Table of Contents

EXSCALABAR Software User Guide

This manual describes the usage and development of the software used on the UK Met Office EXSCALABAR instrument. The software consists of two separate codebases - a server side which communicates directly with the hardware via a PXI chassis developed in National Instruments LabVIEW (LV) and a client side which communicates with the server side via the http protocol and is written in a mixture of HTML, Javascript and CSS.

This manual covers the

- use of the client side software which focuses on the user interaction with the user interface.
- patterns and execution of both the client side and sever software
- tips for debugging both the client and server side software
- description of hardware

A Word about Nomenclature

Much of the software on both the client and server side are object oriented. As such, in many of the software descriptions, methods and parameters may be referred to using the following nomenclature - `Object::ObjectMethod` - which will be familiar to any developer who has worked with code in C++.

Rendering the Manual

This manual was written and compiled using Gitbook. The manual itself can be found here. The guide may be edited by those who have access to the manual via the web interface or [the command line tool or local editor] (https://toolchain.gitbook.com). Access to the code behind the book requires a Gitbook account (which can be linked to your Github account).

Using Git

All code in this project is housed in a git repository. Git is a distributed and free version control system. Git can be hosted locally, on a private server or on a variety of services provided free of charge such as github and gitlab. The code in this project is hosted publically at the MSR Consulting github site in two separate repositories.

In order to retrieve the code, the developer will need the git client on their computer. For Windows users, the best source for the Windows binaries can be found on git-scm. I am familiar with the command line tool (`git-bash`) so all references to git commands will be command line references. There is a gui available for Windows users but that is for the user to research if they wish to know how to use this tool.

The Server-side Code

The server side code consists of *mostly* LabVIEW code (the exception being the code concerning the web service client). To retrieve the current head of this project, simply run the command

```
git clone https://github.com/msr-consulting/exscalabar_server
```

This will create a folder in the directory that the user clones *into* called `exscalabar_server` . This will be a complete local copy of the git repository. This will also connect the local repository to the remote (the one hosted on github). You can find more details about cloning remote repositories here.

When the user makes changes that they want to add to the repo, they will first commit to their local repository and then push to the remote using the following commands:

```
git commit -m "This is a a commit message"
git push
```

The commit should have a useful message describing why the user is committing changes. You can reference issues raised in the repository by the issue number. For instance, if the developer is working on issues 119, then the commit message should look something like

```
git commit -m "Making some awesome changes in relation to #119"
```

This will create a link to issue 119 in the repository. The user can even close an existing issue with the message by using keywords such as `closes` or `fixes` before the issue number. So, to close issue 119 with the commit, the developer will add a message that looks like

```
git commit -m "This commit closes #119."
```

If the developer forgets to add a commit message with the `-m` flag, then the system will request one using a VI like environment. To insert text, simply press the `i` key and then begin adding a message. To save, press `ESC` and then `:w` . If the developer wishes to continue editing, press `i` again to insert text. To quit, press `ESC` and then `wq` - this will save the message and finish the commit.

If the code has been updated in the repository, use the following command to update the local (if no changes have been made)

```
git update
```

Client side code

The user interface code is also stored in a git repository on the MSR Consulting account at github under `exscalabar`. The user may use a similar command to the one above to get the repository

```
git clone https://github.com/msr-consulting/exscalabar
```

but this repository **contains submodules**. This mean that the clone will clone the structure but the folders containing the submodules will be empty. In this case, the folders that the submodules can be found in are `cm` (this is the context menu code), `ad` (this is the dygraph related code) and `cui` (contains custom user interface bits and pieces). If the developer uses the above command, then that command will have to be followed by

```
git submodule init
git submodule update
```

However, the developer *can* recursively retrieve *all* of the code using the command

```
git clone --recursive https://github.com/msr-consulting/exscalabar
```

The code found in these submodules are units that are not specific to the EXSCALABAR project. It is advised not to change the code as this will have a significant impact on the look and feel of the UI. If the developer wishes to modify these modules, it is advised that they read up on submodules at git-scm as this is beyond the scope of the current work.

