The run manager program is comprised of three files: manager.py, input_output.py and acquisition.py. I'll go over some of the important functions and variables to make acquisition code run smoothly with this new setup.

manager.py

The Manager Class

class Manager(self)

This object is the main control hub of the run queue. There is one Manager object needed to start up a run queue. The Manager object talks to and facilitates all communication between all other objects/processes/threads involved in running acquisitions. No other processes communicate directly to each other.

The State Variable

string state = 'STARTUP', 'PHASE', 'ACQUISITION', 'STANDBY', 'END'

This variable determines what is currently happening and can be controlled by the user. The Manager begins in the STARTUP state and initiates a few variables. It begins a phasemonitor.py process and then transitions to the PHASE state. This will continue indefinitely until the user tells the program otherwise. The Manager is alive and ready to receive user input if it is in the PHASE, ACQUISITION or STANDBY states.

When the Manager is in the PHASE or STANDBY states, it will look for run dictionaries in the run queue. The difference between these states is simply whether the phasemonitor.py process is running. The user can also tell the Manager not to look for more run dictionaries when in these states (see Commands below).

The ACQUISITION state is started when the user places a run dictionary into the queue and the Manager begins the acquisition process. During this state, the Manager does not look for more run dictionaries, but they can still be added to the queue.

What Else Does the Manager Do?

When the Manager first starts up, it begins the user interface Tkinter windows as a separate thread using multithreading. Communication between the Manager and user interfaces is facilitated via queues. It also boots up a PhaseMonitor object in a separate process using the multiprocessing module, communicating with the phase monitor via multiprocessing queues (these are like the standard queue objects but process safe; see multiprocessing docs for more info).

When an acquisition is run (including the phase monitor process), the manager begins the acquisition as a new and separate process using the multiprocessing module, and communicates with the process via multiprocessing queue objects. The phase monitor (if running) is paused before the acquisition is started. The manager passes the run dictionary object to the acquisition and in return receives the location of the Google Drive data folder created by the process.

Until notified by the user, the Manager enters a loop, processing user text input and facilitating acquisition output. The Manager will also continually check whether the user input windows have been closed. If so, the Manager will stop all other processes and end itself. The Manager will also check to see if the acquisition process is still alive or if it has ended. If the acquisition is dead, the Manager will attempt to copy the output files from the acquisition to the data folder created. Whether or not this copying succeeds, the Manager will revert to the state it was in before the acquisition began (e.g. if the phase monitor is running and an acquisition begins, upon completion of the acquisition the manager will resume the phase monitor process).

Ending Acquisitions Forcefully

If the user wants to end an acquisition, they can enter a command into the user input window (see Commands below). The user can choose to end an acquisition when convenient for an acquisition (i.e. after the next completion of the acquire() function of the acquisition; see Acquisition below), or immediately.

If the user chooses to end the acquisition when convenient, the Manager will send a message to the acquisition asking it to close. The Manager then waits up to ten minutes for a response from the acquisition that indicates that it has shut down. This is perhaps not the best way to go about ensuring the acquisition has closed, but I'm working on an alternative and for now this should suffice. In the meantime, there may be a case where the acquisition takes more than ten minutes to exit the acquire function. In this case, the process will not close within the ten-minute time limit, and the Manager does not receive the shut down notification. The Manager then notifies the user that the process was not confirmed to have shut down. The user can then simply re-enter the command and wait again.

If the user chooses to end the acquisition immediately, the acquisition will receive a KeyboardInterrupt exception and will close properly (this is handled in the Acquisition class).

input_output.py

UserInput Class

class UserInput(output queue, input queue)

This class is a Tkinter TopLevel widget initiated by the RunScheduler Tkinter Window widget. The user can enter text input which will affect the acquisition. This input will be recorded to a file "userinput.txt" and will be moved to the Google Drive folder after the acquisition is complete.

Commands

1) "quit acq" – ends the current acquisition next time the acquisition finishes its acquire method
2) "quit acq now" – ends the current acquisition immediately via a KeyboardInterrupt exception

3) “quit” – closes all user input/output windows and ends the acquisition the next time the acquisition finishes its acquire method
4) “quit now” – closes all user input/output windows and ends the acquisition immediately via a KeyboardInterrupt exception
5) “pause” – pauses the current acquisition
6) “resume” – resumes the current acquisition
7) “progress” – asks the acquisition to print its progress variable (see Creating Acquisition Code below)
8) “pause queue” – stops the run manager from looking for a new run dictionary
9) “resume queue” – allows the run manager to look for a new run dictionary

UserOutput Class

class UserOutput(output queue, input queue)

This class is a Tkinter TopLevel widget initiated by the RunScheduler Tkinter Window widget. Any output from the run manager or the acquisition will be displayed in this window. The acquisition output will also be logged in a file. This class is also used for the acquisition error output, which is opened as a separate window.

