Table of Contents

Adding a New Device

There are two parts to adding a new device:

1. Write device methods.
2. Set-up UI for your devices columns in the SetList Table.

First, you'll have to provide methods which instruct the program how to interact with your device. It is likely another device is similar, and you can inherit many methods, only overwriting those that you need to tweak.

This documentation should be fleshed out as the first groups actually work through this process.

Adding a new .lvclass

Your new device will be another `.lvclass` which must inherit from Device.lvclass.[1] While you can play around with your local copy and revert it to recover the master branch, **you'll want to first make a new branch on the GitHub (email Neal Pistenti, Zach Smith or Daniel Barker to be a collaborator on SetList) with a descriptive name like add-<NewDev>.** After switching to your new branch, open `SetList.lvproj`.

Create a folder on disk for your new device at `codes/DeviceVIs/<NewDev>` (To find the disk location of this directory go to the `Files` tab in `SetList.lvproj`, right click the `DeviceVIs` folder, and select `Explore` from the dropdown). This will automatically be created in the project explorer based on the path where you save your new device on disk. To create the class, right-click the `DeviceVIs/<NewDev>` folder, select *New* > *Class* and name it. Then, right click `<NewDev>.lvclass` and select *Properties*. Go to the *Inheritance* page and select the appropriate parent object (using *Change Inheritance...* button). This could be `Device.lvclass` if no similar objects exist, `MasterDevice.lvclass` for a PulseBlaster replacement, or even `PulseBlaster.lvclass` if you want to tweak the methods like `PulseBlaster/LoadHardware.vi` while inheriting the HWI preparation. If you need to ADD any data fields, open `<NewDev>.ctl` and add controls, keeping in mind that you already have all the controls for EACH ancestor up the chain. Finally, save your new class in the `<NewDev>` folder on disk.

It is often convenient to add access methods for any new data fields that you add to the class. To organize this methods, add a Virtual Folder to the new device by right clicking `<NewDev>.lvclass` and selecting *New* > *Virtual Folder*. The convention is to name this Virtual Folder `Accessors`. Next, right click `<NewDev>.lvclass` and select *New* > *VI for Data Member Access*. In the dialog, use the radio buttons to create a *dynamic accessor*, check the box labelled *Make available through Property Nodes*, and select the Virtual Folder that you just created from the drop down menu under *Advanced Options*. The dialog also shows all data members of `<NewDev>.lvclass`, which can be selected by clicking. For each member, create a *Read* and a *Write* method by selecting the appropriate option from the *Access* drop down menu and then clicking the *Create* button.

Programming for your <NewDev>.lvclass

Your first task is to provide the necessary methods for your <NewDev>. Ancestors like `Device.lvclass` and `MasterDevice.lvclass` provide structure for the methods you'll need to program. LabVIEW has automated method vi creation to save some repetitive tasks. Begin by right-clicking `<NewDev>.lvclass` and select *New > VI for override...*.[2] Any item with an * MUST be overridden while the others are optional.[3] In this way, the generic parent classes determine what sorts of things your object must do. When in doubt, look at another device to see what each method is doing.

Any method added as *VI for override...* will initially contain a “parent call method.” If you want to completely override the parent method, delete it and write your code in it's place (typically within case structure for handling error in); You can also choose to run the parent call method in addition to your code. Note that many of the methods of `Device.lvclass` are blank, so you'll need to look at other child devices to see what needs to be done.

I suggest going down to the “Configure Table Appearance” section before returning here (— *Daniel Schaeder Barker 2015/12/31 11:31*).

The following is a generic description of what each forced override VI needs to do. Some of the following VIs also have mouse over context help, which can be seen by pressing `Ctrl`-H.

Note: Master device ramp and triggering information is passed in reverse time order.

1. `ParseSWI.vi` - This VI gets the columns of the SetList table associated with the device (the SoftWare Image) and converts the information into a form that allows programming of the device. This means evaluating variable and user-defined function calls as well as correctly generating any ramps for the device.

2. `RequestTrig.vi` - This VI takes the output from `ParseSWI.vi` and identifies changes in the device output that require a trigger from the master device.

3. `Prune.vi` - Double checks the parsed data from `ParseSWI.vi` and trigger information from `RequestTrig.vi`. The purpose of this VI is to remove lines from the parsed SWI data when no trigger is required for that line (i.e. remove duplicate lines so that they don't propagate to the HardWare Image).

4. `BuildHWI.vi` - This VI takes the output of `Prune.vi`, the pruned parsed SWI data, and converts it to a time-ordered list that can be used to program the physical device.

5. `ErrorCheckHWI.vi` - This VI looks at the HardWare Image for the device created by `BuildHWI.vi` to confirm that there are no errors. That is to say, the HWI from `BuildHWI.vi` is a valid and programmable HWI for the physical device.

6. `Load Hardware.vi` - This VI is called by CycleX before the experimental cycle executes. It checks the last HWI programmed to the device against the HWI requested for the current cycle and reprograms the device if the two HWIs do not match.

7. `CollectData.vi` - CycleX calls this VI after every cycle and before `Clear Hardware.vi`. It is used

to pull and distribute any data a device may collect. Hence, it is empty for many devices. One example implementation may be found in `NI Card.lvclass`

8. `Clear Hardware.vi` - CycleX calls this VI after every cycle to clean up the hardware in preparation for the next cycle. For example, `Load Hardware.vi` often expects to open connections to serial devices, so `Clear Hardware.vi` must close these connections to avoid errors on the next cycle.

Additional note: Many “legacy” devices, such as NI cards, are programmed differently. They have `ParseSWI.vi` and `RequestTrig.vi` bundled into a single VI, `PrepareHWI.vi`, and have `Prune.vi` and `BuildHWI.vi` bundled into `FinalizeHWI.vi`. If necessary, this method can be applied to a new device as well (simply override `PrepareHWI.vi` and `FinalizeHWI.vi` so that they no longer call `ParseSWI.vi`,`Prune.vi`,`RequestTrig.vi`, or `BuildHWI.vi`). However, this approach is not recommended since it reduces clarity and makes programming of future child devices more complicated.

Configuring Table appearance

Column objects determine how a user interacts with your device in the Table. There are a number of existing classes in `codes/ColumnVIs` which all inherit from `Column.lvclass`. Again, you can right-click any class within the *Project Explorer* and select *Show Class Hierarchy* to view the inheritance tree. After familiarizing yourself with how the different column types are used at Column Types, you may decide to add a new class (follow the instructions above to create, save, and change inheritance).

There are two VIs that must be programmed for the new device to show up correctly in the table:

1. `Init Device.vi` - This VI instantiates the columns for the device so that they appear in the SetList table. It should begin by calling `GetUserLayouts.vi` and end by calling `Set Columns.vi`. In between these two calls define the columns for the device by inserting the appropriate `<ColumnType>.lvclass` objects and configuring them by calling `ConfigFunc.vi` followed by `ConfigAppearance.vi`.

2. `Display Info.vi` - This VI pops up the dialog box seen after clicking *Edit* or *Add Device* in the manage devices pane. Currently, the `Device.lvclass` version of this method does not contain the required structures for this VI to function as desired. As such, it is **strongly** recommended that you copy the full “Error”/“No Error” case structure from another implemented device into the `Display Info.vi` method for your device and edit it appropriately. The purpose of this VI is to, first, extract important device info (e.g. the Serial Port, number of AO channels, Pulseblaster trigger line, etc.) and column appearance information; second, allow the user to change/configure these settings; third, detect any changes that were made and update the device data and columns correctly.

The device method `Init Device.vi` is called by the `DevUtil/Add Device.vi` and is immediately followed by the method `Manage Device.vi`.

Adding your new device

Once you're ready to test and use as part of SetList, you need to make your device show up as an option under `SetList → Manage Devices → Add Device`. To do so, place a plain text file named

`deviceName.txt` in the same directory as your lvclass file. The contents of the file should be the display name of the new device.

For example:

Your new class `CoolWidget.lvclass` should have a text file `deviceName.txt` that looks like this:

deviceName.txt

```
My Cool Widget
```

and both of these (and all of your supporting vis) should be located in the `codes/DeviceVIs/CoolWidget` directory.

[1] To view the tree, right-click any class and choose *Show Class Hierarchy*.
[2] This option is greyed out if you haven't set up Inheritance.
[3] Hopefully, the first few device programmers will help ensure that all the necessary generic methods are marked as “override required” to smooth this process. The necessary methods should all be marked now (— *Daniel Schaeder Barker 2015/12/31 11:29*).

Analog Inputs

Some NI cards come with Analog Input functionality. SetList can be configured to use this and report the data as a shared variable.

Device Configuration

The usual setup for a device applies:

- Channel range and configuration must be done in NIMax first.
- All inputs/output for a physical device belong under the same physical device in SetList's device manager. For Analog inputs specifically, you must configure:
  - **AI Pause Line**: The gate used to control when the device actually samples its inputs
  - **AI Sample Rate**: A *requested*[4] sample rate, in Hz. Applies to all channels on this device.
  - **Analog Input Channels**: A separate row for each input channel to be used, *including a unique name*. Names will be used in generating shared variables, so be both descriptive and concise.
  - **AI Lines** will be automatically updated based on the above
  - **Name** will also be used in generating Shared Variables, so be both descriptive and concise.