Some useful commands

There are several additional commands that the user might find useful. First, as the LabVIEW code is mostly binary, merging in the repository is not usually possible. When the user finds themselves with changes that have been made unintentionally, they can reset the local copy to that of the head in the repository. To do this, use the following command:

```
git reset --hard HEAD
```

To checkout a particular version found in the repository, use

```
git checkout <commit>
```

where `<commit>` is the commit hash. Or you can go `N` versions back from the head using

```
git checkout HEAD-N
```

To create a new branch for experimental purposes (say if you want to try out the latest version of LabVIEW), then you would use the command

```
git checkout -b <branch-name>
```

where `branch-name` is the tag for the new branch. Once the branch is created, you can switch to that branch using

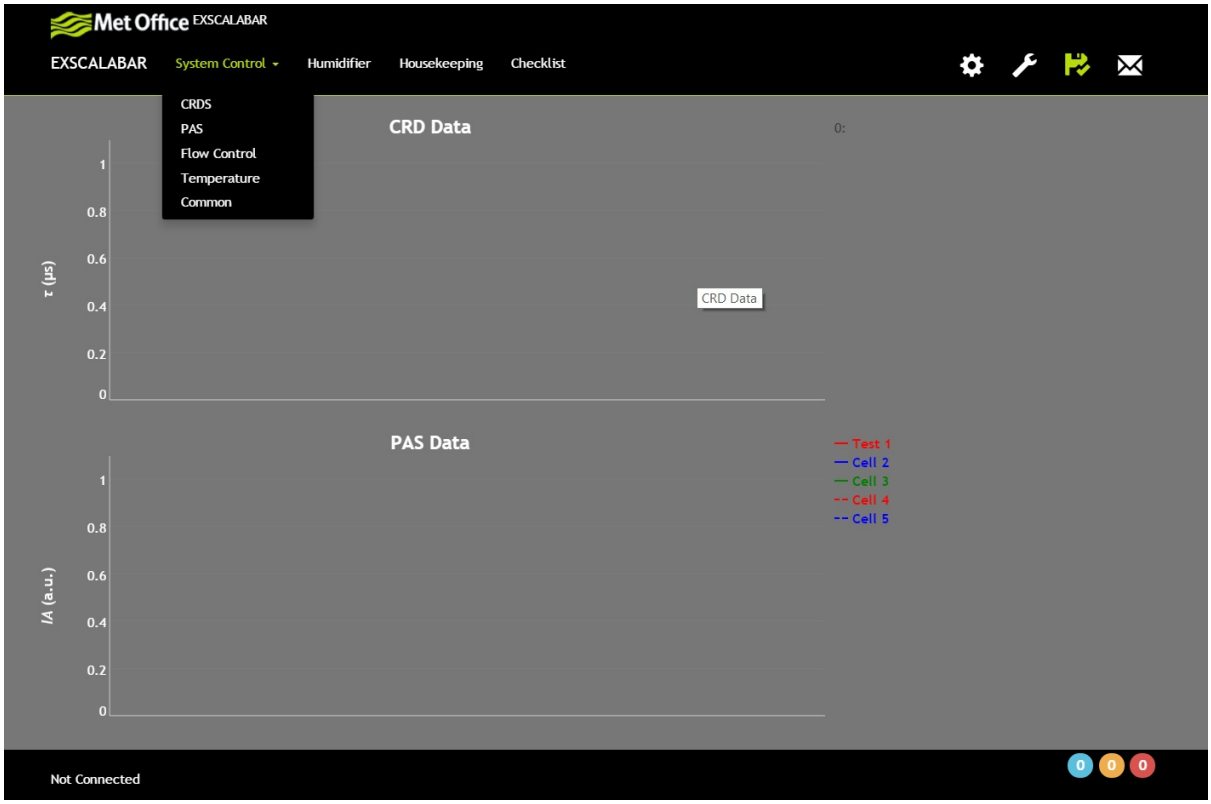
```
git checkout <branch-name>
```

to switch to that branch.

There is much that the user can do with a git repository. Check the manuals at git-scm for more information.

