# Overview/Using this guide

**Title:**

Wet Test Fixture Python Programming Guide

**Client:**

Profound Medical Inc, 2400 Skymark Ave #6, Mississauga, ON L4W 5K5, Canada

**Date:**

June 15th, 2022

**Contributions:**

|  |  |
| --- | --- |
| Program function & Design | Isaiah Baker |
| Programming & development | Isaiah Baker, Rameen Housseini |
| Report Preparation | Isaiah Baker, Rameen Housseini, Axel Montano |
| Corresponding Author | Rajiv Chopra (rajiv@8foldmfg.com) |

**Report Revisions:**

|  |  |
| --- | --- |
| Revision 1.0 | Initial Guide |

**Overview/Summary of utilizing this guide**

This guide is intended to provide a general understanding of how the parts of the Wet Test Fixture Python Interface codebase work and interact with each other. It will provide information on where to find the relevant classes and functions so future programmers can troubleshoot problems, maintain the code, and make improvements and additions in the future. It will also walk the programmer through the examples found in <repository folder>\Examples which provide simplified examples of key concepts to this application’s design.

More detailed information on how the code works can be found in the docstrings at the top of most functions and classes, as well as the comments found throughout the code. This document also contains a linked glossary to key terms, so it is recommended to click on those whenever available to fully understand the topics being discussed.

Below is an interactive table of contents that will hyperlink you to the relevant chapters and pages for the topic you select.

## Table of Contents

[Overview/Using this guide 1](#_Toc108084084)

[Table of Contents 2](#_Toc108084085)

[Technologies and Dependencies 6](#_Toc108084086)

[General Programming 6](#_Toc108084087)

[Python 3.8 6](#_Toc108084088)

[GitHub 6](#_Toc108084089)

[Conda 6](#_Toc108084090)

[**User Interface/Application Architecture/Multithreading** 7](#_Toc108084091)

[PyQt5 7](#_Toc108084092)

[**Hardware Interfacing Libraries** 7](#_Toc108084093)

[Mini Circuits PWR-SEN-8FS Power Meter (PythonNet, import clr) 7](#_Toc108084094)

[Keysight DSOX2002A Oscilloscope, 33500B Signal Generator (PyVISA) 7](#_Toc108084095)

[NI USB-6009 Digital IO board, NI USB-TC01 thermocouple (NI-DAQmx) 7](#_Toc108084096)

[Galil DMC 4143 Motor Controller (gclib) 7](#_Toc108084097)

[UA Interface (subprocess) 7](#_Toc108084098)

[Serial library (MT Balance, Parker Motor Controller, Power relay) 7](#_Toc108084099)

[**Debugging** 8](#_Toc108084100)

[Logging Library 8](#_Toc108084101)

[Termcolor library 8](#_Toc108084102)

[**Data Analysis** 8](#_Toc108084103)

[NumPy, SciPy, and Statistics Libraries 8](#_Toc108084104)

[Re library 8](#_Toc108084105)

[**File input / output** 8](#_Toc108084106)

[Shutil Library 8](#_Toc108084107)

[Yaml library 8](#_Toc108084108)

[ConfigParser 8](#_Toc108084109)

[**Abstraction** 9](#_Toc108084110)

[ABC (abstract base class) library 9](#_Toc108084111)

[**High Level Overview** 9](#_Toc108084112)

[**Class Diagram** 9](#_Toc108084113)

[Architecture Explanation 10](#_Toc108084114)

[Sequence of Events 11](#_Toc108084115)

[Startup 11](#_Toc108084116)

[Scripts 11](#_Toc108084117)

[Abort Immediately Button Clicked 12](#_Toc108084118)

[Abort After Step Button Clicked 12](#_Toc108084119)

[**Summaries of Complex Script Actions** 13](#_Toc108084120)

[Measure Element Efficiency RFB 13](#_Toc108084121)

[Find Element ‘n’ 13](#_Toc108084122)

[Organization of Files 14](#_Toc108084123)

[Top Level Files 14](#_Toc108084124)

[User Interface files and classes: 14](#_Toc108084125)

[Hardware Classes and Tests: 14](#_Toc108084126)

[Abstract Hardware Classes 15](#_Toc108084127)

[Real Hardware Classes 16](#_Toc108084128)

[Graphical File and Folder Structure Representation 17](#_Toc108084129)

[**Other Files** 18](#_Toc108084130)

[definitions.py: 18](#_Toc108084131)

[formulas.py: 18](#_Toc108084132)

[environment.yml: 18](#_Toc108084133)

[Systeminfo.ini: 18](#_Toc108084134)

[Utilities folder: 18](#_Toc108084135)

[**Explanation of Key Classes** 19](#_Toc108084136)

[Manager 19](#_Toc108084137)

[**Programming Guidelines and Best Practices** 21](#_Toc108084138)

[Type Hinting 21](#_Toc108084139)

[Units 23](#_Toc108084140)

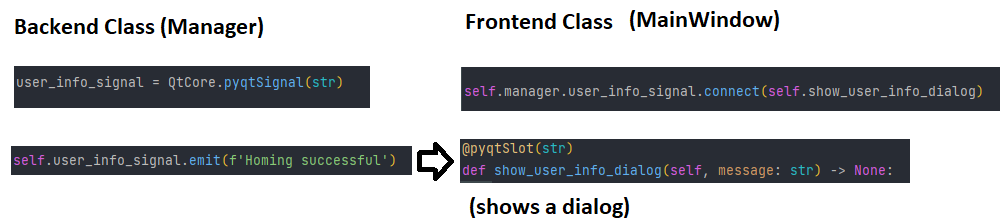
[Developing New Hardware Classes 23](#_Toc108084141)

[Developing New Script Actions 27](#_Toc108084142)

[**Troubleshooting** 30](#_Toc108084143)

[Debugging 30](#_Toc108084144)

[Key Concepts with Examples 31](#_Toc108084145)

[Example 1: Triggering frontend methods from a backend class **** Having a single UI thread is key to the framework of the graphics library (pyQt5), and programmers should avoid calling any Qt methods that change the appearance of the UI from any other thread, as doing so will very likely cause crashes. For this reason, one should only call methods that draw UI elements from the MainWindow class or a UI class, and refrain from triggering these methods with anything other than a pyqtSignal. For a minimal example of this technique in action, refer Examples\change\_ui\_from\_manager. 31](#_Toc108084146)

[Example 2: Triggering backend methods from a frontend class 31](#_Toc108084147)

[Example 3: Triggering long-running backend methods from a frontend class 32](#_Toc108084148)

[Example 4: Sensor Capture Multithreading: 33](#_Toc108084149)

[**Basic Usage** 34](#_Toc108084150)

[Installation 34](#_Toc108084151)

[Script Manipulation 38](#_Toc108084152)

[Change Step Variable Value When a script is loaded, you can expand a script step to see its parameters. 40](#_Toc108084153)

[Add Step 41](#_Toc108084154)

[Deleting Step(s) 43](#_Toc108084155)

[Updating Tree 45](#_Toc108084156)

[Moving Steps 45](#_Toc108084157)

[Saving the Edited/New Script 46](#_Toc108084158)

[Running a Script 47](#_Toc108084159)

[Glossary 49](#_Toc108084160)

[UI Classes 49](#_Toc108084161)

[Config Classes 49](#_Toc108084162)

[Qt Designer (https://build-system.fman.io/qt-designer-download) 49](#_Toc108084163)

[Creating a New UI Element 49](#_Toc108084164)

[Adding Widgets and Layouts to the UI Element 50](#_Toc108084165)

[Manipulating Specific Widgets and their Attributes 52](#_Toc108084166)

[Adding the UI Element to the Application 53](#_Toc108084167)

[Nesting Widgets 56](#_Toc108084168)

[Spacers 60](#_Toc108084169)

[Conclusion 60](#_Toc108084170)

[Unit Test 61](#_Toc108084171)

[Folder Hierarchy of Unit Tests 61](#_Toc108084172)

[Syntax Structure for Unit Tests 62](#_Toc108084173)

[Signals 63](#_Toc108084174)

[Log 64](#_Toc108084175)

# Technologies and Dependencies

This section includes a summary of the technologies used in the development of this application, as well as their use and justification. For instructions on installing these dependencies, refer to the readme.md file in the root of the repository.

## General Programming

### Python 3.8

Python 3.8 is the latest release of Python that can be used with Python.NET, which is necessary for the power meter’s hardware class. Python.NET is also compatible with Python 2.7, 3.5, 3.6, and 3.7, but we recommend using Python 3.8 since all our testing has been done with this version of Python, specifically 3.8.13 for the Conda environment, or at least 3.8.10 for running the program from the Windows environment.

### GitHub

GitHub’s GIT VCS (version control system) allows for seamless branch creating and merging. As well as issue tracking and change blaming. The most important aspect of GitHub is the ability to review changes before a merge, before a commit, and to rollback to previous versions of the code in case something important was lost via a bad merge, or a computer crash, etc. GitHub also has a user-friendly GUI and is free to use.

### Conda

While not required, Conda makes installing required libraries and packages seamless and easy. It also allows you to write/export an environment file that others can use to automatically install all required packages on their machine for quick setup. The GUI aspect of Conda is called **Anaconda Navigator** which is also essential for working with Conda. PyCharm comes with integrated support for Conda, we are unsure about other Python IDEs, so we recommend you research the IDE’s support for Conda before adopting it if you wish to use a different IDE. Conda also allows installing packages and libraries from different repositories from the default one. So far, the only repository we had to install from in addition to the default one is the “forge” repository. These can also be pre-defined in the environment file.

## **User Interface/Application Architecture/Multithreading**

### PyQt5

PyQt5 can be installed through Conda, or pip. This library must be imported by any class that uses a item type that has anything to do with the Qt GUI library. Even items that are not explicitly GUI related (such as ‘qWait,’ or ‘QEvent’) need the PyQt5 library imported. It may be theoretically possible to upgrade the project to PyQt6, but we do not recommend this unless the software team has a lot of time to dedicate fixing potential issues from arising, and there is no substantial difference between the two.

## **Hardware Interfacing Libraries**

### Mini Circuits PWR-SEN-8FS Power Meter (PythonNet, import clr)

PythonNet is a required library for interpreting DLL files and using methods built into them in the Python source code. As mentioned above, this was only needed for the power meter hardware class and is only compatible with a few Python versions including Python 3.8. Refer to the power meter programming manual in the Hardware Programming Manuals folder for more information.

### Keysight DSOX2002A Oscilloscope, 33500B Signal Generator (PyVISA)

The PyVISA library is required to communicate with the Keysight equipment using python. Its installer includes the NI-Visa interactive control tool for sending and receiving commands with these instruments directly in a terminal-like interface, which is useful for debugging and testing commands before using them in Python.

### NI USB-6009 Digital IO board, NI USB-TC01 thermocouple (NI-DAQmx)

The NI-DAQmx library is required for communicating with the digital IO board, as well as the thermocouple.

### Galil DMC 4143 Motor Controller (gclib)

The Galil motor controllers need to have gclib imported for Python to communicate with it. You may either download the API through our Setup.bat file, the official Galil website (<http://www.galil.com/downloads/api/>) or from Conda directly into your working environment.

### UA Interface (subprocess)

Subprocess is a built-in library that allows python to run executables with arguments and direct their output. This is required to use the proprietary executable for reading and writing data using the UA interface. This executable can be found in the abstract\_ua\_interface folder.

### Serial library (MT Balance, Parker Motor Controller, Power relay)

This is a built-in library used for sending and receiving data over a USB connection, usually as a stream of text or bytes. This is used for communicating for COM devices such as the MT Balance, Parker Motor Controller, Power relay. Each has a specific COM port, which can be found using device manager, as well as a specific Baud rate (check the hardware manuals if issues arise) and may require a newline or a carriage return character between commands.

We recommend using a simple serial terminal software for troubleshooting these devices (such as the one built into the Arduino IDE)

## **Debugging**

### Logging Library

The logging library is what unifies all log statements in all classes to one handler and processor. The self.log(level:str, message:str) method is accessible to most of the frontend and backend classes. Hardware devices inherit it from the AbstractDevice class, whereas UI classes inherit it either from the MyQWidget or MyQDialog custom classes. These inherited classes handle the setup of these methods, so it does not need to be repeated in every class. This includes the necessary imports, instantiating the root logger, et cetera. This allows all the classes to collectively modify the WTF.log file in the logs folder. This method automatically adds the line number, class name, and thread name of where the log message was coming from.

### Termcolor library

Allows for colored output, very useful for debugging via print statements.

**Data Analysis**

NumPy, SciPy, and Statistics Libraries

NumPy and SciPy basically handle the heavy lifting of writing mundane to complex mathematical functions. We’ve used them to integrate a list of squared values using Simpson’s rule. Particularly when finding the voltage squared integral of the oscilloscope waveform. NumPy and SciPy are not built in but can be installed via Conda.

Re library

This python library allows us to search through strings via a regular expression, a very useful search syntax pattern that allows for seamless searches of complex and unconventional patterns.

**File input / output**

Shutil Library

Used in the FileSaver class, gives python the ability to copy files amongst other high-level file operations.

Yaml library

A library that allows for the automatic and integrated interpretation of .yaml files in Python. Can be installed through the package ‘pyyaml’ in Conda or through pip.

ConfigParser

A built-in library for parsing .INI files such as systeminfo.ini, used in the ui\_system\_info class.

**Abstraction**

ABC (abstract base class) library

This library is needed to write abstract classes and methods, as well as allowing multiple abstraction layers for a concrete class via importing ABCMeta from ABC (a good example of this is the abstract\_device.py file in the /Hardware/Abstract/ directory. This abstract class inherits QObject, which itself is an abstract class. To allow this, ABCMeta must be imported and used via: \_\_metaclass\_\_ = ABCMeta inside the class definition.

**High Level Overview**

**Class Diagram**

Diagram

Description automatically generated

Architecture Explanation

At the core of this application’s design are two threads which run as the application is open. The frontend thread “ui thread”, controlled by the overarching MainWindow class, and the backend thread “manager thread” controlled by the overarching manager class. The manager class creates and commands hardware classes which utilize low level APIs from the device manufacturers to control the hardware. The MainWindow class instantiates ui classes which provide the layout and functionality of the user interface. The UI classes include widgets that are populated into the MainWindow’s tabs, and pop-up dialogs, which are created by the MainWindow class in response to events.

Backend classes should never call frontend methods directly and vice versa, and in general methods from one thread should not call methods from another thread. Doing so can lead to a host of issues from race conditions to crashes. The solution is to use pyqtSignals and pyqtSlots. In a nutshell, emitting a signal is used as an indirect way of calling a function (the slot). Just like how a function call signal contains zero or more parameters, which are passed to a special function called a slot. When one QObject emits a signal, and the signal has been connected (i.e., another QObject is “listening”), the slot function in the recipient class is called with the signal’s parameters.

Sequence of Events

Startup

1. Main.py begins running
2. Create QApplication
3. Create and show  MainWindow
4. Load system info and config files and populate UI with them
5. Launch a password prompt
6. Begin the manager thread and connect all devices

Scripts

1. User clicks load script
2. (MainWindow.load\_script\_clicked) QFileDialog launches asking the user to choose a script file. The path is emitted as a signal.
3. (Manager.load\_script) The path is received by Manager, which aborts the current script if there is one and sets the step index to -1. It loads the script file and parses it into lists with length equal to the total number of actions to be completed, including repetitions but not including the begin loop or end loop tasks.
   1. task\_names: a list of strings containing the task names (and repetition number)
   2. task\_args: a list of dictionaries containing the parameters for each task, to be passed to the method for each script action in manager
   3. task\_execution\_order: a list containing the task number of each step in the script, and if it is a looped step the loop number and element number as well.
4. Manager emits a parsed list of dictionaries containing script info (containing loop items but not repeated steps loop iterations)
5. (MainWindow.visualize\_script) script is displayed in a QTreeWidget (script data is now accessible to the script editor as well)
6. User clicks “Run script”
7. (Mainwindow.run\_button\_clicked) buttons are disabled, serial number is read from the UA, and a dialog appears prompting the user for more metadata.
8. (PretestDialog class) a TestData object is created to package test metadata.
9. (PretestDialog.ok\_clicked) metadata is emitted as a signal containing the TestData object.
10. (Manager.test\_metadata\_slot) the Manager’s test\_data attribute is cleared and then populated with the metadata sent by the PretestDialog.
11. (Manager.run\_script) Flag variables like “currently\_scripting” are set so that the manager’s run loop knows to begin script execution in its core run loop.
12. (Manager.advance\_script) Step index is increased by 1 and run\_script\_step is called
13. (Manager.run\_script\_step) The step with the current script index is executed using task\_names[i] to identify which function to execute and task\_args[i] as the sole parameter, containing all necessary arguments for the function
14. Steps 12 and 13 are repeated until the end of the script is reached, the user clicks “abort after step” or “abort immediately”, the user closes the window, or an interrupt or pass/fail action aborts the script.

Abort Immediately Button Clicked

1. MainWindow.abort\_immediately\_button.clicked signal triggers manager.abort\_immediately slot
2. (Manager.abort\_immediately)Script control variables like step\_index, and currently\_scripting are reset, step number of zero is emitted to the script visual in the UI.
3. (Manager.abort\_immediately) abort\_immediately\_var is set to true.
4. (One of manager’s script methods) abort\_immediately\_var is checked by an if statement and the method returns false via its continue variable, meaning to stop the current script action and the higher-level methods that called it (if any)
5. The higher-level methods (if any) check the continue Boolean returned by the lower-level method and if it is false, they return False as well
6. The script does not continue because currently\_scripting is false

Abort After Step Button Clicked

1. MainWindow.abort\_after\_step\_button.clicked signal triggers manager.abort\_after\_step slot
2. (Manager.abort\_immediately)Script control variables like step\_index, and currently\_scripting are reset, step number of zero is emitted to the script visual in the UI.
3. On the next iteration of the run loop, the script is not advanced because currently\_scripting is false.