Shared Variables

Results deploy to network shared variables generated automatically by SetList. These are updated at the end of each cycle for any AI channels in use.

Naming Scheme

Names are generated programmatically based on the device and channel names in SetList.

The NI PSP URL for a given analog input is: `\\<path to control computer>\SetList-<device name>-AI\<Analog Input Channel Name>(<Analog Input DAQmx Channel Name>)`

For example, suppose you have a test device named `AITest` connected to you computer `Testiest-Ctrl`. On it there is an AI DAQmx channel name `AI1` that has been named `Test Sweep` in SetList. Its scan would be available under the name `\\Testiest-Ctrl\SetList-AITest-AI\Test Sweep(AI1)`

Data Format

The shared variable is a cluster of 3 elements:

- **t** an array of time point values, in seconds. *Gaps due to gating are transparent to the software!*
- **y** an array of input point values, in whatever scaling/volts was configured in NIMax.
- **Channel Name** the SetList-defined name for the analog input channel.

Sample Reader

There is a sample AI Reader packaged in `codes/DeviceVIs/NI Card/AnalogInput/AIReader.zip`. It demos the bare-minimum interface needed to retrieve Analog Input variables. It is packed as a ZIP to keep it from interacting with the rest of the SetList project.

[4] NI will automatically choose the closest sample rate it can actually achieve. Reported time offsets will reflect actual sample rate, not the requested one

Bugs & Feature Requests

With the code now hosted on GitHub, we have access to their issue tracker. We now use that location for reporting bugs as well as for making feature requests.

Anyone can view the tracker, but you'll need a GitHub account (Sign Up) to open a new issue for any purpose. You're free to set up an main account for your entire lab, and it helps the coders to know where issues/bug reports are coming from if they need more input.

There are several levels of issues, Each of these has their own tag you can apply when creating the issue, so that others will new the level of attention it requires. ***Please be careful to distinguish between:***

Feature Requests

New things that you would like to see added to SetList.

Open Issues in JQIamo/SetList with label feature request
No Issues Found!
View this list on GitHub

Major Bugs

Problems that prevent functional control via SetList. These are “If this doesn't get fixed, I can't use this program” scale problems.

Open Issues in JQIamo/SetList with label major bug
No Issues Found!
View this list on GitHub

Minor Bugs

annoyances such as missing error checking, UI problems, etc.

Open Issues in JQIamo/SetList with label minor bug
No Issues Found!
View this list on GitHub

Column Types

Several column types are already available for use in new devices. A couple should be reserved as Master-Only Columns

Generic Columns

These columns provide the functionality you'll need to implement most devices.

Column

- Use for: Analog values that can be variables
- Parent: None (This is the root class for other column types)

These columns allow any test inside them, and spawn the variable options inside the right-click menu. They have only background and text colors, with no special color handling for their contents.

DigitalCol

- Use for: Digital values that can be variables
- Parent: Column

These columns are like the Column class with special handling depending on contents. If it contains either a '0' or '1', a single click toggles to the other value. Their background color changes depending on the contents: entering '0' uses the off coloring, '1' the on coloring, and any other value the variable coloring.

SingleValCol

- Use for: Analog values that can only be updated once at the beginning of the cycle
- Parent: Column

This is exactly like the standard Column, except all rows except the first one are unable to be changed.

TriggerCol

- Use for: Placeholder for a channel that exists but shouldn't be manipulated from the table
- Parent: Column

This column has entry disabled so the user cannot manipulate it from the table.

MenuCol

Still under development

- User for: Letting the user pick only options from a menu
- Parent: Column

Clicking this column allows the user to pick from a number of pre-defined choices. These choices are passed to handling functions via Name-Tag string pairs that allow you to set behavior based on what was chosen.

Master-Only Columns

These columns were implemented specifically for use with processing the Master table, mostly to handle timing and cycle-building functions

GroupCol

- Use for: Configuring grouping at the Master-level
- Parent: DigitalCol

Implements the group column, and is a special-case of the DigitalCol. It contains extra logic to handle functions like hiding the whole row for a given group.

ModeCol

- Use for: Handling different modes of Master-Object
- Parent: Column

Contains the various modes the Master device can use. It contains logic to disable modes (for instance, Ramp inside a Loop-EndLoop pair), force Ramp steps on the first line, etc. etc. For historical reasons it is left as an independent class instead of folding it in as a subclass of MenuCol.

Feedback Overview

Sometimes one desires the ability to change the procedure based on the results of incoming data. This could be as simple as retaking a single shot, or as complex as manipulating variables to alter a ramp shape and duration.

To enable this kind of communication, we've built a JSON-based interface to a dynamically chosen TCP port. The port is looked up by your client via the NI Service Locator. This interface is available since SetList version v2.0.0 [5)]

TCP Communications

Getting the Port: NI Service Locator

The NI Service Locator is available at port 3580 on any machine with an active LabView installation. Visiting the link http://localhost:3580/dumpinfo? takes you to a page listing the services it currently knows about. When SetList is running, it will create a named service `SetList/JSON`. The service locator will be informed and the port number will be available at http://localhost:3580/SetList/JSON on the computer running SetList. You should be able to access it from another computer by replacing `localhost` with the IP address of the computer running SetList.

The response is of the form `Port=<port #>` which should be easily parse-able by whatever software is initiating the feedback. This port is static over individual runs of `SetList.vi`, so it is wise to cache it and only look it up at startup or when communication fails.

Communication Protocol

Once you have the port number, you can initiate a TCP connection with that port on the SetList computer. SetList is expecting a JSON object string preceded by its length in binary. That is four bytes representing the integer number of bytes in the string being sent, followed by the string itself.

When it finishes processing the data you've sent, SetList will send a response (formatted as above) and then close the connection. If more information needs to be sent, open a new connection to the same port.

Expected JSON Contents

General Format

The string is expected to conform to the JSON format specification[6].

SetList parses the top level of the JSON object for its `members`. It assumes each member's name[7][8] is a string matching to some internal command. The `value` part of the pair is passed to the command as a string.

If a matching command is not found, SetList will respond with an error and skip that member.

SetList's response is formatted as a JSON object as well, usually containing an array for responses that are normal and a separate array for errors.

Variable Format

Representing a variable as a JSON object works as follows:[9]

```
{
  "name": <string (required)>,
  "defaultValue": <number>,
  "sequenceFunction": <string>,
  "informIgor": <bool>,
  "sequence": <bool>
}
```

Where:

- All members' names correspond to their function in the SetList Variable manager.
- The "`name`" member has a **required** string value.
- All other members may be given the value `null` (e.g. "`informIgor`":`null`) to indicate the variable should retain the previous (or, if creating a new variable, the default) value.

Variable Set

Variables can also be grouped into "Variable Sets" using JSON arrays:[10]

```
[
  <variable 1>,
  <variable 2>,
  ...,
  <variable N>
```

```
]
```

Command Formats

Immediately Apply Variables

This command has SetList update the values of the variables right away, regardless of the status of a running sequence. The format is:

```
"instantVariables":<variable set>
```

where <variable set> is a single variable set object as specified above.

Sequence Set Variables

This command has SetList update the values of the variables after the end of the currently-running sequence. SetList maintains an ordered (FIFO) list of variable sets, and applies the next set at the end of a sequence just before re-starting the sequence.

The full command JSON is then:[11)]

```
"sequenceSets":[
  <variable set 1>,
  <variable set 2>,
  ...,
  <variable set N>
]
```

Where <variable set i> is a variable set as defined above.

Mulligans

Mulligans have the simplest format:

```
"mulligan":[<number>,...]
```

where <number> is the file number of an element you are trying to mulligan.

SetList will only check if the elements of the array are numbers, not if they are in the history (and thus mulligan-able). It reports the total count of numbers it finds, and sends errors for the non-numbers.

Example Clients

`codes→Utils→ExternalFeedback→ExampleClients`[12] contains subdirectories with simple example clients for:

- Igor
- LabView
- Python

which should be enough of a starting point to either integrate into your data acquisition package or build a client in some other language.

Why TCP?

Because TCP libraries exist for many programming languages (Python 3.5, Python 2.7, C++, IGOR, MatLab)[13] in addition to LabView, this allows you to "close the loop" however you wish.

LabView has a built-in TCP Listener which waits for incoming connections on a specified port before handing the connection off to your program to handle. This allows us to do a low-overhead loop that just sits and waits for outside data or to be shutdown.

[5] Previously we used a custom protocol over fixed TCP ports to do this. It was too clunky given the existence of standard data exchange formats like JSON, so it has been discontinued. Documentation is still available on old_feedback.
[6] see also JSON
[7] The first part of a `name:value` pair, described in the spec as a `string:value` pair
[8] not to be confused for the value of a member named 'name'
[9], [10], [11] with optional white space added for legibility
[12] or on GitHub
[13] These are the results of a quick Google search, no guarantee implied

Getting SetList: help for the uninitiated

If you've made your way to this page, working with a code repository is probably a foreign concept. While it's not necessary to use GitHub to use the SetList, it is well documented and easy. And for programmers, it also has some nice bells and whistles.

