Contents

[Running the Program 2](#_Toc37929172)

[Directories 2](#_Toc37929173)

[.github 2](#_Toc37929174)

[Configs 2](#_Toc37929175)

[Custom\_Tests 3](#_Toc37929176)

[IBERT\_bitstreams 3](#_Toc37929177)

[Results 3](#_Toc37929178)

[Saved\_Queues 3](#_Toc37929179)

[System 4](#_Toc37929180)

[ResultsConfiguration 4](#_Toc37929181)

[temp 4](#_Toc37929182)

[ConfigFileValidationSchema.json 4](#_Toc37929183)

[Devices.json 4](#_Toc37929184)

[DeviceFileValidationSchema.json 5](#_Toc37929185)

[src 5](#_Toc37929186)

[Instruments 5](#_Toc37929187)

[Scripts 5](#_Toc37929188)

[GUI 5](#_Toc37929189)

[Util 5](#_Toc37929190)

[Flat Files 6](#_Toc37929191)

[.gitignore 6](#_Toc37929192)

[\*.tcl 6](#_Toc37929193)

[LICENSE 6](#_Toc37929194)

[PICTestControlSoftware.pyw 6](#_Toc37929195)

[README.md 6](#_Toc37929196)

[requirements.txt 6](#_Toc37929197)

[template.json 6](#_Toc37929198)

[Running RunAConfigFileMain.main 6](#_Toc37929199)

[The GUI 7](#_Toc37929200)

# Running the Program

The entry point for the PTCS GUI is the file PICTestControlSoftware.pyw file which is flat in the codebase. This is the file that should be run to start the application. All it does is call the main application in the GUI directory and set the matplotlib backend. This will open with pythonw when run standalone and not open a terminal window behind the application. Only Python 3.7 is supported at the moment.

There is also a way to test one config file without any GUI. If there are any command line arguments, this mode will be triggered. To see the required command line arguments run the program with the –h flag.

# Directories

## .github

Has issue templates for users of GitHub to use to create a new issue. Don’t worry about this directory too much.

## Configs

json formatted files that correlate 1 to 1 to a test (“experiment” is interchangeable with “test”) that can be selected from the drop-down of the GUI to add to a queue. These could either be written by hand, or generated when someone building a test on the UI saves said test. Each config file is validated by the schema located in System/ConfigFileValidationSchema.json at runtime. A config file consists of the following:

**name** – the name to display when the user clicks the drop-down button to add an experiment to the queue. This is the only mandatory item in a config file object.

**data** – a dictionary of parameters that have to do with the given experiment. The reason why someone would specify a parameter in the config file rather than just use a value in a test is so that if a user in the future wants to change a value that is used in a test, they do not have to rebuild the test on the test build page just to change one line of the experiment. When an experiment is added to a queue, a user can click on the experiment and change the values of these parameters. If a user has changed a parameter of an experiment, its config file is not changed. A copy of the config file with the changed parameters is put in the tmp folder of System, and is deleted when the queue finishes. A copy of every config file of each queue run is in the results folder if someone wants to reference it later.

**devices** – the names of the devices used in the test. These devices can be chosen from the ones specified in System/Devices.json. An experiment does not have to use instruments though, so this can be omitted.

**tcl** – If this test is a TCL based test, the name of the TCL file to be run will be here. Tcl tests also must have their name end with the string “ (Tcl)”.

**experiment** – a list of objects. These objects each represent the files that should be run that comprise the experiment. Each object should have defined the following items:  
 **type** – the type of file to run. Currently, only “PY\_SCRIPT” is supported, but this functionality could be added onto in the future  
 **source** – the name of the file to run. It is assumed that the file will reside in the src/Scripts directory.  
 **order** – a whole number starting from 1, that designates what order the scripts will be run in. Script multithreading is currently not implemented but if it needs to be in the future, multiple files could have the same order, which means they will be run at the same time before the next number of order is run.