RunScheduler Class

class RunScheduler(output queue, input queue, master)

This class is a Tkinter Frame widget initiated by the Manager in a separate thread. This is the main window of the user interface. This class handles all the run dictionary creation, editing and scheduling for the user. This part of the manual will go over the types of files used by the RunScheduler class and a step-by-step process for how to create each type of file using the run scheduler.

File Types

1) Run Manager (.rm) file

This type of file is opened by the run manager when you click on File → Open Run Dictionary. It is a type of CSV file with the particular format required by the RunScheduler. The first line of this file contains: the name of the python script with which this Run Manager file works, and the location of this file. The second line of this file contains the headings “Property” and “Value”. All lines below the second line contain properties and values to be read in by the script specified in line 1. An example of a .rm file is shown below.

```
fakeacquisition.py,C:/Users/Travis/Google Drive/analyzed_data/valdez/instrument control modules/
Property,Value
Experiment Name,test
Experiment Name Addon,example
Repeats,1
Averages,10
Quenches,C:/Users/Travis/Google Drive/analyzed_data/valdez/instrument control modules/quenchfile

```

2) Run Dictionary (.rd) File

This filetype is read by the acquisition. When the user schedules an acquisition, they are asked to save a run dictionary file in the run queue folder (see instructions below). This is the type of file they save. This filetype differs only slightly from .rd files, but keeping these filetypes separate allows us to use the .rm files as a template and have the acquisition read in various .rd files for the same type of acquisition. This way we don't have to keep creating new .rm files. These files are still CSV type files. The first line includes the headers "Property", "Value", and "Order". The "Property" and "Value" columns are the same as described above. The "Order" column is created by the run scheduler and will specify the order in which the properties and values are to be written by the acquisition into its data.txt file. The order can be changed by the user by simply changing the order of properties displayed in the run dictionary table (see below). An example of a run dictionary file is shown below.

```
Property,Value,Order
Experiment Name,name,0
Experiment Name Addon,addon test,1
Repeats,1,2
Averages,10,3

```

3) Quench Dictionary (.quench) File

These files are also CSV type files and specify a few properties for the quench cavities in the hydrogen experiment. They require five columns: "Quench Name", "Attenuation Voltage", "Cavity", "Open", and "Status". These files also require at least six rows; one for each quench cavity. The "Quench Name" column must include: "pre-quench_910", "post-quench_910" and similar names for the 1088 and 1147 cavities. All six of these must be included. No exceptions. If you want to include more rows (i.e. notes that you want to write down in the quench dictionary file) you can. However, these extra rows cannot be added from the user interface and may as well just be included in the run dictionary file. An example of a quench dictionary file is shown below.

```
Quench Name,Attenuation Voltage,Cavity,Open,Status
pre-quench_910,0.0,pre-910,True,on
pre-quench_1088,0.0,pre-1088,False,off
pre-quench_1147,0.0,pre-1147,False,off
post-quench_910,0.0,post-910,False,off
post-quench_1088,0.0,post-1088,False,off
post-quench_1147,0.0,post-1147,False,off
```

The “Attenuation Voltage” for each cavity can be any float value from 0.0 through 8.0, or “None”. If the value specified is “None”, the cavity will use its default/pi-pulse voltage as indicated in the “quench.csv” file located in the main folder (see below).

The “Cavity” column is just a key used by the acquisitions. This column may eventually get merged with the “Quench Name” column. Do not change these values from what is shown above.