The https://github.com/JQIamo/SetList repository is the central storage for SetList hosted on GitHub. Presently the repository is public meaning anyone can download the code; however, only collaborators may make changes.

Having a central repository means that individual labs can

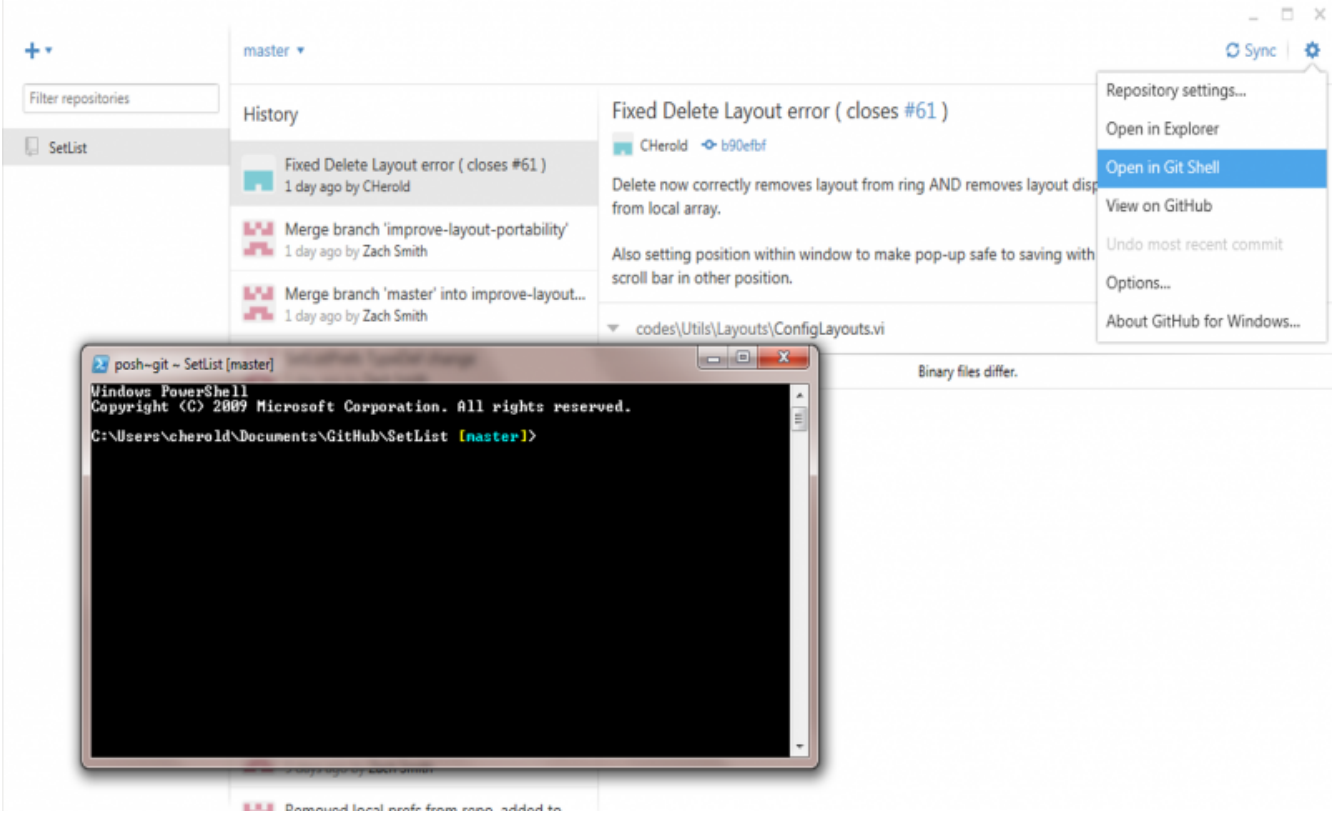
- Share code
- keep up with code improvements from other labs
- track issues and collect feature requests in one place

Step-by-step

Note: I thought when I started this that GitHub for Windows would be easiest, however tags do not show up!

With that in mind, here are very specific instructions about getting SetList from our GitHub repository:[14)]

1. Download and install GitHub for Windows client. Check out the “help tab” at the top!
2. Sign up for lab account at https://github.com. While not expressly required, it will give you a way to start versioning other things in the lab. Also, GitHub is extremely generous with free private repositories for educational use; go ahead and request a discount.
3. Navigate to the repository at https://github.com/JQIamo/SetList and sign in to your (lab's) account.
4. Click “Clone in Desktop” on the right sidebar. The standard location is Documents/GitHub but you can choose whatever you like in the resulting popup.
5. You now get to choose by always having the latest-and-greatest “beta” version (`master` branch) or working of a stable `tag`.
    1. In GitHub for Windows, ensure that SetList is selected on the left:



2. The pull down menu likely says `master`; here you can choose between development `branches`.
3. In order to use the stable tag, you'll need to use the command line.
    1. As shown above, click the settings gear at the top right and choose “Open in Git Shell.”
    2. The command line will show you the path to the SetList repository as well as your current branch.
    3. Type `git tag` to see the available tag names.
    4. switch to the latest tag by typing `git checkout <version>` where `<version>=v1.2.0`, e.g.
6. GitHub for Windows will always show you the branch you are using. The neat thing is when you switch (checkout something else) it leaves all unchanged files in place and only adds/removes/overwrites what is different. **Note: when you checkout a “tag” GitHub for Windows will say “DETACHED HEAD: <SHA hash #>.” This is normal and points to the commit from which the tag was generated.**
7. Finally, please don't be discouraged by the load time the first time you open SetList.lvproj. In order to make LabVIEW more compatible with the repository, the code will be compiled locally ONCE. Every subsequent load should be much faster!

Git Shell tips

- If you checkout a different branch with GitHub for Windows while the Git Shell is open, you can press enter to update the displayed branch on the command line.
- To close the shell, simply type `exit`.

Switching branches

To switch branches, following the above instructions starting from below the screenshot. Remember, you'll need to use the command line to checkout a new tag. Also, you can browse the tags at https://github.com/JQIamo/SetList/releases.

Your lab might be very happy simply using `master` so you always have the latest and greatest additions. However, periodically, you'll want to start GitHub for Windows and look at the top right corner next to Sync. If there are new changes on the central repository, you can pull them down by clicking Sync. There is no need to sync with a tag because it is static by definition.

[14)] Of course, there are many other ways to accomplish the same as described at where_can_i_get_it

New SetList

See SetListFAQs for frequently asked questions about how the SetList behaves. Many of the pages linked herein need more content. If you've found your way here, please take the time to add helpful tips/tricks/hints for the benefits of others!

Description

This documentation is under development as we roll out the new SetList. Please be an active wiki user and augment/edit it as you see fit!

The underpinnings of the old group CycleX LabVIEW program have been rewritten using object-oriented LabVIEW. In the new structure, the SetList has been streamlined into a single table with customizable views. Implementation of hardware programming is now device agnostic and relies on *device classes* to provide *methods* for translating the SetList to device-readable instructions. Previously, to add a new type of hardware device to the old CycleX, one had to re-program the guts; now a new device can easily be added to SetList by programming a set of prescribed methods.

It is my sincere hope that the **new device flexibility will enable a single version to be used throughout the laser cooling and trapping labs at JQI instead of the frequent, lab-wise branching that has been typical.** To that end, we are setting up a group-wide code repository. See the next section (Where can I get it?) for more. If you find yourself needing new functionality, please talk to one of the collaborators on the SetList code repository about how it can be achieved without a permanent fork for your lab!

For a detailed description of the group approach to experimental timing and control, see Approach.

Where can I get it?

Please note: For acceptable graphical (UI) performance, we need to use LabVIEW 2013[15)] (32-bit to work with present PulseBlaster API). Prior versions of LV took *many ms* for each table cell coloring operation! LabVIEW 2013 can be downloaded through the UMD site license (Terpware) or the NIST site license.

The SetList and LuaEvaluator are now hosted on the JQIamo organizational repository on GitHub. Presently, the repository is public. Email Neal Pisenti or Zach Smith (zsmith12@umd.edu) with your GitHub username if you'd like to become a code contributor, and they will add you as collaborator.

Anyone can clone the repositories (SetList **and** LuaEvaluator) to get started using SetList in their lab. It's probably easiest to set-up a GitHub account and download their Windows client (although command-line is there if you'd prefer). Alternately, you can use the remote URL and your preferred Git or even

Subversion client.

If you've gotten this far and have no idea what the last paragraphs mean, see these step-by-step instructions.

Releases

If you will only be using but not modifying SetList, and you'd rather stay at a fixed point on the revision history, we've set up a system for making & numbering releases, and you can always find the latest release at https://github.com/JQIamo/SetList/releases/latest. The description for each release should list the changes since the last release so you can tell what you are getting. The version numbers also carry meaning, the should be parse as v`Major`.`Minor`.`Revision`, where:

- `Major` is incremented only when we make not-backwards compatible changes (hopefully never)
- `Minor` is incremented for major features, e.g. a new Device is added, a new Column class is added, etc.
- `Revision` is for sets of bug fixes between minor releases. To keep from clogging the pipeline with a lot of releases, these will be bundled together *at most* once a week.