**Summaries of Complex Script Actions**

Measure Element Efficiency RFB

1. Unpacks variables from a dictionary and typecasts them as necessary
2. Create an RFBData object to store the raw and analyzed data captured by this action
3. Configure the AWG to the desired frequency and amplitude in continuous mode.
4. Select the desired UA channel using the relay in the power module box
5. Create and start an RFBDataLogger thread to capture data from the forward and reflected power meters, as well as the RF Balance for the desired amount of time
6. In a loop with a specified number of iterations, turn the ultrasound output on and off at specified intervals (by turning the AWG output on and off), while continually retrieving data from the RFBDataLogger and emitting it as a signal to be visualized in the rfb tab in the UI
7. Stop the rfb logger and sensor threads
8. Perform data analysis on the completed data (see data analysis documentation)
9. If the data is invalid or has an abnormally high standard deviation, trigger an interrupt action which may prompt the user if they wish to re-run the test, abort automatically or retry automatically (depending on the “Interrupt action” parameter in the config file)

Find Element ‘n’

1. Unpacks variables from a dictionary and typecasts them as necessary
2. Configures the AWG to the desired frequency and amplitude in toneburst mode.
3. Configures the timebase of the oscilloscope
4. Rotates the UA to face the hydrophone
5. (If enabled) Loops through a given range of X coordinates
6. Captures a hydrophone waveform (or the RMS value only) at each coordinate
7. Moves the UA to the X coordinate with the highest voltage squared interval (or rms voltage)
8. (If enabled) Loops through a given range of R coordinates
9. Captures a hydrophone waveform (or the RMS value only) at each coordinate
10. Stores the element X and R coordinates in the element\_x\_coordinates and element\_r\_coordinates variables in manager so they can be referenced in later script actions and saved to disk

This method cycles the ultrasound output from the ultrasound applicator on and off for specified intervals while continuously capturing data from the forward power meter, reflected power meter, and radiation force balance. The readings are captured from all 3 sensors at the same time using 4 additional threads directed by using a RFBDataLogger object.

The refresh\_rfb\_tab helper method scrapes data from the RFBDataLogger and sends it as a signal to be displayed in the RFBTab of the UI. Finally, this method analyzes the data (stored in a RFBData object) and sends it to the FileSaver to be saved.

Organization of Files

Top Level Files

The three highest level python files are main.py, the script which directs the sequence of events upon application launch, MainWindow.py, the parent of all UI elements, and central frontend class, and manager.py, the central backend class which creates and directs hardware devices. All three of these classes are in the root of the repository.

User Interface files and classes:

All the graphical elements of this program were designed in Qt designer, which outputted .ui files specifying various aspects, attributes, properties, behaviors, and contents of each individual window/prompt/dialog. We then subsequently converted the .ui files to python files with the command: “*python -m PyQt5.uic.pyuic -x <filename.ui> -o <filename.py>*”. All these UI and python files are in the **Widget\_Library** folder. This allowed us to interact with the individual objects in the UI element to various program files. Specifically, we imported the converted, now Python classes to new classes contained within the folder **ui\_elements**. While the Widget\_Library folder has no organized structure, the adapted GUI python classes in ui\_elements are organized nicely. Suppose persistent changes are needed towards any of the GUI elements. In that case, you must either modify its respective adapted python file in ui\_elements or open the UI file in Qt Designer, make and save the changes via Qt Designer, then run the *pyuic* command described above to have the change(s) appear during execution.

More specifically regarding the organized structure of the ui\_elements folder: within it contains four self-evidently named folders: “Dialogs,” “Images,” “script\_editor\_menus,” and “tabs.” The tab folder contains operation logic for the tabs you see in the main window. The images folder contains the PNG images used in the RFB tab’s graph legend. There are also three files in ui\_elements. “*my\_qwidget.py*” is a base class for all the task setting options in the script editor tab. It passes the main logger and enables all buttons to the class that inherits it. “*switch.py*” is a custom-made Qt Object that acts as a toggle switch, since the Qt designer suite didn’t come with one built-in. “*ui\_oscilloscope\_plot.py*” is, as the name suggests, the python class that controls the plots in the RFB and Scan tabs of the main window.

Hardware Classes and Tests:

Hardware classes are contained within the “Hardware” folder from the project’s root directory. Each hardware class represents an interface for programmable equipment. There are both real and simulated hardware classes, so the code can still be run for testing purposes even if some devices are not connected. Both real and simulated hardware classes inherit from one or more abstract classes.

Abstract Hardware Classes

The abstract hardware classes are in the **Abstract** folder. They contain function declarations but not implementations. The base class of all hardware devices is AbstractDevice, which defines the methods and attributes every device should have, namely a connect method, a disconnect method, a get serial number method, and a wrap up method which calls the disconnect method. It also includes a method called log(message, level) which does not need to be overridden and gives all hardware devices the ability to log information to the wtf.log file. Note that this is a separate file from the scriptlog and is intended mainly for debugging. See also: Log

More specific abstract classes inherit the functions and attributes of AbstractDevice and build upon them, for example AbstractSensor adds a get\_reading() method. Each real and simulated hardware class inherits from the abstract class of the most specific abstract class it realizes. For example, MTBalance and SimulatedBalance are both realizations of AbstractBalance, and implement all the required abstract methods thereof.

The purpose of this hierarchy is to define the methods and attributes that external classes can count on a certain type of device to have, without worrying about the implementation details for a specific model of device, allowing for a new hardware class for one type of device to be developed according to the abstract template and swapped out easily, ideally changing only the line of code where the device is imported.

In Manager, notice that the type hints for devices are the name of the abstract class. For example, “Oscilloscope: AbstractOscilloscope” This allows the IDE to give type hints and code autofill suggestions for the classes’ abstract template.

Although the type hints say “abstract”, keep in mind that the instances of the device classes in Manager will be a subclass, either for a real device or a simulated device.

To see the documentation for the underlying abstract class, mouse over the method name in PyCharm. If a docstring for the abstract method exists, it will show up in the mouseover interface, unless it is overridden by the docstring of the child class

One drawback to this approach is that if you use ctrl-click to step into a hardware method from the manager class, it will bring you to the abstract class when you may instead want to see the actual hardware class. To get around this, just go to the desired class in the hardware folder and use ctrl-f to find the desired method.

Simulated Hardware Classes

The simulated hardware classes used for testing without real hardware present are in the **Simulated** folder. Unit tests to test these hardware classes, real or simulated, are in the **unit\_test** folder. These should be run to ensure no change in the code breaks any of the hardware code logic. The python classes for real hardware are in the root of the hardware folder with self-descriptive names. The hardware folder also contains the dynamic link libraries (.dll) used for the power meter class in the **power\_meter\_dlls** folder. The **interface\_box\_executable** folder contains the .exe file that the ultrasound actuator box uses to relay information back and forth between the program and itself. The power meter will require PythonNet to be installed, which makes the maximum Python version 3.8.

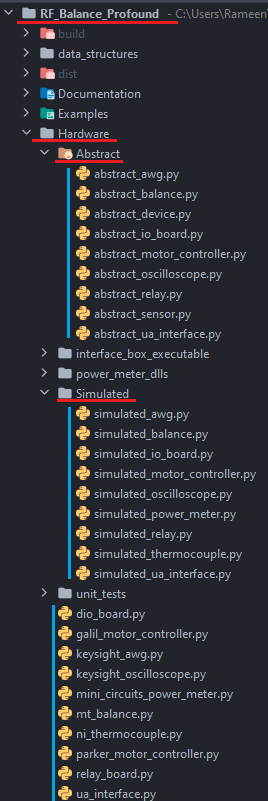
Inside the root project folder also lies “local.yaml” and “default.yaml,” both these files act as the “settings” or “preferences” of the application, allowing the user to set various variables and flags that the source code will refer to. The program will first look in local.yaml to see if the setting value is set in there, if not, it will then check the default.yaml file.

Inside the manager python class, all the hardware is initialized and signals that allow the manager thread to communicate to the main window thread are established. A logger is set, and the manager then waits for the user to load a script or issue a command via the main window. Some of the hardware classes are given a reference to the configuration files described earlier in the manager class. The main window class concurrently initializes all the tab python classes and signals to communicate to the manager thread. It uses signals from the manager to toggle various hardware monitoring indicators in real-time while the user is on the main screen.

Real Hardware Classes

Hardware classes for real programmable devices begin by inheriting from an abstract class and implementing all abstract methods. In general, they must also import a library made by the device’s manufacturer for interfacing with said device. Programming manuals for these devices can be found online or in the “\Hardware manuals” folder in the repository. For predictable performance between different devices, the behavior of any method that overrides an abstract class should obey the docstring of the underlying abstract method. In addition, they should use variables and class attributes from the underlying abstract classes whenever possible and strive to use them in a sensible and self-explanatory way. This is to allow future programmers to program with hardware classes from external classes such as manager without needing to know all the implementation details or syntax of the underlying hardware class. If a piece of hardware is replaced or upgraded, for example the transition from Galil motion controllers to Parker motion controllers, the two classes should realize the same abstract class and behave as similarly as possible. That way, once the new hardware class is implemented, the only thing that needs to be changed is the code in Manager.add\_devices() that imports and instantiates the object.

### Graphical File and Folder Structure Representation



**Other Files**

definitions.py:

The definitions.py file initializes the root directory of the project, config paths, the power meter dll, and the system info ini path. All these values are automatically generated, so you do not need to worry about changing them, since they are in the project folder and python can find the directory it’s placed in. The directories are formatted for Windows.

formulas.py:

This file contains key scientific and physical formulas for this application so they can easily be viewed and changed if necessary. If any features that rely on scientific formulas are added, the formulas themselves should be implemented in this file.

environment.yml:

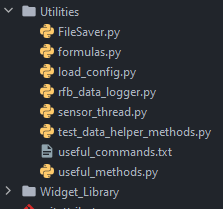
This project uses Conda to organize the required dependencies. The project will come with a file called environment.yaml which specifies the name of the environment, channels to get packages from, dependent package lists from both Conda channels and pip, and the directory to setup this environment. The README.md will describe how to use this file to set up the Conda environment (method 2). Without doing this, if you run the project from an IDE and do not point the IDE to use the Conda environment, the IDE will complain about several missing imports that it cannot identify, and the project will fail to run.

Systeminfo.ini:

This file comprehensively initializes hardware information that pertains to the wet test fixture. These values are then neatly displayed in the “System Info” tab of the main window.

Utilities folder:

This folder contains modularized aspects of the project that pertain to a particular objective, such as loading configuration data, saving information to a file, etc. These python classes are imported into several other classes that need them. You may see what classes are using these python files in PyCharm by selecting one of these files and pressing alt+F7.



**Explanation of Key Classes**

Manager

The manager thread serves several central key functions as the executive class for the backend of the application.

First, it takes in a .WTF script file and parses its contents into various local data structures, such as a list of attribute-value pairs and a list of task names representing the sequence of all actions to be performed in the script in the order they will be executed, as well as a list of dictionaries representing all the parameter names and their values for each action. When a script loops over elements, these lists include entries for each loop iteration but do not include entries for the beginning and end of the loop itself. The script parser can even detect if the “Element” value within a loop is a static element number instead of the “Current” dynamic value usually used in looped tasks, and automatically correct it for you if desired.

Second, Manager manages the order of execution in the script process, including when tab switches should occur, when buttons/inputs should be enabled/disabled, and when dialog boxes occur.

Third, Manager contains the sequential code for each of the script actions themselves, each of which accepts a single dictionary containing all the necessary parameters, unpacks them, and performs the step-by-step sequence associated with that script action.

The main window class mostly contains methods that the manager calls upon to change GUI elements. For example, if the manager wants to disable a certain button, a method written in the main window class (since it has direct access to those elements) will have a pyqtSlot decorator above it and the manager will use its pyqtSignal to toggle those elements on or off. The main window also initializes all the outer layer tabs. The outer layer tabs may have other tabs or multiple layouts that are managed within their own nested classes.

MainWindow

As the central class controlling the user interface, MainWindow is responsible for instantiating all the tab widgets that appear within the application’s main window, as well as all the dialog boxes that appear when certain events happen in the software’s backend classes. It is responsible for disabling the buttons in the user interface when the application is busy and re-enabling them when the process is complete (this involves calling the set\_buttons\_enabled method in all the tab widgets)

MainWindow inherits its layout from the window\_wet\_test.py file in the Widget\_Library, which is generated programmatically based on the .ui file with the same name. This QTDesigner file contains promoted widgets for each of the tabs, which means that although the tabs will appear empty in the QTDesigner file preview, all the tab widget classes will be instantiated and added to the MainWindow when it calls the self.setupUi method. Note that the tab widget classes that are added to the mainwindow are not the .py files in the widget\_library folder, rather they are the files from ui\_elements\tabs folder, which create a layer of functionality on top of the QTDesigner files (see QTDesigner section for more details).

MainWindow triggers actions in Manager, for example when load\_script is clicked it gets the file path using a QFileDialog, and then emits it as a signal to the Manager class which loads and parses it. The run button causes a similar chain of events, except instead of the MainWindow emitting a signal, MainWindow instantiates and shows a custom dialog, connects its signals to Manager, and when the dialog is filled out and dismissed it emits a signal to Manager, triggering the start of the script.

Buttons such as “Abort immediately”, “Abort after step”, “Insert UA”, and “Retract UA” are connected more or less directly to backend classes, so despite being in a backend class their code actually runs in the UI thread. This is permissible because the functions either run quickly or contain one or more self.app.processEvents() calls, ensuring that the UI remains responsive. See example 2 for more details.

MainWindow passes the Manager hardware classes to its child widgets and dialogs so that they can directly communicate using signals. It also connects signals from the Manager and hardware classes to UI elements to display feedback to the user.

Finally, MainWindow visualizes the current script, which it receives as a list of dictionaries emitted as a signal by Manager, by populating it into a QTreeWidget. This QTreeWidget is an object shared with the Script Editor widget, which can modify it. Note that when the script is modified it no longer matches the copy in Manager, so must be saved (and automatically reloaded) before it is run.

**Programming Guidelines and Best Practices**

Type Hinting

The flexibility of python as a dynamically typed language means that, unless otherwise specified, the input(s) of a function can be any type, and the output(s) can be anything, including nothing. On one hand this can be more convenient and allow the programmer to do things that could not be done in a statically typed language, but on the other hand this flexibility can mask eventual TypeErrors that can occur in hard-to-detect edge cases. For example, if a function usually returns a float but occasionally returns a none if a rare error occurs, this can cause errors in other parts of the code that assume the function returns a float. The solution to this is to specify the intended types of a function’s inputs and outputs whenever possible.

Here is an example:



Figure - Notice how parameter hints follow the form: <parameter name>**: <parameter type>** and return hints follow: def func\_name (params) **-> <return type>**

The capital names that appear yellow are imported from the typing library, which includes useful features like Union[...,...] to specify that a parameter can be one of multiple different types. If a function returns more than one value, use Tuple[...,...] to package the types of each returned variable.

Python will **not** stop you from running the code if these hints are violated, IDEs such as PyCharm will flag the violations with strong warnings, which helps to prevent errors before they appear.

Docstrings

Generally, any but the simplest classes and methods should include a docstring. Docstrings should be placed beginning on the line beneath the class or function header, between triple quotation marks “””...”””. The docstring should explain in straightforward terms the intended effect of the code within, without getting into much implementation detail. It is not a replacement for commenting the intention behind sections of code within a function. The docstring should provide a high-level description of how the function turns the inputs into the output (if any), and any other side effects the code has for the class, UI, file outputs, et cetera. Decorators may be used to make the main text (not :param: nor :return:) bold (\*\*text\*\*), code font (``text``), or italic (`text`). Underscore rays can also be used as line separators, as shown below.

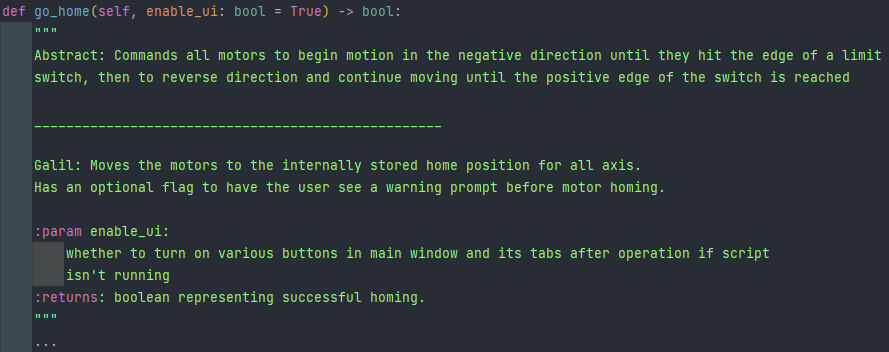


Figure 2 - Format (inside abstract\_motor\_controller.py)

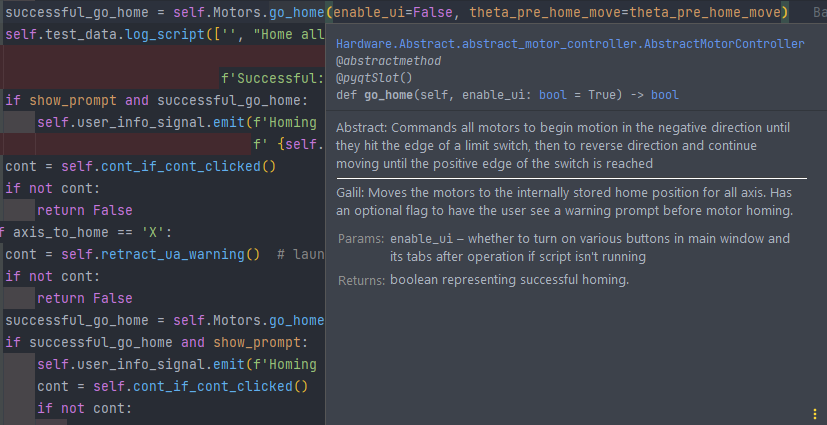


Figure 3 - Result (inside manager.py):

Units

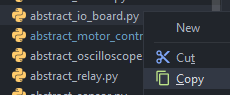
For any variables that contain measurements or data, it is strongly recommended to include an abbreviation for the data’s units in the variable name. Use conventional letter case for the abbreviation, for example “mV” for millivolts, even if this violates variable naming conventions.

