __future__ import annotations
from dataclasses import dataclass, field
from typing import Dict, Any, List, Tuple
import os

@dataclass
class Settings:
    # Board configuration
    ai_board_number: int =0 # Analog input board number
    dio_board_number: int =0 # Digital I/O board number (for valves)
```

```
    ai_channel: int =0 # Analog input channel

    # Acquisition parameters
    sampling_frequency: int = 10_000  # Hz, e.g. 10_000 for 10 kHz
    block_size: int = 1_000  # samples grabbed per driver call
    run_duration: float = 595.0  # seconds, total length of a run (595 for 10 min interval)

    # Misc / operator info
    operator_initials: str = "NULL"  # appears in data-file names
    save_directory: str = field(default=os.getcwd())

    # Auto-run parameters
    auto_run: bool = False  # Enable auto-run feature
    auto_run_interval: int = 600  # Seconds between runs (default 10 minutes)

    # Valve scheduling - now a list of (time, valve) pairs
    valve_schedule: List[Tuple[float, str]] = field(default_factory=lambda: [
        (15.0, "B")  # Default: swap to B at 15s
    ])

    # Properties for synchronization
    @property
    def effective_run_duration(self) -> float:
        """Run duration minus 5s buffer"""
        return max(0, self.run_duration - 5.0)

    # Helpers
    def validate(self) -> None:
        """Raise ValueError if any field is outside a sane range."""
        if not (0 <= self.ai_board_number <= 15):
            raise ValueError("AI board number must be between 0 and 15 (inclusive)")
```

```
        if not (0 <= self.dio_board_number <= 15):
            raise ValueError("DIO board number must be between 0 and 15 (inclusive)")
        if not (0 <= self.ai_channel <= 15):
            raise ValueError("AI channel must be between 0 and 15 (inclusive)")
        if self.sampling_frequency <= 0:
            raise ValueError("sampling_frequency must be positive")
        if self.block_size <= 0:
            raise ValueError("block_size must be positive")
        if self.run_duration <= 0:
            raise ValueError("run_duration must be positive")
        for time, valve in self.valve_schedule:
            if time < 0:
                raise ValueError(f"valve time cannot be negative")
            if valve not in ("A", "B"):
                raise ValueError(f"valve must be 'A' or 'B'")
        if self.auto_run_interval <= 0:
            raise ValueError("auto_run_interval must be positive")

    # Easy-to-read dump (handy for logging)
    def as_dict(self) -> Dict[str, Any]:
        """Return a plain dict of the current settings."""
        return {
            "ai_board_number": self.ai_board_number,
            "dio_board_number": self.dio_board_number,
            "ai_channel": self.ai_channel,
            "sampling_frequency": self.sampling_frequency,
            "block_size": self.block_size,
            "run_duration": self.run_duration,
            "operator_initials": self.operator_initials,
            "auto_run": self.auto_run,
            "auto_run_interval": self.auto_run_interval,
```

```
            "valve_schedule": list(self.valve_schedule),
        }

    def __str__(self) -> str:
        items = [f"{k}: {v}" for k, v in self.as_dict().items()]
        return "\n".join(items)


# Global singleton – import once, everywhere
settings = Settings()

# Validate immediately so typos are caught on launch
settings.validate()
```

USB1408FS.py

```
from Display import main as run_gui

if __name__ == "__main__":
    # Launch the Tkinter interface
    run_gui()
```

Valves.py

```
from mcculw import ul
from mcculw.enums import DigitalPortType
from time import sleep

from Settings import settings

class Valves:
```

```
    def __init__(self):
        self.board_num = settings.dio_board_number
        self._hardware_available = False

        try:
            # Try to access the board to verify it exists
            ul.get_board_name(self.board_num)
            ul.d_config_port(self.board_num, DigitalPortType.FIRSTPORTA, 1)
            self._hardware_available = True
        except Exception as e:
            print(f"Warning: Digital I/O board {self.board_num} not found. Valve controls will be simulated.")
            self._hardware_available = False

    # Function to switch to Position A (open Valve A, close Valve B)
    def set_valve_position_a(self) -> None:
        if not self._hardware_available:
            print("Simulating valve A open")
            return

        try:
            ul.d_bit_out(self.board_num, DigitalPortType.FIRSTPORTA, 0, 0)  # Valve A ON
            ul.d_bit_out(self.board_num, DigitalPortType.FIRSTPORTA, 1, 1)  # Valve B OFF
        except Exception as e:
            print(f"Error setting valve A: {str(e)}")

    # Function to switch to Position B (open Valve B, close Valve A)
    def set_valve_position_b(self) -> None:
        if not self._hardware_available:
            print("Simulating valve B open")
            return
```

```
        try:
            ul.d_bit_out(self.board_num, DigitalPortType.FIRSTPORTA, 0, 1)  # Valve A OFF
            ul.d_bit_out(self.board_num, DigitalPortType.FIRSTPORTA, 1, 0)  # Valve B ON
        except Exception as e:
            print(f"Error setting valve B: {str(e)}")


def testValves():
    testValve = Valves()
    testValve.set_valve_position_a()
    sleep(3)
    testValve.set_valve_position_b()

# Enable this to test valves
# testValves()
```

Measurement Computing's USB-1608FS

DataAcquisition.py

```
# DataAcquisition.py
import numpy as np
from mcculw import ul
from mcculw.enums import ULRange
from mcculw.ul import ULError
import ctypes as ct
import threading, queue, time
from datetime import datetime
import os

from Settings import settings
```

```
class DataAcquisition:
    blockSize = 100
    samplingFrequency = 10_000  # Hz

    def __init__(self):
        # Board-specific settings (updated to use settings)
        self.board_num = settings.ai_board_number
        self.channel = settings.ai_channel
        self.ai_range = ULRange.BIP10VOLTS
        self._hardware_available = False

        try:
            # Try to access the board to verify it exists
            ul.get_board_name(self.board_num)
            self._hardware_available = True
        except ULError:
            print(f"Warning: Analog input board {self.board_num} not found. Running in simulation mode.")
            self._hardware_available = False

        # Pre-allocate one-block buffer
        self._buf = (ct.c_uint16 * self.blockSize)()

        # Run bookkeeping
        self.data: list[tuple[float, float]] = []

        # initial file name
        self.filename = None

        # Threading helpers
```

```
        self._queue: queue.Queue | None = None
        self._thread: threading.Thread | None = None
        self._running = threading.Event()

    def set_filename(self, filename):
        self.filename = filename

    # Public control surface
    def attach_queue(self, q: queue.Queue) -> None:
        """GUI supplies a queue to receive (t_rel, volts)."""
        self._queue = q

    def start(self) -> None:
        if self._thread and self._thread.is_alive():
            return
        self._running.set()
        self._thread = threading.Thread(target=self._worker, daemon=True)
        self._thread.start()

    def stop(self, join_timeout: float = 1.0) -> None:
        self._running.clear()
        if self._thread:
            self._thread.join(timeout=join_timeout)
            self._thread = None
        self.writeData(self.data)  # auto-save
        self.data = []  # clear for next run

    # Background worker — runs in its own thread
    def _worker(self) -> None:
        t0 = time.perf_counter()
        while self._running.is_set():
```

```
            volts = self.getSignalData()
            if volts is None:
                continue  # skip bad scan, keep running

            t_rel = time.perf_counter() - t0
            epoch1904 = self.getTimeData()
            self.recordData(epoch1904, volts)

            if self._queue:
                try:
                    self._queue.put_nowait((t_rel, volts))
                except queue.Full:  # drop oldest if GUI lags
                    _ = self._queue.get_nowait()
                    self._queue.put_nowait((t_rel, volts))
            # a_in_scan blocks ≈ blockSize/samplingFrequency
            # so no extra sleep is needed

        # Low-level helpers
        def getSignalData(self) -> float | None:
            if not self._hardware_available:
                # Simulate a sine wave when hardware isn't available
                return 2.5 + 2.5 * np.sin(time.perf_counter() * 2 * np.pi * 0.1)

            try:
                ul.a_in_scan(self.board_num,
                             self.channel, self.channel,
                             self.blockSize, self.samplingFrequency,
                             self.ai_range, self._buf, 0)

                counts = np.ctypeslib.as_array(self._buf).mean()
                return float(ul.to_eng_units(self.board_num, self.ai_range, int(counts)))
```

```
            except ULError as e:
                print("UL error:", e.errorcode, e.message)
                return None

    def getTimeData(self) -> float:
        return (datetime.now() - datetime(1904, 1, 1)).total_seconds()

    def recordData(self, epoch1904: float, volts: float) -> None:
        self.data.append((epoch1904, volts))

    # File I/O
    def writeData(self, data: list[tuple[float, float]]) -> None:
        if not data or not self.filename:
            return

        try:
            # Get directory path and ensure it exists
            dir_path = os.path.dirname(self.filename)
            if dir_path:  # Only try to create if there is a directory component
                os.makedirs(dir_path, exist_ok=True)

            # Write the data file
            with open(self.filename, "w", encoding="utf-8") as f:
                for epoch, v in data:
                    # Format: "[time since Jan 1st 1904] TAB [Signal up to 4 decimal places]"
                    f.write(f"{epoch:.4f}\t{v:.4f}\n")

            print(f"Successfully saved {len(data)} rows to {self.filename}")

        except PermissionError as e:
            print(f"Error: Permission denied when writing to {self.filename}: {str(e)}")
```

```
        except OSError as e:
            print(f"Error writing to {self.filename}: {str(e)}")
        except Exception as e:
            print(f"Unexpected error saving data: {str(e)}")
```

Display.py

- ```
  import time
  import queue
  import tkinter as tk
  from tkinter import ttk, messagebox, filedialog
  import os
  from datetime import datetime, timedelta
  import math

  import matplotlib

  matplotlib.use("TkAgg")
  from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
  import matplotlib.pyplot as plt

  from DataAcquisition import DataAcquisition
  from Valves import Valves
  from Settings import settings


  class Display:
      """Real-time GUI with variable valve-swap schedules, X/Y autoscaling, and auto-run."""