The User Interface

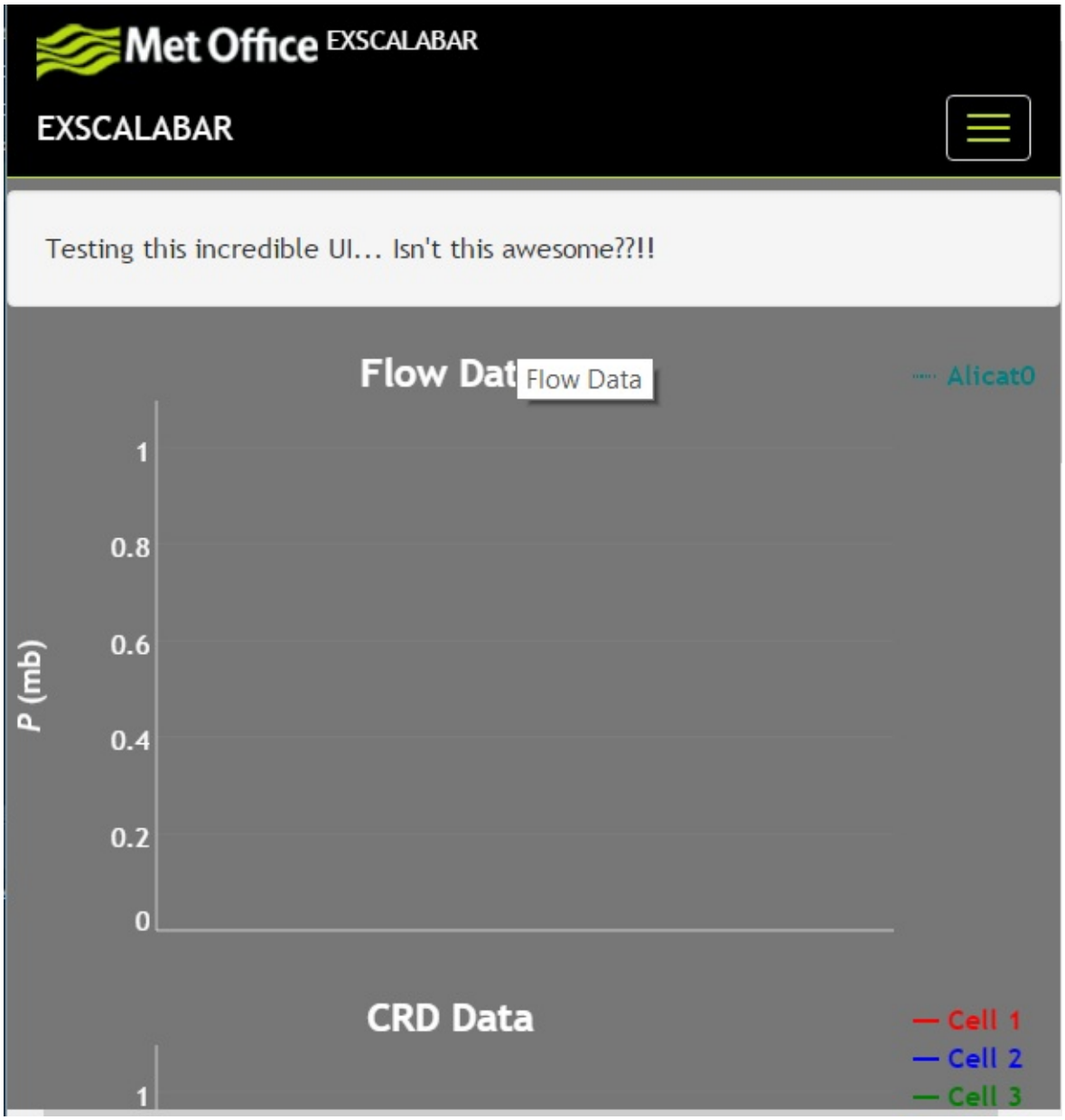
An overview of the user interface is show below. The user interface is a responsive, single page application (SPA) developed using the AngularJS framework and Twitter Bootstrap version 3. The former is a javascript framework that provides the functionality to develop an SPA while the latter provides the css and some javascript for styling the responsive behavior (more information concerning these technologies will be provided in the following chapters).