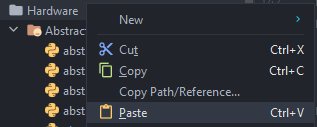
Developing New Hardware Classes

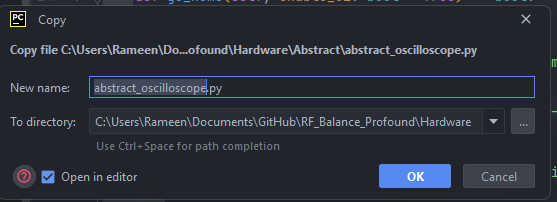
**Note**: It is recommended to read the “*Hardware Classes*” section of the “*Organization of Files*” chapter before reading the following section.

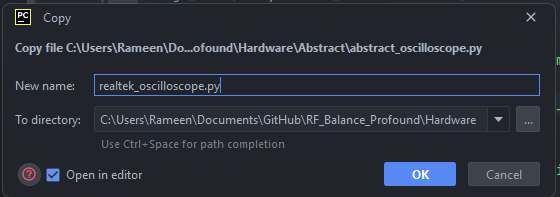
This application was designed to allow for easy development with new or upgraded hardware devices. To develop a class for a different model of a device type that already has a hardware class, follow these basic steps:

* Copy its abstract class for this specific type of device from the ***\Hardware\abstract*** folder into the **root** ***\Hardware*** folder and **rename** the file and class with a descriptive name with its manufacturer and device type

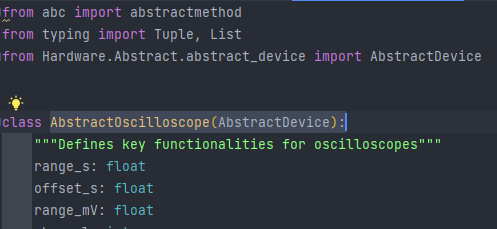


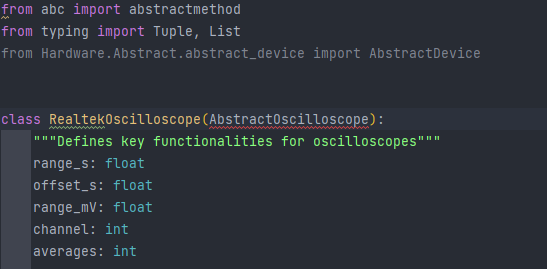




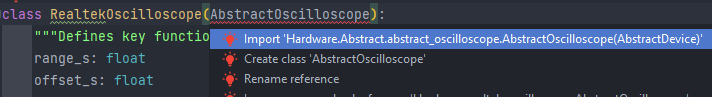


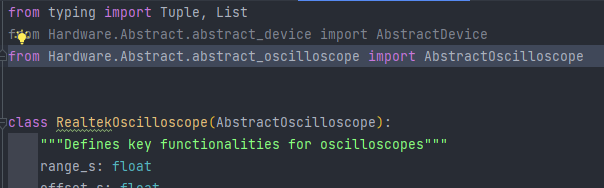
* Replace the inherited class in the parentheses of the class header with the name of the abstract class you just copied, and import that name from the abstract folder (in PyCharm you can right click the name with a red underline and click import this name from… in the context menu)



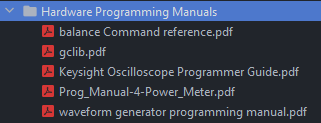




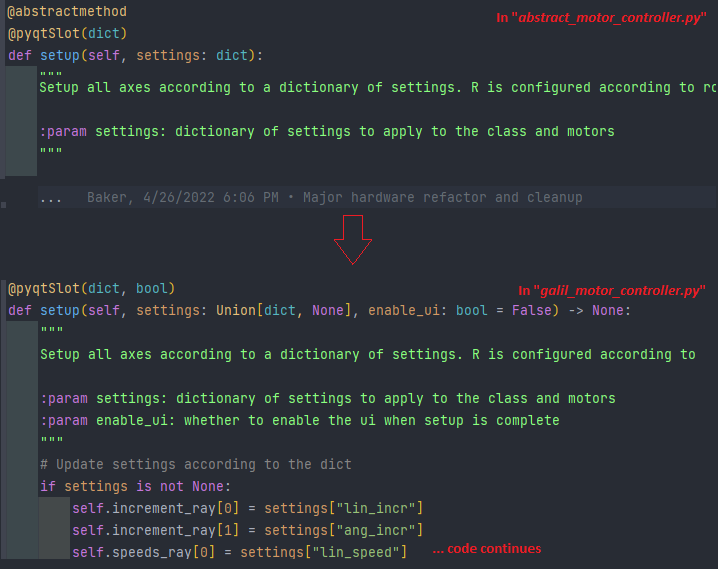




* Learn how to program the device in question by reading its user manual and/or programming manual, contacting the manufacturer, seeking out advice on forums, et cetera.



* Implement the methods labeled *@abstractmethod* and delete this tag from all the implemented methods.

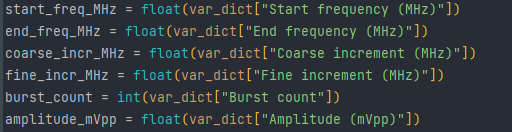


* Make sure to follow the type hints and check for warnings in PyCharm for incorrect types.
* To use the new hardware class, go to the *add\_devices* section in Manager, and find where the device type in question is implemented. Notice that there is an if statement that imports a simulated hardware class if the corresponding variable in the config file is true, or the real device if said variable is false. If you are developing a real hardware interface, change the part of this if statement that does not say “simulated”
* Replace the statement that imports the previous hardware class with one that imports the new (replacement) hardware class. Also change the line of code that instantiates the class, making sure to use the correct class name.
* If the new class is a functional equivalent to the old class, the code should run and perform all the same functionality as before, since the same methods will be called with the same parameters. Some testing and debugging are almost always necessary.
* Try running the unit test from the previous hardware class with your new hardware class (copy the unit test file in Hardware\unit\_tests and swap out the old hardware class with the new one. This should put the hardware class through its paces and alert you if any errors or discrepancies arise.
* Be sure to test every use case of the device that you can think of using a variety of test scripts and user actions in the application, validating that it behaves as intended.

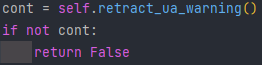
Developing New Script Actions

When developing a new script action, begin by creating a function with a descriptive name that takes a single parameter, var\_dict : dict, and returns a Boolean indicating whether to continue the script.

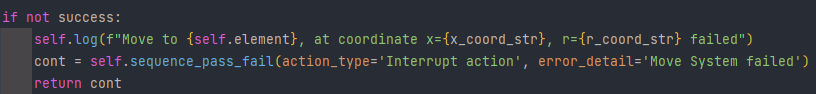
* The var\_dict dictionary will contain all the necessary parameters for the function as name-value pairs. At the beginning of the function, unpack all these variables. The dictionary keys should match the names of the variables in a script file, and the values will usually be expressed as strings since they are read directly from a script file. When unpacking them from the dictionary, give them a variable name like their key in the dictionary, and typecast them as a suitable variable type (bool, int, float, etc.). If it is a string containing an integer such as “Element 1”, use the find\_int method in useful\_methods to parse it out.



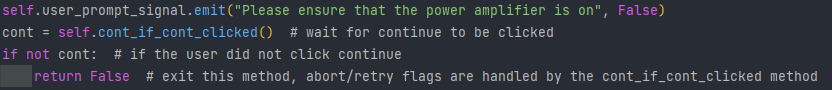
* The returned Boolean is used when this function may be called as a subroutine. The parent routine should also abort early if the subroutine returns false. Put a return True statement at the bottom of the function since this should be the default if everything runs properly.
* Between unpacking the parameters and the default return true statement, implement the script action. Feel free to call other script actions as subroutines.  This usually involves creating and loading a new parameter dictionary to pass to the other script action. Look at the method to see what keys this dictionary requires.
* When calling other script actions as subroutines, remember to handle the continue Boolean returned by the subroutine like so:



* Check for hardware errors and handle them by logging an error message to both the hardware log and scriptlog, and calling sequence\_pass\_fail, with action\_type = ‘Interrupt action’



* To prompt the user for action, emit the user\_prompt\_signal with a string containing the message for the user, and a Boolean indicating whether to restrict the continue button to engineers and administrators. Then handle the case where the user did not click continue as shown here:



* To give the Manager the ability to call this function while parsing a script, add an entry to this else/if tree in the run\_script\_step method, so the manager calls the new function when it appears in a script.

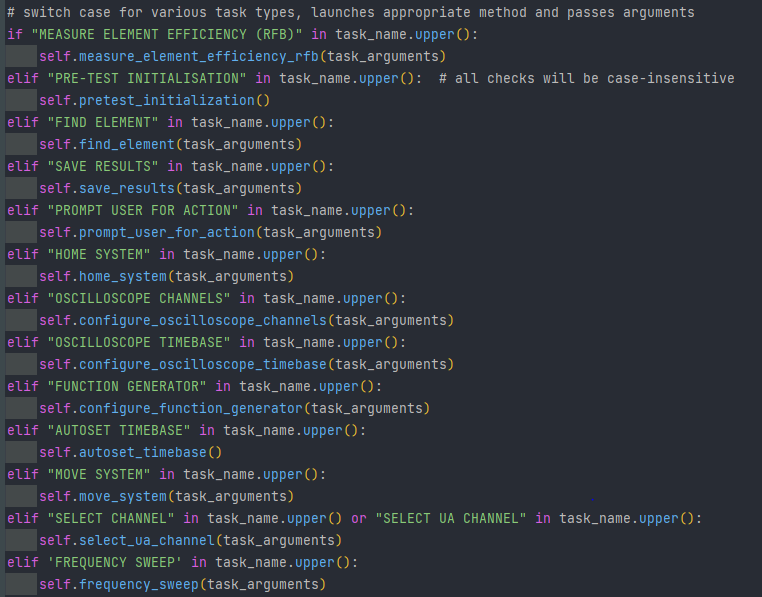
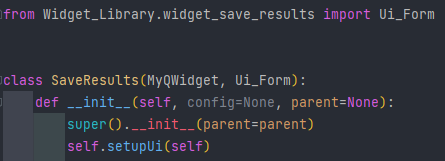


Figure - note: using \_\_\_.upper() in \_\_\_.upper() instead of == makes the condition less picky.

After developing the function, the next step is to add it to the script editor.

* To do this, create  a widget in QTDesigner with fields for all the desired parameters, using the appropriate input widget for the variable in question.
* This menu should have a maximum size of about 635 x 565 pixels (set the minimum and maximum width and heights to these values in QTDesigner).
* Change the names of these input widgets to descriptive names using snake case such as “element\_number\_spinbox”.
* Dump the .ui file to a .py file with the same name in the Widget\_Library folder. See QTDesigner section for more details.
* Next, copy the ui\_script\_edit\_template file in ui\_elements/script\_editor\_menus, renaming it according to the name of the script action, importing Ui\_Form from the .py file created with QTDesigner.
* Add it to the class header and make sure to call super().\_\_init\_\_ and self.setupUi() like so: This is how the class inherits the layout created by QTDesigner.



* From here, implement the orderedDict\_to\_ui and ui\_to\_orderedDict methods, making sure to keep the dictionary keys of each variable the same as those unpacked in the function in manager (case sensitive).
* Use the example code and comments in ui\_script\_edit\_template and refer to other files in ui\_elements\script\_editor\_menus for more guidance.

Add the name of the new task to the script editor widget

* Open the widget\_script\_editor file in QTDesigner
* Click on the Task Type dropdown and add an entry with the name of the new script action
* Save the .ui file and dump it to a .py file with the same name in the same location. See QTDesigner section for more details.

Finally, add this new menu to the *ui\_elements\tabs\script\_editor\_menus* file.

* Import it at the top of the file like so:



* Add it to the show\_task\_type\_widget method like so:



Figure 5 - note: using \_\_\_.upper() in \_\_\_.upper() instead of == makes the condition less picky.

* Now when you run the application, the new menu should appear, when you click “*Add to Script*”, it should appear on the left, and when expanded it should contain all the variables from the UI menu.
* When you save and run the script it should execute the corresponding method in Manager with all the correct variables. Test and debug as needed.

**Troubleshooting**

Debugging

If a bug or issue occurs when the program is running, one can usually look at the program’s *execution* *stack traceback* to solve the problem. The program is configured to not only output the exception to the command prompt (if ran from there) but to also show a dialog displaying the stack trace and our contact information for more assistance. The stack trace always has the line causing the issue on the bottom, with the preceding lines showing what called it and what called its caller, etc. Each line includes the file, line number, and method name in that order. At the very bottom, the specific error/exception will be shown and described. You can navigate to that part of the code, insert breakpoints, and run the program in debug mode to see what is happening step-by-step. You can also use print statements to see what is happening in the code, if various parts of it are being reached, if a code block is running as many times as it should, etc. The print(**colored**(… method from the “*termcolor”* library and Python’s “*pprint/pformat”* methods are especially helpful if using this approach.

You may also use PyCharm’s expression evaluator to see what the result of a code snip equals.

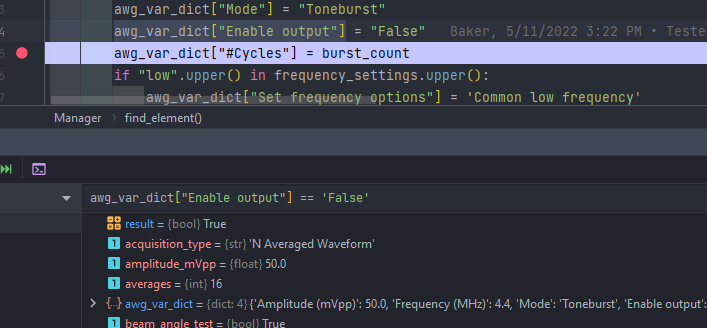
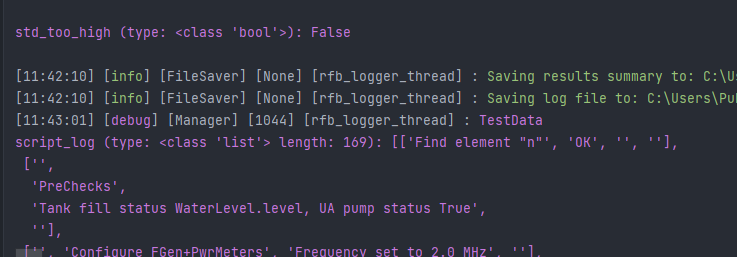


Figure - Enter an expression in the text box (blue oval) and press the enter key to get the output (orange oval)

There is an option in the configuration file under **Debugging** called “*print\_detailed\_verbose*”, if this value is true, you will see more detailed logs that appear as purple text in the PyCharm.



What you see in the screenshot above is the result of running an object’s \_\_repr\_\_ function. In the code, this is how such data is printed.

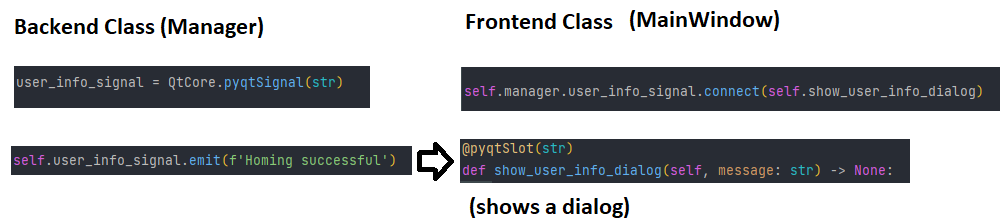
**Calling code:**



**Method in test\_data.py**



Key Concepts with Examples

Example 1: Triggering frontend methods from a backend class  
****  
Having a single UI thread is key to the framework of the graphics library (pyQt5), and programmers should avoid calling any Qt methods that change the appearance of the UI from any other thread, as doing so will very likely cause crashes. For this reason, one should only call methods that draw UI elements from the MainWindow class or a UI class, and refrain from triggering these methods with anything other than a pyqtSignal. For a minimal example of this technique in action, refer Examples\change\_ui\_from\_manager.

Example 2: Triggering backend methods from a frontend class

Signals allow events and information to be sent between frontend and backend classes safely. This can also be used the other way around to indirectly call backend methods from a frontend class. **However**, counterintuitively, **not all code running within the Manager Qthread** object (or an object instantiated therein) **runs in the manager thread.**

This is critical to understand because the purpose of delegating tasks to a background thread is to keep the UI thread responsive to user input. Even if user input is not required, long-running code anywhere in the UI thread will cause the application to “stop responding” and turn white, as any computer user has likely encountered at one point or another. Even if the application is still working behind the scenes the user will likely spring for the close button.

For short running tasks (1 second or less) this is not a big issue because the task will complete before the window “stops responding”. However, for long-running tasks this must be addressed in one of two ways.