New Features

There are many new features which improve the functionality, flexibility, and usability of SetList. Below is a summary of the most requested:

- **copy/paste**: by block or row
- **undo/redo**
- **grouping**: Variably enable/disable sets of lines
- **functional variable sequencing**: No longer are you limited to linear ramps
- …

Also, check out UI tips for hints, tips, and tricks.

Automatic Mulligans

SetList now allows other programs to request mulligans over a TCP connection as specified at Mulligans. This effectively enables your imaging computer to automatically request a retake of a shot during a sequence.

Feedback via Variables

Also available is the ability to have an external program change the values/configuration of variables over a (different) TCP connection, as described in Variables. This opens up opportunities for automatic

optimization of parameters or adaptive sampling routines based on the results of your shots.

Analog Inputs as Shared Variables

Since v2.0.0, SetList has supported exporting Analog Inputs from NI Cards as shared variables, where they are available to the acquisition computer. Variables are automatically named based on device and channel names. See Analog Inputs

Bugs & Feature Request

With the code now hosted on GitHub, we have access to their issue tracker. More information is available on Bugs & Feature Requests

UI tips

Much of the new user interface features are in right-click “context” menus. These change with column, row/column header, etc. Hopefully, most things are intuitive, but if something takes you a while to figure out, please document it at UI tips.

For Programmers

While you can always dig into the block diagram to look at the (well-?)commented code, here's an introduction to the Structure. But if we did our job, there should be no reason to touch most of the underpinnings.

To add a new device you will need to provide (1) methods for your device and (2) initialization of your device appearance in table columns. For more details, see Adding A New Device.

In addition, a label specifically to point out potential pitfalls has been added to the issue tracker:

All Issues in JQIamo/SetList with label programmer warnings

- #94: Slow Table Performance Fix may need to be re-applied in the future. programmer warnings Reported by: ZJ on 2016/04/29 12:58
- #55: Update git remote programmer warnings Reported by: CHerold on 2014/09/19 08:44
- #53: It is possible to program infinite recursion when disabling other columns programmer warnings wontfix Reported by: ZJ on 2014/09/16 13:50
- #51: PreviewCell for digitals minor bug programmer warnings Reported by: CHerold on 2014/09/16 09:27

View this list on GitHub

Export SetList namespace

Export SetList Documentation as PDF

[15] see also repacking_lua for more about using future versions

LuaVIEW in SetList

String expression parsing has been done in the group control software with LuaVIEW for nearly 10 years. It is a port of the scripting language lua for LabVIEW, however it appears to have ceased active development around 2009. Lately, we've been kicking around alternate languages, e.g. Python, however existing LabVIEW pluggability is reportedly pretty slow.

Here, the minimal use of LuaVIEW in the new SetList is described.

Packed library

The LuaVIEW distribution is pretty large and in order to minimize what needs to be loaded and to make it easier for end users, we placed what is needed into a LabVIEW packed library (SetList/codes/PackedLibraries/LuaEvaluatorAPI.lvlibp). As a result, there is no need to install LuaVIEW on each end-user machine.

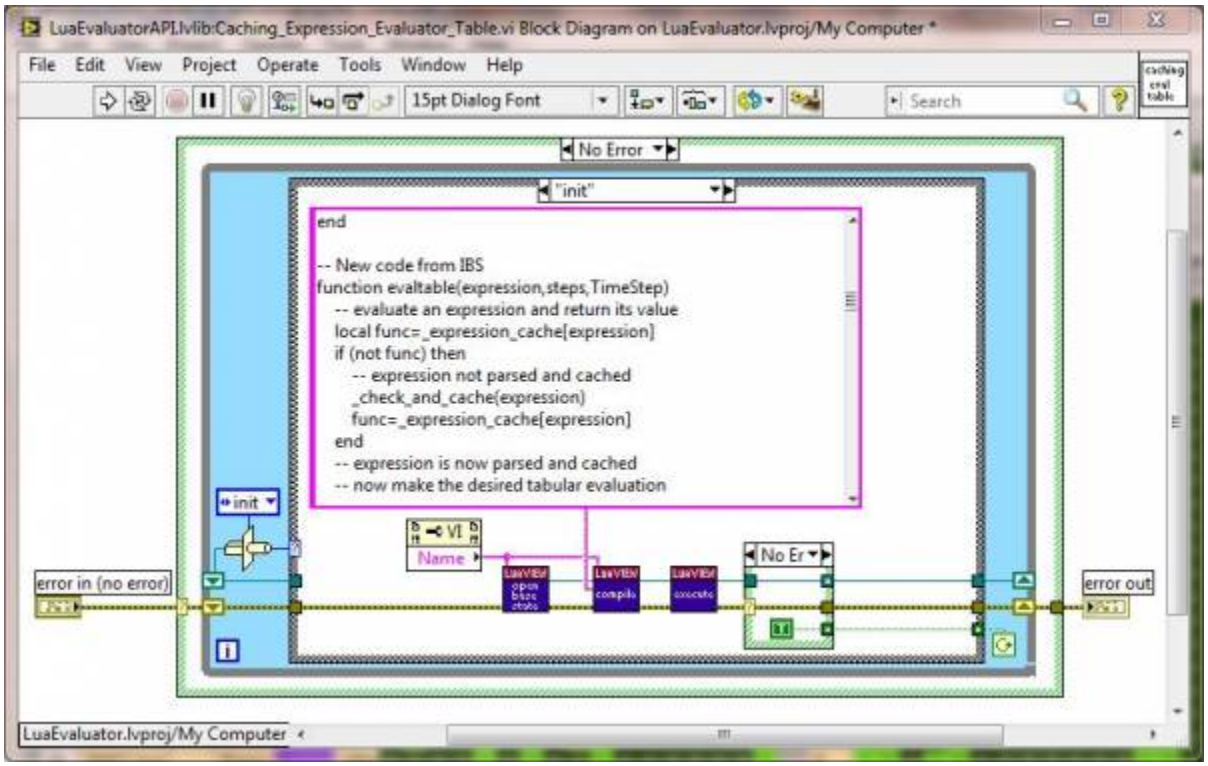
Packed libraries only work in the LabVIEW version for which they were built (2013 in this case); see instructions on repacking Lua for another version.

LuaEvaluatorAPI

Historically, two separate VIs were called, both of which are modified from LuaVIEW methods: Caching_Expression_Evaluator_v4.vi and Caching_Expression_Evaluator_Table.vi. They are configured for non-parallel operation so that only one instance is ever called. Furthermore, they only run their init code (which creates a Lua state) ONCE EACH, after which they continue to act on the same state *until it is stopped*.

However, by calling two separate functions (with nearly duplicate functionality) TWO Lua states were being initialized! As a result, we couldn't take advantage of the fact that variables and functions are cached (since the two states don't talk to each other) and they were being reset every single time a cell needed evaluation. **Speed improvements are significant (better than 4x faster)** by removing unnecessary variable and function setting by only using one Lua state.

The full initialization code is below and highlights how the _Table version incorporates step size and total delay time into IBS's function `evaltable`. As this is the only addition, the full function is also shown after he block diagram image:

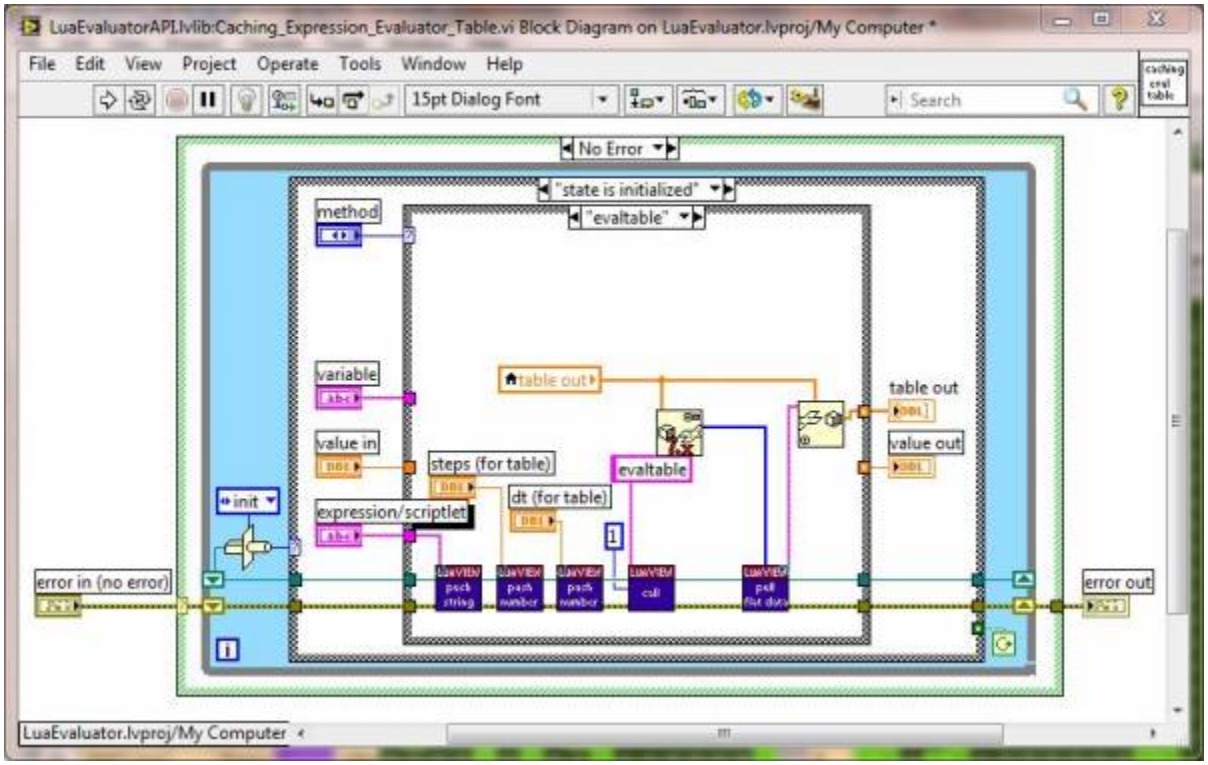



```
-- New code from IBS
function evaltable(expression,steps,TimeStep)
    -- evaluate an expression and return its value
    local func=_expression_cache[expression]
    if (not func) then
        -- expression not parsed and cached
        _check_and_cache(expression)
        func=_expression_cache[expression]
    end
    -- expression is now parsed and cached
    -- now make the desired tabular evaluation

    local i;
    local StepsSafe;
    answer={}

    dt = TimeStep;
    tMax = (steps+1)*dt;
    StepsSafe = max(steps,1);
    for i=0,steps do
           f = i/StepsSafe;
           t = i*dt;
        answer[i+1] = func()
    end
    return answer
```

```
end
```



Init code

While much of what is below was written for LuaVIEW, there are a few new pieces. One is marked — New code from IBS and defines the function `evaltable`. In addition, the variables f, t, tMax, and dt are pre-defined with value 0 (unnecessarily?) at the bottom.

```
function set(variable,value)
    -- set a variable to a value
    _G[variable]=value
end

function get(variable)
    -- get the value of a variable
    return _G[variable]
end

function _check_and_cache(expression)
    -- check an expression, if OK cache it and return its evaluation
    if (expression=="") then
        _error("empty expression string")
    end
    local func,err = _loadstring("return "..expression)
    if (not func) then _error(err) end
    if 1~=_table_getn({func()}) then
```

```
        _error("expression \""..expression.."\" must evaluate to a single
value. Commas are not allowed.")
    end
    local result=func()
    if _type(result)~="number" then
        _error("expression \""..expression.."\" does not evaluate to a
number")
    end
    _expression_cache[expression]=func
    return result
end

function eval(expression)
    -- evaluate an expression and return its value
    local func=_expression_cache[expression]
    if (func) then
        -- expression already parsed and cached
        return func()
    end
    return _check_and_cache(expression)
end

-- New code from IBS
function evaltable(expression,steps,TimeStep)
    -- evaluate an expression and return its value
    local func=_expression_cache[expression]
    if (not func) then
        -- expression not parsed and cached
        _check_and_cache(expression)
        func=_expression_cache[expression]
    end
    -- expression is now parsed and cached
    -- now make the desired tabular evaluation

    local i;
    local StepsSafe;
    answer={}

    dt = TimeStep;
    tMax = (steps+1)*dt;
    StepsSafe = max(steps,1);
    for i=0,steps do
           f = i/StepsSafe;
           t = i*dt;
        answer[i+1] = func()
    end
```

```
    return answer
end


function assign(variable,expression)
    -- evaluate an expression and assign it to a variable
    local func=_expression_cache[expression]
    if (func) then
        -- expression already parsed and cached
        _G[variable]=func()
        return
    end
    _G[variable]=_check_and_cache(expression)
end

function execute(statements)
    -- execute statements, e.g. setting initial variable values
    -- or defining custom functions
    local func,err=_loadstring(statements)
    if (not func) then
        _error(err)
    end
    func()
end

function isvalid(variable)
    -- check if a variable name is valid for assignment
    if _name_reserved[variable] then
        _error("\""..variable.."\" is not a valid variable name: it is
reserved for a predefined ".._type(_G[variable]))
    end
    if _string_find(variable,"[^_%w]") then
        _error("\""..variable.."\" is not a valid variable name: only
underscores and alphanumerical characters are allowed.")
    end
    if (not _loadstring("return {"..variable.."=42}")) then
        _error("\""..variable.."\" is not a valid variable name: leading
digits and Lua reserved names are prohibited.")
    end
end

--start with an empty expression cache
_expression_cache={}

-- hide functions and tables that do not belong in expressions
_error=error
_type=type
```

```
_table_getn=table.getn
_string_find=string.find
_loadstring=loadstring
retain={
retain=true,
set=true,
get=true,
eval=true,
evaltable=true,
assign=true,
execute=true,
isvalid=true,
string=true,
math=true,
}
for k,v in pairs(lv) do
    if (string.find(k,"I8") or string.find(k,"U8") or string.find(k,"I16") or
string.find(k,"U16") or string.find(k,"I32") or string.find(k,"U32")) then
        _G[k]=v
        retain[k]=true
    end
end
_name_reserved={}
for k,v in pairs(_G) do
    if _type(k)=="string" and not (retain[k] or math[k] or
string.sub(k,1,1)=="_") then
        _G[k]=nil
    else
        _name_reserved[k]=true
    end
end
math=nil
_name_reserved.math=nil
retain=nil
_name_reserved.retain=nil
string=nil
_name_reserved.string=nil

f=0
t=0
tMax = 0
dt = 0
```

Deprecated Feedback Interface

This feedback interface is deprecated as of v2.0.0[16]. For up-to-date documentation, please see Feedback.

Variables

Sometimes one desires the ability to change the procedure based on the results of incoming data. This could be used for anything from autotuning AOM frequencies to adaptive sampling during a data-gathering run. To allow these kinds of activities to happen in an automated way, we've implemented a way to communicate new values for variables over a standard TCP connection.

Once a connection is made, it expects to receive a Big Endian 32-bit signed integer (Labview type I32) giving it a command, followed by some number of bytes of data (set by the command). Depending on the data received, it will send back a two-character ASCII response, followed by an I32.

The commands allow you to build a set of variable names and values that SetList will store in a queue, applying those sets to the next cycle to run.

TCP Settings

SetList is setup to listen on port #55928 which is in the IANA range of private ports and thus should always be available.

The IP address will be whatever the local IP is of the machine is running SetList.

SetList will expect the other party to close the TCP port when it is done[17], if you do not do this the feedback system may hang for quite a while waiting for a response. Issuing a Close Connection command will signal SetList to stop listening for data, but it still expects the client to close the TCP on its own.

Protocol

The commands section details the command codes and data format SetList will expect, and the Responses section details the possible response codes and their meaning.

Commands

This is the list of command integers you can send over the TCP connection to SetList.

Code	Command	Data Field[18]		Description
Control Commands				
OK	**OK**?	*No Data*		Response is either “OK” or “ER” followed by an **I32** representing the number of errors since the last OK? check.
QU	**QU**iet	*No Data*		Stop responses from SetList, except for “OK?”. Will still send “OK” to confirm that “QU” was processed successfully.
LD	**L**ou**D**	*No Data*		Turns off “QU”, response as in “OK?”.
Set Commands				
SS	**S**tart **S**et	**I32**	# in Set	Marks the beginning of a variable set, and lets SetList know how many variables to expect. If called without ending the previous set, it is dropped silently.
CS	**C**ancel **S**et	*No Data*		Drops whatever set is currently in progress.
EN	**E**nd **N**ow	*No Data*		Ends the current variable set. If the total number of variables matches what was given by **SS**, SetList will immediately execute the changes laid out in the variable set. If not, will report either **EE** or **ME** and drop the set.
ES	**E**nd After **S**equence	*No Data*		As in **EN**, but instead of applying the changes immediately will wait until the end of the currently running sequence to do so. Sets closed with **ES** are added to a queue, so you can do this many times.
NN	**N**ext **N**ow	**I32**	# in Next Set	As in **EN**, but immediately start a new set as in **SS**.
NS	**N**ext After **S**equence	**I32**	# in Next Set	As in **ES**, but immediately start a new set as in **SS**.
SZ	**S**i**Z**e	*No Data*		Asks SetList how many variables it is currently aware of. See the **SZ** response.
Variable Commands				
DV	Set **D**ouble **V**ariable	**Byte**	Change Flags	Sets parameters for the named variable, masked by the Change Flags byte as described below. The “String Value” section currently will have no effect on SetList, as this functionality does not exist.
		String[19]	Variable Name	
		DBL	Default Value	
		String	Sequence Function	
SV	Set **S**tring **V**ariable	**Byte**	Change Flags	***Not Supported by SetList*** Sets parameters for the named variable, masked by the Change Flags byte as described below. The “String Value” section currently will have no effect on SetList, as this functionality does not exist.
		String	Variable Name	
		String	Default Value	
		String	Sequence Function	

The Change Flag is a single byte whose bits are encoded as bF X I S CI CS CD CSF

Responses

Each response code indicates the action taken by SetList in response to the command. It also includes data to help debug any problems that occurred.