      OPEN_CLR = "#90EE90"
      CLOSED_CLR = "#D3D3D3"
  ```
  ```

```
    #
    # ------------------------------------------------------------------
    #  Construction
    # ------------------------------------------------------------------
    def __init__(self, root: tk.Tk):
        self.root = root
        self.root.title("Real-Time Signal Monitor")

        # Create notebook for tabs
        self.notebook = ttk.Notebook(self.root)
        self.notebook.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

        # Create control tab
        self.control_tab = ttk.Frame(self.notebook)
        self.notebook.add(self.control_tab, text="Control")

        # Create config tab
        self.config_tab = ttk.Frame(self.notebook)
        self.notebook.add(self.config_tab, text="Configuration")

        self.daq = DataAcquisition()
        self.dataQueue = queue.Queue()
        self.daq.attach_queue(self.dataQueue)

        self.blockMS = max(
            1,
            round(1000 * self.daq.blockSize / self.daq.samplingFrequency),
        )
```

```
        # Run-state
        self.recording = False
        self.maxDuration = settings.effective_run_duration
        self.currentValve = "A"
        self.swap_job_ids = []  # List to store all swap job IDs

        # Auto-run state
        self.auto_run_job = None
        self.next_run_time = None

        # Initialize valves after settings
        self.valves = Valves()
        if not hasattr(self.daq, '_hardware_available') or not self.daq._hardware_available:
            messagebox.showwarning("Hardware Not Found",
                                   f"Analog input board {settings.ai_board_number} not found. Running in simulation mode.")

        if not hasattr(self.valves, '_hardware_available') or not self.valves._hardware_available:
            messagebox.showwarning("Hardware Not Found",
                                   f"Digital I/O board {settings.dio_board_number} not found. Valve controls will be simulated.")

        # Build GUI
        self._build_widgets()

        self.jobId = self.root.after(self.blockMS, self.updateLoop)
        self.root.protocol("WM_DELETE_WINDOW", self.closeWindow)

        # Start auto-run scheduler if enabled
        if settings.auto_run:
            self._start_auto_run_scheduler()
```

```
    #
    # ------------------------------------------------------------------------
    #  GUI layout
    #
    # ------------------------------------------------------------------------
    def _build_widgets(self):
        # Build configuration tab first
        self._build_config_tab()

        # Build control tab
        self._build_control_tab()

    def _build_config_tab(self):
        """Build the configuration tab"""
        config_frm = ttk.LabelFrame(self.config_tab, text="Board Configuration", padding=10)
        config_frm.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

        # AI Board
        ai_frm = ttk.Frame(config_frm)
        ai_frm.pack(fill=tk.X, pady=5)
        ttk.Label(ai_frm, text="AI Board Number:").pack(side=tk.LEFT, padx=(0, 10))
        self.ai_board_var = tk.IntVar(value=settings.ai_board_number)
        ai_board_spin = ttk.Spinbox(ai_frm, from_=0, to=15, width=5,
                                    textvariable=self.ai_board_var)
        ai_board_spin.pack(side=tk.LEFT)

        # DIO Board
        dio_frm = ttk.Frame(config_frm)
        dio_frm.pack(fill=tk.X, pady=5)
        ttk.Label(dio_frm, text="DIO Board Number:").pack(side=tk.LEFT, padx=(0, 10))
```

```
1690            self.dio_board_var = tk.IntVar(value=settings.dio_board_number)
                dio_board_spin = ttk.Spinbox(dio_frm, from_=0, to=15, width=5,
                                             textvariable=self.dio_board_var)
                dio_board_spin.pack(side=tk.LEFT)

1695            # AI Channel
                chan_frm = ttk.Frame(config_frm)
                chan_frm.pack(fill=tk.X, pady=5)
                ttk.Label(chan_frm, text="AI Channel:").pack(side=tk.LEFT, padx=(0, 10))
                self.ai_channel_var = tk.IntVar(value=settings.ai_channel)
1700            ai_channel_spin = ttk.Spinbox(chan_frm, from_=0, to=15, width=5,
                                              textvariable=self.ai_channel_var)
                ai_channel_spin.pack(side=tk.LEFT)

                # Apply Button
1705            btn_frm = ttk.Frame(config_frm)
                btn_frm.pack(fill=tk.X, pady=(20, 5))
                apply_btn = ttk.Button(btn_frm, text="Apply Configuration",
                                       command=self._apply_config)
                apply_btn.pack(pady=10)
1710
                # Status message
                self.config_status = tk.StringVar(value="")
                ttk.Label(config_frm, textvariable=self.config_status, foreground="blue").pack()

1715        def _build_control_tab(self):
                """Build the main control tab"""
                # Top bar
                top = ttk.Frame(self.control_tab, padding=(10, 5))
                top.pack(fill=tk.X)
1720
```

```
        ttk.Label(top, text="Operator Initials:").grid(row=0, column=0, sticky="w")
        self.initialsVar = tk.StringVar()
        ttk.Entry(top, width=5, textvariable=self.initialsVar).grid(
            row=0,
            column=1,
            padx=(2, 15),
        )

        self.startBtn = ttk.Button(top, text="Start", command=self.startRecording)
        self.startBtn.grid(row=0, column=2, padx=(10, 2))

        self.stopBtn = ttk.Button(top, text="Stop", command=self.stopRecording, state="disabled")
        self.stopBtn.grid(row=0, column=3)

        ttk.Label(top, text="File:").grid(row=0, column=4, padx=(10, 2))
        self.filenameVar = tk.StringVar(value="")
        self.filenameLabel = ttk.Label(top, textvariable=self.filenameVar, foreground="blue")
        self.filenameLabel.grid(row=0, column=5, sticky="w")

        ttk.Separator(self.control_tab, orient="horizontal").pack(fill=tk.X, pady=4)

        # Main frame
        main = ttk.Frame(self.control_tab)
        main.pack(fill=tk.X, padx=10)

        info = ttk.Frame(main)
        info.pack(side=tk.LEFT, expand=True)

        # New: Save Directory selection
        dir_frm = ttk.Frame(info)
        dir_frm.grid(row=0, column=0, columnspan=6, sticky="we", pady=(0, 5))
```

```
        ttk.Label(dir_frm, text="Save Directory:").pack(side=tk.LEFT, padx=(0, 5))
        self.save_dir_var = tk.StringVar(value=settings.save_directory)
        dir_entry = ttk.Entry(dir_frm, textvariable=self.save_dir_var, width=40)
        dir_entry.pack(side=tk.LEFT, fill=tk.X, expand=True)
        ttk.Button(
            dir_frm,
            text="Browse",
            width=8,
            command=self._select_directory
        ).pack(side=tk.LEFT, padx=(5, 0))

        # New: File path display
        self.file_path_var = tk.StringVar(value="")
        ttk.Label(
            info,
            textvariable=self.file_path_var,
            foreground="green",
            wraplength=500
        ).grid(row=1, column=0, columnspan=6, sticky="w", pady=(0, 5))

        # Live readouts
        ttk.Label(info, text="Time Elapsed (s):").grid(row=2, column=0, sticky="w")
        self.timeVar = tk.StringVar(value="0.0000")
        ttk.Label(info, textvariable=self.timeVar).grid(row=2, column=1, padx=(4, 20))

        ttk.Label(info, text="Current Signal (V):").grid(row=2, column=2, sticky="w")
        self.signalVar = tk.StringVar(value="0.0000")
        ttk.Label(info, textvariable=self.signalVar).grid(row=2, column=3, padx=(4, 20))

        ttk.Label(info, text="Time Remaining (s):").grid(row=2, column=4, sticky="w")
```

```
        self.remainingVar = tk.StringVar(value="0.0000")
        ttk.Label(info, textvariable=self.remainingVar).grid(row=2, column=5)

        # Run-duration row
        dur = ttk.Frame(info)
        dur.grid(row=3, column=0, columnspan=6, sticky="w", pady=(6, 0))

        ttk.Label(dur, text="Run Duration (s):").pack(side=tk.LEFT, padx=(0, 2))
        self.durationVar = tk.StringVar(value=str(settings.run_duration))
        self.durationEntry = ttk.Entry(dur, width=7, textvariable=self.durationVar)
        self.durationEntry.pack(side=tk.LEFT)
        self.durationEntry.bind("<FocusOut>", self._update_duration)
        self.durationEntry.bind("<Return>", self._update_duration)

        ttk.Label(dur, text="Presets:").pack(side=tk.LEFT, padx=(10, 2))
        preset_btns = [
            ("2 min", 120),
            ("5 min", 300),
            ("10 min", 600)
        ]
        for text, sec in preset_btns:
            ttk.Button(
                dur,
                text=text,
                width=7,
                command=lambda s=sec: self._set_duration(s)
            ).pack(side=tk.LEFT, padx=(0, 2))

        # Valve schedule controls
        valve_schedule_frm = ttk.LabelFrame(info, text="Valve Schedule")
        valve_schedule_frm.grid(row=4, column=0, columnspan=6, sticky="we", pady=(10, 5), padx=5)
```

```
        # Initial valve
        initial_frm = ttk.Frame(valve_schedule_frm)
        initial_frm.pack(fill=tk.X, padx=5, pady=(5, 0))
        ttk.Label(initial_frm, text="Initial Valve:").pack(side=tk.LEFT, padx=(0, 5))
        self.initialValveVar = tk.StringVar(value="A")
        ttk.Radiobutton(initial_frm, text="A", variable=self.initialValveVar,
value="A").pack(side=tk.LEFT)
        ttk.Radiobutton(initial_frm, text="B", variable=self.initialValveVar,
value="B").pack(side=tk.LEFT)

        # Valve swap table
        swap_frm = ttk.Frame(valve_schedule_frm)
        swap_frm.pack(fill=tk.X, padx=5, pady=5)

        # Table header
        header = ttk.Frame(swap_frm)
        header.pack(fill=tk.X, pady=(0, 5))
        ttk.Label(header, text="Swap #", width=8).pack(side=tk.LEFT)
        ttk.Label(header, text="Time (s)", width=8).pack(side=tk.LEFT, padx=5)
        ttk.Label(header, text="Valve", width=8).pack(side=tk.LEFT, padx=5)
        ttk.Label(header, text="Action", width=8).pack(side=tk.LEFT)

        # Container for swap rows
        self.swap_rows_frame = ttk.Frame(swap_frm)
        self.swap_rows_frame.pack(fill=tk.X)

        # Add/remove controls
        ctrl_frm = ttk.Frame(valve_schedule_frm)
        ctrl_frm.pack(fill=tk.X, padx=5, pady=(0, 5))
        ttk.Button(ctrl_frm, text="+ Add Swap", command=self._add_swap_row).pack(side=tk.LEFT)
```

```
        ttk.Button(ctrl_frm, text="- Remove Last", command=self._remove_last_swap).pack(side=tk.LEFT, padx=5)

        # Populate with existing schedule
        self.swap_vars = []
        for time, valve in settings.valve_schedule:
            self._add_swap_row(time, valve)

        # Auto-run row
        auto_run_f = ttk.Frame(info)
        auto_run_f.grid(row=5, column=0, columnspan=6, sticky="w", pady=(10, 0))

        self.autoRunVar = tk.BooleanVar(value=settings.auto_run)
        ttk.Checkbutton(
            auto_run_f,
            text="Auto-run every",
            variable=self.autoRunVar,
            command=self._toggle_auto_run
        ).pack(side=tk.LEFT)

        # Interval entry
        self.autoIntVar = tk.StringVar(value=str(settings.auto_run_interval))
        self.autoIntEntry = ttk.Entry(auto_run_f, width=5, textvariable=self.autoIntVar)
        self.autoIntEntry.pack(side=tk.LEFT)
        self.autoIntEntry.bind("<FocusOut>", self._update_auto_interval)
        self.autoIntEntry.bind("<Return>", self._update_auto_interval)

        ttk.Label(auto_run_f, text="seconds").pack(side=tk.LEFT, padx=(0, 2))

        # Next run display
        self.nextRunVar = tk.StringVar(value="Next run: --:--:--")
```

```
        ttk.Label(auto_run_f, textvariable=self.nextRunVar).pack(side=tk.LEFT, padx=(10, 0))

        # Y-axis controls
        yctrl = ttk.Frame(info)
        yctrl.grid(row=6, column=0, columnspan=6, sticky="w", pady=(6, 0))

        self.autoscaleVar = tk.BooleanVar(value=True)
        ttk.Checkbutton(
            yctrl,
            text="Autoscale Y",
            variable=self.autoscaleVar,
            command=self._toggleAutoscale,
        ).pack(side=tk.LEFT)

        ttk.Label(yctrl, text="Y min:").pack(side=tk.LEFT, padx=(10, 2))
        self.yMinVar = tk.StringVar(value="0")
        self.yMinEntry = ttk.Entry(yctrl, width=7, textvariable=self.yMinVar, state="disabled")
        self.yMinEntry.pack(side=tk.LEFT)

        ttk.Label(yctrl, text="Y max:").pack(side=tk.LEFT, padx=(6, 2))
        self.yMaxVar = tk.StringVar(value="1")
        self.yMaxEntry = ttk.Entry(yctrl, width=7, textvariable=self.yMaxVar, state="disabled")
        self.yMaxEntry.pack(side=tk.LEFT)

        # Manual valve buttons
        valve_f = ttk.Frame(main, padding=(20, 0))
        valve_f.pack(side=tk.RIGHT, anchor="ne")

        self.buttonA = tk.Button(valve_f, text="Open A", width=10, bg=self.CLOSED_CLR,
    command=self.toggleValveA)
        self.buttonA.pack(pady=(0, 5))
```

```
        self.buttonB = tk.Button(valve_f, text="Open B", width=10, bg=self.CLOSED_CLR,
    command=self.toggleValveB)
        self.buttonB.pack()

        # Matplotlib figure
        self.fig, self.ax = plt.subplots(figsize=(6, 4))
        self.ax.set_xlabel("Time (s)")
        self.ax.set_ylabel("Signal (V)")

        self.line, = self.ax.plot([], [], lw=1.3)

        FigureCanvasTkAgg(self.fig, master=self.control_tab).get_tk_widget().pack(fill=tk.BOTH,
    expand=True)

        self.xData = []
        self.yData = []

    #
    # Configuration methods
    #
    def _apply_config(self):
        """Apply new configuration settings"""
        try:
            if self.recording:
                messagebox.showerror("Error", "Cannot change configuration while recording")
                return

            # Validate inputs
```

```
            ai_board = int(self.ai_board_var.get())
            dio_board = int(self.dio_board_var.get())
            ai_channel = int(self.ai_channel_var.get())

            if not (0 <= ai_board <= 15):
                raise ValueError("AI board number must be 0-15")
            if not (0 <= dio_board <= 15):
                raise ValueError("DIO board number must be 0-15")
            if not (0 <= ai_channel <= 15):
                raise ValueError("AI channel must be 0-15")

            # Update settings
            settings.ai_board_number = ai_board
            settings.dio_board_number = dio_board