The first way is to trigger the slot directly and repeatedly force UI updates within the long-running method. In the constructor of a backend class, retrieve the currently running QApplication to a class attribute for example:

And then within the long-running method (e.g., a loop) call self.app.processEvents()

This method works but is not recommended for anything longer than a handful of seconds because the code is still running in the UI thread, defeating the purpose of the background thread to some extent. There may be a slight advantage in responsiveness, however. This technique is used for the incremental move buttons in the Position tab.

For a minimal example refer to Examples\trigger\_small\_job from UI.

Example 3: Triggering long-running backend methods from a frontend class

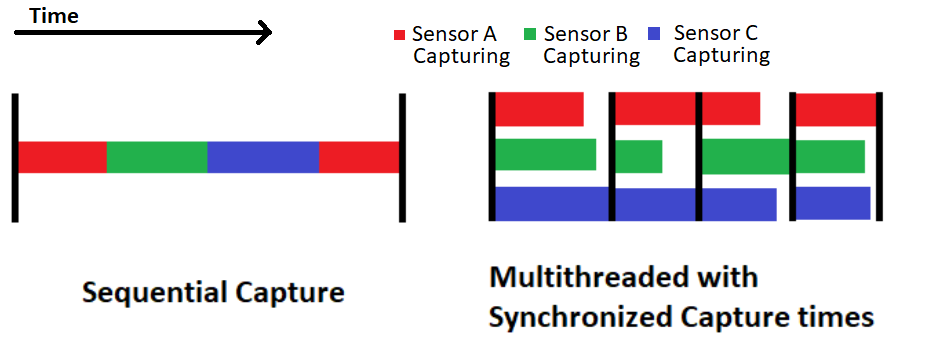
As mentioned in the previous example, code run directly in a slot connected to a UI class will run in the UI class, even if it is a backend class or a QThread object. It is possible to work around this by forcing UI updates, but in a long and complex method that is not ideal, and it will still bog down the UI thread to a noticeable extent. The code for long running methods such as, connecting hardware devices, running a scan or frequency sweep, and all script methods should run in the manager thread. Code will run in the manager if and only if it is called via the manager’s *run()* loop.  
  
In order to trigger code in the run loop, an if condition can be added to the run loop that calls the appropriate method when a certain condition is met. The recommended way to do this in the manager class is by adding a *pyqtSignal(str)* called *command\_signal* to a UI class, connecting it to the Manager’s *exec\_command* method, and using it to emit a string indicating the operation to be completed and any necessary parameters. This string will be stored in the manager’s *self.command* variable using a slot. However, the command will only be run once the manager’s run loop acts on that string using an else/if tree and calling the corresponding method. Therefore, the operation will run in the manager thread and will not bog down the UI.

Note that while the manager is running this code its core run loop will be on hold, and no command strings sent to it in the intervening time will be acted upon. The manager is “busy”. There is a signal in Manager called the *busy\_signal* used to communicate this to other classes and to the user.

Refer to ***Examples\trigger\_big\_job\_from\_ui*** for a minimal example

Example 4: Sensor Capture Multithreading:

The main advantage of multithreading is improving an application’s performance by allowing it to do more than one thing at once. The part of the RF Balance application in which fast performance is most important is the Measure\_element\_efficiency\_rfb method, in which readings from three different sensors (the forward power meter, reflected power meter, and RF balance) must be repeatedly captured in real-time at a high sampling rate (between 10 and 20 Hz). Some delays are unavoidable when  capturing a reading from a sensor, including the response time of the sensor itself, the performance of low-level API code, the speeds associated with a USB interface, etc. Therefore, capturing data from each sensor in sequence within a loop can be rather slow and result in a low capture rate. Furthermore, the time of  capture for the first sensor will be significantly earlier than the last. The solution to this problem is to utilize a thread for each sensor, so all three can capture at once. To coordinate the capture times of each sensor and assemble their data into lists, an additional thread is used.



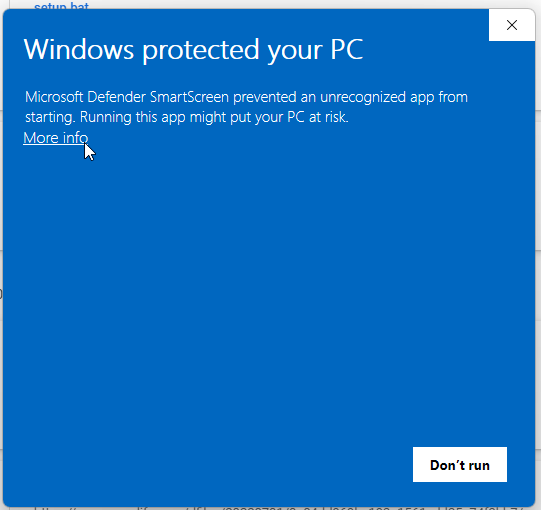
The result is that all three sensors initiate each capture at the same time, and as soon as all three are finished, they begin another capture. This method achieves nearly three times as many captures in the same amount of time, and all three datasets share the same time axis. The classes used for this are RFBDataLogger and SensorThread. The RFBDataLogger emits trigger\_capture\_signal, which is connected to trigger\_capture\_slot in each SensorThread. This flags the SensorThread to initiate a capture in their core event loop and emit it as a signal back to RFBDataLogger. The RFBDataLogger appends the data from each sensor to their respective list, and once it has received data from all 3 it initiates another capture.

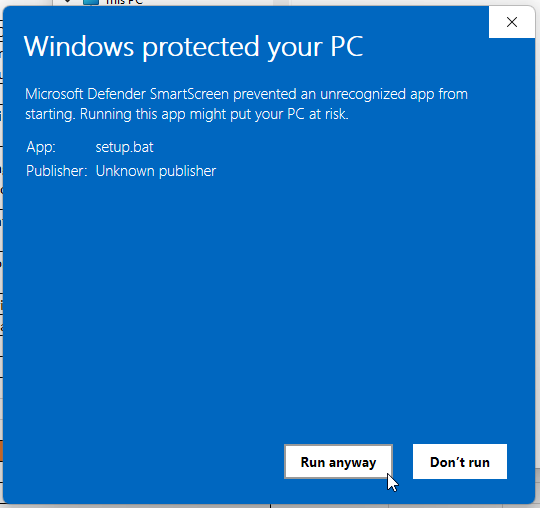
For a minimal example refer to ***Examples\simple\_multithreaded\_capture\_demo***

# Basic Usage

Installation

You will be provided with a file called “*Setup.bat*”. This is a Batch file that interacts with Windows’ command prompt API. This file will not work with any operating system other than Windows. You may get a warning about launching this file from Microsoft Defender SmartScreen or some other antivirus program. Ignore this warning, as it is a false positive.



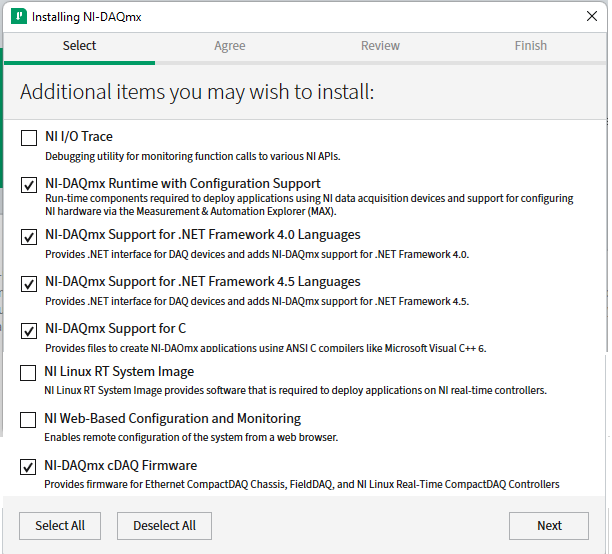


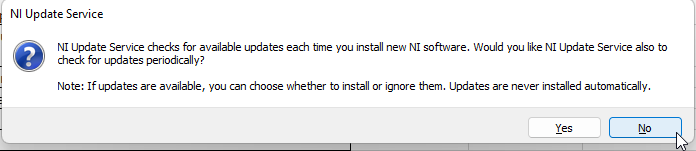
Upon launch, the script will ensure it has administrator privileges since it will be adding/modifying files on your system. The setup will then automatically create the following directories on your PC where *%username%* is the currently logged in user.



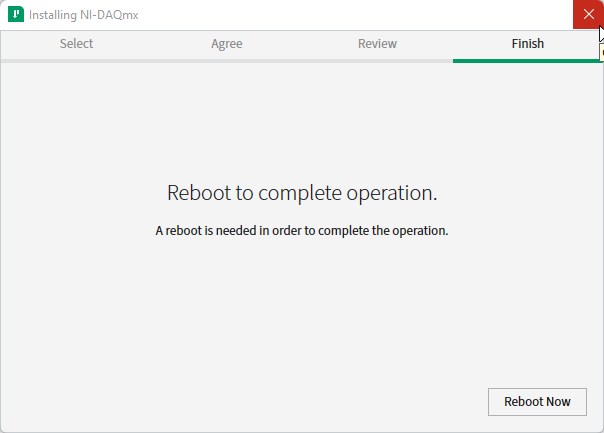
The script will then step you through a series of optional dependency installations. Starting with ***Git***, the setup will ask you if you want to install this software. If you select ‘Y’, the script will double check to see if it’s already installed and if not. If it is not installed, the script will automatically download the installer from the internet and launch it for you to complete. The command prompt may have special notes or instructions that display in a special color. This process will then repeat for ***gh*** (GitHub command line interface). *Gh* will step you through the authentication process, which can be achieved either through logging into an account with permissions in your browser, or via an access token that can be generated from a privileged account (<https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token>). Once authenticated, the script will download the repository to the ***\Documents\GitHub*** folder shown above. If the repo already exists, the user will have an option to download it again.

The script then checks if ***Conda*** (Python environment manager) is already installed . If not, the script will download and launch the installer for it. The script then repeats this process for ***Python 3.8.10*** (latest version of 3.8 with an installer for Windows). This process is also repeated for the ***National Instruments-VISA*** and ***NI-DAQmx*** ***drivers***. The installer may ask in the end if you would like to automatically check for updates. For stability purposes, we recommend **not** turning on this option as an update may break the code’s programing logic.





The installer will then prompt for a reboot, **do not reboot *unless* this is the final package you need to setup**, click the window exit button on the top right corner of the window prompt.



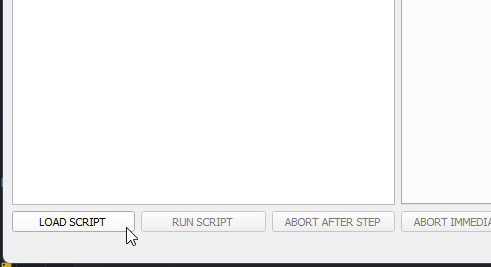
The script will then ask if you want to set up the Conda environment given the environment.yml file that was downloaded along with the repository. If ‘Y’ is selected, the environment will be created with a trace-level verbose to reassure the user the process is not stuck (it will take several minutes most likely, as several Python packages are being installed).

The script then asks if you want to install the Galil Python Libraries, if ‘Y’ is selected, it will download and launch the installer for it. The script will then ask if you want to apply gclib to the program’s local Python environment, if yes it selected, a Galil Python wrapper will be automatically launched and the *gclib.py* library file will be copied over to the environment files.

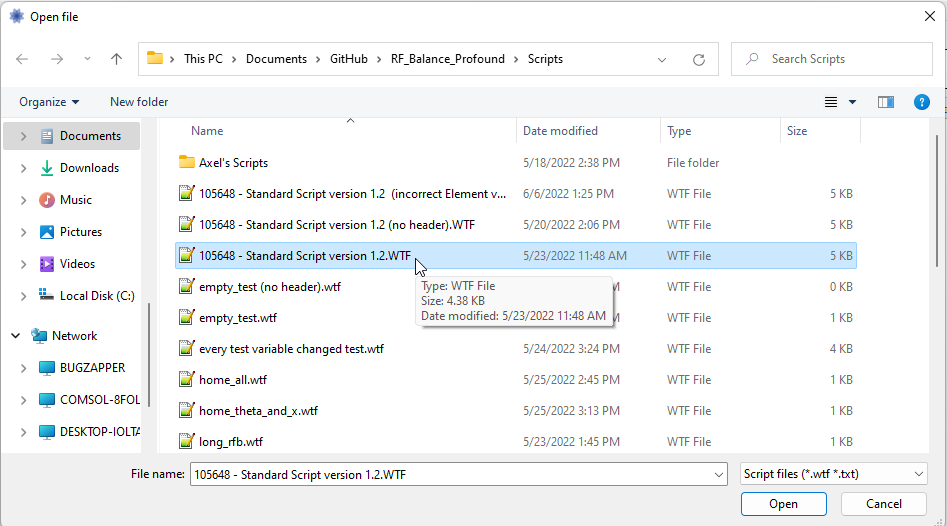
The script will then give an option to copy the dynamic link library for the power meter to Windows’ SysWOW64 folder, which is needed for functional interaction with the power meter. Then, the user can optionally download and install PyCharm community, and if the user wants to create a desktop shortcut for the newly configured Python application. The user will then be told that they should restart their PC for all changes to take effect.

## Script Manipulation

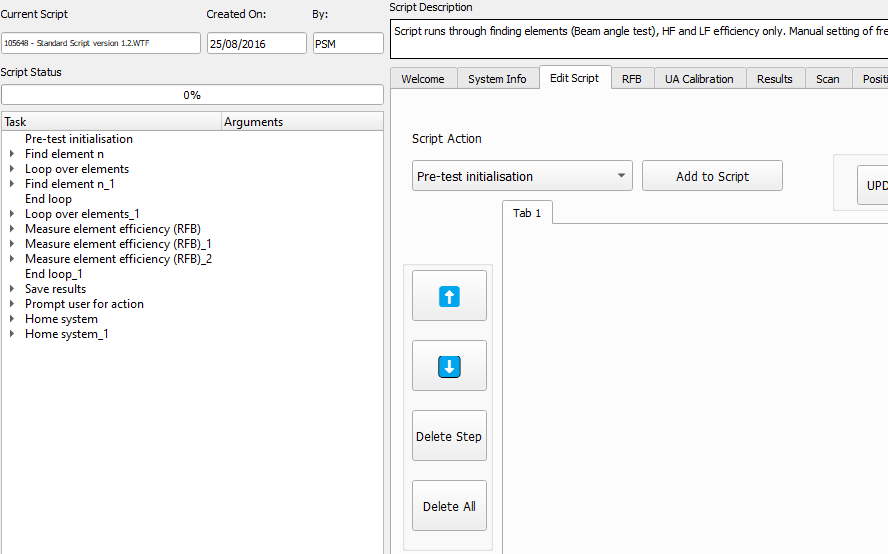
Select the “LOAD SCRIPT” button on the lower left corner of the main window.



Select the script you wish to open and either double click it, press enter, or select the “Open” button. The browser window will automatically filter out any non-script files (\*.wtf or \*.txt)



The description box will flash blue once if there’s a description for the script file and the tab will automatically switch to the “Edit Script” tab. The visualizer will display all the steps specified in the script in an expandable, user-friendly view. The script name, creation date, and author will also be updated if provided in the file.

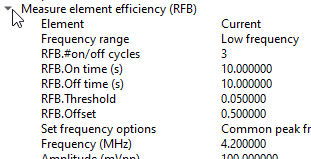


The “RUN SCRIPT” button on the lower left corner should now be pressable, to **begin** the script test, press it once you are ready to do so. The opened script is now **editable** as well.

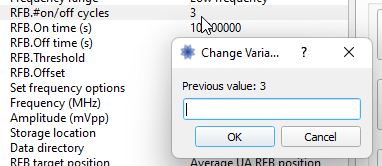
Editing/Creating Script

### Change Step Variable Value When a script is loaded, you can expand a script step to see its parameters.





You may then click on a variable value to change it via a popup box.

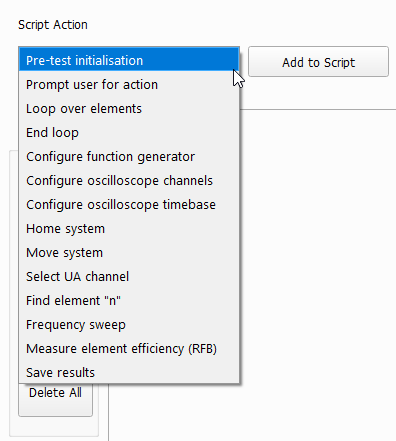


Enter the new value (I chose to input 2 for the new value) and press enter or the “OK” button. You should now see the updated value in the left-hand visualizer and the “RUN SCRIPT” button should now be a disabled, red button with “SAVE BEFORE RUNNING” as the text.