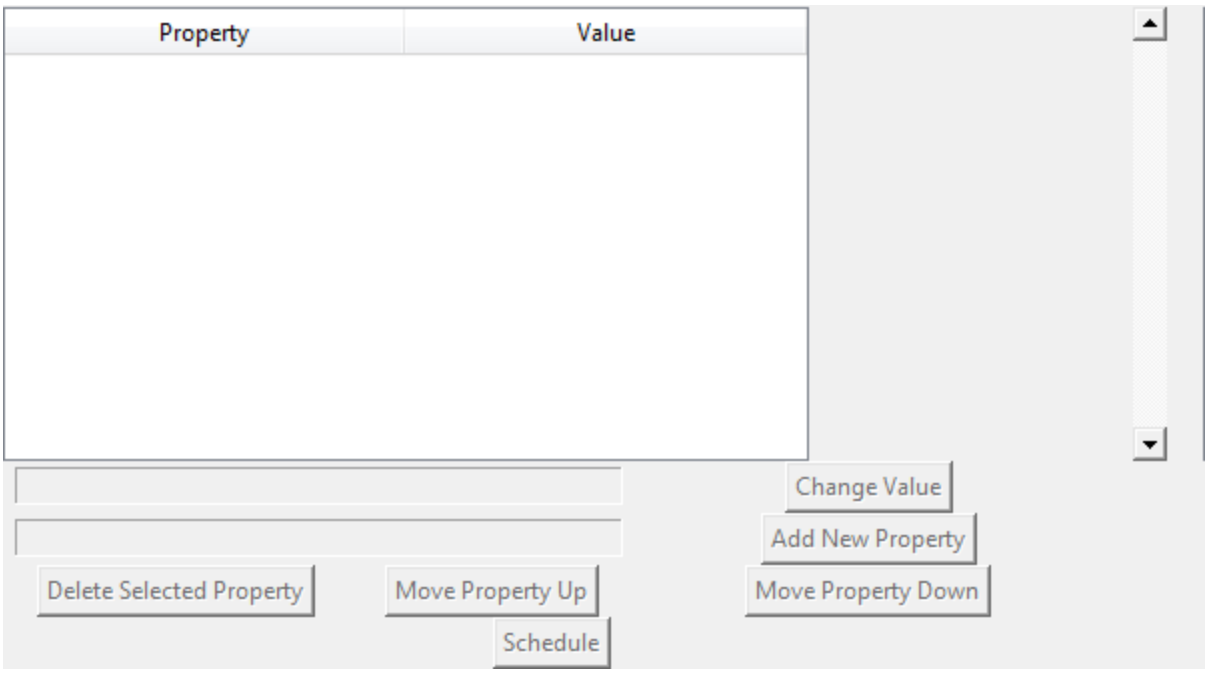
The “Open” column tells the Quench Manager whether to attempt to open the USB Synthesizer hooked up to this cavity. Items in this column are Booleans. Generally, they should all be set to “True” unless the cavity is not working. If you specify “True” and “off” (in the “Status” column) for a cavity that you do not want to use, the Quench Manager will make sure it is off. Otherwise, the synthesizer will not even be opened and may be left on from the last acquisition.

The “Status” column values can be “off” or “on”. This tells the Quench Manager if the cavity will be on at the beginning of the acquisition.

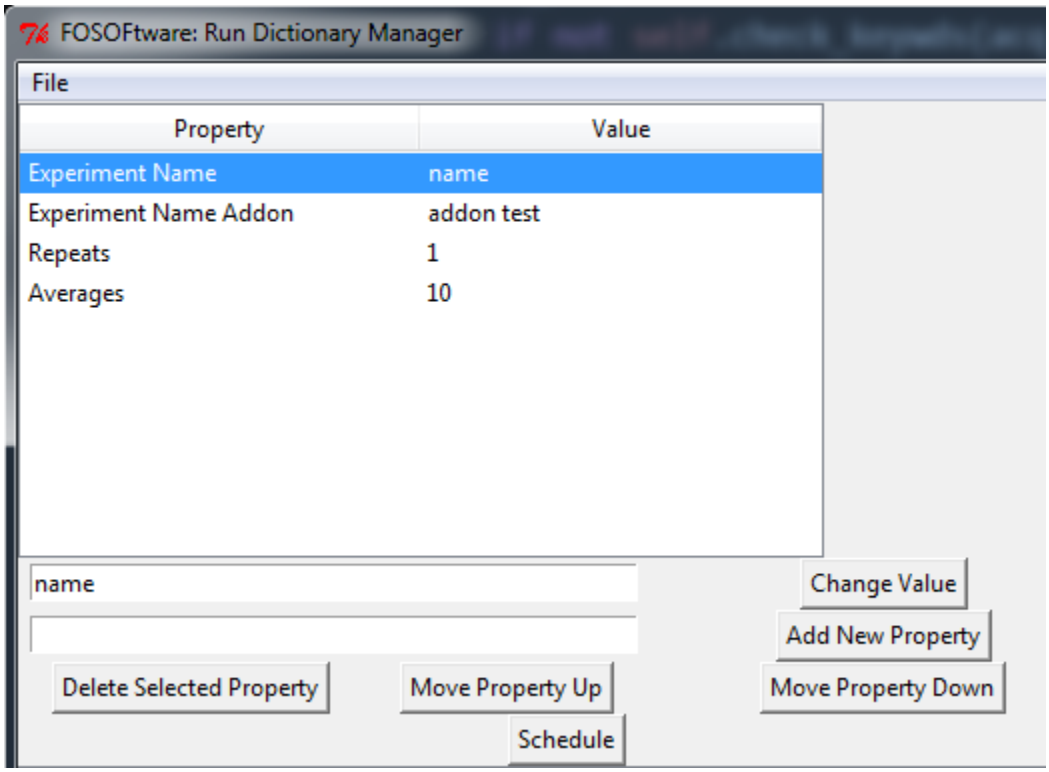
Introduction to the User Interface

The Run Dictionary Panel

Most of the user interface is self-explanatory, but I’ll go through it here anyway. The run dictionary section (shown below) is where all the run dictionary properties/values are created, edited, deleted and rearranged. It is also where you can schedule an acquisition to be run.