1: Overview

Responsive Behavior

A responsive user interface is one that will respond appropriately to changes in the window size. The EXSCALABAR user interface will resize user interface element at certain "break points". At each of these break points, the elements will resize to fit the window. At the last break point, the navigation menu will condense to a single "hamburger" icon in the top, right-hand corner of the screen as seen below.

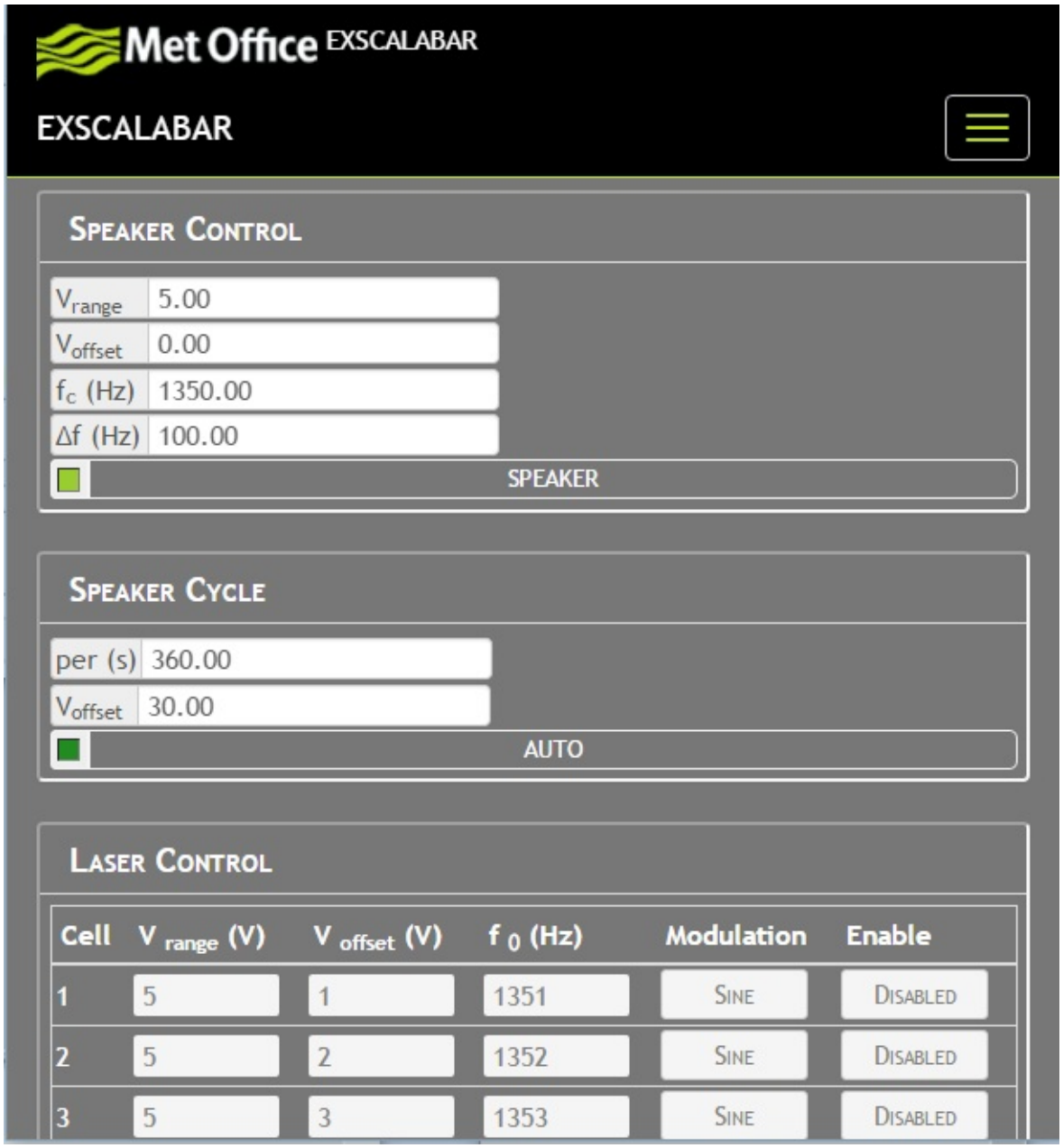


2: Responsive menu

To access the navigational menu items, simply click on the "hamburger" icon and the menu will drop down as seen below.