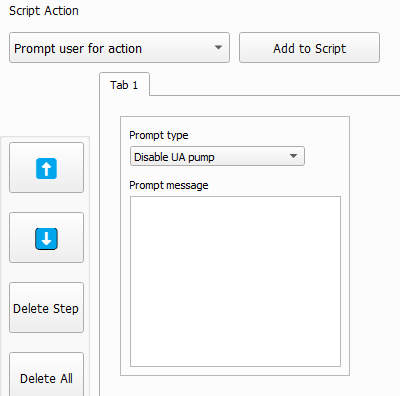


### Add Step

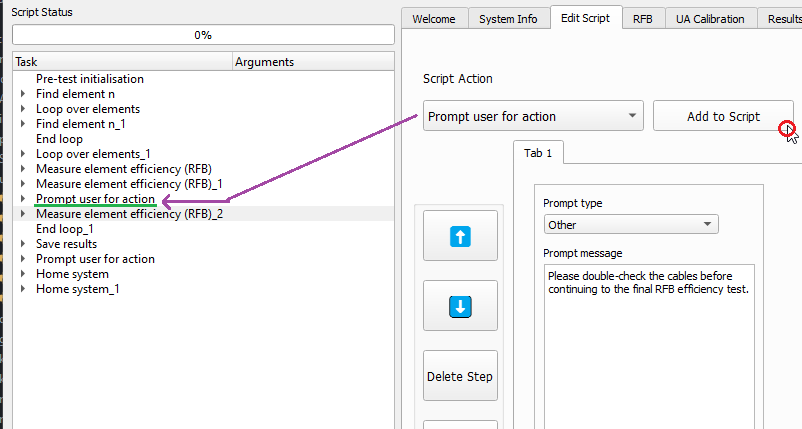
To add a script step, in the “Edit Script” tab, select the script action via the drop down on the upper left corner of the tab area, it starts at “Pre-test initialization” by default.



All actions except for “Pre-test Initialisation” and “End loop” will have GUI options you can set before adding the step. For example, let’s prompt the operator to “Check the cables” before the last measure RFB step in our script. Select the “Prompt user for action” script action from the dropdown shown above. You should now see this in the main window.

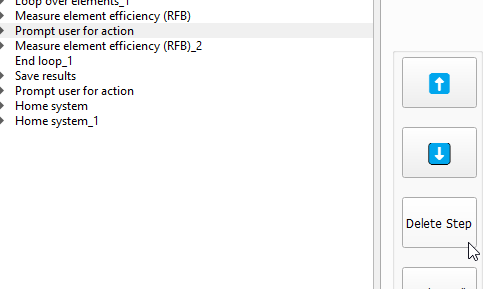
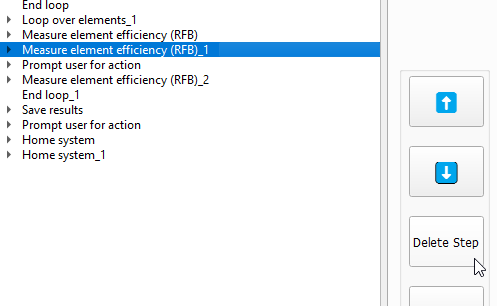


Change the prompt type to “Other” and type in the message box conveying what you want the operator to do. Then press the “Add to Script” button to the right of the script action combo box. Note: steps are added above the currently selected item in the visualizer. So, in our case, we need to highlight the “Measure element efficiency (RFB)\_2” before pressing the “Add to Script” button.

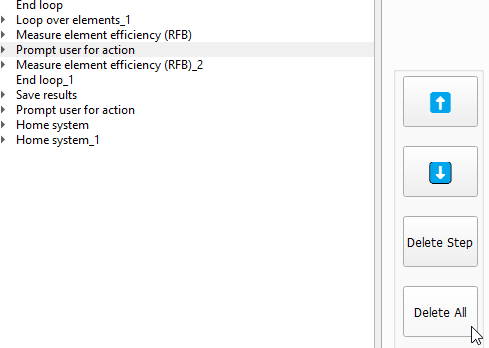
****

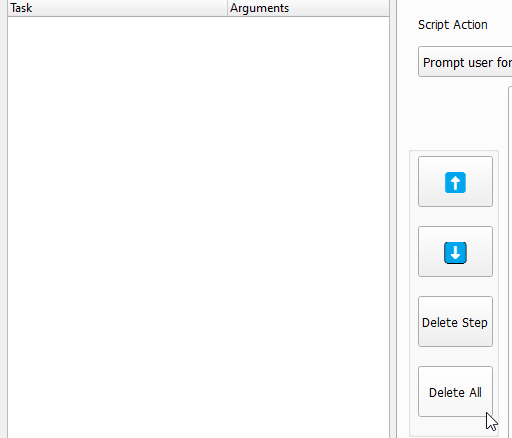
### Deleting Step(s)

You may either delete a single step at a time or wipe all steps. To do delete a single step, simply select the step on the left-hand visualizer, and press the “Delete Step” button. We can delete the second measure RFB efficiency test as an example.



Pressing the “Delete All” button will delete all steps immediately. No selection needed.

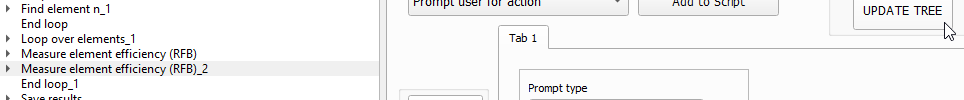


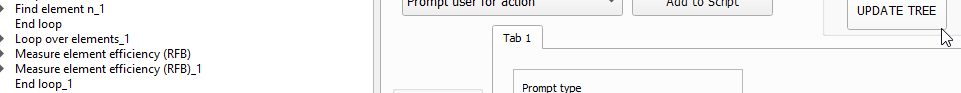


You will also need to save the script either as a new one or overwriting the old one to run it.

### Updating Tree

You may have noticed that when we deleted the second RFB test, the suffix distinguisher did not update to reflect the new script steps (“(RFB)\_2” should be “(RFB)\_1” after the deletion). To address this, press the “UPDATE TREE” button on the upper-right side of the tab screen.

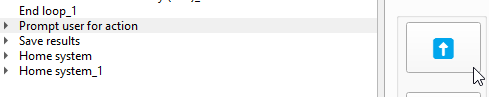




### Moving Steps

To move steps up or down, simply select the step in the visualizer and select the up or down button above the delete step buttons.

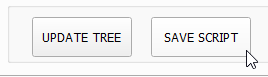


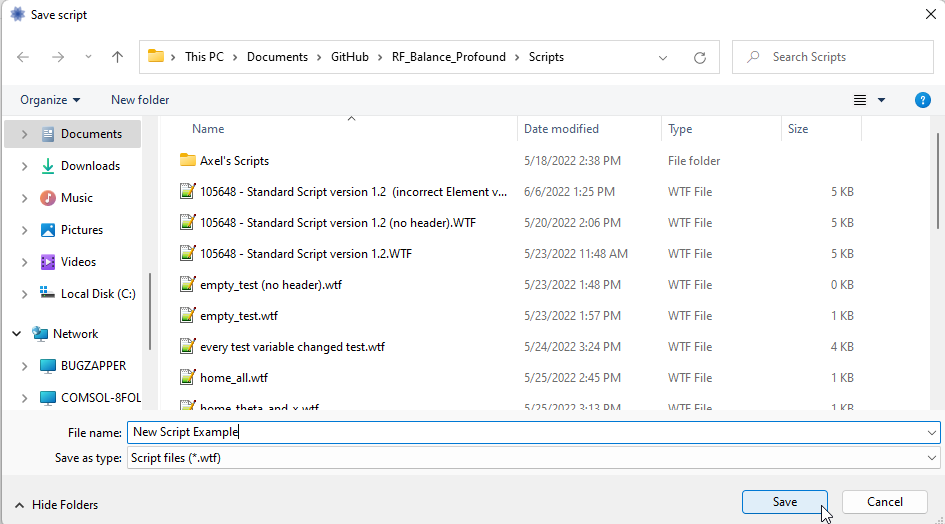


Unlike the previous operation, this action will automatically rename/update the steps in the tree to reflect order. (e.g., moving the second RFB test above the first one will automatically change their suffix distinguishers while keeping their distinct parameters).

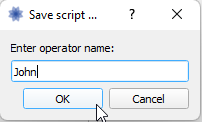
### Saving the Edited/New Script

Simply click the “SAVE SCRIPT” button in the upper-left hand corner to launch a file browser box. Either select the previously selected script to overwrite its previous steps or make a new file.

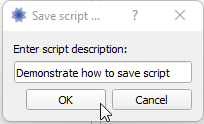




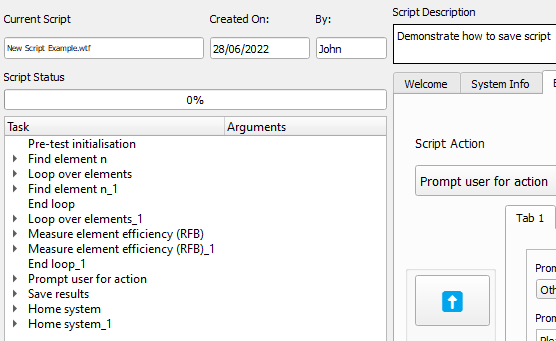
Enter the desire script author’s name **(can be blank)**



Then enter a description for the script **(can be blank)**

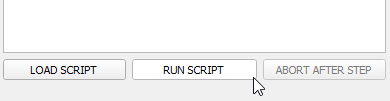


You should now see the fields in the main window have updated to reflect the script name, creating date, author name, and description; you may now run the script via the “RUN SCRIPT” button.

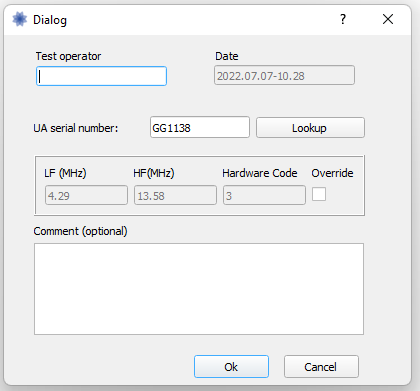
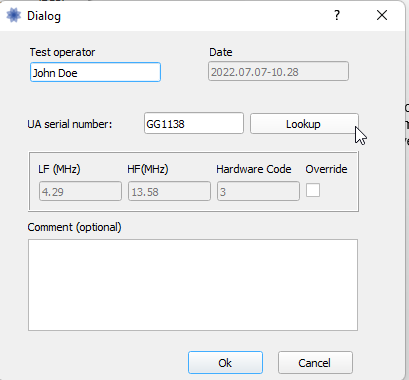
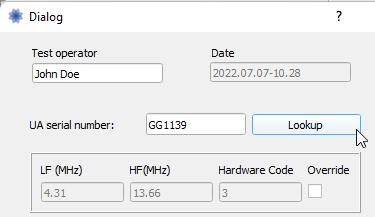
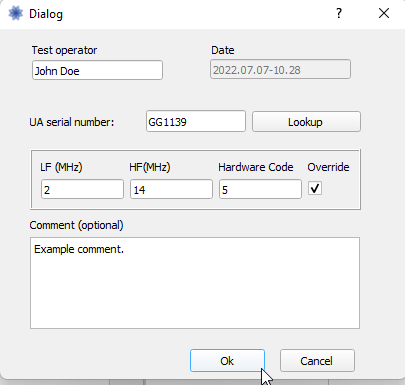


## Running a Script

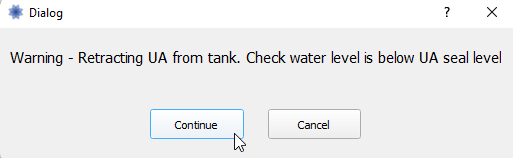
Once you have a valid script loaded, the program’s “RUN” button should be available for you to click. Go ahead and do so when you are ready to launch the test script.



If your script has “Pretest Initialisation” as the first script action (which it normally should), you will be greeted with this dialog prompt.

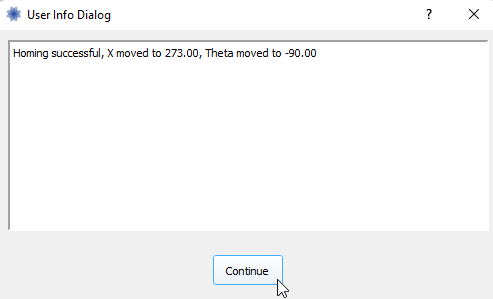
   

You may enter the UA’s serial number and have the program look through a list of presets matching the serial number once you click the “Lookup” button. You may also check the “Override” checkbox and manually fill out the frequency parameters and hardware code. You may also leave an optional comment, click the “Ok” when done.

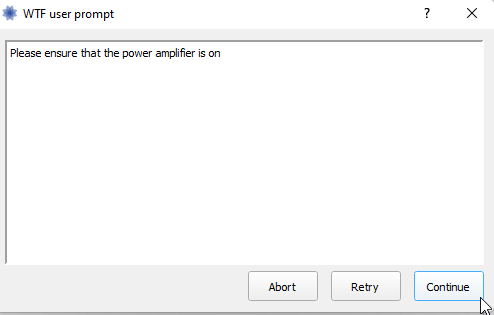


You might then be greeted with a warning prompt to ensure the water level is below the UA seal level, there currently is no automated way to check, so if you see this message, follow it and press continue when confirmed.

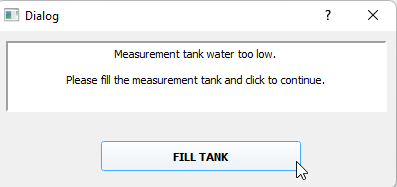
You may also see homing prompts like this.



There is currently no way to automatically check if the power amplifier is on, so the pretest Initialisation will manually ask you to check if it’s on, please do so and select continue.

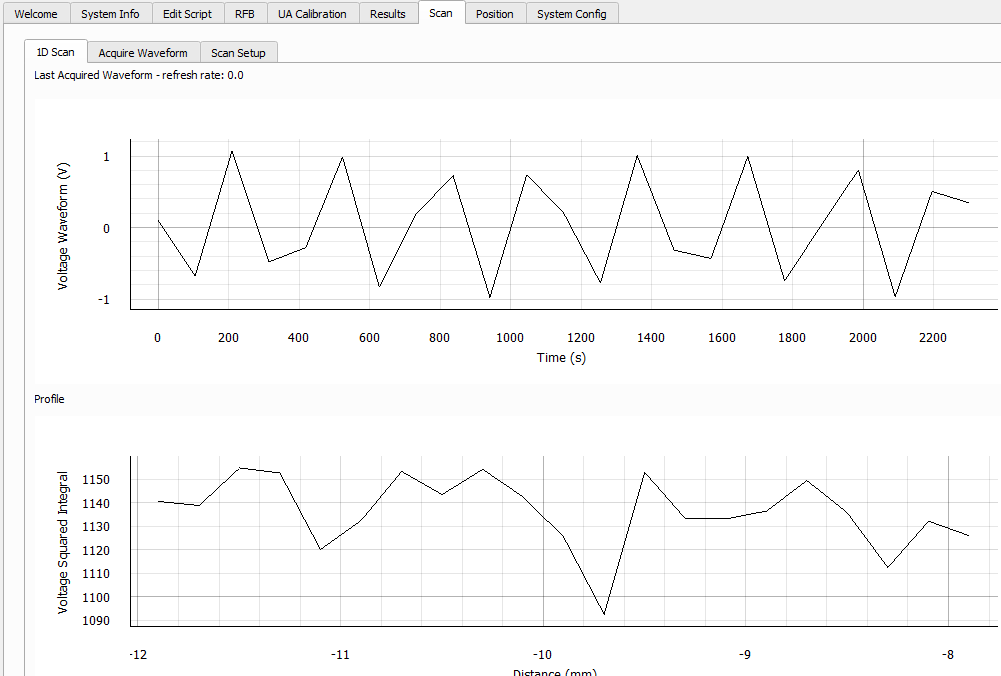


If the tank is not level, this prompt will come up and you will either be able to select “FILL TANK” or “DRAIN TANK” depending on the situation.

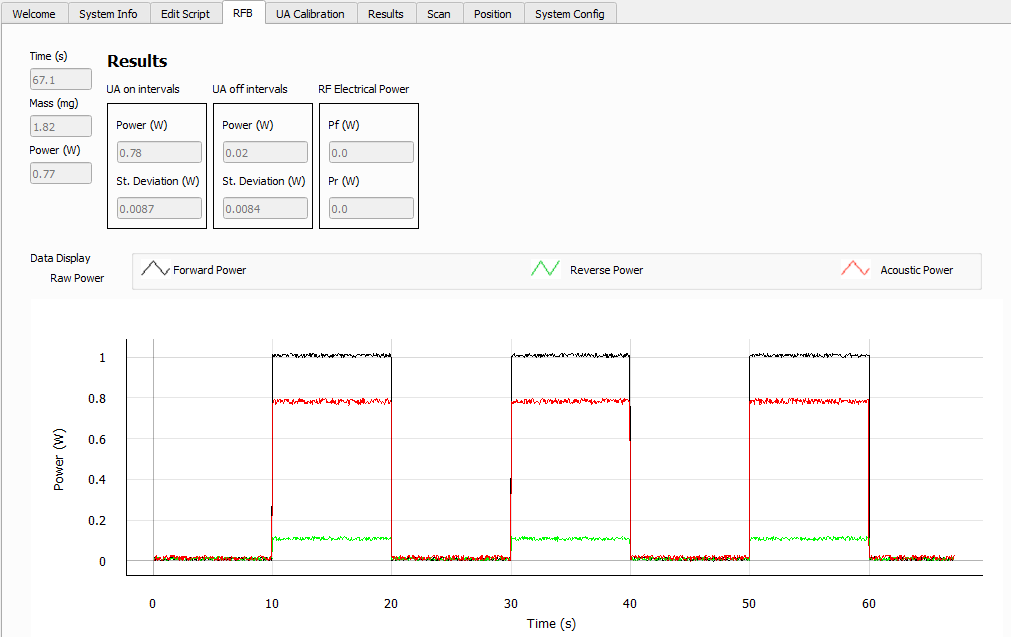


The filling/draining dialog will pop up and will automatically continue once the water in the tank is level.

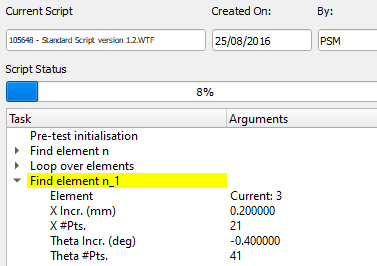
The find element action will automatically switch the main application to the scan tab’s 1D Scan section.