**display\_order –** The order the experiment will display on the queue drop-down list on the GUI. Valid ordering starts at 1 and ends at 1 million. Any test that has the same value as another for this field will either display above or below it (nondeterministically), but will definitely be above experiments with a higher value, and below experiments with a lower value. Experiments without this field set will internally be assigned to 1 million and display at the bottom in the section with the other 1 million valued experiments if there are any.

## Custom\_Tests

This directory will only be created when a user first successfully builds a test on the Test Build tab. It has files that are text-based descriptions of the experiment(s) created. These files are not used for much after an experiment is created since it is converted into a python file at the same time, but it may be useful to figure out how to re-parse these files back into the test-build tab so one can edit the tests when PTCS is reopened.

## IBERT\_bitstreams

These are hardware bitstream files to flash the VCU108 with.

## Results

Every experiment that is run, be it in a queue or not, receives its own folder in this directory based on the name of the config file and the date and time started. The folder will minimally have the config file run as Config.json. Other files generated from running that config file will be put in that folder. If they are not explicately created in that folder to begin with, calling add\_result\_file on a script’s ExperimentResultModel object will copy the file into that directory anyway. As you can see by running the Fake Voltage Accuracy Test, the collected results, the reduced results, and a plot of the reduced results are stored in that directory.

If a series of TCL tests are run, they will receive one folder in the Results directory labeled “Tcl Experiment” and inside this folder will be subdirectories for each TCL test that had output. The subdirectories are numbered to ensure unique names, but the numbers can look weird because any test with no output has its subdirectory deleted. If you run a TCL test and you don’t see a subdirectory for it here, it means the test did not produce any output for whatever reason.

## Saved\_Queues

Saving and loading a queue will be facilitated by result of the files stored in this directory. When you save a queue from the GUI, a file is created in this directory with entries of the path to the experiment config files used saved with an asterisk before it. Below that are the values of all the parameters that test will be run with. One reason for the creation of this was out of necessity of combining TCL files into one file. See the document “the new queue features” to know more about running TCL tests with the queue. Also, see the document “How to Use Alternate Parameter Passing Techniques” because at the bottom it will show the layout of a parameter file. This is the same layout used in a saved queue file.

## System

This folder currently holds two folders and three files.

### ResultsConfiguration

This is the directory that stores json files of the metadata of each queue and experiment run.

For each experiment run, there will be a copy of the config file under the name "experiment\_config\_location", the paths to all the files created by the experiment under “experiment\_results\_files” (aka in the Results directory), the start date and time under “start\_datetime”, the end date and time under “end\_datetime”, and the folder name where the results are saved for this experiment under “experiment\_results\_directory”.

For each queue run, there will be a json file generated by the date and time run. This json file will contain the date started, the date ended, and the names of each of the experiment metadata json files stored in the same directory. Representing each experiment that was run in that queue.

### temp

When a queue is run, each experiment has its configuration output to a file (yes it is also created if no one changed any of the test parameters) within this directory. This path will be input as the –c parameter to RunAConfigFileMain.main. If you have not realized already, the real meat and potatoes of PTCS lies within the subsystem RunAConfigFile. All experiments in a queue just basically get fed into RunAConfigFileMain sequentially. Each config file put in this directory gets os.remove’d at the end of it getting run through RunAConfigFileMain If an exception is raised during its run, it still gets deleted because the os.remove call is in a finally block.

ConfigFileValidationSchema.json

This file is the schema for validating a config file. When the GUI starts up, it takes all the config files in the Configs directory and validates them against this schema. It then adds them to the drop-down box in the queue tab. If a file turns out to be not valid, an exception will be raised and the console will give the most likely not too helpful error details.

Fun fact: each config file run in the headless mode is validated also.

I am pretty sure this schema conforms to draft 7 syntax

Devices.json

This is the file that stores all the devices able to be connected to the instrument. All the details about this file either have already been explained in this document or are explained in the “How to Add a Device to PTCS” word document.