Code[20)]	Data Field [21)]	Meaning	Action Taken
OK	Echo of received command code	**OK**	Commanded action
ER	Number of errors since last “OK?”	**ER**ror(s)	No action requested
TO	Echo of received command code	**T**imed **O**ut waiting for the rest of the command	Closes TCP connection after a short wait
NC	Always 0	**N**o **C**ommand: Timed out waiting for the next command	Closes TCP connection after a short wait
BC	Echo of received command code	**B**ad **C**ommand: Received an unknown command code	Ignore last command
NS	Echo of received command code	**N**ot **S**upported: Command code for an unimplemented command	Last command was read in as specified, but will not be processed downstream.
ME	Number of elements received	**M**issing **E**lement: Received fewer total variables than told to expect	Drops last variable set
EE	Number of elements received	**E**xtra **E**lement: Received more total variables than told to expect	Drops last variable set
SZ	Number of variable received	**S**i**Z**e: Size of current variable set.	No action, response to SZ command

Mulligans

If your data processing has the ability to detect a bad shot, you may desire the ability to automatically inform SetList to retake that point.

TCP Settings

Mullgians are setup to listen on port #50291 which is in the IANA range of private ports and thus should always be available.

The IP address will be whatever the local IP is of the machine is running SetList. SetList will expect the other party to close the TCP port when it is done.

Protocol

The protocol for the Mulligans mechanism is simple: Once the connection is opened, SetList opens for Big-Endian I32 values, echoing them back as it receives them, until the connection is closed. Once the connection closes it goes back to listening for a new connection at the designated port.

Example Clients

`codes→Utils→ExternalFeedback→ExampleClients` contains subdirectories with simple example clients for:

- Igor
- LabView
- Python

which should be enough of a starting point to either integrate into your data acquisition package or build a client in some other language.

Why TCP?

Because TCP libraries exist for many programming languages (Python 3.5, Python 2.7, C++, IGOR, MatLab)[22] in addition to LabView, this allows you to "close the loop" however you wish.

LabView has a built-in TCP Listener which waits for incoming connections on a specified port before handing the connection off to your program to handle. This allows us to do a low-overhead loop that just sits and waits for outside data or to be shutdown.

[16] specifically since commit 3507fb0
[17] Except for when it issues a "TO" or "NC" response, see the Protocol section
[18] All types are to be communicated in Big-Endian form
[19] Encoded as an I32 giving string length followed by ASCII characters
[20] Always a 2-character ASCII string
[21] Always a Big-endian I32
[22] These are the results of a quick Google search, no guarantee implied

SetList Preferences

Adding a Preference

1. Add new preference data to `SetListPrefs→PrefCluster.ctl`. If your preference will require only one piece of data, add that, otherwise add a cluster for all the data related to that function. Name the data/cluster something descriptive in CamelCase.
2. Add a short description to the data/cluster in the “Description and tooltip” right-click menu.
3. Set the defaults for the preference in `SetListPrefs.lvclass`'s private data using the using “Use current values as default” methodology.
4. Add read/write accessors to the class to use in the main program. For organization, create a sub-virtual-folder to group just the accessors for that particular preference.
5. Add whatever front panel interface you'll want to the “Preferences” tab of `SetList.vi`
6. Doing the above will cascade a bunch of TypeDef changes withing the SetListPrefs class, so ***make sure to save all the modified files*** before committing/pushing the result.

Once you've finished adding your preference, you'll want to:

Using Preferences

There is a global class in `Globals→PrefsGlobal` that holds the current preferences in memory while SetList is running. You can use this to read/write values whenever you like, with one important exception:

When the program first starts, it need to initialize the preferences before you can use their stored values. To facilitate this process, there are two VIs provided:

- `GetPrefsAfterInitDone.vi` waits for initialization to finish, then returns the now initialized global preferences on its output.
- `WaitForInitDone.vi` is pass-through for the Preferences terminals, but still waits for the init signal before it finishes execution. You can use the error out line on this to hold execution of other VIs.

Warning: we are not guarded against race conditions when accessing the global. Be careful about when you are choosing to write data.

Preferences are only saved to file when the quit button is pressed.

Removing a Preference

1. Copy the preference name into a new entry of the “KnownOldPrefs” string array in `SetListPref.lvclass`'s private data.
2. Delete corresponding accessor(s) from `SetListPref.lvclass`, if present.
3. Clean whatever configuration interface exists off of the preferences tab.
4. Delete the corresponding cluster from `SetListPrefs→PrefCluster.ctl`
5. Doing the above will cascade a bunch of TypeDef changes withing the SetListPrefs class, so ***make sure to save all the modified files*** before committing/pushing the result.

XML Preferences Format

The Preference class handles reading and writing the XML file for you, so you should never need to edit it manually. Nevertheless, an effort has been made to make it human-readable. On load the file is checked against a schema to ensure it is properly formatted, and an error will be reported if it is not.

The XML schema used to validate the preference file is included with SetList, but for completeness we'll outline the structure of the file.

The root-level tag is `<SetListPrefs>`, and it has several attributes to setup how the XML Parser will interact with it. For the actual preferences it contains elements (in order):

- `<DocLoc>`: Locations of documentation for the file format
  - `<WikiURL>`: A link to this page, the location of format definition on the wiki
  - `<LocalCopy>`: A link to a local markdown file containing the same documentation
- `<ActivePrefSet>`: The `<Name>` of the currently active PrefSet. Must match the `<Name>` of one of the `PrefSets`.
- One or more of `<PrefSet>`: The actual stored preferences.
  - `<Name>`: The name of this preference set. Should be in CamelCase. (this is what is referenced by `<ActivePrefSet>`)
  - Zero or more of `<PrefItem>`: Individual elements of the Preferences cluster in LabView
    - `<Name>`: The name of this item, pulled from LabView's label. Should be in CamelCase.
    - `<Description>`: The description of this item, pulled from LabView's description.
    - `<ModTimestamp>`: The last time this particular item was saved to file. Given as in ISO-8601 form.
    - `<PrefKnown>`: One of:
      - `Current` meaning this was a preference item in-use at time of saving
      - `Old` meaning this item is not is use, but a known former preference item
      - `Unknown` meaning this item is neither in use nor a known old item
    - `<LVData>` contains the XML form of the preference item's value generated by LabView

Example Preferences file

For reference, here is a properly formatted preferences file. It was in active use when pulled.

exampleSetListPrefs.xml

```
<?xml version="1.0"?>
<SetListPrefs xmlns:jqisetlist="https://jqi-
wiki.physics.umd.edu/d/documentation/software/computercontrol/setlist/home
"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="https://jqi-
wiki.physics.umd.edu/d/documentation/software/computercontrol/setlist/home
setListPrefs.xsd"
xmlns="https://jqi-
wiki.physics.umd.edu/d/documentation/software/computercontrol/setlist/home
">
    <DocLoc>
        <WikiURL>https://jqi-
wiki.physics.umd.edu/d/documentation/software/computercontrol/setlist/pref
erences</WikiURL>
        <LocalCopy>./XMLWikiPage.md</LocalCopy>
    </DocLoc>
    <ActivePrefSet>Default</ActivePrefSet>
    <PrefSet>
        <Name>Default</Name>
        <PrefItem>
            <Name>StartupWindowPosition</Name>
            <Description>Sets a standard position for the main SetList
window to snap to, and whether or not this should happen automatically at
startup.</Description>
            <ModTimestamp>2014-10-03T13:14:28-04:00</ModTimestamp>
            <PrefKnown>Current</PrefKnown>
            <LVData>
                <LvVariant>
                <Name>Value</Name>
                <Cluster>
                <Name>StartupWindowPosition</Name>
                <NumElts>2</NumElts>
                <Cluster>
                <Name>WindowBounds</Name>
                <NumElts>4</NumElts>
                <I16>
                <Name>Left</Name>
                <Val>110</Val>
                </I16>
                <I16>
                <Name>Top</Name>
                <Val>117</Val>
                </I16>
                <I16>
                <Name>Right</Name>
```

```
                <Val>1362</Val>
                </I16>
                <I16>
                <Name>Bottom</Name>
                <Val>886</Val>
                </I16>
                </Cluster>
                <Boolean>
                <Name>ResetPositionAtStartup</Name>
                <Val>1</Val>
                </Boolean>
                </Cluster>
                </LvVariant>
            </LVData>
        </PrefItem>
        <PrefItem>
            <Name>HiddenDevices</Name>
            <Description>Holds the names of devices that shouldn't show up
in the Add Devices menu.</Description>
            <ModTimestamp>2014-10-02T20:25:16-04:00</ModTimestamp>
            <PrefKnown>Current</PrefKnown>
            <LVData>
                <LvVariant>
                <Name>Value</Name>
                <Array>
                <Name>HiddenDevices</Name>
                <Dimsize>4</Dimsize>
                <String>
                <Name>String</Name>
                <Val>DDS</Val>
                </String>
                <String>
                <Name>String</Name>
                <Val>Novatech</Val>
                </String>
                <String>
                <Name>String</Name>
                <Val>PTS via PB</Val>
                </String>
                <String>
                <Name>String</Name>
                <Val>SRS DS345</Val>
                </String>
                </Array>
                </LvVariant>
            </LVData>
        </PrefItem>
```

```
        <PrefItem>
            <Name>AutoPressHWUpdate</Name>
            <Description>Sets whether the user must press the 'Update HWI'
button or if any table changes are automatically processed.</Description>
            <ModTimestamp>2014-10-02T20:25:16-04:00</ModTimestamp>
            <PrefKnown>Current</PrefKnown>
            <LVData>
                <LvVariant>
                <Name>Value</Name>
                <Boolean>
                <Name>AutoPressHWUpdate</Name>
                <Val>0</Val>
                </Boolean>
                </LvVariant>
            </LVData>
        </PrefItem>
        <PrefItem>
            <Name>DetachVMUIonStart</Name>
            <Description>Sets whether the variable manager starts out in
the subpanel of popped-out.</Description>
            <ModTimestamp>2014-10-03T13:14:38-04:00</ModTimestamp>
            <PrefKnown>Current</PrefKnown>
            <LVData>
                <LvVariant>
                <Name>Value</Name>
                <Boolean>
                <Name>DetachVMUIonStart</Name>
                <Val>0</Val>
                </Boolean>
                </LvVariant>
            </LVData>
        </PrefItem>
    </PrefSet>
</SetListPrefs>
```