3: Expanded responsive menu

Controls and indicators on the page will adjust to accommodate new screen sizes as shown below.



4: Responsive controls

Configuration

The user interface uses a configuration file to determine how certain aspects are displayed. The file is written in javasctript object notation (JSON). The file is called `ui.json` and resides in the main client directory (i.e. same place as `index.html`).

This configuration file is used to primarily define how plots are rendered, but contains two additional entries:

- `name` - this is a string that defines system name (in this case this is EXSCALABAR).
- `version` - this defines the current version of the user interface. This value should only be adjusted in conjunction with the `package.json` file. The version consists of 4 numbers that follow the semantic versioning format (with the exception of the final number). The first number represents the major version, the second represents the minor, and the third represents the patch number. The final number represents the build number. The `gulp.js` file contains three tasks associated with the changing of the version - `bump-major`, `bump-minor` and `bump-patch`.

Configuring How Plots are Displayed

The configuration file determines how individual plots are rendered. At the time of writing this documentation, the configuration file contained the following keys for configuration plots:

- **crdplot**: This is the plot that contains derived CRD data such as ringdown times, extinction, etc.
- **pasplot**: Displays 1 Hz data from the photoacoustic spectrometer.
- **flowplot**: Displays 1 Hz data from the Alicats including pressure, temperature and flow rate.
- **ppt**: Displays 1 Hz data returned from the Honeywell pressure transducers. These meters return pressure and temperature.

Each section of the configuration file associated with plotting may contain the following entries for configuring the plots:

- `color` : array of strings containing colors for rendering the individual plots on the graph
- `strokewidth` : array of integers that contain the width of the strokes for rendering the lines.
- `pattern` : array of strings that represent the dygraph associated stroke patterns. The value can be `null`. If the value is `null`, then line will be solid. Available dygraph values are:
  - `DASHED_LINE`
  - `DOTTED_LINE`
- `ygrid` : boolean indicating whether to display the ygrid in the plot.
- `xgrid` : boolean indicating whether to display the xgrid in the plot.

For the array data, there does not need to be a 1:1 correspondence between the number of entities being graphed and the number of entries in the arrays. The plots will be rendered in order using the existing values and when there are no futher values to be used, the arrays will be recycled (i.e. the values will be populated by beginning at the front of the array and working backward).

As an example of this in action, let us say we have three devices with the following plotting entries:

```
"vaisala": {
    "color": [
      "red"
    ],
    "strokeWidth": [
      2
    ],
    "pattern": [
      null,
      "DASHED_LINE",
      "DOTTED_LINE"
    ],
    "xGrid": true,
    "yGrid": true
```

```
    }
```

In this case, we will plot three lines, all red with a stroke width of 2. The first line is solid (as indicated by the first entry in `pattern`, `null`), the second is dashed and the third is dotted.

Additional Plotting Inputs

In addition to the values settable by for all plots as defined above, the user may also configure the names of the CRD and PAS cell plots. In each of these objects, there is a field called `names`. This field represents an array of strings that will be visible to the right of the plot. These represent the names of the cells and the user may use this field to distinguish or provide specificity to the visual representation of the data.

Labels for individual devices are defined in the configuration file on the server.