            settings.ai_channel = ai_channel

            # Reinitialize hardware
            self.valves = Valves()
            self.daq = DataAcquisition()
            self.daq.attach_queue(self.dataQueue)

            self.config_status.set("Configuration updated successfully")
        except ValueError as e:
            messagebox.showerror("Error", f"Invalid configuration: {str(e)}")

    #
    # Valve schedule management
    #

    def _add_swap_row(self, time_val: float = 0.0, valve_val: str = "B"):
```

```
        """Add a new row to the valve schedule table"""
        row = ttk.Frame(self.swap_rows_frame)
        row.pack(fill=tk.X, pady=2)

        # Swap number
        swap_num = len(self.swap_vars) + 1
        ttk.Label(row, text=f"#{swap_num}", width=8).pack(side=tk.LEFT)

        # Time entry
        time_var = tk.StringVar(value=str(time_val))
        time_ent = ttk.Entry(row, width=8, textvariable=time_var)
        time_ent.pack(side=tk.LEFT, padx=5)

        # Valve selection
        valve_var = tk.StringVar(value=valve_val)
        valve_cmb = ttk.Combobox(row, width=8, textvariable=valve_var, state="readonly")
        valve_cmb['values'] = ("A", "B")
        valve_cmb.pack(side=tk.LEFT, padx=5)

        # Remove button
        remove_btn = ttk.Button(row, text="Remove", width=8,
                                command=lambda r=row: self._remove_swap_row(r))
        remove_btn.pack(side=tk.LEFT)

        # Store variables
        self.swap_vars.append((time_var, valve_var, row))

    def _remove_swap_row(self, row):
        """Remove a specific row from the valve schedule"""
        # Find and remove the row from our list
        for i, (time_var, valve_var, row_widget) in enumerate(self.swap_vars):
```

```
            if row_widget == row:
                self.swap_vars.pop(i)
                row.destroy()
                break

        # Renumber remaining swaps
        for i, (_, _, row_widget) in enumerate(self.swap_vars):
            swap_num_label = row_widget.winfo_children()[0]
            swap_num_label.config(text=f"#{i + 1}")

    def _remove_last_swap(self):
        """Remove the last swap from the schedule"""
        if self.swap_vars:
            _, _, row = self.swap_vars.pop()
            row.destroy()

    def _get_valve_schedule(self) -> list[tuple[float, str]]:
        """Get the current valve schedule from the UI"""
        schedule = []
        for time_var, valve_var, _ in self.swap_vars:
            try:
                time_val = float(time_var.get())
                valve_val = valve_var.get()
                if time_val >= 0 and valve_val in ("A", "B"):
                    schedule.append((time_val, valve_val))
                else:
                    messagebox.showerror("Invalid Input",
                                         f"Invalid valve schedule: time={time_val}, valve={valve_val}")
            except ValueError:
                messagebox.showerror("Invalid Input", "Time must be a number")
        return sorted(schedule, key=lambda x: x[0])
```

```
        #
# ------------------------------------------------------------------------
        # Directory selection
        #
# ------------------------------------------------------------------------
    def _select_directory(self):
        """Open directory dialog and update save path"""
        dir_path = filedialog.askdirectory(
            initialdir=self.save_dir_var.get(),
            title="Select Save Directory"
        )
        if dir_path:
            self.save_dir_var.set(dir_path)
            settings.save_directory = dir_path

        #
# ------------------------------------------------------------------------
        # Duration and interval synchronization
        #
# ------------------------------------------------------------------------
    def _set_duration(self, seconds: float):
        """Set duration without affecting auto-run interval"""
        settings.run_duration = seconds
        self.durationVar.set(str(seconds))

    def _update_duration(self, event=None):
        """Update from GUI entry"""
        try:
            seconds = float(self.durationVar.get())
            settings.run_duration = seconds
```

```
            except ValueError:
                pass

    def _update_auto_interval(self, event=None):
        """Update auto-run interval from GUI"""
        try:
            interval = int(self.autoIntVar.get())
            settings.auto_run_interval = interval
        except ValueError:
            pass

    #
    # Auto-run methods
    #
    def _toggle_auto_run(self):
        settings.auto_run = self.autoRunVar.get()
        if settings.auto_run:
            # Update settings from GUI
            try:
                settings.auto_run_interval = int(self.autoIntVar.get())
            except ValueError:
                pass  # Keep previous value
            self._start_auto_run_scheduler()
        elif self.auto_run_job:
            self.root.after_cancel(self.auto_run_job)
            self.auto_run_job = None
            self.next_run_time = None
            self.nextRunVar.set("Next run: --:--:--")
```

```
    def _calculate_next_run(self):
        """Calculate next run time at exact second interval."""
        now = datetime.now()
        interval_seconds = settings.auto_run_interval

        # Calculate next time that is multiple of interval seconds
        current_seconds = now.hour * 3600 + now.minute * 60 + now.second
        remainder = current_seconds % interval_seconds

        if remainder == 0:
            # Already at interval, run immediately
            next_seconds = current_seconds
        else:
            next_seconds = current_seconds + interval_seconds - remainder

        # Convert seconds back to time
        next_hour = next_seconds // 3600
        next_minute = (next_seconds % 3600) // 60
        next_second = next_seconds % 60

        # Create next run time
        next_run = now.replace(hour=next_hour, minute=next_minute, second=next_second,
    microsecond=0)

        # Handle day rollover
        if next_hour >= 24:
            next_run = next_run + timedelta(days=1)

        return next_run

    def _start_auto_run_scheduler(self):
```

```
        if not settings.auto_run:
2125            return

        # Calculate next run time
        self.next_run_time = self._calculate_next_run()
        self.nextRunVar.set(f"Next run: {self.next_run_time.strftime('%H:%M:%S')}")
2130
        # Calculate delay in milliseconds
        now = datetime.now()
        delay_ms = int((self.next_run_time - now).total_seconds() * 1000)

2135    # Schedule next run
        if self.auto_run_job:
            self.root.after_cancel(self.auto_run_job)
        self.auto_run_job = self.root.after(delay_ms, self._execute_auto_run)

2140    def _execute_auto_run(self):
        if not settings.auto_run:
            return

        # Start recording
2145    self.startRecording()

        # Schedule next run after current run completes
        self.auto_run_job = self.root.after(
            int(settings.auto_run_interval * 1000),  # Use full interval
2150        self._start_auto_run_scheduler
        )

    #
```

```
        # Valve helpers
        #
        # ------------------------------------------------------------------
        def _setValveState(self, valve: str):
            if valve == "A":
                self.valves.set_valve_position_a()
                self.buttonA.config(bg=self.OPEN_CLR)
                self.buttonB.config(bg=self.CLOSED_CLR)
            else:
                self.valves.set_valve_position_b()
                self.buttonB.config(bg=self.OPEN_CLR)
                self.buttonA.config(bg=self.CLOSED_CLR)

            self.currentValve = valve

        #
        # ------------------------------------------------------------------
        # Autoscale toggle
        #
        # ------------------------------------------------------------------
        def _toggleAutoscale(self):
            state = "disabled" if self.autoscaleVar.get() else "normal"
            self.yMinEntry.config(state=state)
            self.yMaxEntry.config(state=state)

        #
        # ------------------------------------------------------------------
        # Start / Stop
        #
        # ------------------------------------------------------------------
        def startRecording(self):
```

```
        if self.recording:
            return

        # Set operator initials from GUI
        settings.operator_initials = self.initialsVar.get().strip() or "NULL"

        # Generate filename for this run
        initials = settings.operator_initials.upper()
        stamp = datetime.now().strftime("%y%m%d_%H%M%S")
        self.current_filename = f"{initials}_{stamp}"

        # Generate directory path
        base_dir = self.save_dir_var.get()
        yyyy_mm = datetime.now().strftime("%Y-%m")
        full_dir = os.path.join(base_dir, yyyy_mm)

        # Create directory if needed
        try:
            os.makedirs(full_dir, exist_ok=True)
        except Exception as e:
            messagebox.showerror(
                "Directory Error",
                f"Could not create directory {full_dir}: {str(e)}"
            )
            return

        # Set filename with full path
        full_path = os.path.join(full_dir, self.current_filename + ".txt")
        self.daq.set_filename(full_path)
        self.file_path_var.set(f"File will be saved to:\n{full_path}")
```

```
            # Show filename in UI
            self.filenameVar.set(self.current_filename)

            # Use effective duration (with 5s buffer)
            self.maxDuration = settings.effective_run_duration

            # Get valve schedule from UI
            settings.valve_schedule = self._get_valve_schedule()

            # Set initial valve
            self.currentValve = self.initialValveVar.get()
            self._setValveState(self.currentValve)

            # Clear any existing swap jobs
            for job_id in self.swap_job_ids:
                self.root.after_cancel(job_id)
            self.swap_job_ids = []

            # Schedule all valve swaps
            for swap_time, valve_target in settings.valve_schedule:
                if swap_time > 0 and swap_time < self.maxDuration:
                    job_id = self.root.after(
                        int(swap_time * 1000),
                        lambda v=valve_target: self._setValveState(v)
                    )
                    self.swap_job_ids.append(job_id)

            self.xData.clear()
            self.yData.clear()
            self.line.set_data([], [])
```

```
        self.startTime = time.perf_counter()
        self.recording = True
        self.daq.start()

        self.startBtn.config(state="disabled")
        self.stopBtn.config(state="normal")

    def stopRecording(self):
        if not self.recording:
            return

        self.recording = False
        self.daq.stop()

        # Cancel all swap jobs
        for job_id in self.swap_job_ids:
            self.root.after_cancel(job_id)
        self.swap_job_ids = []

        self.startBtn.config(state="normal")
        self.stopBtn.config(state="disabled")

        # Clear filename after short delay to show it was saved
        self.root.after(2000, lambda: self.filenameVar.set(""))

    #
    # ---------------------------------------------------------------------------
    #  Main update loop
    #
    # ---------------------------------------------------------------------------

    def updateLoop(self):
```

```
        while not self.dataQueue.empty():
2280        t_rel, v = self.dataQueue.get_nowait()
            self.xData.append(t_rel)
            self.yData.append(v)

        if self.xData:
2285        elapsed = self.xData[-1]
            remaining = max(0.0, self.maxDuration - elapsed)

            self.timeVar.set(f"{elapsed:.4f}")
            self.remainingVar.set(f"{remaining:.4f}")
2290        self.signalVar.set(f"{self.yData[-1]:.4f}")

            self.line.set_data(self.xData, self.yData)

            xmin = self.xData[0]
2295        xmax = self.xData[-1]
            pad_x = max(1e-6, (xmax - xmin) * 0.02)
            self.ax.set_xlim(xmin - pad_x, xmax + pad_x)

            if self.autoscaleVar.get():
2300            ymin = min(self.yData)
                ymax = max(self.yData)
                pad_y = max(1e-6, (ymax - ymin) * 0.05)
                self.ax.set_ylim(ymin - pad_y, ymax + pad_y)
            else:
2305            lims = self._manualYLimits()
                if lims:
                    self.ax.set_ylim(lims)

            self.fig.canvas.draw_idle()
```

```
            if self.recording and elapsed >= self.maxDuration:
                self.stopRecording()

        self.jobId = self.root.after(self.blockMS, self.updateLoop)

    #
    # ------------------------------------------------------------------
    #  Utility helpers
    #
    # ------------------------------------------------------------------
    def _manualYLimits(self):
        try:
            ymin = float(self.yMinVar.get())
            ymax = float(self.yMaxVar.get())
            if ymin >= ymax:
                raise ValueError
            return ymin, ymax
        except ValueError:
            return None

    def _safe_float(self, tk_var: tk.StringVar, default: float) -> float:
        try:
            return float(tk_var.get())
        except ValueError:
            return default

    # Manual override buttons
    def toggleValveA(self):
        self._setValveState("A")
```

```
            def toggleValveB(self):
                self._setValveState("B")

            # Clean shutdown
2345        def closeWindow(self):
                if self.recording:
                    self.stopRecording()

                if self.jobId:
2350                self.root.after_cancel(self.jobId)

                for job_id in self.swap_job_ids:
                    if job_id:
                        self.root.after_cancel(job_id)
2355
                if self.auto_run_job:
                    self.root.after_cancel(self.auto_run_job)

                self.root.destroy()
2360

        # Stand-alone entry point
        def main():
            root = tk.Tk()
2365        Display(root)
            root.mainloop()


        if __name__ == "__main__":
2370        main()
```