When the Run Scheduler is first opened, all the buttons will be disabled as shown above. To enable the buttons, open or create a new run manager file. Once a run manager file is open and being edited, this panel will look like the image below.

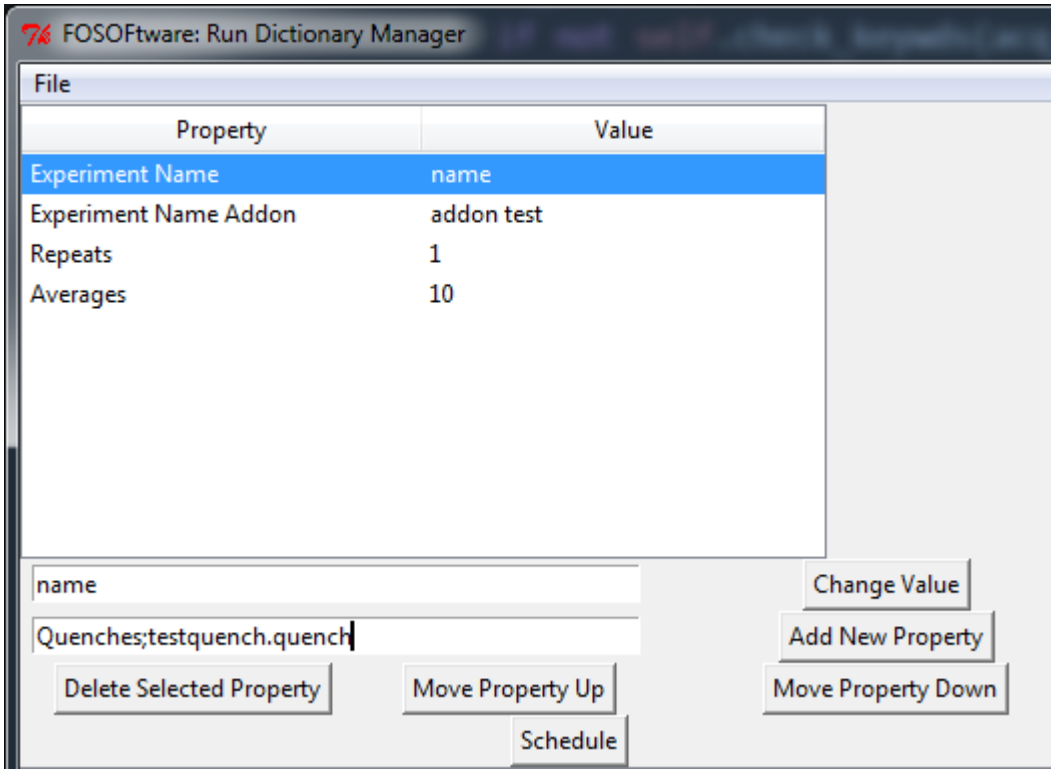


When a run manager file is opened, its properties can be edited except for the acquisition script that will be run when the run dictionary is scheduled. This must be done manually or by

creating a new run manager file. To edit a property, click on it in the table and the current value will appear in the first text box below the table. Enter a new value into the text box and click "Change Value". No restrictions on values are imposed by the run dictionary, but they may be imposed by the acquisition itself (e.g. the "Repeats" value should be an integer).

To rearrange the order of the properties, use the "Move Property Up" and "Move Property Down" buttons below the new property text box. This will not affect the acquisition, but will affect how the run dictionary is written by the acquisition to the head of its data.txt file.

To create a new property, enter text as shown below into the new property text box (the second text box below the table). The format of input should be "property;value". That is, the property and value should be separated by a semicolon. Unfortunately, any other semicolons will be separated from the value and property. I may change that if it becomes a problem, but we don't typically use semicolons in our run dictionaries as of right now.