Navigating the User Interface

The user interface can be divided into three separate areas:

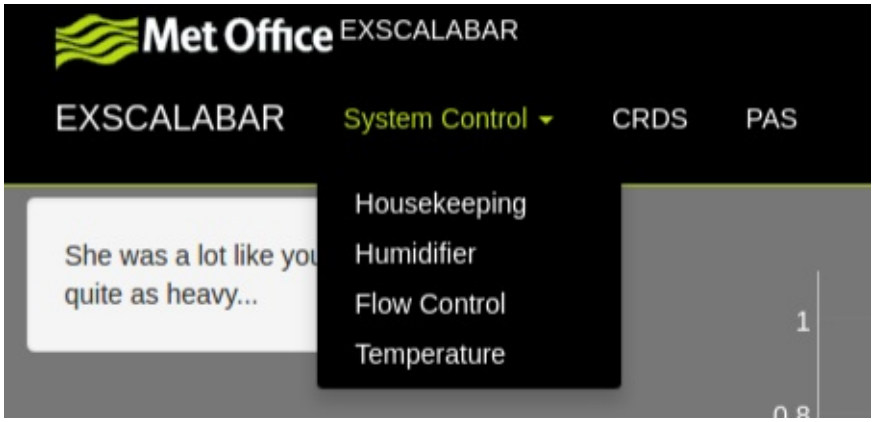
- the top portion contains all of the navigational input. Clicking on buttons or links in this region (with the exception of the save functionality described below) will display the page selected in the middle portion of the page. This portion of the page is *always visible*.
- the right hand side contains controls indicators that the user may require regular access to regardless of the current display.
- the bottom portion displays information regarding system status

The Navigation Bar

The navigation bar resides at the top of the page, is always visible and consists of two parts. The left hand side, shown immediately below, contains a dropdown menu for displaying functionality that is not necessarily core to the instrument under the heading `System Control`. Clicking on any of the items in the dropdown menu will pull up a page corresponding to the selection. The heading for the dropdown menu, `System Control`, is not selectable.

In addition to this dropdown menu, the left hand side contains two more selections. These two selections are for viewing core functionality for two separate instruments - the CRDS and the PAS. Clicking on either of these pages will bring up a page that will contain the control and data visualization.

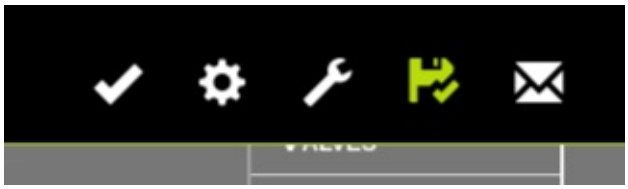
Finally, on the left hand side is also located a less obvious button with the label `EXSCALABAR`. Clicking on this button will display primary data for both of the instruments as well as the flow controller data. This page, which is hthe index page for the application, contains no control functionality other than what is provided on the right hand side of the page itself.



5: Left hand portion of the navigation bar.

On the right hand side of the navigation bar, there are five buttons that provide access to different pages (as shown below). They are defined in this order:

- A checklist page which allows the user to define a system checklist for operation. **This functionality has not been fully implemented yet.**
- A configuration page.
- A calibration configuration page
- A save button. This button will be green with a check mark if the system is currently saving data and red with an x mark if not.
- A system messages page



6: Right hand portion of the system navigation bar.

System Status Bar

The system status bar is located at the bottom of the page and contains two elements. The left hand side serves a dual purpose: when the system is connected, the server time is returned in the data packet. The time is updated regualarly. If the system is working properly, it should update on 1 second intervals. If it is not, the intervals will not be regular *or* there will be no update and time will appear to be frozen. When the system is not connected, it will display the text `Not Connected` . **It is important to note that if the system was previously connected and is no longer connected then the last time retrieved will be displayed but not updated.**

7: Server heartbeat.

The right hand side (shown below) displays message counts in the following order: system messages (blue), system warnings (yellow) system errors(red). The actual messages can be read on the message page which can be accessed through the mail button on the right hand side of the navigation bar (as shown above).

8: Message indicator.

Developing Operational Checklists

> The features described below are still in development. While they are utilizable, they still lack intended capabilities that make them more useful for actually using checklists.