Settings.py

```
from __future__ import annotations
from dataclasses import dataclass, field
from typing import Dict, Any, List, Tuple
import os

@dataclass
class Settings:
    # Board configuration
    ai_board_number: int =0 # Analog input board number
    dio_board_number: int =0 # Digital I/O board number (for valves)
    ai_channel: int =0 # Analog input channel

    # Acquisition parameters
    sampling_frequency: int = 10_000  # Hz, e.g. 10_000 for 10 kHz
    block_size: int = 1_000  # samples grabbed per driver call
    run_duration: float = 600.0  # seconds, total length of a run (595 for 10 min interval)

    # Misc / operator info
    operator_initials: str = "NULL"  # appears in data-file names
    save_directory: str = field(default=os.getcwd())

    # Auto-run parameters
    auto_run: bool = False  # Enable auto-run feature
    auto_run_interval: int = 600  # Seconds between runs (default 10 minutes)

    # Valve scheduling - now a list of (time, valve) pairs
    valve_schedule: List[Tuple[float, str]] = field(default_factory=lambda: [
        (15.0, "B")  # Default: swap to B at 15s
    ])
```

```
    # Properties for synchronization
    @property
    def effective_run_duration(self) -> float:
        """Run duration minus 5s buffer"""
        return max(0, self.run_duration - 5.0)

    # Helpers
    def validate(self) -> None:
        """Raise ValueError if any field is outside a sane range."""
        if not (0 <= self.ai_board_number <= 15):
            raise ValueError("AI board number must be between 0 and 15 (inclusive)")
        if not (0 <= self.dio_board_number <= 15):
            raise ValueError("DIO board number must be between 0 and 15 (inclusive)")
        if not (0 <= self.ai_channel <= 15):
            raise ValueError("AI channel must be between 0 and 15 (inclusive)")
        if self.sampling_frequency <= 0:
            raise ValueError("sampling_frequency must be positive")
        if self.block_size <= 0:
            raise ValueError("block_size must be positive")
        if self.run_duration <= 0:
            raise ValueError("run_duration must be positive")
        for time, valve in self.valve_schedule:
            if time < 0:
                raise ValueError(f"valve time cannot be negative")
            if valve not in ("A", "B"):
                raise ValueError(f"valve must be 'A' or 'B'")
        if self.auto_run_interval <= 0:
            raise ValueError("auto_run_interval must be positive")

    # Easy-to-read dump (handy for logging)
```

```
    def as_dict(self) -> Dict[str, Any]:
        """Return a plain dict of the current settings."""
        return {
            "ai_board_number": self.ai_board_number,
            "dio_board_number": self.dio_board_number,
            "ai_channel": self.ai_channel,
            "sampling_frequency": self.sampling_frequency,
            "block_size": self.block_size,
            "run_duration": self.run_duration,
            "operator_initials": self.operator_initials,
            "auto_run": self.auto_run,
            "auto_run_interval": self.auto_run_interval,
            "valve_schedule": list(self.valve_schedule),
        }

    def __str__(self) -> str:
        items = [f"{k}: {v}" for k, v in self.as_dict().items()]
        return "\n".join(items)


# Global singleton – import once, everywhere
settings = Settings()

# Validate immediately so typos are caught on launch
settings.validate()
```