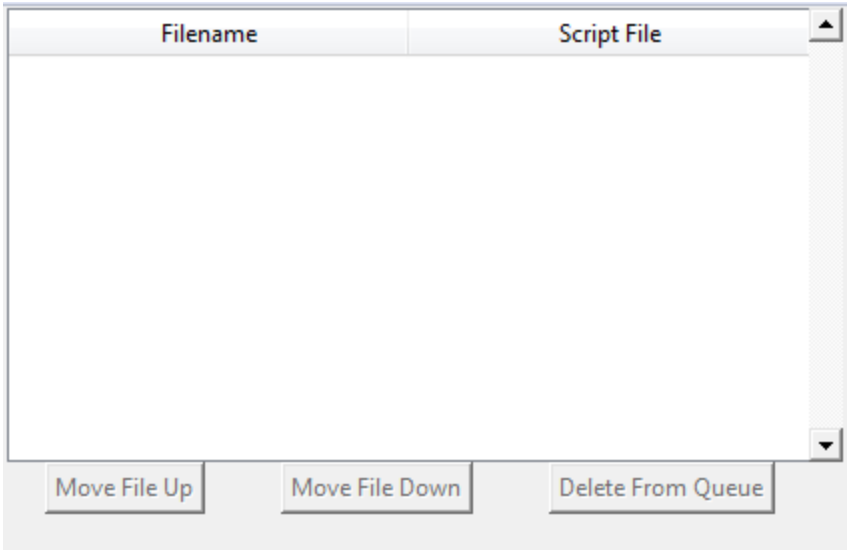


If you have incorrectly entered a property name and wish to change it, simply delete it with the "Delete Selected Property" button and recreate it.

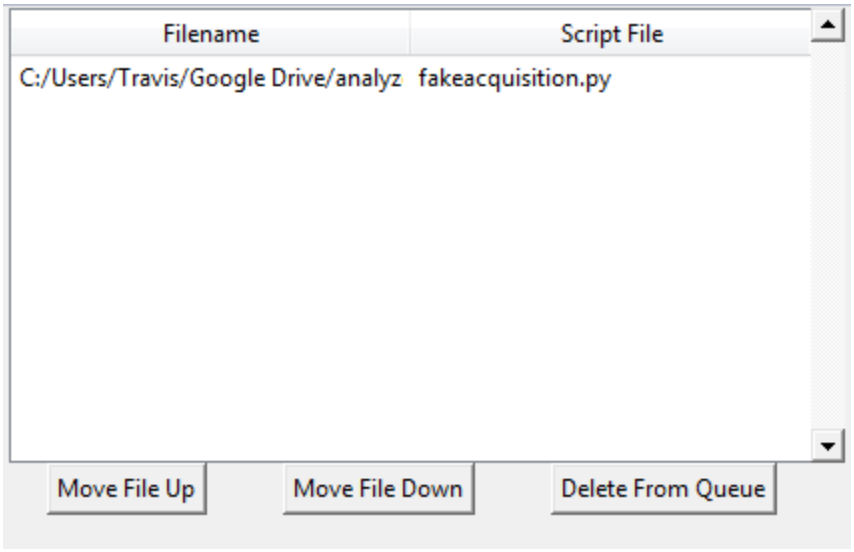
To schedule an acquisition, click the "Schedule" button. Instructions for this will follow below.

The Run Queue Panel

The run queue panel shows all current and upcoming acquisitions in the queue. This will be empty when the Run Scheduler is started, but will become active once the "Schedule" button in the run dictionary panel has been pushed.

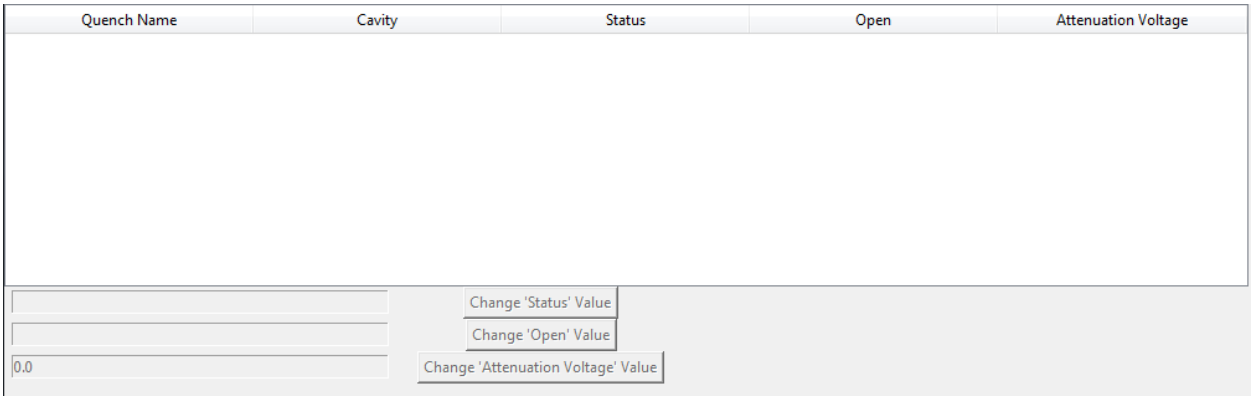


Once the run queue is active, the run manager will read in files from the top of the table down to the bottom. Files that are currently being processed or have been processed will no longer remain in the queue. You can edit the order of the queue by clicking on its name in the table and using the “Move File Up” or “Move File Down” buttons. If you made a mistake and want to remove an acquisition from the queue, use the “Delete From Queue” button. If you only want to pause acquisitions after the current one, you can do that with the command line input interface (see above).