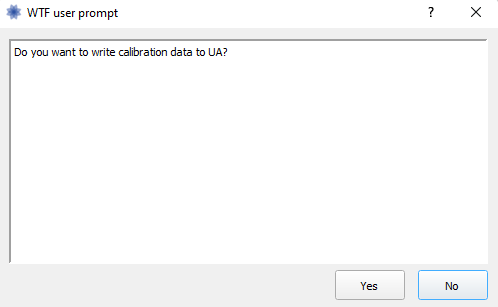
Measure element efficiency tasks will automatically switch the application over to the RFB tab.



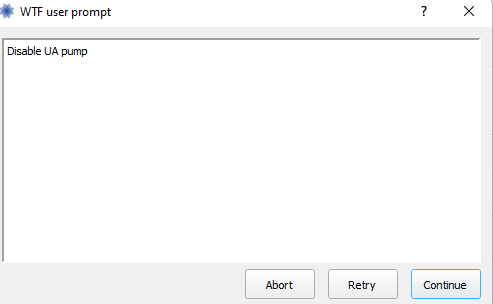
As the script progresses, you can see its current progress, step, and element in the left-hand info column. Current steps will automatically expand.



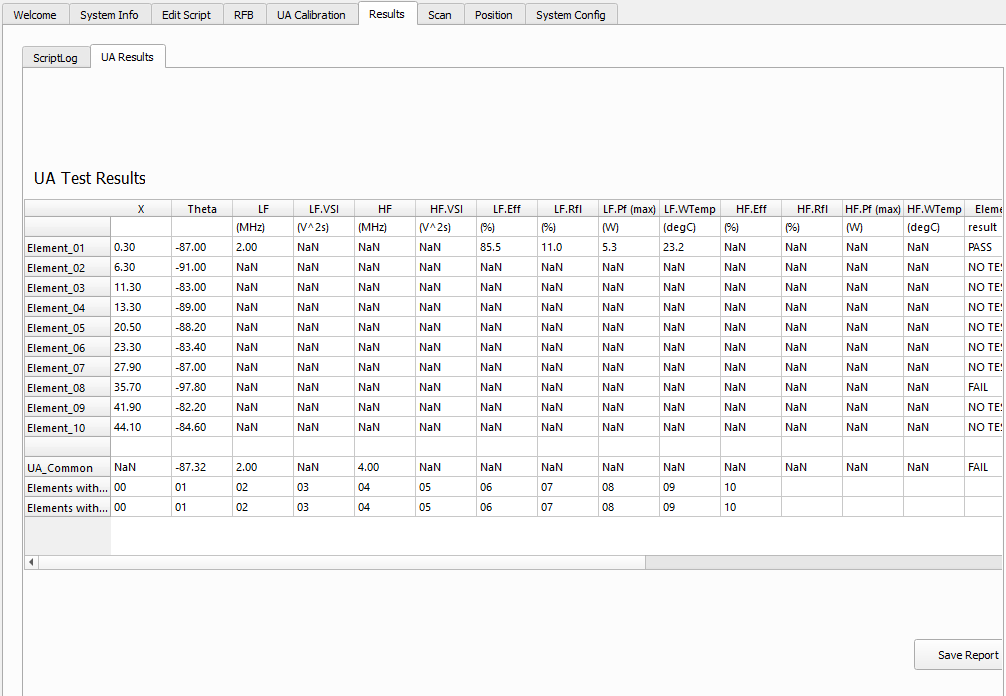
The save results task will launch this dialog if “PromptForCalWrite” is True in the task parameters, make your selection to continue.



A custom “Prompt user for action” prompt may be launched as a task, please follow its direction, and select continue when done.

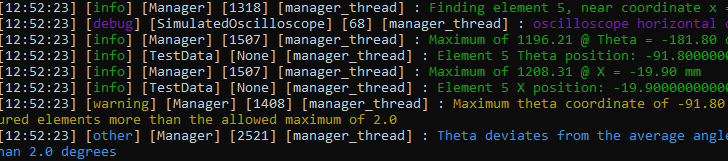


At the end of the tests, the application will automatically jump to the Result tab’s UA Results section.



You may save the results summary and script log manually by clicking the “Save Report” button in the bottom right corner. This may have already been done as a script task though, so keep that in mind. A confirmation dialog will popup if this button is pressed, answer it to proceed. The files will be saved to the **Paths\UA results root directory** and **Paths\Script log directory** value in either of the configuration files (local.yaml takes precedence over default.yaml).

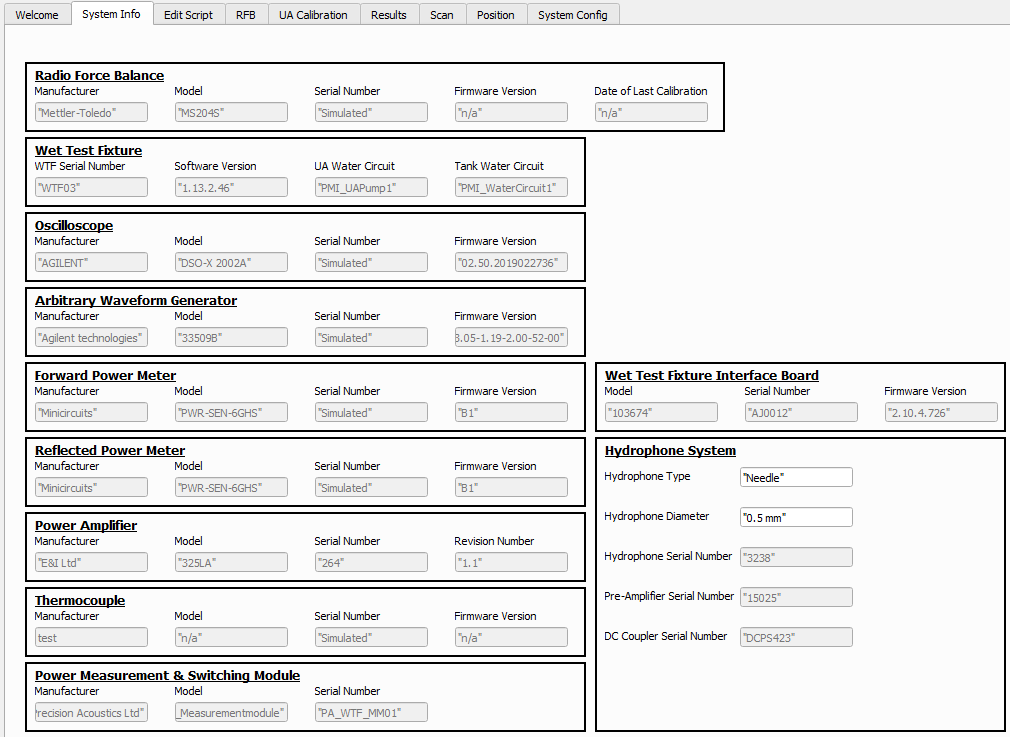
You can also observe the color-coded log when launched with the command prompt. Both Conda’s Python and Windows’ native Python will have the command prompt run in the background.



# Main Window Tabs

## System Info

As mentioned before in the “Organization of Files” section of this document. This tab reflects information regarding the hardware attached to this system in a more user-friendly manner than looking at the text file in an organized manner.

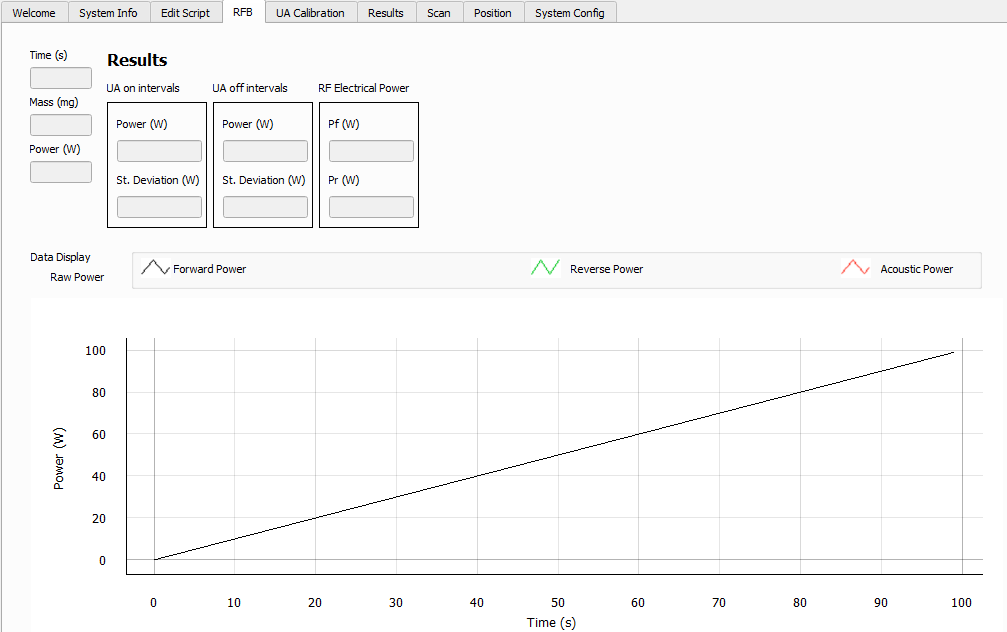


## Edit Script

See the “[Basic Usage\Script Manipulation](#_Script_Manipulation)” section of this document.

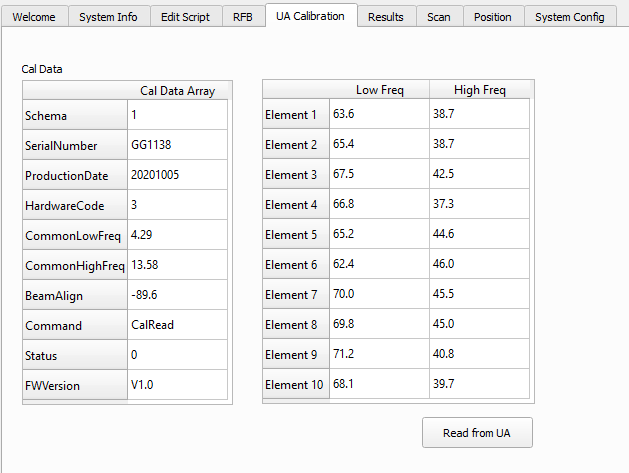
## RFB

This tab displays all readings pertaining to the element efficiency test. It emulates the previous software accurately and shows the power readings in real-time during the test on the graph with a legend for the colors. Numeric readings are also displayed above the graph in real-time.



## UA Calibration

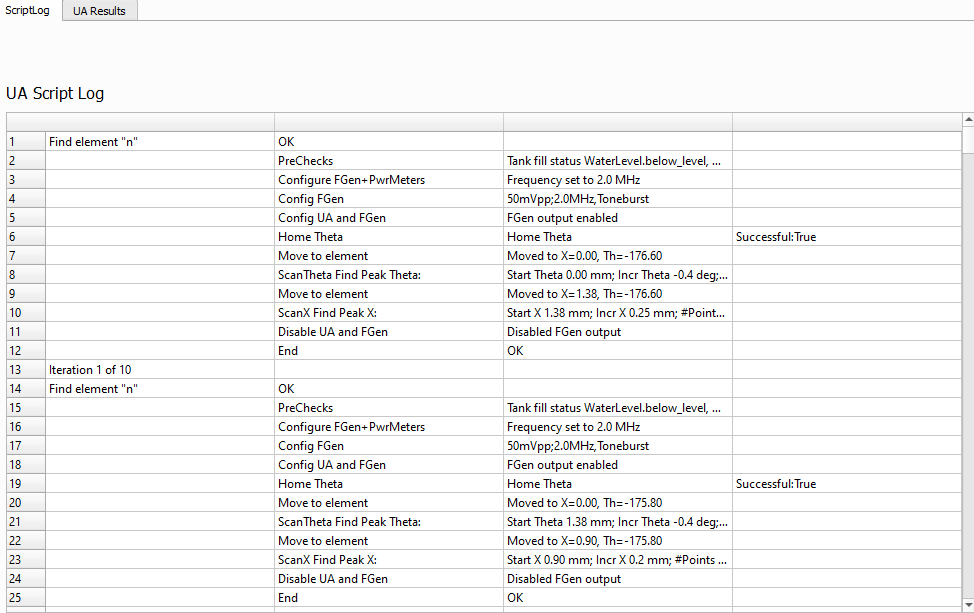
Initially, the tables will appear empty until a read command is issued to the UA launchpad via ethernet. This command can be issued during a script or manually by clicking the “Read from UA” button.



## Results

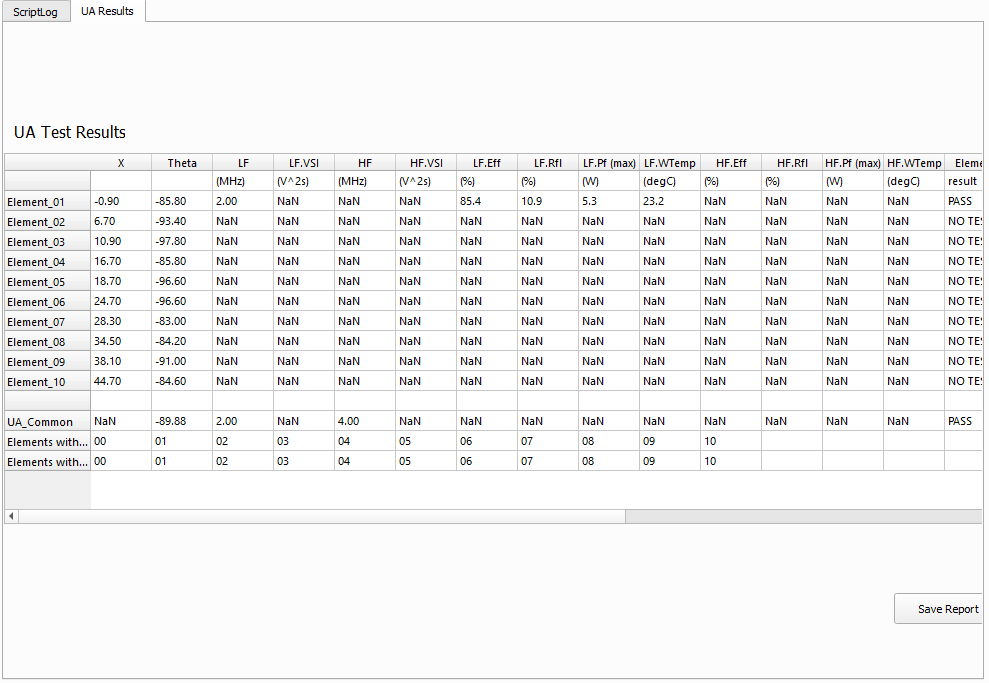
### ScriptLog

Here, you can see a detailed log of a test via a 4-column spread, just like the previous iteration of the software.



### UA Results

As previously mentioned, this tab reflects the position and efficiency data of all tested actuators in a table. In this screenshot, a custom script that only goes over the first element has been ran.



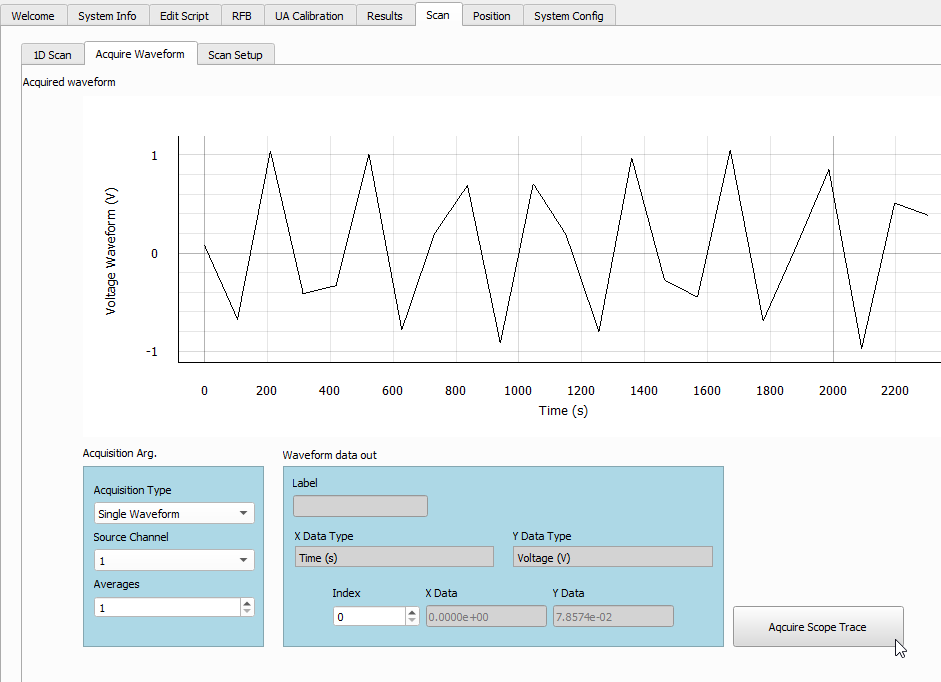
## Scan

### 1D Scan

As mentioned before, this tab is automatically switched to during the find element phase. The screenshot is not a realistic representation of what you will likely see. You will more likely see a smooth bell curve for both graphs, where the bottom graph’s peak represents the element’s position.