USB1608FS.py

```
from Display import main as run_gui

if __name__ == "__main__":
```

```
    # Launch the Tkinter interface
    run_gui()
```

Valves.py

```
from mcculw import ul
from mcculw.enums import DigitalPortType
from time import sleep

from Settings import settings



class Valves:
    def __init__(self):
        self.board_num = settings.dio_board_number
        self._hardware_available = False

        try:
            # Try to access the board to verify it exists
            ul.get_board_name(self.board_num)
            # Configure all digital I/O bits as outputs
            ul.d_config_port(self.board_num, DigitalPortType.AUXPORT, 1)
            self._hardware_available = True
        except Exception as e:
            print(f"Warning: Digital I/O board {self.board_num} not found. Valve controls will be simulated.")
            self._hardware_available = False

    def _set_valve(self, a_state: int, b_state: int):
        """Set both valve control lines to specified states"""
        if not self._hardware_available:
            return
```

```
            try:
2495            # Set Position B control (DIO1)
                ul.d_bit_out(self.board_num, DigitalPortType.AUXPORT, 1, b_state)
                # Set Position A control (DIO3)
                ul.d_bit_out(self.board_num, DigitalPortType.AUXPORT, 3, a_state)

2500        except Exception as e:
                print(f"Error setting valve states: {str(e)}")

        def set_valve_position_b(self) -> None:
            if not self._hardware_available:
2505            print("Simulating valve B open")
                return

            try:
                # Set A high, B low
2510            self._set_valve(a_state=1, b_state=0)
                print("Valve set to Position B")
            except Exception as e:
                print(f"Error setting valve B: {str(e)}")

2515    def set_valve_position_a(self) -> None:
            if not self._hardware_available:
                print("Simulating valve A open")
                return

2520        try:
                # Set B high, A low
                self._set_valve(a_state=0, b_state=1)
                print("Valve set to Position A")
```

```
            except Exception as e:
                print(f"Error setting valve A: {str(e)}")

        def testValves():
            testValve = Valves()
            testValve.board_num = 0
            ul.d_bit_out(0, DigitalPortType.AUXPORT, 1, 1)
            sleep(5)
            ul.d_bit_out(0, DigitalPortType.AUXPORT, 1, 0)

        # Uncomment the line below to run the test
        # testValves()
```

National Instruments' USB-6008 and USB-6009

ConfigWindow.py

```
        import tkinter as tk
        from tkinter import ttk


        class ConfigWindow(tk.Toplevel):
            def __init__(self, parent, daq, channel_config, valve_config):
                super().__init__(parent)
                self.title("DAQ Configuration")
                self.daq = daq
                self.channel_config = channel_config
                self.valve_config = valve_config
                self.parent = parent

                # Create notebook for tabs
```

```
        self.notebook = ttk.Notebook(self)
        self.notebook.pack(fill='both', expand=True, padx=10, pady=10)

        # Analog Channels Tab
        self.analog_tab = ttk.Frame(self.notebook)
        self.notebook.add(self.analog_tab, text="Analog Channels")

        # Device configuration
        ttk.Label(self.analog_tab, text="Device Name:").grid(row=0, column=0, padx=5, pady=5)
        self.device_var = tk.StringVar(value=self.daq.device_name)
        ttk.Entry(self.analog_tab, textvariable=self.device_var).grid(row=0, column=1, padx=5, pady=5)

        # Channel mapping
        ttk.Label(self.analog_tab, text="Controller").grid(row=1, column=0)
        ttk.Label(self.analog_tab, text="Output Channel").grid(row=1, column=1)
        ttk.Label(self.analog_tab, text="Input Channel").grid(row=1, column=2)

        self.ao_vars = []
        self.ai_vars = []

        for i, config in enumerate(self.channel_config):
            ttk.Label(self.analog_tab, text=f"Controller {i + 1}").grid(row=i + 2, column=0, padx=5, pady=5)

            ao_var = tk.StringVar(value=config['ao'])
            ai_var = tk.StringVar(value=config['ai'])

            ttk.Entry(self.analog_tab, textvariable=ao_var, width=8).grid(row=i + 2, column=1, padx=5, pady=5)
            ttk.Entry(self.analog_tab, textvariable=ai_var, width=8).grid(row=i + 2, column=2, padx=5, pady=5)

            self.ao_vars.append(ao_var)
            self.ai_vars.append(ai_var)
```

```
        # Digital Channels Tab
        self.digital_tab = ttk.Frame(self.notebook)
        self.notebook.add(self.digital_tab, text="Digital Channels")

        # Valve configuration table
        ttk.Label(self.digital_tab, text="Valve").grid(row=0, column=0, padx=5, pady=5)
        ttk.Label(self.digital_tab, text="Port/Line").grid(row=0, column=1, padx=5, pady=5)

        self.valve_vars = []
        for i, valve in enumerate(self.valve_config):
            ttk.Label(self.digital_tab, text=valve['name']).grid(row=i + 1, column=0, padx=5, pady=5)
            port_var = tk.StringVar(value=valve['port_line'])
            ttk.Entry(self.digital_tab, textvariable=port_var).grid(row=i + 1, column=1, padx=5, pady=5)
            self.valve_vars.append(port_var)

        # Save button
        ttk.Button(self, text="Save Configuration", command=self.save_config).pack(pady=10)

    def save_config(self):
        """Save configuration to main application"""
        self.daq.set_device_name(self.device_var.get())

        for i in range(len(self.channel_config)):
            self.channel_config[i]['ao'] = self.ao_vars[i].get()
            self.channel_config[i]['ai'] = self.ai_vars[i].get()

        # Save valve config
        for i in range(len(self.valve_config)):
            self.valve_config[i]['port_line'] = self.valve_vars[i].get()
```

```
            self.parent.update_channel_labels()
            self.parent.update_valve_config()
            self.destroy()
```

DAQController.py

```
import nidaqmx

class DAQController:
    def __init__(self, device_name="Dev2"):
        self.device_name = device_name
        self.digital_states = {}  # Track valve states

    def set_device_name(self, device_name):
        self.device_name = device_name

    def write_voltage(self, channel, voltage):
        """Write voltage to analog output channel (clamped to 0-5V)"""
        clamped_voltage = max(0.0, min(5.0, voltage))

        with nidaqmx.Task() as task:
            task.ao_channels.add_ao_voltage_chan(
                f"{self.device_name}/{channel}",
                min_val=0.0,
                max_val=5.0
            )
            task.write(clamped_voltage)

    def read_voltage(self, channel):
        """Read voltage from analog input channel"""
        with nidaqmx.Task() as task:
```

```
            task.ai_channels.add_ai_voltage_chan(f"{self.device_name}/{channel}")
            return task.read()

    def write_digital(self, port_line, state):
        """Write digital output (on/off) to a specific port/line"""
        with nidaqmx.Task() as task:
            task.do_channels.add_do_chan(f"{self.device_name}/{port_line}")
            task.write(bool(state))
        # Update state tracking
        self.digital_states[port_line] = bool(state)

    def read_digital_state(self, port_line):
        """Read last set digital state"""
        return self.digital_states.get(port_line, False)
```

Display.py

- ```
  import tkinter as tk
  from tkinter import ttk, messagebox, Menu
  import time
  from datetime import datetime, timedelta
  from Flowrate import MFCManager
  from DAQController import DAQController
  from ConfigWindow import ConfigWindow
  from ValveControlFrame import ValveControlFrame
  from ValveScheduler import ValveScheduler


  class MFCControlFrame(tk.LabelFrame):
      def __init__(self, parent, index, daq, mfc_manager, channel_config, *args, **kwargs):
          super().__init__(parent, *args, **kwargs)
  ```
  ```

```
        self.daq = daq
        self.mfc_manager = mfc_manager
        self.index = index
        self.channel_config = channel_config

        # Channel info labels
        self.channel_info = tk.StringVar()
        ttk.Label(self, textvariable=self.channel_info, font=("Arial", 8)).grid(
            row=0, column=0, columnspan=4, sticky="w", padx=5
        )

        # MFC selection
        ttk.Label(self, text="MFC Type:").grid(row=1, column=0, padx=5, pady=5, sticky="e")
        self.mfc_var = tk.StringVar()
        self.mfc_dropdown = ttk.Combobox(self, textvariable=self.mfc_var, state="readonly", width=12)
        self.mfc_dropdown['values'] = self.mfc_manager.get_all_mfc_names()
        self.mfc_dropdown.grid(row=1, column=1, padx=5, pady=5, columnspan=3, sticky="w")

        # Set flow section
        ttk.Label(self, text="Set Flow:").grid(row=2, column=0, padx=5, pady=5, sticky="e")
        self.flow_var = tk.StringVar()
        self.flow_entry = ttk.Entry(self, textvariable=self.flow_var, width=8)
        self.flow_entry.grid(row=2, column=1, padx=5, pady=5, sticky="w")

        # Set flow unit display
        self.set_unit_var = tk.StringVar(value="SCCM")
        ttk.Label(self, textvariable=self.set_unit_var, width=6).grid(row=2, column=2, padx=5, pady=5,
    sticky="w")

        # Set button
        self.set_btn = ttk.Button(self, text="Set", command=self.set_flow, width=6)
```

```
        self.set_btn.grid(row=2, column=3, padx=5, pady=5, sticky="w")

        # Actual flow section
        ttk.Label(self, text="Actual Flow:").grid(row=3, column=0, padx=5, pady=5, sticky="e")
        self.actual_flow_var = tk.StringVar(value="0.0")
        ttk.Label(self, textvariable=self.actual_flow_var, width=8).grid(row=3, column=1, padx=5, pady=5,
sticky="w")

        # Actual flow unit display
        self.actual_unit_var = tk.StringVar(value="SCCM")
        ttk.Label(self, textvariable=self.actual_unit_var).grid(row=3, column=2, padx=5, pady=5,
sticky="w")

        # Scaling information
        self.scaling_info = tk.StringVar()
        ttk.Label(self, textvariable=self.scaling_info, font=("Arial", 8)).grid(
            row=4, column=0, columnspan=4, sticky="w", padx=5
        )