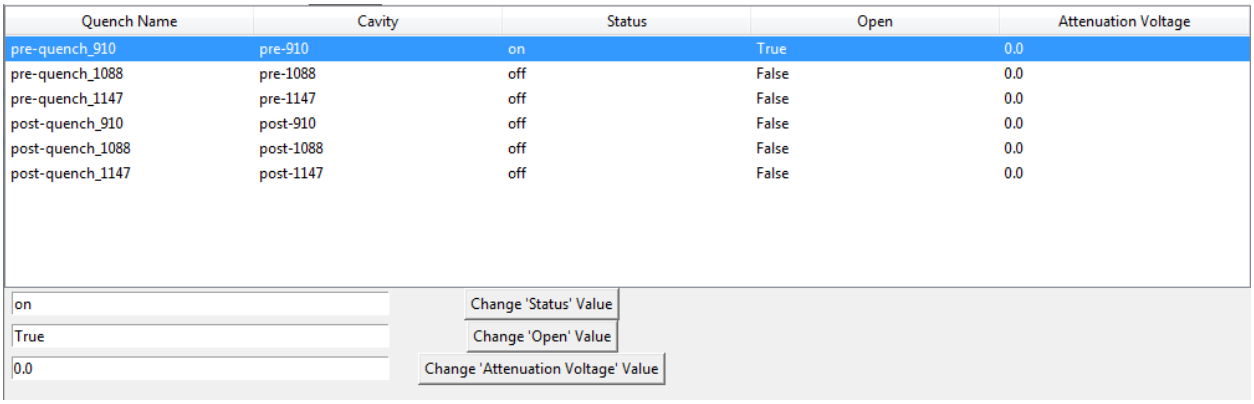


Quench Dictionary Panel

The quench dictionary panel shows the properties for the quench cavities that will be used by the current acquisition. Once again, on startup this part of the UI will be disabled. It is enabled by opening a run manager file with a “Quench” property (the capital Q must be included).



Once the quench panel is active, just as with the run dictionary panel, the properties can be edited by selecting the quench cavity to edit and entering text into the corresponding entry box. Then, click the "Change _____ Value" button. Protection has been included for "dumb" values (values that don't make sense or that would cause an error) since all quenches have the same allowed values.



How to Create and Run an Acquisition

Step 1: Creating an Acquisition Script

While writing this run manager code, I've tried to streamline writing acquisition scripts as much as possible. All the necessary communication between the acquisition and the run manager is done in the acquisition.py file in the Acquisition object.

When creating a new acquisition script, import the acquisition.py file and any other modules needed. Create the acquisition as a class object, and make that class a subclass of the Acquisition class in the acquisition.py file.

```
2  from acquisition import Acquisition
3
4  class Name(Acquisition):
```

The __init__ method required by all classes should be written as follows:

```
def __init__(self, queue_in, queue_out, queue_err):
    super(Name, self).__init__(queue_in, queue_out, \
                               queue_err)
```

Here, queue_in, queue_out and queue_err are Queue objects of the multiprocessing module. As you can probably guess, queue_in will be read by your acquisition, while queue_out and queue_err will be used as the sys.stdout and sys.stderr channels.

In its initialization method, Acquisition sets up in/out communication with the manager, asks the manager for the run dictionary and creates a Google Drive data folder and starts a user input checking thread. To successfully create the data folder, **you must include "Experiment Name" and "Experiment Name Addon" as properties in your run dictionary.** These properties will be used to create the folder name: "date-time Experiment Name - Experiment Name Addon". I am working on a smarter method for run dictionary creation, but this is what we have for now. By making your acquisition a subclass of Acquisition and using the Acquisition __init__ method, you ensure that communication between the manager and your acquisition will run smoothly.

After its initialization method, the Acquisition class calls the initialize_acquisition method. You must create this method and specify what is initialized. This is where you can create useful global variables, open communication to instruments as global handles, etc. The run dictionary can be accessed by the "self.run_dictionary" variable. This is a pandas.DataFrame object with keys corresponding to the properties in the .rd file that was sent to the acquisition (the Property column of the run dictionary tale; above).

The "**self.progress**" variable should also be initialized here. When a user enters "progress" into the command line interface, this variable is printed to the standard output. They can do this at any time they like, so updating this variable regularly is a good habit. This way, you can see how far along your acquisition is. The only requirement is that this variable remain a string.