### DeviceFileValidationSchema.json

This is the json schema for validating a device file. When the GUI starts up, it loads the Devices.json file and uses it to instantiate the drivers for the instruments found in that file. If a user entered a device incorrectly into that file and restarted the GUI, the software will error before the GUI starts.

## src

This directory holds all the python files used by PTCS and some others that may or may not be used (ex. Scripts and Instruments).

This directory has 3 subfolders.

Instruments

Holds all the front-end drivers. Instruments are a software object that models some external equipment, either physical or virtual. See the word document “How to Add a Device to PTCS” for more information on how to write a driver, because it is kind of complex. Don’t forget to also conform to the naming conventions for parameters and return types: “How to Let User Know of Parameter and Return Units”

Scripts

This folder is where PTCS will look to run scripts. These scripts are python files. They need to have at least one method which is named “main” (that does not return anything) and have two parameters: the data map and the results model object (ExperimentResultsModel). The data map uses a similar structure to the JSON config, except that the parameters are found in data\_map[‘data’][‘Initial’]; collected data should be placed in data\_map[‘data’][‘Collect’] and reduced data in data\_map[‘data’][‘Reduce’].

The results model object has some cool features. Look through the file and see that it can export a bunch of file types without a user needing to remember the syntax of writing a file, or even where the results directory is. It can also save data to a csv file, and make a scatter chart so you do not have to remember matplotlib syntax.

GUI

The design pattern is called UI-Application-Model. I am not sure the theory of it, but it sort of resembles MVC. The only problem is that all Application classes are only instantiated once and are globally accessible through the SystemConfigManager. Because of this it is hard to model the interactions between the SCM and all the other classes. I did my best at modeling everything by creating the PTCSArchetecture diagram using draw.io.

As you can see in that diagram, PTCS is made up of two separate subsystems less some coupling between 4 classes. These systems are the GUIMainApp system, and the RunAConfigFile system. This is nice because one can run the program two distinctly different ways as represented by the subsystems.

Util

This directory holds a file of global functions (Functions.py), a global file holding just one variable: the SystemConfigManager, and a file that holds CONSTANTS that can be accessed wherever, for example the names of absolute file directories, constants to make the GUI look how we want. It also has Timestamp.py which is a class that makes it easier and simpler to deal with timestamps around the program.

# Flat Files

### .gitignore

Tells git what files not to track changes on. Look this file up for more information

### \*.tcl

A bunch of TCL files that are either ready to run, or are partial ones that will be synthesized together with other files when a TCL test is run. All these TCL files should probably be put in another folder sometime in the future, no need to have these hanging out.

### LICENSE

This repository is licensed under GNU GPL

### PICTestControlSoftware.pyw

This is the main entry point to the GUI and the command line interface. Read the below section to find out more about this file and how it is run.

### README.md

Gives a short, human readable description of what is found in this repository.

### requirements.txt

The python dependency packages that need to exist in order to run the software. See the build guide in order to run the command to install all the dependencies.

### template.json

A template of a config file.

# Running RunAConfigFileMain.main

Whether the program is run in headless mode, or experiments are added to a queue and then run, each experiment passes through RunAConfigFileMain.main. This function takes the command line arguments, an optional SystemConfigManager object, and an optional QueueManager object. The two are optional because if running in headless mode, there will be no references passed into the main function because the call is initiated from the command line.

The function first parses the command line arguments. The argument specification is that there needs to be a mandatory –c CONFIGFILE argument. Optional arguments include –p PARAMFILE which lets a user give the path to a parameter file, and any other arguments desired. See the “How to Use Alternate Parameter Passing Techniques” document to know the specification of a param file and inputting parameters on the command line.

Next, the json config file is validated and constructed into the ConfigFile domain object class. A data map is then created (this is a python dictionary which will hold all the data useful for each script to use while running) and loaded with a dictionary representation of the config file.