        # Initialize
        if self.mfc_dropdown['values']:
            self.mfc_var.set(self.mfc_dropdown['values'][0])
            self.update_unit_display()
            self.update_scaling_info()

        # Bind MFC selection change
        self.mfc_var.trace_add("write", self.on_mfc_change)

        # Update channel labels
        self.update_channel_info()
```

```
    def on_mfc_change(self, *args):
        self.update_unit_display()
        self.update_scaling_info()

    def update_scaling_info(self):
        """Update scaling information display"""
        mfc = self.get_current_mfc()
        if mfc:
            self.scaling_info.set(f"5V = {mfc.max_flow} {mfc.unit}, 0V = 0 {mfc.unit}")

    def update_unit_display(self):
        """Update unit displays based on MFC selection"""
        mfc = self.get_current_mfc()
        if mfc:
            unit = mfc.unit
            self.set_unit_var.set(unit)
            self.actual_unit_var.set(unit)

    def update_channel_info(self):
        """Update channel information display"""
        ao = self.channel_config[self.index]['ao']
        ai = self.channel_config[self.index]['ai']
        self.channel_info.set(f"Output: {ao} | Input: {ai}")

    def get_current_mfc(self):
        return self.mfc_manager.get_mfc(self.mfc_var.get())

    def set_flow(self):
        try:
            mfc = self.get_current_mfc()
            if not mfc:
```

```
                messagebox.showerror("Selection Error", "No MFC selected")
                return

            flow = float(self.flow_var.get())
            voltage = mfc.flow_to_voltage(flow)
            ao_channel = self.channel_config[self.index]['ao']
            self.daq.write_voltage(ao_channel, voltage)
        except (ValueError, AttributeError) as e:
            messagebox.showerror("Input Error", f"Invalid flow value: {e}")
        except Exception as e:
            messagebox.showerror("Error", f"Failed to set flow: {str(e)}")

    def update_reading(self):
        try:
            mfc = self.get_current_mfc()
            if not mfc:
                return

            ai_channel = self.channel_config[self.index]['ai']
            voltage = self.daq.read_voltage(ai_channel)
            flow = mfc.voltage_to_flow(voltage)
            self.actual_flow_var.set(f"{flow:.2f}")
        except Exception as e:
            # Fail silently for read errors to avoid spamming
            pass


class ScheduledRunTab(ttk.Frame):
    def __init__(self, parent, daq, valve_config, *args, **kwargs):
        super().__init__(parent, *args, **kwargs)
        self.daq = daq
```

```
        self.valve_config = valve_config
        self.swap_job_ids = []
        self.recording = False
        self.auto_run_job = None
        self.next_run_time = None
        self.position_job = None

        # Configure grid for columns
        for i in range(4):
            self.columnconfigure(i, weight=1, uniform="valve_cols")

        # Valve position display (top)
        self.position_frame = ttk.LabelFrame(self, text="Current Valve Positions")
        self.position_frame.grid(row=0, column=0, columnspan=4, sticky="we", padx=5, pady=5)

        self.valve_position_vars = []
        for i in range(4):
            frame = ttk.Frame(self.position_frame)
            frame.pack(side=tk.LEFT, padx=10, pady=5)
            ttk.Label(frame, text=f"Valve {i + 1}:").pack(side=tk.LEFT)
            pos_var = tk.StringVar(value="OFF")
            ttk.Label(frame, textvariable=pos_var, width=5).pack(side=tk.LEFT)
            self.valve_position_vars.append(pos_var)

        # Run controls frame (middle)
        self.control_frame = ttk.LabelFrame(self, text="Run Controls", padding=10)
        self.control_frame.grid(row=1, column=0, columnspan=4, sticky="we", padx=5, pady=10)

        # Run duration
        ttk.Label(self.control_frame, text="Run Duration (s):").grid(row=0, column=0, padx=(0, 5))
        self.duration_var = tk.StringVar(value="60")
```

```
        ttk.Entry(self.control_frame, width=8, textvariable=self.duration_var).grid(row=0, column=1)

        # Start/Stop buttons
        self.start_btn = ttk.Button(self.control_frame, text="Start Run", command=self.start_recording)
        self.start_btn.grid(row=0, column=2, padx=10)

        self.stop_btn = ttk.Button(self.control_frame, text="Stop Run", command=self.stop_recording,
state="disabled")
        self.stop_btn.grid(row=0, column=3)

        # Auto-run controls
        auto_frame = ttk.Frame(self.control_frame)
        auto_frame.grid(row=0, column=4, padx=(20, 0))

        self.auto_run_var = tk.BooleanVar(value=False)
        ttk.Checkbutton(auto_frame, text="Auto-run every", variable=self.auto_run_var,
                        command=self.toggle_auto_run).pack(side=tk.LEFT)

        self.interval_var = tk.StringVar(value="300")
        ttk.Entry(auto_frame, width=5, textvariable=self.interval_var).pack(side=tk.LEFT, padx=5)
        ttk.Label(auto_frame, text="seconds").pack(side=tk.LEFT)

        self.next_run_var = tk.StringVar(value="Next run: --:--:--")
        ttk.Label(auto_frame, textvariable=self.next_run_var).pack(side=tk.LEFT, padx=(10, 0))

        # Valve schedulers (bottom)
        self.valve_schedulers = []
        for i in range(4):
            scheduler = ValveScheduler(
                self, i, daq, valve_config,
                padding=10,
```

```
                relief="groove"
            )
            scheduler.grid(row=2, column=i, padx=5, pady=5, sticky="nsew")
            self.valve_schedulers.append(scheduler)

    def toggle_auto_run(self):
        if self.auto_run_var.get():
            self.start_auto_run_scheduler()
        elif self.auto_run_job:
            self.master.after_cancel(self.auto_run_job)
            self.auto_run_job = None
            self.next_run_time = None
            self.next_run_var.set("Next run: --:--:--")

    def calculate_next_run(self):
        """Calculate next run time at exact second interval."""
        now = datetime.now()
        try:
            interval_seconds = int(self.interval_var.get())
        except ValueError:
            interval_seconds = 300

        current_seconds = now.hour * 3600 + now.minute * 60 + now.second
        remainder = current_seconds % interval_seconds

        if remainder == 0:
            next_seconds = current_seconds
        else:
            next_seconds = current_seconds + interval_seconds - remainder

        next_hour = next_seconds // 3600
```

```
        next_minute = (next_seconds % 3600) // 60
        next_second = next_seconds % 60

        next_run = now.replace(hour=next_hour, minute=next_minute,
                               second=next_second, microsecond=0)

        if next_hour >= 24:
            next_run = next_run + timedelta(days=1)

        return next_run

    def start_auto_run_scheduler(self):
        if not self.auto_run_var.get():
            return

        self.next_run_time = self.calculate_next_run()
        self.next_run_var.set(f"Next run: {self.next_run_time.strftime('%H:%M:%S')}")

        now = datetime.now()
        delay_ms = int((self.next_run_time - now).total_seconds() * 1000)

        if self.auto_run_job:
            self.master.after_cancel(self.auto_run_job)
        self.auto_run_job = self.master.after(delay_ms, self.execute_auto_run)

    def execute_auto_run(self):
        if not self.auto_run_var.get():
            return

        self.start_recording()
        self.auto_run_job = self.master.after(
```

```
                int(self.interval_var.get()) * 1000,
                self.start_auto_run_scheduler
            )

        def start_recording(self):
            if self.recording:
                return

            self.recording = True
            self.start_btn.config(state="disabled")
            self.stop_btn.config(state="normal")

            # Set initial states for all valves
            for scheduler in self.valve_schedulers:
                scheduler.set_initial_state()

            # Schedule valve swaps
            for scheduler in self.valve_schedulers:
                scheduler.schedule_actions(self.master)

            # Schedule run end
            try:
                duration = float(self.duration_var.get())
                job_id = self.master.after(
                    int(duration * 1000),
                    self.stop_recording
                )
                self.swap_job_ids.append(job_id)
            except ValueError:
                messagebox.showerror("Invalid Duration", "Please enter a valid number for run duration")
```

```
        # Start updating valve positions
        self.update_valve_positions()

    def update_valve_positions(self):
        """Update valve position display"""
        for i, scheduler in enumerate(self.valve_schedulers):
            self.valve_position_vars[i].set("ON" if scheduler.current_state else "OFF")

        # Continue updating if recording
        if self.recording:
            self.position_job = self.after(500, self.update_valve_positions)

    def stop_recording(self):
        if not self.recording:
            return

        self.recording = False

        # Cancel all swap jobs
        for job_id in self.swap_job_ids:
            self.master.after_cancel(job_id)
        self.swap_job_ids = []

        # Cancel valve schedules
        for scheduler in self.valve_schedulers:
            scheduler.cancel_schedules()
            scheduler.turn_off()

        # Cancel position updates
        if self.position_job:
            self.after_cancel(self.position_job)
```

```
2985
            self.start_btn.config(state="normal")
            self.stop_btn.config(state="disabled")

            # Final position update
2990        self.update_valve_positions()


    class MainApp(tk.Tk):
        def __init__(self):
            super().__init__()
2995        self.title("MFC & Valve Control System")
            self.geometry("1000x700")

            # Initialize managers
3000        self.mfc_manager = MFCManager()
            self.daq = DAQController()

            # Default channel configuration
            self.channel_config = [
3005            {'ao': 'ao0', 'ai': 'ai0'},
                {'ao': 'ao1', 'ai': 'ai1'},
                {'ao': 'ao2', 'ai': 'ai2'},
                {'ao': 'ao3', 'ai': 'ai3'}
            ]
3010
            # Default valve configuration
            self.valve_config = [
                {"name": "Valve 1", "port_line": "port1/line0"},
                {"name": "Valve 2", "port_line": "port1/line1"},
3015            {"name": "Valve 3", "port_line": "port1/line2"},
```

```
            {"name": "Valve 4", "port_line": "port1/line3"},
        ]

        # Add available MFCs with output range 5.0V
        self.populate_mfcs()

        # Create menu
        self.create_menu()

        # Create notebook (tabs)
        self.notebook = ttk.Notebook(self)
        self.notebook.pack(fill='both', expand=True, padx=10, pady=10)