After calling the initialize_acquisition method, the Acquisition class enters a loop known as the **main acquisition loop**. This loop performs the following tasks:

1) Look for user input and respond appropriately
2) Run the acquire method if not paused
3) Check the self.acquisition_complete variable to see if the acquisition has finished

The user input checking thread mentioned above constantly looks for user input and determines the best course of action. If the user enters "pause", "resume" or "quit acq", the data will be passed to the main thread. In step (1) of the **main acquisition loop**, the main thread checks to see if the user input checking thread has passed on a command. This is the point at which the process will pause, resume, or quit if the user has asked. Keep this part of the main acquisition loop in mind as it is important for writing step (2) of the loop.

If the Acquisition has not paused itself, it will now call the **acquire()** method. This method should be overridden by the user and should include all steps to acquire data. These include updating the "**self.progress**" variable, changing iterator variables, querying the digitizer for data, writing to a file, etc. You should be smart about writing this function and think about how often you'd like the acquisition to check again for user input. The longer this function takes, the longer you'll have to wait for the acquisition to pause or shut down. As an example, I've created a fakeacquisition.py script for use with the run manager. The acquire method is shown here.

```
23      def acquire(self):
24
25          sys.stdout.write("Acquiring some data.")
26          time.sleep(1)
27
28          self.progress = "Repeat: " + str(self.rep) + "\nAverage: " + \
29                          str(self.av)
30
31          self.av = self.av + 1
32
33          if self.av == self.averages:
34              self.av = 0
35              self.rep = self.rep + 1
36
37          if self.rep == self.repeats:
38              self.acquisition_complete = True
39
40          return
```

In this example, I've decided that I want to check for user input after every "trace" of data we take. Instead of taking a trace from the digitizer, I've just used a time.sleep() command to simulate acquisition time. Notice that at the end of the acquisition, I have to iterate the self.av (averages) variable and if necessary, iterate the self.rep (repeats) variable or change the self.acquisition_complete variable. A similar approach could be taken for any list of items over which you would like to loop by creating a randomly-ordered list and using an iterator variable to loop through the items in the list.

In addition to the methods above, you should also override the shut_down(), pause(), and resume() methods. In the shut_down/pause methods, all open instruments should be closed to allow access to them during the pause phase or after the acquisition has finished. The resume() method should reopen all connections closed during the pause() method.

One last thing to note about creating a new acquisition file is the use of the '# Run Dictionary Keys' line to make .rm file creation faster (see below). To use this feature, create a comment block in the code that begins with '# Run Dictionary Keys'. Anything else entered in

this comment block will be read by the user interface as a required key for the run dictionary. You can also specify values for these keys or guidelines for users to follow when creating a new run manager file. An example of this comment block is shown below for the fakeacquisition.py file.

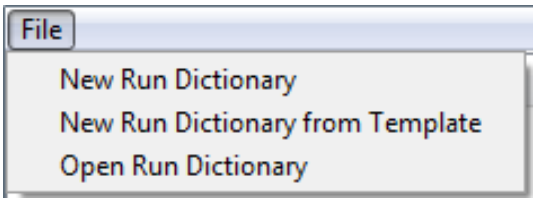
```
5
6   # Run Dictionary Keys
7   # Repeats = int > 0
8   # Averages = int > 0
```

Once you have created your acquisition file, make sure you include the acquisition in a folder included in your PYTHONPATH. The Manager will import your acquisition code automatically if it can find your code.