Program only changes

While writing many table lines takes less than a second for most devices, we require the ability to only reprogram slow devices when absolutely necessary. For example, the Novatech can support 2^{15}=32,768 outputs for each of two channels, but with only serial communication, the time it takes to write such a program to the device is prohibitive. Luckily, the Novatech supports line-by-line modifications so a long program can be written once (slow) and modified (quick, provided changes are line-wise only).

As programmed currently, devices (objects) prepare a “hardware image” or HWI (PrepareHWI.vi, FinalizeHWI.vi), which, along with communication information, is actually programmed to the physical hardware in the LoadHardware.vi method. Then, after waiting an appropriate time[23] Clear Hardware.vi is called for each device. Presently, many devices close their “task” or communication channel. This requires the NI cards to always be reprogrammed since the “task” had been deleted. In a future implementation, it would be possible to leave open a “task” (stop and restart if no changes) until it needed to be reprogrammed.

As a result, of the implemented devices (PulseBlaster, Novatech, NI card, NIST DDS box) only the Novatech and DDS are able to implement “program only changes.”[24]

Novatech

To implement line-by-line functionality, Novatech/Load Hardware.vi tracks the “last HWI” for each unique device it encounters. Then, if “Program only changes” is requested, the present HWI is compared to last line-by-line. Differences are marked in a “skip” array and the two tableable channels are selectively reprogrammed.

However, even if NOTHING changes, some serial commands still MUST be sent presently:

- Nova Init: ensure fastest possible serial communication (3 writes, 2 reads)
- Nova FreqPowPhase: called 4 times to set static output for each channel (only really necessary for channels 2 and 3; 4 x 3 write, 3 read = 12 write, 12 read). These could be called selectively after checking for a change.
- Nova Table mode: set the tableable outputs for channel 0 and 1
  - “M 0”: table programming mode (1 write/read)
  - (line-wise setting of any changed table lines; 2 writes per line, 1 each per channel)
  - “M t”: start tabled output (execute first line then wait for trigger, provided duration = 0xFF; 1 write/read)

Thus, the required overhead is 17 writes and 16 reads even if no table lines are programmed.

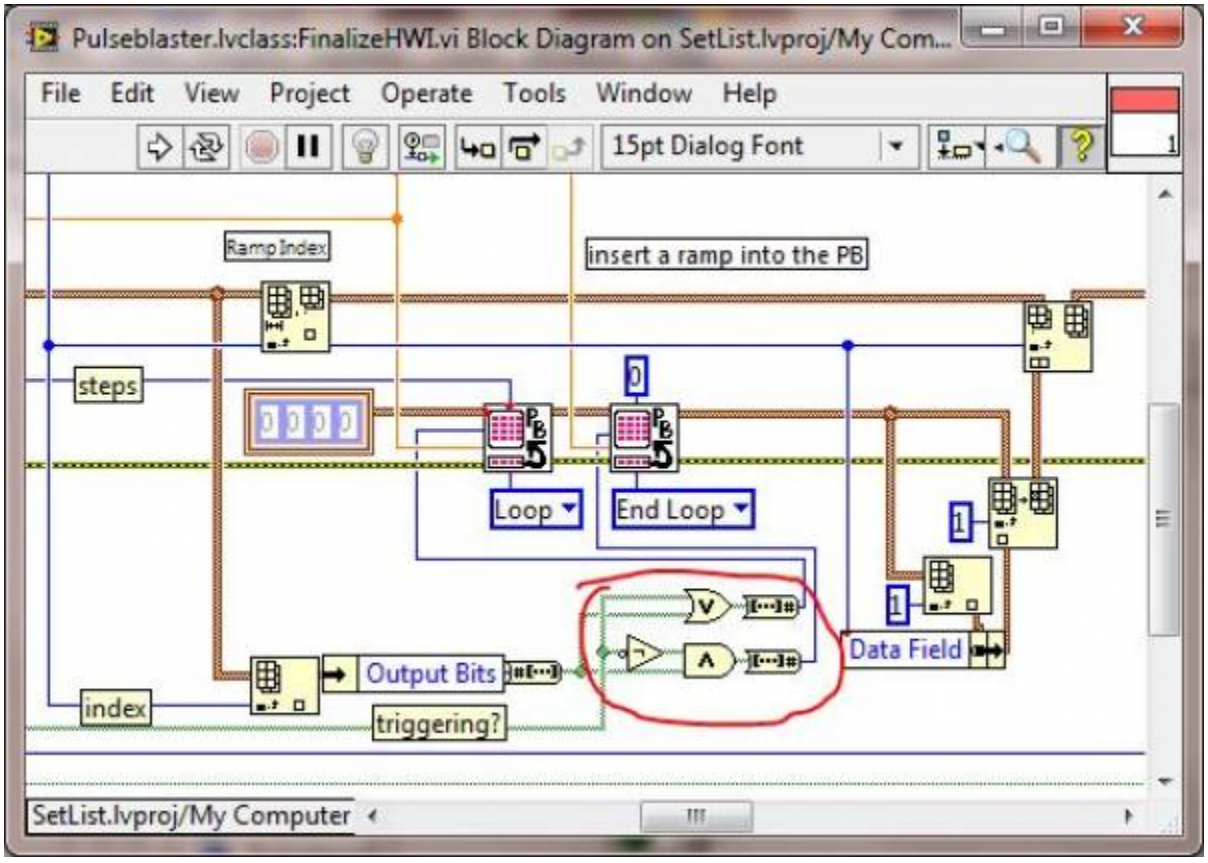
[23] Only accounts for the time to complete the requested procedure, but the user can add “additional delay” to ensure devices finish before they are stopped
[24] As of 7/9/2014, this DDS functionality was untested.

PulseBlaster

Throughout many years, the group has relied on SpinCore PulseBlaster (PB) cards to provide digital outputs and triggered timing for BEC experiments. There are many card versions around which will probably result in proliferation of PulseBlaster device classes if their SpinAPI's are incompatible. **However, this no longer necessitates a fork of the SetList; rather it requires that each lab instantiate the correct PB object version.**

Trigger Polarity

A number of people have asked about variable polarity for trigger lines. Presently, every “Ramp” line is implemented as a Loop/EndLoop so that the PB compactly executes the number of steps requested. For standard (non-triggering) outputs, Loop and EndLoop are each fed the same value from “Output Bits,” while for a line where “triggering?” is true Loop is HIGH and EndLoop goes LOW. The logic is highlighted below:



Since we need to set output on Loop and EndLoop, the logic corresponds to two truth tables.

Loop: out \ trig?	**0**	**1**
0	0	1
1	1	1

EndLoop: out \ trig?	**0**	**1**
0	0	0
1	1	0

If one wants to force rising-edge triggering, it is apparent that one sends “out” OR “trig?” to Loop and “out” AND NOT(“trig?”) to EndLoop.

However, for selectable polarity the truth tables could be modified so that the “output” of a trigger line defines it's “off” state (i.e. 0 for rising edge, 1 for falling edge):

Loop: out \ trig?	**0**	**1**
0	0	1
1	1	0

EndLoop: out \ trig?	**0**	**1**
0	0	0
1	1	1

Then, Loop gets passed “out” XOR “trig?” while EndLoop simply gets “out.” If this change is implemented, one should look at the TriggerCol column class AND the way PB initializes (Pulseblaster.lvclass/InitDevice.vi and /ManageDevice.vi) so provide the user with a protected way to modify the polarity.