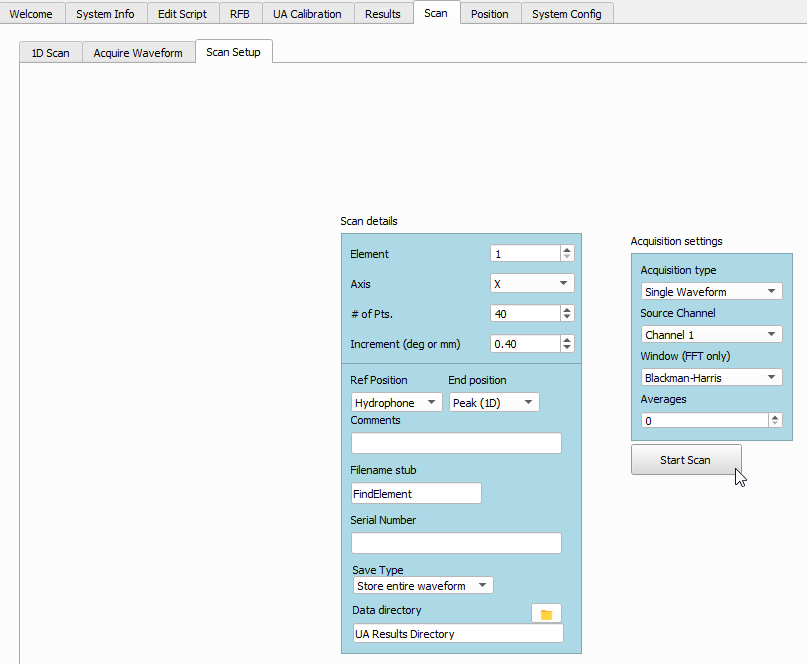
### Acquire Waveform

This tab shows the voltage over time graph like the previous tab, but also has data acquisition settings the user may specify. The user can manually launch a frame capture of the oscilloscope.



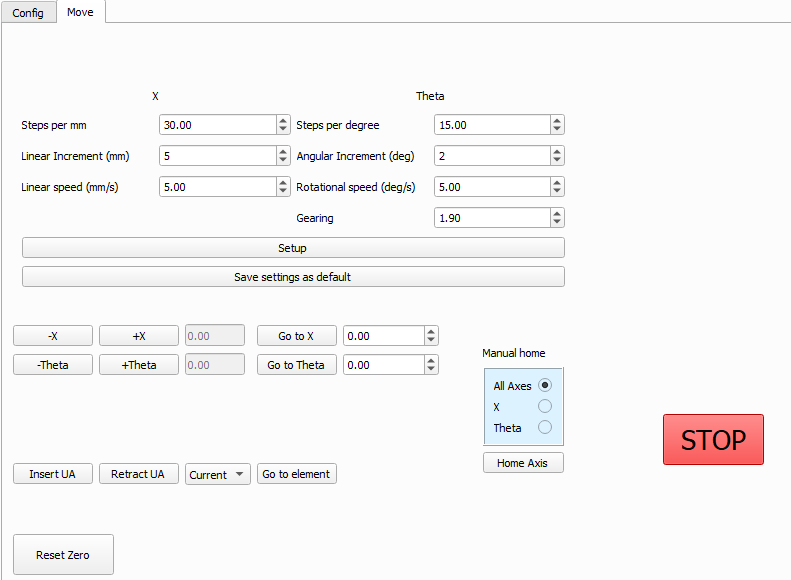
### Scan Setup

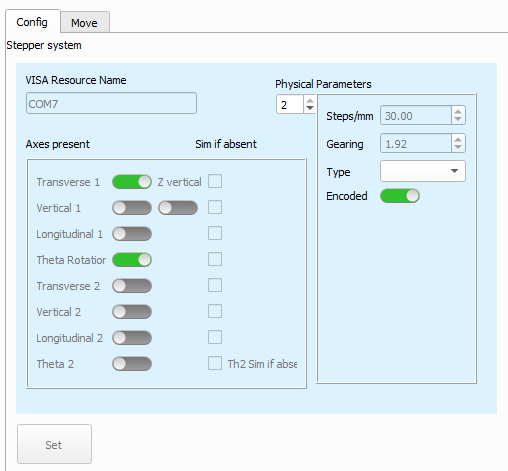
This tab allows an automated multi-frame capture of the oscilloscope with specific settings that the user may specify. Clicking the “Start Scan” button will jump the application to the 1D Scan tab so you may see the oscilloscope trace graphically.



## Position

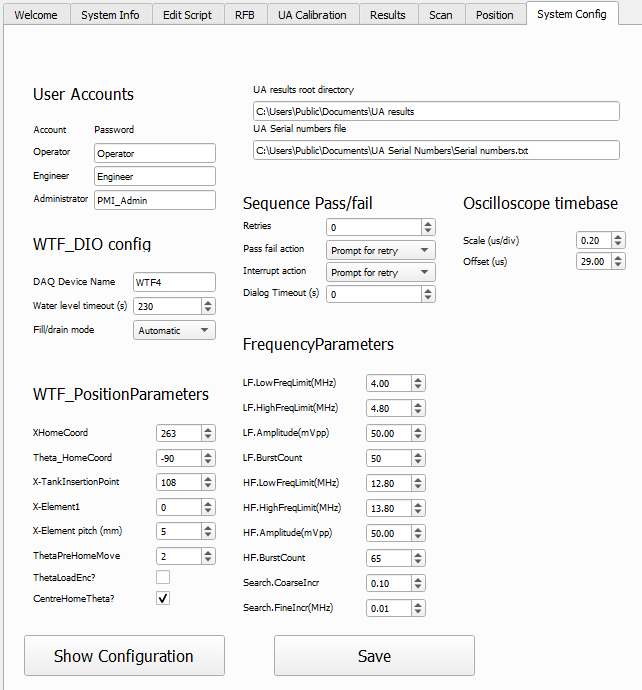
In the Position tab, you can manually move the motors in the “Move” sub-tab. You may also adjust the stepper system in the Position tab’s “Config” sub-tab.





## System Config Tab

There is a tab in the application called “System Config”. This tab allows the user to alter some settings in the default/config.yaml in a GUI fashion.



Clicking the “Show Configuration” button will launch the local.yaml file in your default text editor program to modify some settings that are not present in the System Config tab. Clicking the “Save” button will write the setting specifications in the GUI tab to the configuration text file.

# Glossary

## UI Classes

A class subclassing QWidget or QDialog that provides the layout and functionality for a component of the UI.  Generally, the UI classes in this project are in the ui\_elements folder, and instead of coding the layout and functionality in one file, they inherit their layout from a file created with QTDesigner. See also: QTDesigner

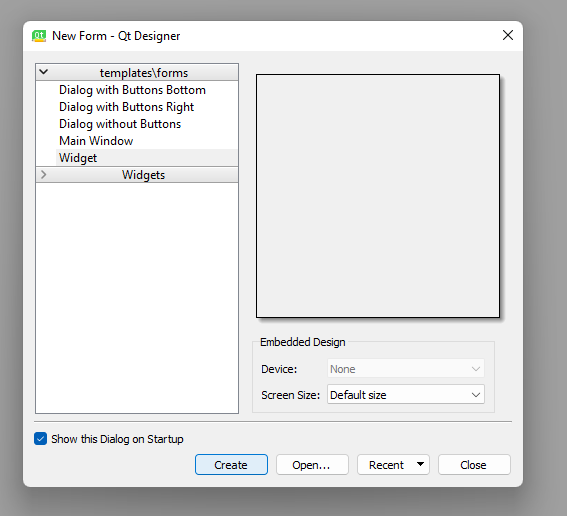
## Config Classes

Variables for any aspects of the application’s behavior designed to be configurable prior to runtime are in a file called default.yaml. However, this file should not be changed by the end user if they simply wish to modify the values of these parameters. To modify the values of these parameters, the user should make a copy of default.yaml and rename it local.yaml (if it does not already exist).

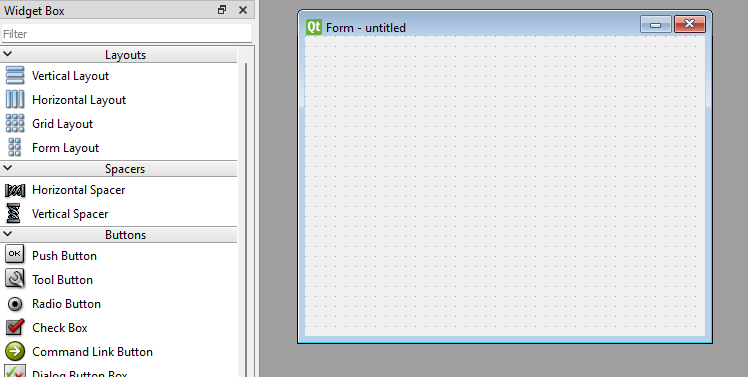
## Qt Designer (<https://build-system.fman.io/qt-designer-download>)

### Creating a New UI Element

To create a new UI element via Qt Designer, launch the application. You should be greeted with a new form dialog (if not, press Ctrl+N). You can select anything except for the “Main Window” option from templates.



You will then see an empty window that you can drag various elements into. Including layout elements, such as frames, horizontal/vertical layouts, grid layouts, form layouts, etc. These layouts can be nested, but it is generally recommended to use a grid layout if possible.



### Adding Widgets and Layouts to the UI Element

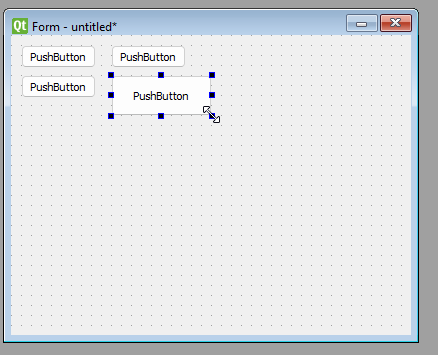
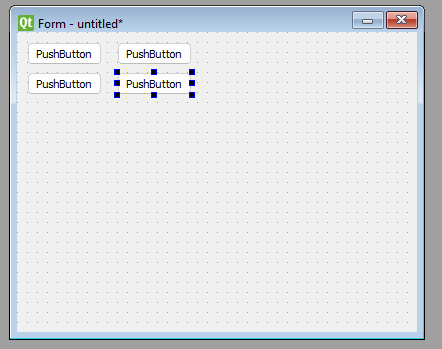
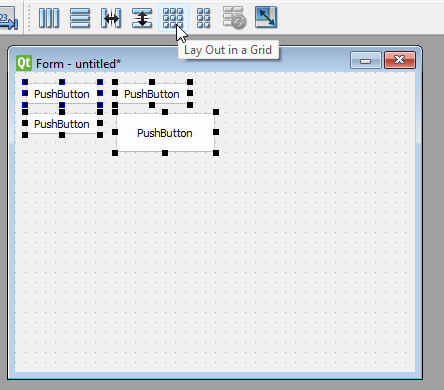
Widgets from the left-hand column can be simply dragged and dropped into the window. You can resize the element (with the occasional exception that it is in a layout) and change aspects and attributes of the widget on the right-hand column, along with renaming those widgets so they can be identified in python code.   
  


Figure 7 - Hold Ctrl and click on multiple objects to multi-select

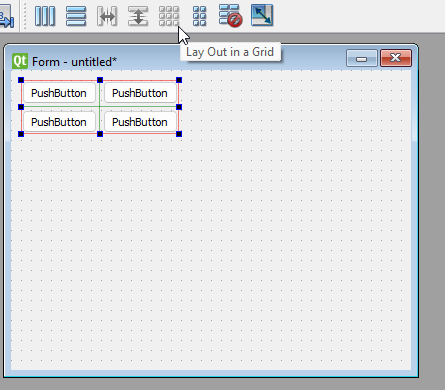
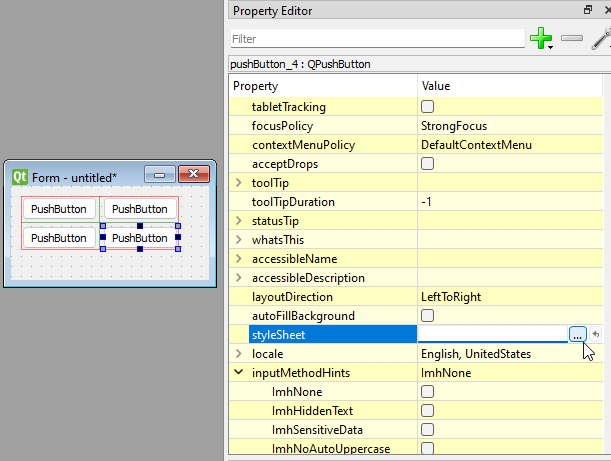
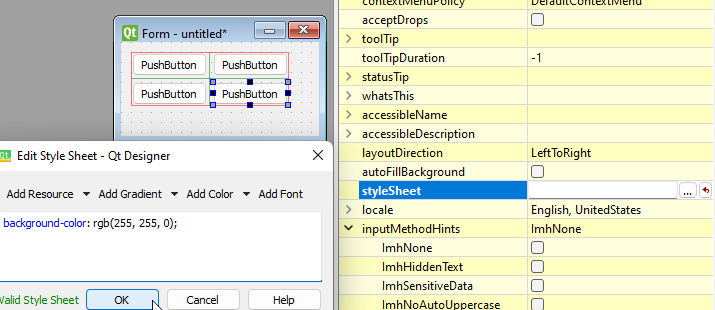


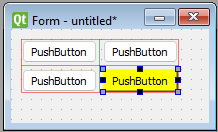
Figure 8 - Layouts can auto change object attributes, constrain them, or even outright lock them

Stylesheets can be used to control the color of various elements, such as frames, spin boxes, etc. Even if they are nested. You may, however, need to look up the syntax for stylesheets online.

### Manipulating Specific Widgets and their Attributes







### Adding the UI Element to the Application

Once you’re satisfied with the window, you may save it as a .ui file. For our project, we save all initial UI files in the Widget\_Library folder. Then, through PyCharm’s terminal tab or command prompt, change the directory to the Widget\_Library folder and run the command “*python -m PyQt5.uic.pyuic -x <filename.ui> -o <filename.py>*”. After that, you may create a new python file in the ui\_elements folder and appropriate subfolders. You **must** have the lines “*from Widget\_Library.<widget\_name> import Ui\_Form*” and “*from ui\_elements.my\_qwidget import MyQWidget*” near the top of the class. When declaring the class, extend the class with “Ui\_Form” from the originating .py file in Widget Library,  
and “MyQWidget” or “QDialog” if it’s a widget or a dialog respectively. In this new file, you may create signals, methods, and variables that pertain to that specific widget.

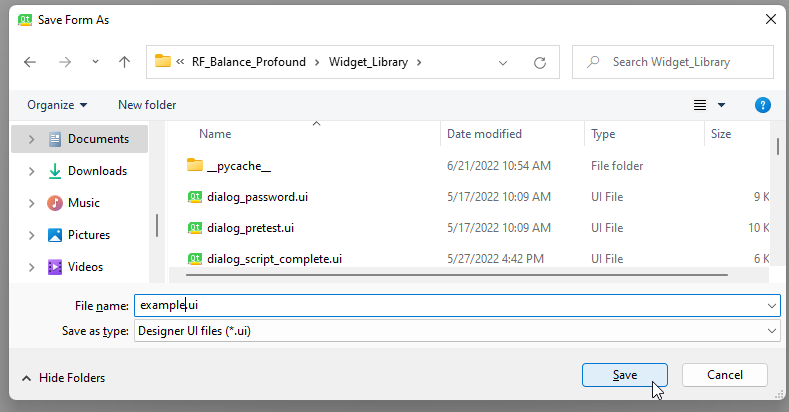


Figure 9 - Press Ctrl+S or using the toolbar select File → “Save As” or “Save”

