**Making a Scanner (and scannable) Instrument in LabVIEW**

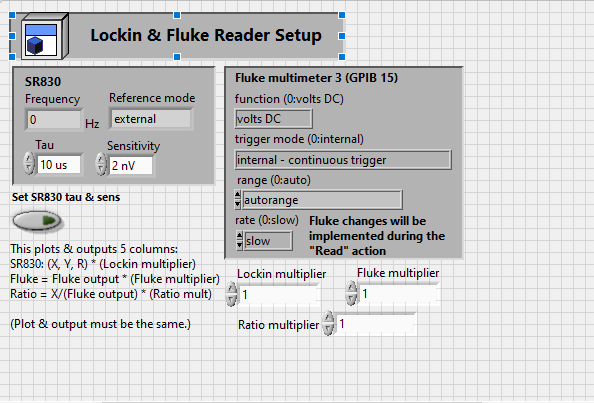
**Background Information**

A scanner is an object that takes another object called a scannable and iterates through a list of values that the scannable has defined it can. Essentially, the scannable is the instrument (like the digikrom) that has a range of numbers or values it can iterate over (like wavelength), and the scanner is someone pushing the button to iterate over that range.

The way the scanner and scannable instruments (objects) work in our code is very similar to how the iterator and iterable objects work in Java; so if this explanation doesn’t make sense look up how an iterator and iterable work in Java.

**Making the Scannable Instrument Class**

1. Create a new folder in LabVIEW-Programs\Instruments
   1. Names are hard to change later on when the file is committed and pushed to the Git repository, so choose well. Make sure the final name choice is made before you commit and push.
2. Open the LabVIEW project in LabVIEW-Programs\LabVIEW.lvproj
3. Right-click on the folder in the LabVIEW project explorer and pick “New” >> “Class” from the dropdown menu. Save the class to give it a name; you should use the same name as the directory you created. The file extension should be *.lvclass*
4. Now that the class has been created, right-click on the class, choose “properties” >> “Inheritance”>> “Change Inheritance”, select “Instruments/Scannable.lvclass” then choose “inherit from selected.”
   1. This breaks everything in our LabVIEW-programs folder because we have not defined the required methods common to the class.
   2. For this class, the only required method is a *\_Read.vi* file, see item 7 below, which when created will make LabVIEW runnable again. (You can do this right away to un-break LabVIEW even without any actual code in the *\_Read.vi* file, if you want.)
5. Define the properties of the object in the *[Class name].ctl* file created when we created the class, i.e. what instruments and properties associated with the instruments and/or scan that we want to implement. Each instrument is different. Typically you should only need to include the VISA information for the instruments, along with those properties that you want the user to be able to adjust from the scan setup box.
   1. Open the .ctl file, which should be a (mostly) blank file reading “Cluster of class private data” at the top, above a recessed empty cluster box.
   2. To add VISA information for each instrument, you can copy & paste from the LabVIEW Programs\Global\*GPIB Numbers.vi* global variable.
6. IMPORTANT!! If a reader object has been created in the past for the instrument, always look at the way the methods were created in that object for reference. That will help reduce the time and the difficulty of the code.
7. Now, create (or finish creating) the *\_Read.vi* file by right-clicking the class and choosing New >> VI for Override >> *\_Read* from the dropdown menu; choosing this initially just creates a file which calls the base *\_Read.vi* of the parent class. It then needs to be edited to give instructions for how to actually read from the instrument(s).
   1. The purpose of this function is to read the data from the instrument each time it is called and output it for that point. For example, in the case of the lockin it reads and then returns the X, Y, and R values.
   2. Use an unbundle function to pick out necessary properties from the object. In general, you will always unbundle the VISA number(s) for whatever instrument(s) we are creating a reader for. In some cases there may be other things that need to be known from the object in order to properly read the data (which channels to read, does the user want any rescaling or division, etc.). Those other things will often be set by the user in the *Setup UI.vi*, see below.
   3. Note: at the end, you only need to bundle the VISA numbers (as good practice) as well as any properties which will change during the experiment. In the case of the lockin, the only property which might change would be the sensitivity (other properties are set at the beginning of the scan).
8. Now that the *\_Read.vi* file is created, we will follow a similar process to create additional methods for the class by right-clicking the class and choosing New >> VI for Override or VI from Dynamic Dispatch to add additional methods which will be needed. See the next few steps.
9. The *Check Status.vi* is to read the current settings from the physical instrument into the LabVIEW object. Create it using the VI for Override option.
   1. In general, you will unbundle the VISA number(s) of the instrument(s) from the class cluster. Then, you will create the code (e.g. using driver VIs) to read the different settings for the instrument and output them as a “Status” string. To make things clearer, if using more than one instrument you could put a single element flat sequence structure around the code for each instrument to keep things together when doing a Control-U cleanup.
   2. When finished, bundle all the different settings and the VISA numbers of the instrument(s), wire their values to the bundle, and wire the bundle to the object output block.
10. You will also have to create a constructor VI. To do this, right-click the class and choose New >> VI from Dynamic Dispatch, and give it the same name as the class. Right click on the inputs and outputs located at the top right corner of the VI and change them all to “recommended” by choosing This Connection Is >> Recommended. You will be using the *Check Status.vi* here, so that must be created before writing the code for this VI.
    1. Using local variables of the instrument clusters, bundle the instrument information (for the case of multiple instruments) and wire it through *Check Status.vi* (which you just created) and *Readable.vi* (which is the constructor of the Readable Instrument class).
    2. Note: the *Readable.vi* takes an array of the channel names and an array of the corresponding units as two separate inputs. There should be a “Status” string output which shows the data values as well as their names and units.
11. Make the *Set Properties.vi* as a new VI from Dynamic Dispatch. This VI is not strictly speaking necessary (it’s not part of the class), but is pretty useful to have. In some sense it’s the reverse of *Check Status.vi*: that sets the LabVIEW object properties based on the physical instrument; this sets the physical instrument based on the LabVIEW object properties.
    1. This one is straightforward to make, you take the object’s properties in and unbundle them then send them to the physical instrument using driver VIs.
    2. Writing to the physical instrument is different for every instrument we have, and there may or may not already be a driver VI to change a certain setting. You will have to write one yourself if we don’t have one.
12. Make the *Setup UI.vi* as a new VI for Override. This VI will be the user interface that appears in the reader settings panel when setting up a scan. The UI looks like this, for one example:



* 1. If you are interfacing with an instrument(s) for which we already have code, you can probably copy and paste a lot from the *Setup UI.vi* for that instrument.
  2. Set up the front panel to have inputs for whatever properties you want the user to be able to set and possibly displays for things you want the user to know. See the image for an example.
  3. The block diagram is tricky so closely follow other examples. It should use a flat sequence structure. The first item of the sequence will have a case structure which either sets the fields in the UI based on what’s in the object properties using unbundle (first time through), or else sets the object properties based on what’s in the fields in the UI using bundle (all other times through). The other items in the sequence structure will mostly or entirely be to enable button presses in the UI, which must be done via case structures due to how this UI gets executed in a frame of the scan setup UI. The button presses generally will set properties on the physical instruments before the scan starts; the *Set Properties.vi* method may well be useful for this. The buttons must be manually “unpressed” after the commands are executed (e.g. properties are set).

1. You may want to add other methods depending on the specific needs of the instrument you are implementing. The VIs mentioned in this document only form a minimum set.
2. To implement the new reader in scans, you must add the new reader to the scan driver options. This is done by modifying two VIs.
   1. The first one is the *Reader List.ctl* type def file.
      1. You can open it directly, but one way to find it is to go to the *Construct Reader.vi* file in the *Scan Driver.lvclass*. Right-click on the “Reader Enum in” variable and choose “Open Type Def.” from the dropdown menu.
      2. Within *Reader List.ctl*, right-click on the “Reader” variable and choose “Edit items…” from the dropdown menu.
      3. Insert an item and type the name of the new reader object which you are trying to add (for instance, “Lockin and Fluke Multimeter”). Keep them in alphabetical order so that John does not get mad.
      4. Save the *Reader List.ctl* file.
   2. The second one is *Construct Reader.vi* (which might already be open, depending on how you find the type def file).
      1. Add a case to the case structure. It should automatically add the case we just added to *Reader List.ctl.*
      2. Add our constructor vi and *Reader.vi* from the Reader class to the case structure and wire the outputs of the constructor vi to *Reader.vi* (which is the constructor of the Reader object class) and from there to Reader Out.
      3. Save the *Construct Reader.vi* file
3. At this point, we can double check that it works by opening *Settings Panel.vi* and running a scan with the new reader object we just created. Double check that set parameters works and that the *Setup UI.vi* interface fits in the space allotted on the Scan Driver setup vi.

Other important notes:

* When you make edits to the methods, you should not only save the methods but ALSO right click the class file in the LabVIEW project manager and click Save. There are other options like “save all”, but you want to select the one that just says “save.”
* Status string is what is automatically saved to the data file