        # Tab 1: MFC Control
        self.mfc_tab = ttk.Frame(self.notebook)
        self.notebook.add(self.mfc_tab, text="MFC Control")

        # Create MFC control frames
        self.control_frames = []
        for i in range(4):
            frame = MFCControlFrame(
                self.mfc_tab, i, self.daq, self.mfc_manager, self.channel_config,
                text=f"MFC Controller {i + 1}", padx=10, pady=10
            )
            frame.grid(row=i // 2, column=i % 2, padx=10, pady=10, sticky="nsew")
            self.control_frames.append(frame)

        # Configure grid weights for MFC tab
        for i in range(2):
            self.mfc_tab.rowconfigure(i, weight=1)
            self.mfc_tab.columnconfigure(i, weight=1)
```

```
        # Tab 2: Valve Control
        self.valve_tab = ttk.Frame(self.notebook)
3050    self.notebook.add(self.valve_tab, text="Valve Control")

        # Create valve control frame
        self.valve_frame = ValveControlFrame(
            self.valve_tab, self.daq, self.valve_config,
3055        text="Manual Valve Controls", padx=20, pady=20
        )
        self.valve_frame.pack(fill='both', expand=True, padx=20, pady=20)

        # Tab 3: Scheduled Runs
3060    self.schedule_tab = ttk.Frame(self.notebook)
        self.notebook.add(self.schedule_tab, text="Scheduled Runs")

        # Create scheduled run frame
        self.schedule_frame = ScheduledRunTab(
3065        self.schedule_tab, self.daq, self.valve_config,
            padding=10
        )
        self.schedule_frame.pack(fill='both', expand=True)

3070    # Setup periodic updates for MFC readings
        self.update_interval = 1000  # ms
        self.update_readings()

    def create_menu(self):
3075    """Create the menu bar"""
        menubar = Menu(self)
```

```
        config_menu = Menu(menubar, tearoff=0)
        config_menu.add_command(label="Device Configuration", command=self.open_config)
        menubar.add_cascade(label="Configuration", menu=config_menu)

        self.config(menu=menubar)

    def open_config(self):
        """Open the configuration window"""
        ConfigWindow(self, self.daq, self.channel_config, self.valve_config)

    def update_channel_labels(self):
        """Update channel info in all control frames"""
        for frame in self.control_frames:
            frame.update_channel_info()

    def update_valve_config(self):
        """Update valve configuration in frames"""
        self.valve_frame.update_valve_ports()
        self.schedule_frame.valve_config = self.valve_config
        for i, scheduler in enumerate(self.schedule_frame.valve_schedulers):
            scheduler.port_line = self.valve_config[i]['port_line']

    def populate_mfcs(self):
        # Add all MFC types with output range 5.0V
        mfc_specs = [
            ("30 SLPM", 30, 5.0),
            ("15 SLPM", 15, 5.0),
            ("5 SLPM", 5, 5.0),
            ("1 SLPM", 1, 5.0),
            ("500 SCCM", 500, 5.0),
            ("100 SCCM", 100, 5.0),
```

```
                ("20 SCCM", 20, 5.0),
                ("10 SCCM", 10, 5.0)
            ]
            for name, max_flow, output_range in mfc_specs:
                self.mfc_manager.add_mfc(name, max_flow, output_range)

        def update_readings(self):
            for frame in self.control_frames:
                frame.update_reading()
            self.after(self.update_interval, self.update_readings)
```

Flowrate.py

```
class MFC:
    def __init__(self, name, max_flow, output_range=5.0):
        self.name = name
        self.max_flow = max_flow
        self.unit = "SLPM" if "SLPM" in name else "SCCM"
        self.output_range = output_range  # Voltage range for feedback signal

    def flow_to_voltage(self, flow_rate):
        """Convert flow rate to voltage (0-5V scale) based on MFC capacity"""
        if self.max_flow <= 0:
            return 0.0

        # Calculate voltage proportionally to max flow
        voltage = (flow_rate / self.max_flow) * 5.0

        # Clamp between 0-5V to prevent out-of-range errors
        return max(0.0, min(5.0, voltage))
```

```
    def voltage_to_flow(self, voltage):
        """Convert voltage to flow rate based on MFC capacity and output range"""
        if self.max_flow <= 0:
            return 0.0
        # Scale based on MFC output range
        raw_value = (voltage / self.output_range) * self.max_flow

        # voltage to flow rate not reading correct values? lets calibrate it.
        calibration_constant = (-0.2681 * self.max_flow) - 0.1454

        calibrated_value = raw_value - calibration_constant

        return calibrated_value

class MFCManager:
    def __init__(self):
        self.mfcs = {}

    def add_mfc(self, name, max_flow, output_range=5.0):
        self.mfcs[name] = MFC(name, max_flow, output_range)

    def get_mfc(self, name):
        return self.mfcs.get(name)

    def get_all_mfc_names(self):
        return list(self.mfcs.keys())
```

NI DAQ.py

```
from Display import MainApp
```

```
if __name__ == "__main__":
    app = MainApp()
    app.mainloop()
```

ValveControlFrame.py

```
import tkinter as tk
from tkinter import ttk, messagebox


class ValveControlFrame(tk.LabelFrame):
    def __init__(self, parent, daq, valve_config, *args, **kwargs):
        super().__init__(parent, *args, **kwargs)
        self.daq = daq
        self.valve_config = valve_config
        self.valve_states = {}
        self.on_buttons = {}
        self.off_buttons = {}

        # Title
        ttk.Label(self, text="Valve Control", font=("Arial", 12, "bold")).grid(
            row=0, column=0, columnspan=4, pady=10
        )

        # Create regular buttons (not ttk) for full color control
        for i, valve in enumerate(self.valve_config):
            # Valve name
            ttk.Label(self, text=valve["name"]).grid(row=i + 1, column=0, padx=10, pady=5, sticky="w")

            # ON button (regular tk.Button)
            on_btn = tk.Button(
```

```
            self,
            text="ON",
            width=6,
            bg="SystemButtonFace",  # Default color
            activebackground="SystemButtonFace",
            command=lambda v=valve["port_line"]: self.set_valve(v, True)
        )
        on_btn.grid(row=i + 1, column=1, padx=5, pady=5)
        self.on_buttons[valve["port_line"]] = on_btn

        # OFF button (regular tk.Button)
        off_btn = tk.Button(
            self,
            text="OFF",
            width=6,
            bg="SystemButtonFace",  # Default color
            activebackground="SystemButtonFace",
            command=lambda v=valve["port_line"]: self.set_valve(v, False)
        )
        off_btn.grid(row=i + 1, column=2, padx=5, pady=5)
        self.off_buttons[valve["port_line"]] = off_btn

        # Store state indicator
        self.valve_states[valve["port_line"]] = tk.StringVar(value="OFF")
        state_label = ttk.Label(self, textvariable=self.valve_states[valve["port_line"]], width=8)
        state_label.grid(row=i + 1, column=3, padx=5, pady=5)

        # Set initial state to OFF
        self.set_valve(valve["port_line"], False)

    def set_valve(self, port_line, state):
```

```
            try:
                self.daq.write_digital(port_line, state)
                self.valve_states[port_line].set("ON" if state else "OFF")

                # Update button colors directly
                if state:
                    self.on_buttons[port_line].config(bg="#39FF14", activebackground="#39FF14")  # Neon green
                    self.off_buttons[port_line].config(bg="SystemButtonFace", activebackground="SystemButtonFace")
                else:
                    self.on_buttons[port_line].config(bg="SystemButtonFace", activebackground="SystemButtonFace")
                    self.off_buttons[port_line].config(bg="#FF5F1F", activebackground="#FF5F1F")  # Neon orange

            except Exception as e:
                messagebox.showerror("Valve Error", f"Failed to control valve: {str(e)}")

        def update_valve_ports(self):
            """Update valve ports after configuration change"""
            for i, valve in enumerate(self.valve_config):
                port_line = valve['port_line']
                state = self.daq.read_digital_state(port_line)
                self.valve_states[port_line].set("ON" if state else "OFF")

                # Update button colors
                if state:
                    self.on_buttons[port_line].config(bg="#39FF14", activebackground="#39FF14")
                    self.off_buttons[port_line].config(bg="SystemButtonFace", activebackground="SystemButtonFace")
                else:
                    self.on_buttons[port_line].config(bg="SystemButtonFace", activebackground="SystemButtonFace")
                    self.off_buttons[port_line].config(bg="#FF5F1F", activebackground="#FF5F1F")
```

ValveScheduler.py

```
import tkinter as tk
from tkinter import ttk



class ValveScheduler(ttk.LabelFrame):
    def __init__(self, parent, valve_index, daq, valve_config, *args, **kwargs):
        super().__init__(parent, *args, **kwargs)
        self.daq = daq
        self.valve_index = valve_index
        self.valve_config = valve_config
        self.port_line = valve_config[valve_index]['port_line']
        self.swap_job_ids = []
        self.current_state = False  # Track current valve state

        # Configure grid for column layout
        self.columnconfigure(0, weight=1)

        # Valve label
        ttk.Label(self, text=f"Valve {valve_index + 1} Schedule", font=("Arial", 10, "bold")).grid(
            row=0, column=0, padx=5, pady=5, sticky="w"
        )

        # Initial state
        ttk.Label(self, text="Initial State:").grid(row=1, column=0, sticky="w", padx=5)
        self.initial_state_var = tk.StringVar(value="OFF")
        initial_frame = ttk.Frame(self)
        initial_frame.grid(row=2, column=0, sticky="w", padx=5, pady=(0, 10))
        ttk.Radiobutton(initial_frame, text="OFF", variable=self.initial_state_var,
                        value="OFF").pack(side=tk.LEFT)
```

```
        ttk.Radiobutton(initial_frame, text="ON", variable=self.initial_state_var,
                        value="ON").pack(side=tk.LEFT, padx=(10, 0))

        # Swap schedule header
        header = ttk.Frame(self)
        header.grid(row=3, column=0, sticky="we", padx=5)
        ttk.Label(header, text="Time (s)", width=8).pack(side=tk.LEFT)
        ttk.Label(header, text="Action", width=8).pack(side=tk.LEFT, padx=5)
        ttk.Label(header, text="Remove", width=8).pack(side=tk.RIGHT)