LuaEvaluatorAPI.lvlibp

Presently, LuaEvaluatorAPI.lvlibp created with 2013 will NOT work with 2014.[25)] Repacking the LuaEvaluatorAPI packed library, is fairly easy but must be done for each version of LabVIEW. If the packed library isn't from the correct version, LabVIEW won't be able to load the project.

To make a packed library, open the LuaEvaluator repository in the desired version of LabVIEW. In the Project Explorer, expand “Build Specifications” and right-click “LuaEvaluatorAPI” and choose Build. LabVIEW will make a packed library in the folder LuaEvaluator/LuaEvaluatorAPI. All that remains to integrate this with SetList is to replace the file at SetList/codes/PackedLibraries/LuaEvaluatorAPI.lvlibp with your new one.

Joe Tiamsuphat figured out how to pack the library, so you could also ask him if you have any trouble.

Unless other packed libraries are used, all other files should open correctly in newer versions of LabVIEW.

[25)] Zach Smith may find time to look into making the library more portable through preferences

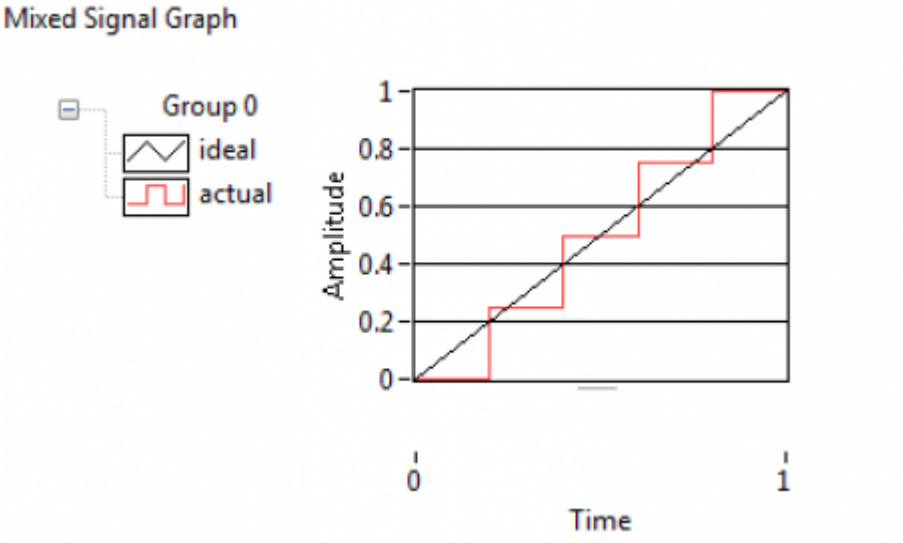
SetList FAQs

These questions come up frequently. While some of them are addressed on the “About SetList” tab within SetList, the wiki affords more room for detailed explanation.

How are Ramps constructed?

Any “rampable” output parses it's table contents through the Lua “Ramp Generator.”

Here's a discretized linear ramp compared to the exact line from 0 to 1:



This was generated with the SetList table

Mode	Delay	Step	Output
Ramp	1 s	0.2 s	LineRamp(f,0,1)

As you can see from the figure, the output is stepped through five steps (`npnts = floor(delay/step)`) with `stepsize = delay/npnts` and each endpoint is *included* in the output points. In the case above, the `step` divided the delay without a remainder. If the floor rounds down, the actual `stepsize` will be *larger* than that specified in `step`.

Ramps pre-define four variables

- `f` - fractional position within a ramp
- `t` - actual time within a ramp (at *evaluated* points)
- `dt` - time between *evaluated* points
- `tMax` - total time for Ramp

The requirement that both start and end points be points in the ramp causes `dt` to differ from the `stepsize`, with `dt = delay/(step-1)`. If you look closely, these points `{0,0.25,0.5,0.75,1}` are the five points where the discretized LineRamp actually samples/intersects the line. However, these are not the times when these sampled values are output! The disconnect comes between calculating output values and when they are actually triggered.

Squinting at the discretized output, there is clearly a bias to be *below at early times* and *above at late times* due to forcing the endpoints to be output with equal `stepsize` duration. This approaches the straight line with fine enough sampling; in some sense the effect of the bias is to increase the slope of the ramp while adding a hold at the start and end points.

In summary, below are listed the calculated values[26] output at the trigger times:

Trig. time	f	t	dt	tMax
0	0	0	0.25	1
0.2	0.25	0.25	0.25	1
0.4	0.5	0.5	0.25	1
0.6	0.75	0.75	0.25	1
0.8	1	1	0.25	1

Think of `dt`, `f`, `t` as the *sampling* interval, fraction, time.

[26] `f` and `t` only match because the delay time is 1 s.

SetList Structure

This is intended as a high-level overview of the program structure for SetList. Is is not even necessary for the typical device programmer to understand anything but the Device Classes and UI Column Classes.

Even experienced programmers may benefit from looking at LabVIEW Object-Oriented Programming FAQ, as some of the ways LabVIEW has implemented object-oriented programming are non-intuitive or counter to other languages. The project explorer is the main place to navigate the SetList project and it's various class definitions.

Project Explorer

- brief description, NI link
- hints for navigating classes:
  - adding_a_new_lvclass
  - hierarchy tree: right-click a .lvclass file in the project explorer and select “Show Class Heirarchy”
  - right click for add new VI from template
  - viewing entire private data for class (need to look at all ancestors too!)
- …

Block Diagram

The block diagram contains three main loops:

- UI input event handling
- Hardware Update
- Sequencing

Device Classes

As it presently exists, the device class hierarchy is

- Device
  - Master Device
    - PulseBlaster
  - NI card
  - DDS
  - Novatech

A Device (or one of its children) consists of “Private Data” and a set of “Methods.” The Device class defines a data and method structure that each child can inherit, override and/or augment. In very broad strokes, the data owned by each instance of a child device is its

- portion of the Setlist table or software image (SWI, human readable)
- communication method
- hardware image (HWI, device readable)
- Column Object array

When possible, general methods are inherited by all devices e.g. for reading/writing SWI, using Columns. Each specific child device has its own implementation of methods like “PrepareHWI.vi” and “Load Hardware.vi,” which interpret the SetList table strings with LuaVIEW. The big improvement in flexibility comes in the class organization–we can dictate what methods each device must provide even if we don't know a priori how they will work. This allows one to program “read SWI and prepare HWI” as a loop over devices whose specific implementation of a method are “dynamically dispatched” at runtime.

This array of devices contain almost all information needed to program an experiment. Additional components which are bundled into a cluster and saved together are Layouts, User Defined Functions, and the Variable Manager.

UI Column Classes

The user interface (UI) for programming devices is the SetList table. At its core, it is simply a 2D string array. Programming everything as a string means that any element can (in principle) be a variable; while this has long been true for Analog outputs, this is also now true of Digital outputs. The Column classes provide a UI skin to the string table to provide some error checking and simplify the programming tasks.

The column class hierarchy is

- **Column** (general one-color column)
  - **DigitalCol** (change color for high/low/variable)
    - **GroupCol** (additional methods for row-wise “grouping” to enable/disable sets of instructions)
  - **SingleValCol** (for static outputs which cannot be updated during a cycle)
  - **ModeCol** (for Pulseblaster, this column restricts input to predefined modes)
  - **TriggerCol** (simplifies non-programmable column display, like a master trigger line)
  - **MenuCol** (for picking from a drop-down menu on a mouse-click)
    - **MenuDisableCol** (for when your menu able to selectively enable/disable neighboring cells)
    - **SingleValMenuCol** (for static outputs picked from a menu)

Every object contains a Column Array in its private data which defines that object's appearance within the SetList table. The column classes contain methods to handle keyboard and mouse input and display “layouts.” Hopefully, any future device SWI programming functionality can be built from these classes, but a new child class might be required.

Hardware Programming

Variable Manager

The Variable Manager handles both static and sequenced variables (which have been combined). The “VariableManagerCodes” take care of may things including

- Controlling sub-panel window within Variables tab
- interfacing with variables directly from SetList table through dialog box
- multi-dimensional sequencing with *functional* point-spacing
- providing current variable list to HWI parser

SetList-Level Preferences

These are preferences that apply at the lab level, rather than to individual procedures. These are saved in an XML file that is loaded and parsed at run-time. It is human-readable but you shouldn't ever need to look at it.

The system and how to use it is described in more detail at SetList Preferences

UI Tips for SetList

While the UI is intended to be as intuitive as possible, please place tips here if there's something that took you a while to figure out how to do. Additionally, if you've discovered some way of using the UI tools that you think is neat and useful, tell us about it.

Left-clicking will allow you to edit/toggle individual cells in the SetList table. Right-click will bring up a context menu. This menu will change based on which column you click and whether you click

- the header (change entire column)
- a cell (insert, add/edit variable, delete, paste…)
- a selection (copy)
- below the table data (insert rows up to, **clear selection**…)

Selecting cells

- Left-clicking a row/column header will select the entire row/column. An additional shift-click will select the spanning rows/cols.
- Within non-digital cels, you can click-and-drag to make a selection
- For digital cells, click-and-drag will work inside the edit box (double click first if a 1/0).