Next, the parameters are added to the data\_map based on those specified in the param file, and those simply passed through on the command line. Notice that some parameters will be overwritten if more than one source has a parameter with the same name.

Next, all the devices used for the experiment are connected. This is done by going through all the devices listed in the “devices” section of the config. If there is an entry in the Devices.json file with the same name, it checks the type of device connection it is.

* If it is “Visa”, it will be connected with PyVisa using the specified connection string in the “Default” entry of the device.
* If it is “Direct”, it will pass whatever string is specified in the “Default” entry of the device into the constructor of the device specified by “Driver” entry as the first and only argument.

When a device needs to be used, its corresponding class is imported in global scope on the spot if it has not been imported already, then instantiated with either the device object, or the “Default” string passed in as the first argument. When this is instantiated, exit\_stack.enter\_context() is used. I believe enter\_context() makes a device close itself automatically if things do not go well at runtime. These connected device classes are then added to the data map.

The scripts are then spawned. Sequentially from low “order” attribute to high “order”. If the script name has not been imported since the program has started, the file with the name specified in the “source” of the entry of the config file’s experiment section is imported globally. When it needs to be run, the function in that file named “main” is called with two parameters: data\_map (which has the config file, additional parameters if specified, and the driver classes for each instrument specified) and the experiment\_result object for this current experiment. The script will assumedly call methods on the drivers, and receive data from them, and save that data in some format. The experiment result object has methods one can call for exporting data in nice formats, for example making a matplotlib plot, exporting to a csv file, or taking binary data read from an instrument into a .png file.

When the last script has finished running, all the data that the experiment\_result object has gathered will be output to files in the Results directory as well as the ResultsConfig directory as specified previously.

The main function then ends.

# The GUI

It uses wxPython which is a python wrapper for the popular cross platform wxWidgets front end desktop framework (our app is not cross platform yet though). The GUI is made by hand, no SceneBuilder like tools were used or anything.

As you can see in the PTCSArchetecture.png, the MainFrame UI component holds all of the other UI components in a tabbed page UI data structure. Each tab conforms to the abstract class Page which pretty much defines that each page should have two panels. One that displays things, and one that controls the things on the one that displays things. You can see this on the GUI on every tab.

Pretty much the design pattern is that if you click a button, an event handler will get called from the UI that will use an Application class inside the global SystemConfigManager while handling that event. Look at any Panel for examples of how this works. That method then calls things on the model objects that the manager classes contain, which manipulates the state of the model objects.

Within the SystemConfigManager, there is a class called the UIController that works in the opposite direction. Methods get called on it from wherever which in turn manipulates the state of the items in the UI, for example it switches the queue from running (with a terminal-like output) to the panel with the drop-down box where one can add things to the queue.

An intricacy of this UI is that every half of a second, it updates (or tries to because it is currently broken) the size of the text of every button or text box. I was not able to take a look at it to see how it works, so I cannot give you any more information currently.

Most of the functionality of the Queue page is pretty evident just from looking at it. The queue is manipulated using add, remove, clear, save, load, and run. This page is a very good reference point for building or editing other pages, since it has a control panel on the right that updates with user actions and an interacting display panel on the right. The queue page has not been changed much since its creation, and it’s not likely that anything more than minor changes will be required in the future. A solved bug with the queue feature can be found on the [github repo](https://github.com/FuturePhotonInitiative/PTCS).

The Hardware and Results pages are extremely straightforward and their functionality is explained in the User’s Guide. The Results page uses os.system(“start [file]”) to open results files; this may in the future be expanded to open more file types or directories.

The Test Build page is entirely new as of Summer 2019, and it is probably the page that could use the most updating and improving. For anyone tasked with working on this page, you should definitely read the Test Build Page – Developer Notes document. That should hopefully familiarize you sufficiently with the page, because its code is by no means easy to understand.