# Using 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.

# 

# Technologies and Dependencies:

## todo

# 

# High level overview

At the core of this application’s design are two threads which run as long 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 a function call signals contain 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.
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.begin\_script\_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.

When the project is opened from its root, the main target file of execution is “main.py”. This script executes the sequence of events that happens on application startup, including Creating and displaying the MainWindow class, loading the config file, prompting the user for a password, and telling the mainwindow to begin the manager thread, which starts connecting devices. In this file, the overarching frontend class, MainWindow is created, which In turn creates the overarching backend thread, Manager. The main window thread powered by Qt is centrally coded in the file “MainWindow.py” and the backend manager thread is coded in “manager.py”.

## Scripting

# 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 located 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

….Todo

## 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 of these values are automatically generated, so you do not need to worry about changing them, since they are located in the project folder and python has the ability to find the directory it’s placed in. The directories are also platform agnostic (i.e. works with Windows, Mac, Linux, etc.).

### 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 packages 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.

# Class Breakdowns:

## Manager

The manager thread serves a few central functions. 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 dictionary of tasks that have the arguments nested within it. Second, it contains the backend logic for all the tasks related to efficiency testing UA devices, along with all other script tasks. It can take loops into consideration and can even detect if the “Element” value is a static one instead of the “Current” dynamic value usually used in looped tasks, and automatically correct it for you if desired. Third, it 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.

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.

# Programming Guidelines and Best Practices

todo

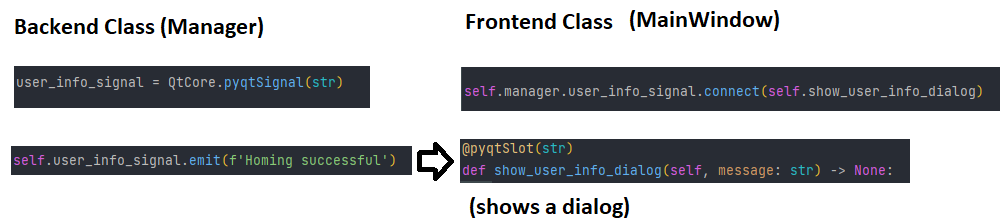
# Troubleshooting

## 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.

# 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 to 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) **actually 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.

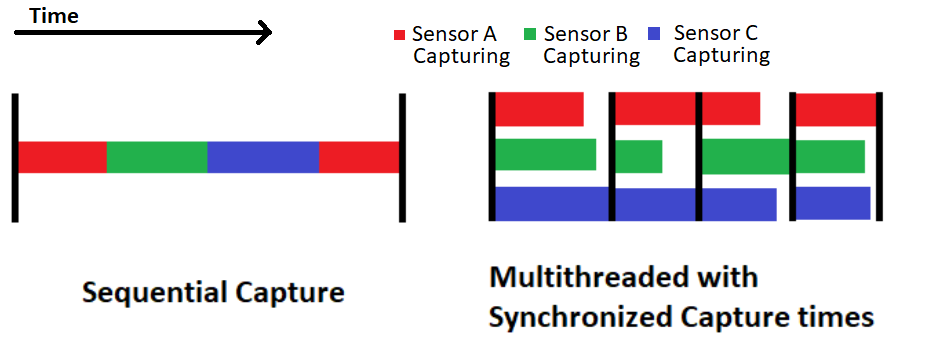
Note that while the manager is running this code its core run loop will be on hold, and no command strings sent to it in the intervening time will be acted upon. The manager is “busy”. There is a signal in Manager called the busy\_signal used to communicate this to other classes and to the user.

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 realtime 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

# 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 located 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

## Qt designer

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.

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.

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

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,” and “MyQWidget.” In this new file, you may create signals, methods, and variables that pertain to that specific widget.

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\_<widgetname>*” where subfolders is 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.

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.

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.

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 initalize all appropriate variables and close all appropriate resources respectively. In between these two classes, you may write as many tests method as you wish as long as 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()

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.

## Signal

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.