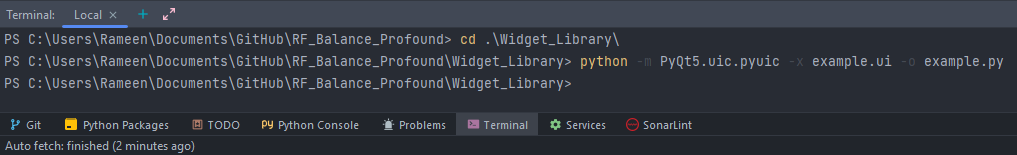
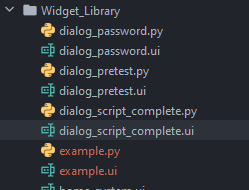
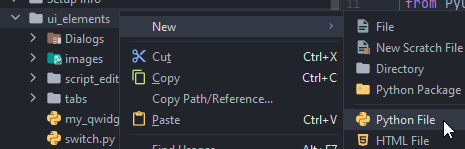
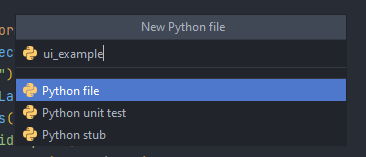
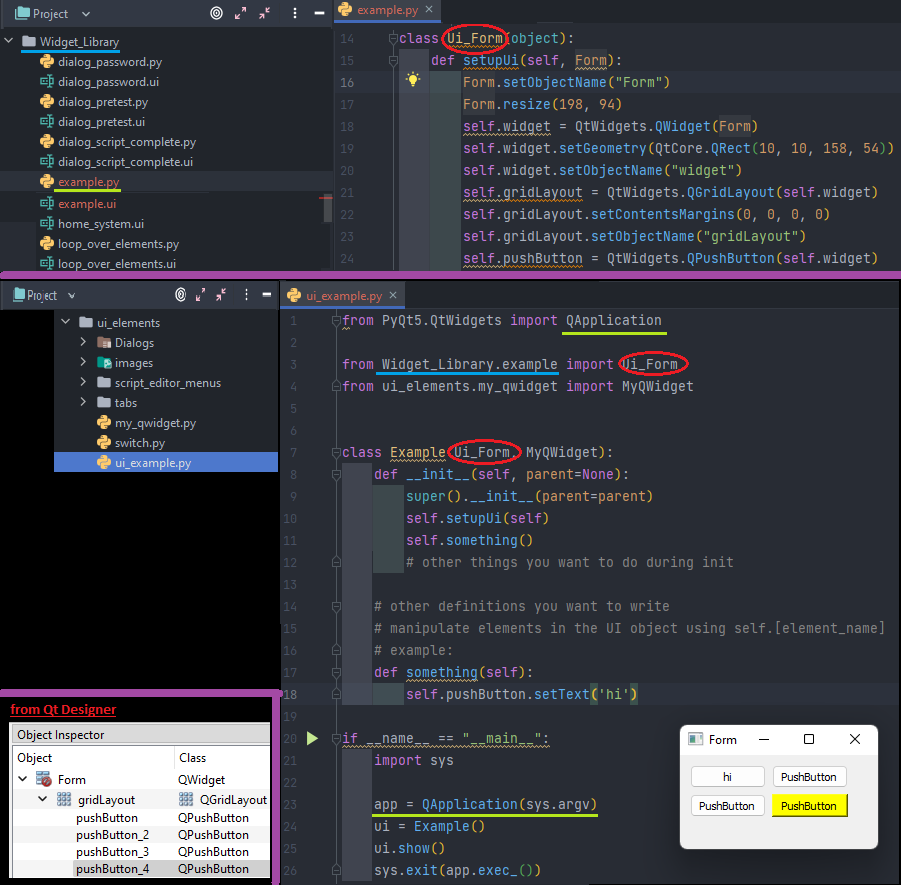
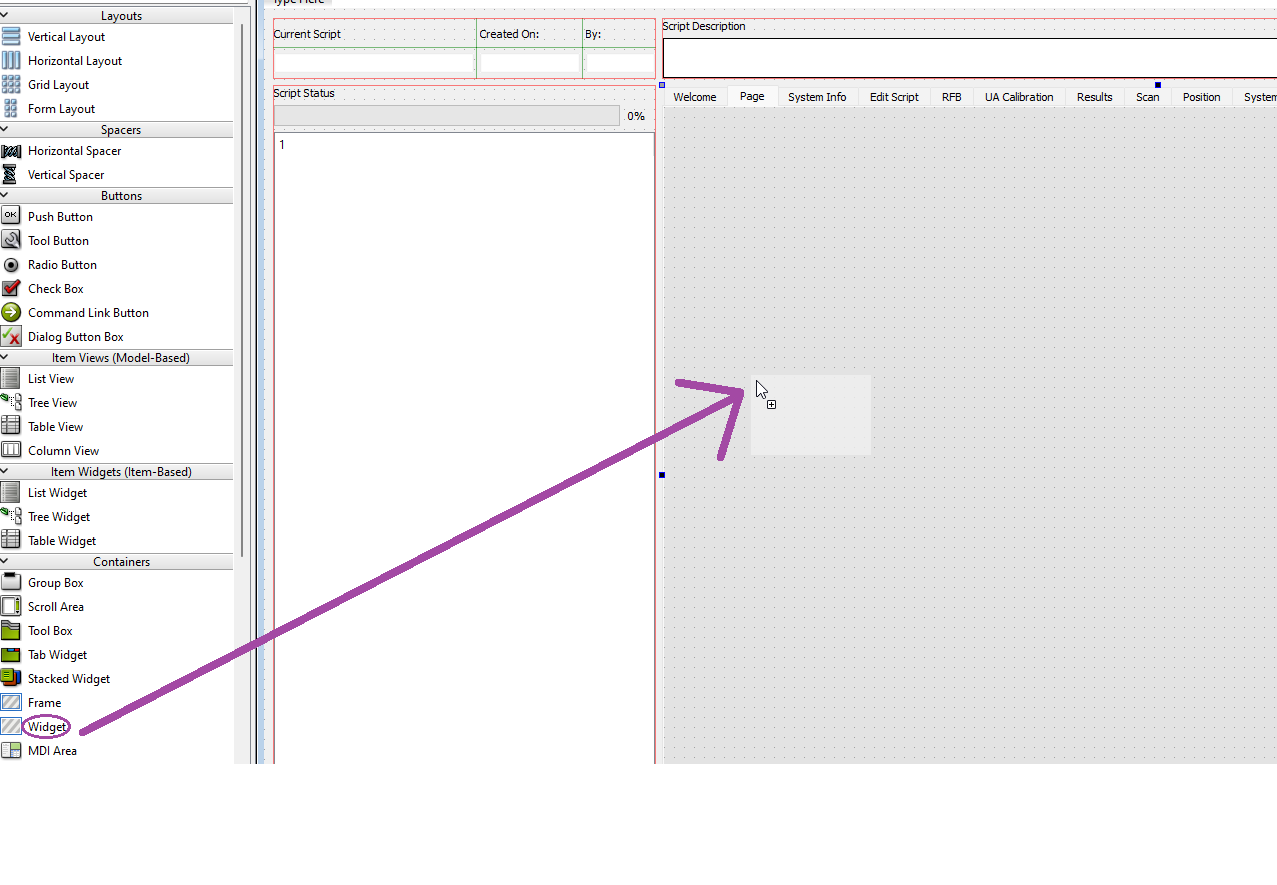
  
  
  
  


Figure 10 - Preview the UI by running the Python file. The main condition in the bottom will allow you to run this file.

### Nesting Widgets

After that, to further modularize the GUI aspect of the project. You may place a “widget” from the left-hand widget column in the general area of a window/section (e.g., The entire area of the position tab). You may then right-click the inside of the widget area and select the “Promote to…” option. The base class name is QWidget, the promoted class name is whatever you’d like to name it, though we encourage you to use similar naming conventions as ours. In the header file section, type in “*ui\_elements.<subfolders>.ui\_<widget\_name>*” where subfolders are optional. Check the global include button then click the “Add” button on the right-hand side. You will then see the item you just created underneath the QWidget section. Select the item, then click the “Promote” button in the bottom right corner. Once saved. Update the file that contains the newly promoted widget via the *pyuic* command.



You may resize the widget area to your liking. In our example, we will maximize it to take up the entire “Page” tab

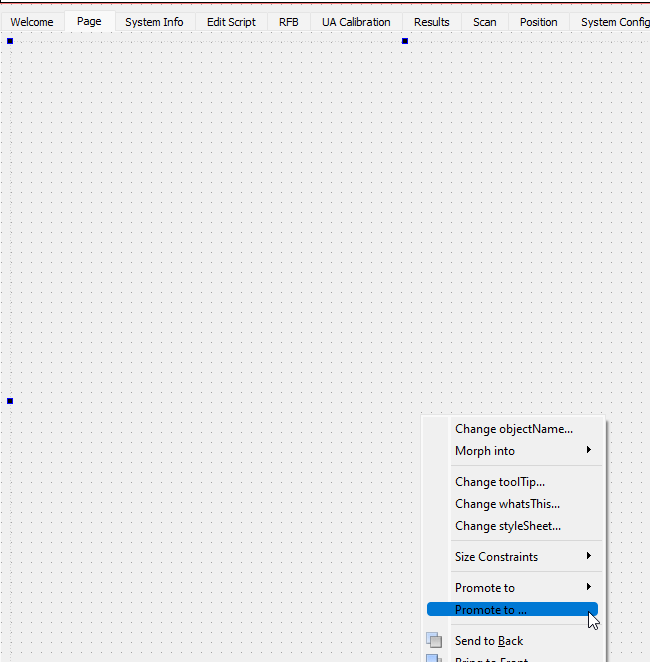


Figure 11 - Right click inside the widget area to see this context menu, select “Promote to…”

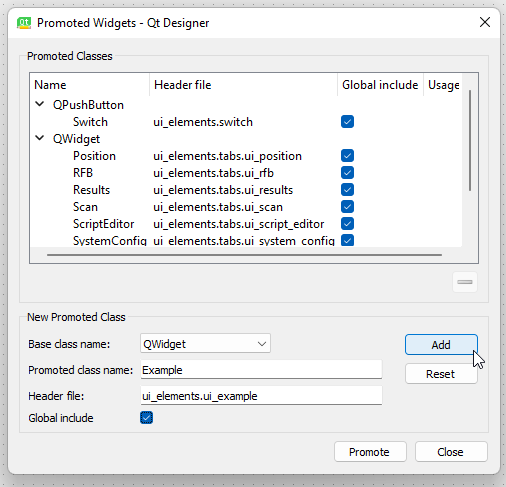


Figure 12 - The promoted class name should follow naming conventions. The header file should point to the ui\_[name].py file you created in the “ui\_elements” folder. Tick the global include checkbox and make sure the base class name is QWidget.  
  


Save the UI file via Ctrl+S (asterisk in window title should go away)



Figure 13 - This command must be ran every time you make a change to a UI element and want it reflected in your application

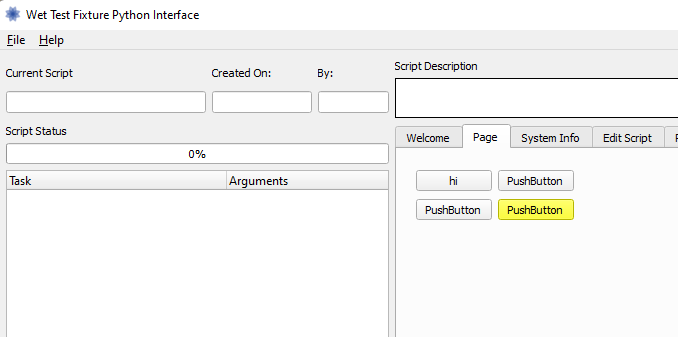
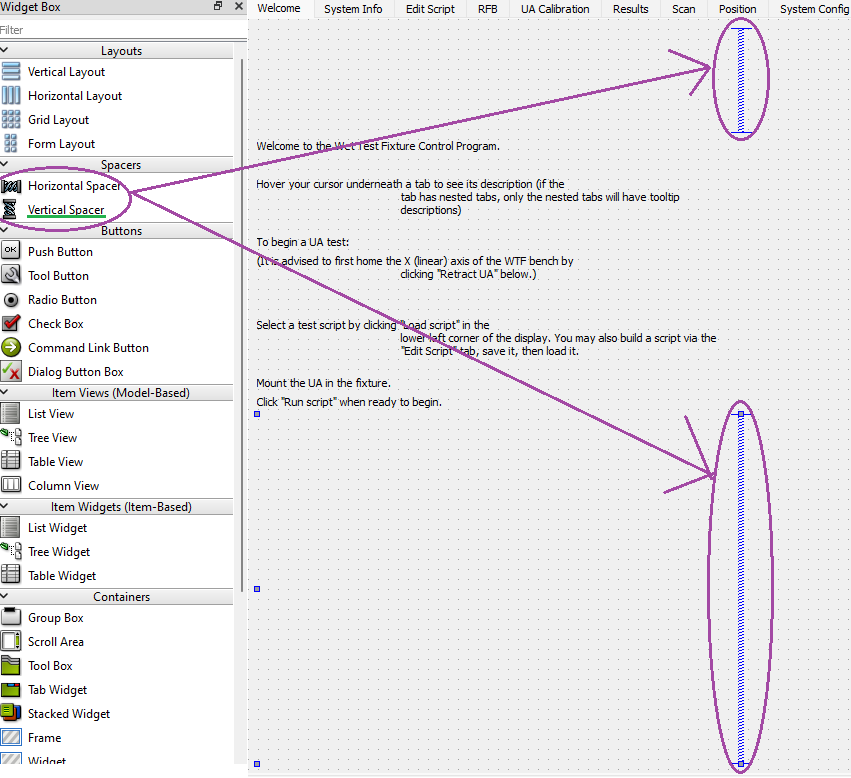


Figure 14 - You will not see this change if you press the Ctrl+R preview shortcut in Designer, you must formally compile and run the project to see the widgets placed in their promoted positions

Now, once you load the application, you will see the widget inside the desired location you placed it, even though they have separate python and UI files.

### Spacers

In addition to the above, you may also add spacers inside the window with appropriate window dimension and spacer attribute constraints to control how the window behaves when resized.  


### Conclusion

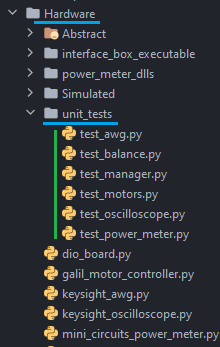
Element properties and methods can be altered and called respectively within python classes that import the widget, this is how the main window and various other GUI elements can change during runtime. We recommend PyCharm since it automatically detects an item’s attributes and methods.

## Unit Test

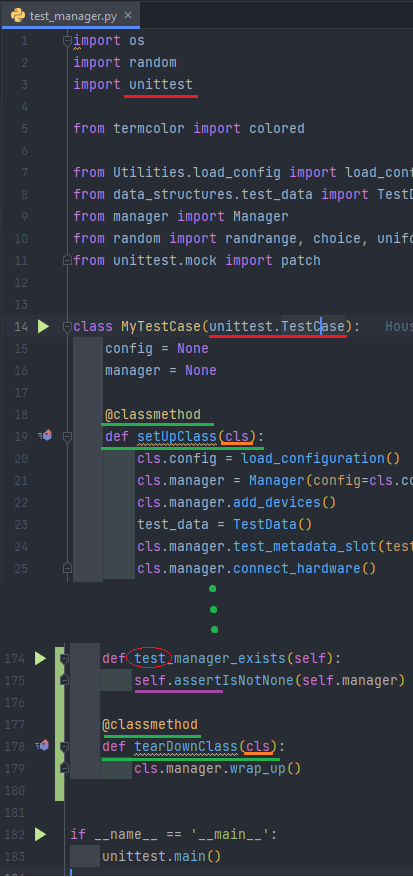
For this project, we placed the unit test classes in Root\_Project/Hardware/unit\_tests/. To create a unit test for a hardware, GUI, or software class, you must import unittest and the class that you are trying to test (e.g., *from manager import Manager*, or *from Hardware.mt\_balance import MT\_Balance*). Inside the class declaration for the unit test, extend it with “*unittest.TestCase*”. There are two methods every test class needs, a “setUpClass(cls)” and a “tearDownClass(cls)” method. Both followed by the class decorator “*@classmethod*”. As the names suggest, they initialize all appropriate variables and close all appropriate resources respectively. In between these two classes, you may write as many tests method as you wish if you suffix the method name with “test\_”. Because you imported unittest, you may use the entire assertion library (i.e., self.assert<False/True/Equals/etc.>(appropriate parameters)) in these methods to ensure methods behave as they’re expected to. You may then end the file with:

*if \_\_name\_\_ == ‘\_\_main\_\_’:*  
 unittest.main()

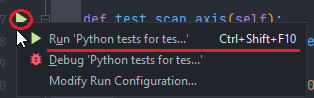
### Folder Hierarchy of Unit Tests



### Syntax Structure for Unit Tests

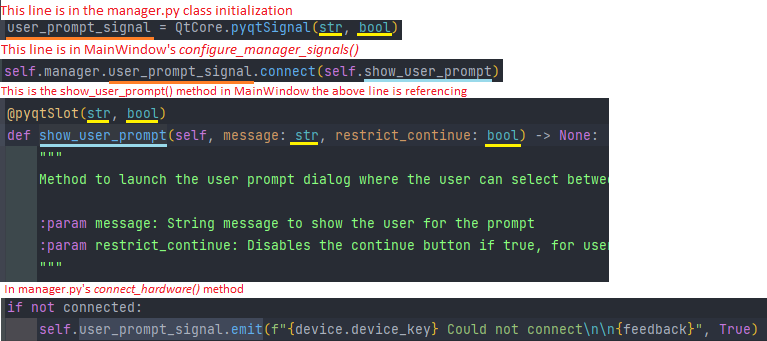


to run all the tests. You may also run tests individually if you are using PyCharm via the green run icons in the left-hand line number column, they are placed next to the method headers.



## Signals

Signals must be used to communicate between the manager thread and the ui thread to prevent memory access issues that could crash the program. The signals are easy to create and use: first, declare a “*pyqtSignal*” that contains the list of variables it will carry in the parenthesis ahead of it (e.g., *command\_signal = QtCore.pyqtSignal(str)*). Then have it connect to a method in the other class via its “connect” method (e.g., *self.command\_signal.connect(self.manager.exec\_command)*). The method “exec\_command” in the manager class must have the slot decorator directly above it with the matching parameter list (e.g., *@pyqtSlot(str)*)*.* Onceconnected, the signal can be used via its “emit” method (e.g., *self.command\_signal.emit(“STEP”)*). This will call the manager’s exec\_command method with the string value “STEP” passed along with it.



## Log

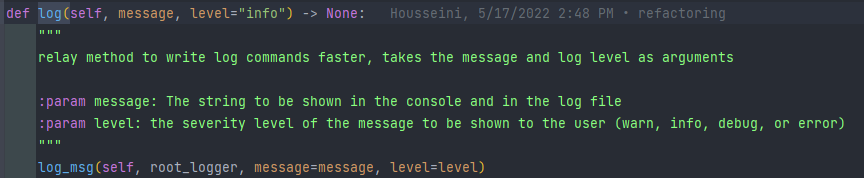
A function which allows frontend and backend classes alike to add entries to the wtf.log file. It has two parameters, message, and level. Message is a string containing the text to be added to the log file, and level indicates whether the feedback conveys ‘info’, an ‘error’ message, or a ’debug’ message. Entries also contain the current date and time, the thread name, the class name, and the line number, making this a powerful function for debugging.  


Figure 15 - Any class that wants to use the main/central logger needs to have this method defined within it



Figure 16 - The class needs to also have this import statement in the beginning