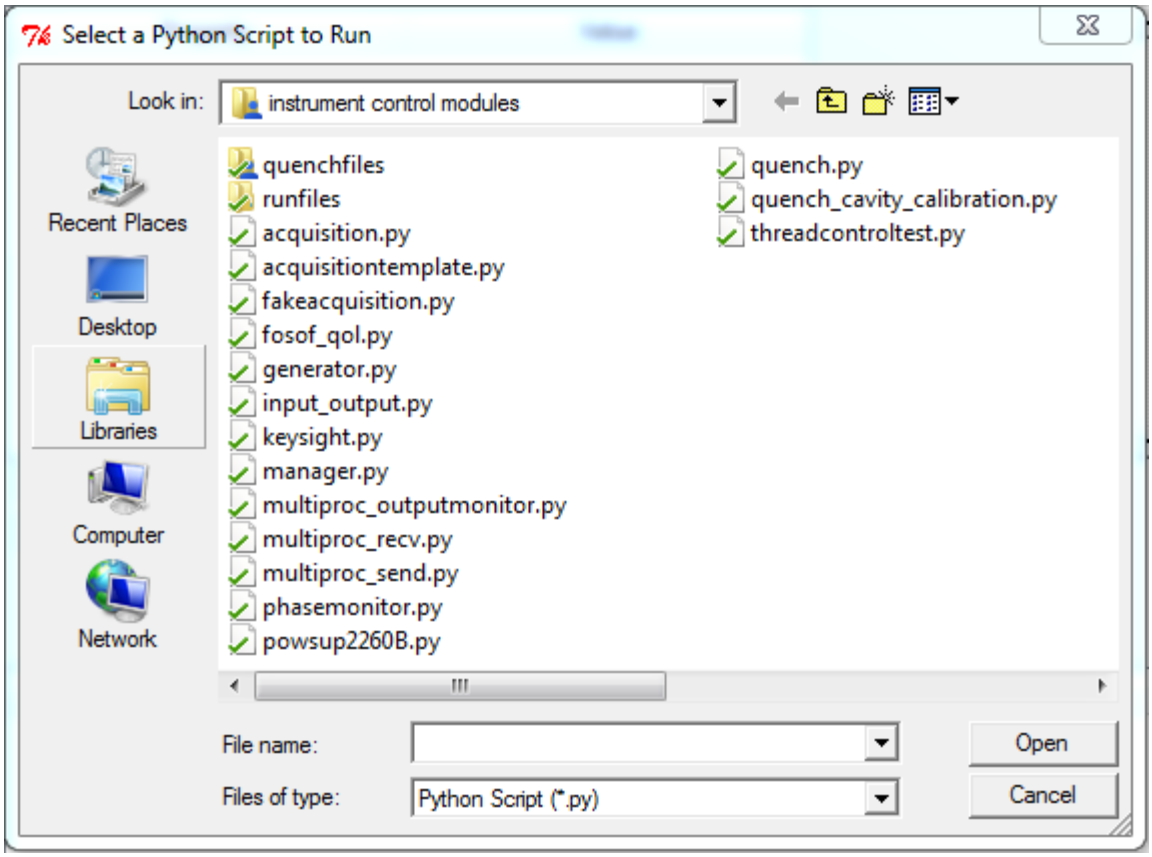
If any of these guidelines are confusing, I've included fakeacquisition.py, acquisitiontemplate.py and a few example run dictionaries/run manager files in this zip file to help further.

Step 2: Run Dictionary File Creation

To run an acquisition, you first need to specify a run manager file which will be used to create the run dictionary. To do this, you have two options: create a new file or open an existing file.



If you create a new file, you'll first be asked to specify the acquisition script that this run dictionary/run manager file will call.



After that, you must specify the name that you'd like to give the .rm file. Protection has been put in place to ensure that the file specified has the .rm extension. The Run Scheduler will create a new file template with the script you specified as the first line. It will populate the columns with an example property. This can be deleted once you have created a second property. If you delete this row before you create a property, the tk Treeview widget (the table) will raise an exception.

If you choose to create a new run dictionary from a template, you must select a Python (.py) file that has a '# Run Dictionary Keys' section (see above). If the file you selected does not have a '# Run Dictionary Keys' line, you will see an error message and nothing will open. If the file does have the required line, you must then select a name for your .rm file just as if you created a run dictionary from scratch. The table will be populated with the values from the .py file.

If you choose instead to open an existing run manager file, you'll be asked to select the file from a Windows Explorer interface. Once you open the file, you can change any of the existing properties.

Step 3: Scheduling the Acquisition

Once your properties have been changed, you can click the "Schedule" button below the property table. This will open a prompt asking you to select a name for the run dictionary (.rd) file. **Make sure you don't overwrite any .rd files that are waiting in the queue.** You can

move the scheduled run dictionary files up and down the queue, provided there is more than one file.

Also Included with This Package

In addition to the user interface, Manager and Acquisition classes listed above, I have included the fosof_qol.py file and the paths.csv file that fosof_qol needs to run. Inside fosof_qol is the only variable you'll need to change inside a script. The variable _DEFAULT_FILE_LOCATION_ will need to be changed to the folder in which you plan to place the paths.csv file. Once this has been done, you'll only need to change values inside of a csv file.

The paths.csv file includes the paths to all important directories required for our FOSOF scripts. These include, the main run queue folder, the .rm file folder, the .quench file folder, the binary traces folder and the Google Drive folder. Change these values in the paths.csv file to match those on your computer.

I've also included a "phase_monitor_DEFAULT.rd" and "phasemonitor.py" files. These files are just templates and need to be edited. They must be placed in the Run Queue folder and in the PYTHONPATH, respectively. The names must remain the same as they are included explicitly in the manager.py file. The phase_monitor_DEFAULT.rd is the run dictionary for the Phase Monitor that starts up with the run manager. The phasemonitor.py script is the phase monitor script itself, as you'd expect.

Finally, there are various txt files and folders within this zipped folder. These are just examples of various output folders and files from simulated acquisitions.