        # Container for swap rows
        self.swap_rows_frame = ttk.Frame(self)
        self.swap_rows_frame.grid(row=4, column=0, sticky="we", padx=5, pady=5)

        # Add/remove buttons
        btn_frame = ttk.Frame(self)
        btn_frame.grid(row=5, column=0, sticky="w", padx=5, pady=(0, 10))
        ttk.Button(btn_frame, text="+ Add Swap", command=self.add_swap_row).pack(side=tk.LEFT)
        ttk.Button(btn_frame, text="- Remove Last", command=self.remove_last_swap).pack(side=tk.LEFT,
    padx=5)

        # Store swap variables
        self.swap_vars = []

        # Add one initial row
        self.add_swap_row()

    def add_swap_row(self):
        """Add a new row to the valve schedule table"""
        row = ttk.Frame(self.swap_rows_frame)
        row.pack(fill=tk.X, pady=2)
```

```
        # Time entry
        time_var = tk.StringVar(value="0.0")
        time_ent = ttk.Entry(row, width=8, textvariable=time_var)
        time_ent.pack(side=tk.LEFT)

        # Action selection
        action_var = tk.StringVar(value="ON")
        action_cmb = ttk.Combobox(row, width=8, textvariable=action_var, state="readonly")
        action_cmb['values'] = ("ON", "OFF")
        action_cmb.pack(side=tk.LEFT, padx=5)

        # Remove button
        remove_btn = ttk.Button(row, text="Remove", width=8, command=lambda r=row:
self.remove_swap_row(r))
        remove_btn.pack(side=tk.RIGHT)

        # Store variables
        self.swap_vars.append((time_var, action_var, row))

    def remove_swap_row(self, row):
        """Remove a specific row from the valve schedule"""
        for i, (time_var, action_var, row_widget) in enumerate(self.swap_vars):
            if row_widget == row:
                self.swap_vars.pop(i)
                row.destroy()
                break

    def remove_last_swap(self):
        """Remove the last swap from the schedule"""
        if self.swap_vars:
```

```
            _, _, row = self.swap_vars.pop()
            row.destroy()

    def get_schedule(self):
        """Get the schedule for this valve"""
        schedule = []
        for time_var, action_var, _ in self.swap_vars:
            try:
                time_val = float(time_var.get())
                action_val = action_var.get()
                if time_val >= 0:
                    schedule.append((time_val, action_val))
            except ValueError:
                pass  # Skip invalid entries
        return sorted(schedule, key=lambda x: x[0])

    def set_initial_state(self):
        """Set the initial state for this valve"""
        state = self.initial_state_var.get() == "ON"
        self.current_state = state
        self.daq.write_digital(self.port_line, state)

    def schedule_actions(self, master):
        """Schedule all actions for this valve"""
        schedule = self.get_schedule()
        for time_val, action_val in schedule:
            state = action_val == "ON"
            job_id = master.after(
                int(time_val * 1000),
                lambda s=state: self.set_valve_state(s)
            )
```

```
            self.swap_job_ids.append(job_id)

    def set_valve_state(self, state):
        """Set valve state and track current state"""
        self.current_state = state
        self.daq.write_digital(self.port_line, state)

    def cancel_schedules(self):
        """Cancel all scheduled jobs for this valve"""
        for job_id in self.swap_job_ids:
            self.master.after_cancel(job_id)
        self.swap_job_ids = []

    def turn_off(self):
        """Turn off this valve"""
        self.current_state = False
        self.daq.write_digital(self.port_line, False)
```

References

Altshuller, A. P.: Assessment of the Contribution of Chemical Species to The Eye Irritation Potential of Photochemical Smog, J. Air Pollut. Control Assoc., 28, 594–598, https://doi.org/10.1080/00022470.1978.10470634, 1978.

Darley, E. F., Kettner, K. A., and Stephens, E. R.: Analysis of Peroxyacyl Nitrates by Gas Chromatography with Electron Capture Detection, Anal. Chem., 35, 589–591, https://doi.org/10.1021/ac60197a028, 1963.

Flocke, F. M., Weinheimer, A. J., Swanson, A. L., Roberts, J. M., Schmitt, R., and Shertz, S.: On the measurement of PANs by gas chromatography and electron capture detection, J. Atmos. Chem., 52, 19–43, https://doi.org/10.1007/s10874-005-6772-0, 2005.

Furgeson, A., Mielke, L. H., Paul, D., and Osthoff, H. D.: A photochemical source of peroxypropionic and peroxyisobutanoic nitric anhydride, Atmos. Environm., 45, 5025–5032, https://doi.org/10.1016/j.atmosenv.2011.03.072, 2011.

Gaffney, J. S., and Marley, N. A.: The Impacts of Peroxyacetyl Nitrate in the Atmosphere of Megacities and Large Urban Areas: A Historical Perspective, ACS Earth Space Chem., 5, 1829–1841, https://doi.org/10.1021/acsearthspacechem.1c00143, 2021.

Gomez, A., Hallett, A., Easterbrook, K., Miller, A., & Osthoff, H.: Measurement of Henry's law solubility and liquid-phase loss rate constants for acryloyl peroxynitrate (APAN) in deionized water at room temperature. Journal of Atmospheric Chemistry, 82, 75–92. https://doi.org/10.1007/s10874-025-09475-4, 2025.

Grosjean, D., Williams, E. L., and Grosjean, E.: Peroxyacyl nitrates at southern California mountain forest locations, Environm. Sci. Technol., 27, 110–121, https://doi.org/10.1021/es00038a011, 1993.

Measurement Computing Corporation. (2023a). USB-1408FS user's guide. https://www.mccdaq.com/PDFs/manuals/USB-1408FS.pdf

Measurement Computing Corporation. (2023b). USB-1608FS user's guide. https://www.mccdaq.com/PDFs/manuals/USB-1608FS.pdf

Mielke, L. H., and Osthoff, H. D.: On quantitative measurements of peroxycarboxylic nitric anhydride mixing ratios by thermal dissociation chemical ionization mass spectrometry, Int. J. Mass Spectrom., 310, 1–9, https://doi.org/10.1016/j.ijms.2011.10.005, 2012.

Pätz, H.-W., Lerner, A., Houben, N., and Volz-Thomas, A.: Validation of a new method for the calibration of peroxy acetyl nitrate (PAN)-analyzers, Gefahrstoffe Reinhaltung Der Luft, 62, 215–219, 2002.

Rider, N. D., Taha, Y. M., Odame-Ankrah, C. A., Huo, J. A., Tokarek, T. W., Cairns, E., Moussa, S. G., Liggio, J., and Osthoff, H. D.: Efficient photochemical generation of peroxycarboxylic nitric anhydrides with ultraviolet light-emitting diodes, Atmos. Meas. Tech., 8, 2737–2748, https://doi.org/10.5194/amt-8-2737-2015, 2015.

Roberts, J. M.: The atmospheric chemistry of organic nitrates, Atmos. Environm. A, 24, 243–287, https://doi.org/10.1016/0960-1686(90)90108-Y, 1990.

Roberts, J. M., Neuman, J. A., Brown, S. S., Veres, P. R., Coggon, M. M., Stockwell, C. E., Warneke, C., Peischl, J., and Robinson, M. A.: Furoyl peroxynitrate (fur-PAN), a product of VOC–NOx photochemistry from biomass burning emissions: photochemical synthesis, calibration, chemical characterization, and first atmospheric observations, Environmental Science: Atmospheres, 2, 1087–1100, https://doi.org/10.1039/D2EA00068G, 2022.

Slusher, D. L., Huey, L. G., Tanner, D. J., Flocke, F. M., and Roberts, J. M.: A thermal dissociation-chemical ionization mass spectrometry (TD-CIMS) technique for the simultaneous measurement of peroxyacyl nitrates and dinitrogen pentoxide, J. Geophys. Res., 109, D19315, https://doi.org/10.1029/2004JD004670, 2004.

Taha, Y. M., Saowapon, M. T., Assad, F. V., Ye, C. Z., Chen, X., Garner, N. M., and Osthoff, H. D.: Quantification of peroxynitric acid and peroxyacyl nitrates using an ethane-based thermal dissociation peroxy radical chemical amplification cavity ring-down spectrometer, Atmos. Meas. Tech., 11, 4109–4127, https://doi.org/10.5194/amt-11-4109-2018, 2018.

Tokarek, T. W., Huo, J. A., Odame-Ankrah, C. A., Hammoud, D., Taha, Y. M., and Osthoff, H. D.: A gas chromatograph for quantification of peroxycarboxylic nitric anhydrides calibrated by thermal dissociation cavity ring-down spectroscopy, Atmos. Meas. Tech., 7, 3263–3283, https://doi.org/10.5194/amt-7-3263-2014, 2014.

Veres, P. R., and Roberts, J. M.: Development of a photochemical source for the production and calibration of acyl peroxynitrate compounds, Atmos. Meas. Tech., 8, 2225–2231, https://doi.org/10.5194/amt-8-2225-2015, 2015.

VICI. (2009). Technical note TN413: Microelectric actuator operating modes. Valco Instruments Company Inc. https://www.vici.com/support/tn/tn413.pdf

Volz-Thomas, A., Xueref, I., and Schmitt, R.: An automatic gas chromatograph and calibration system for ambient measurements of PAN and PPN, Environm. Sci. Poll. Res., 9, 72–76, 2002.

Warneck, P., and Zerbach, T.: Synthesis of peroxyacetyl nitrate in air by acetone photolysis, Environm. Sci. Technol., 26, 74–79, https://doi.org/10.1021/es00025a005, 1992.

Zheng, W., Flocke, F. M., Tyndall, G. S., Swanson, A., Orlando, J. J., Roberts, J. M., Huey, L. G., and Tanner, D. J.: Characterization of a thermal decomposition chemical ionization mass spectrometer for the measurement of peroxy acyl nitrates (PANs) in the atmosphere, Atmos. Chem. Phys., 11, 6529–6547, https://doi.org/10.5194/acp-11-6529-2011, 2011.