Checklists may be accessed for operation via the user interface. These checklists are recorded in the JSON file format.
The format is written as an array of objects. Each object represents a section in the checklist. A checklist file will look like the following:

```
{
  "main": [
    {
      "title": "Consumable",
      "items": [
        "Check and record zero pressure.",
        "Turn on O2 cylinder.  Record O2 pressure.",
        "Replace scrubber as necessary (every 2 to 3 flights).",
        "Replace CLAP filter, if needed.  White side up.",
        "Clean impactor (20 to 30 flight hours)",
        "Check 2 water reservoir levels (clear reservoirs at rear of instrument)."
      ],
      "records": [
        {
          "name": "Pressure",
          "type": "numeric"
        },
        {
          "name": "Pressure",
          "type": "numeric"
        },
        {
          "name": "Replaced?",
          "type": "boolean"
        },
        {
          "name": "Replaced?",
          "type": "boolean"
        },
        {
          "name": "Cleaned?",
          "type": "boolean"
        },
        null
      ]
    },
    {
      "title": "Power Up",
      "items": [
        "Check that the 5 PAS Laser breakers are off.",
        "Check that bus 1 is selected for 60 Hz.",
        "Start instrument by turning on the AC master breaker. Wait for valves to cycle on before continuing. "
,
        "Start host on laptop. "
      ]
    }
  ]
}
```

Each object in the array contains an entry called `title`. This represents the section heading. The checklist for each section is defined by an array of strings that is called out by the entry `items`. On the user interface, this will be rendered as a list with each item preceded by a checkbox.

System Messages

System messages are generated by the server and returned in the data stream from the server under the entry `Msg`. This entry contains an array of strings representing these messages. The system messages may be viewed by clicking the mail icon on the right hand side of the navigation bar. System messages are usually green but may appear red in the case of an error or yellow in the case of a warning

Messages may arrive from a variety of places on the server. Each message string is preceded by a time stamp and then the location where the message was generated followed by the message itself. Aside from the color, error messages are also indicated by the bracketed word ERROR. The string will look like

where `systime` is the time stamp of the generated message, `location` describes where the message was generated and `msg` is the message text. The following represent some of the location strings that the user might see:

- `ppt` - Honewell pressure transducer
- `rh[#]` - Vaisala probe at address `#`
- `alicat` - Alicat flow device
- `mTEC` - Meerstetter TEC
- `TEC` - TE Technology TEC
- `ctl` - message produced by the `Controller` object on the server (this is the object through which all instrument communication is funneled)
- `instr` - generic message produced by an `Instrument Actor`
- `udp` - message produced by the `Network Actor`
- `pas` - message produced by the `ePAS` or `PAS` object
- `crd` - message produced by the `eCRDS` or `CRDS` object

System Configuration Page

The system configuration page is accessed via the cog on the right hand side of the navigation bar. This page contains three main elements

- a network settings box labeled `Network`
- three buttons to change the state of the server
- a table containing a list of devices and their current state

Changing the network settings

The user can change the settings for the network communication via the controls in the box labeled `Network`. This box contains two text fields and they are as follows:

- `IP` - this is the IP address of the server
- `Port` - port that the server web service is listening on

These settings are used to generate an http address that will look like `http://[IP address]:[port]/xService/`. This is the base address for all http communications with the web service. These values are stored using html5 local storage variables. These variables are labeled `ip` and `port`. Local storage is used to make sure that the address is maintained across browser sessions (the user will notice that, unless the cache is cleared, these values will be the same between sessions if the user does not change them).

Changing the server state

The three buttons are used to change the server state. If the server is currently running and communication has been established with the client via correct configuration of the network settings, the the LED indicator on the `Stop` button will be lit green. To stop the system, simply press this button. If the server is on but not running(as when the user stops the system via the `Stop` button), then the `Start` button will be lit green. To restart the system, press this button.

If the server is in need of a reboot, press the `Reboot` button. This will cause the web service to send a signal for a hard reboot. **Warning: This will put the system into an indeterminate state just as if the user had physically turned the chassis off and on.**

Device status

The device status table represents a list of all devices found in the INI file on the server and cross-checks this with the devices found in the CVT under the `device` heading. If a device is present in the INI file but not present in the CVT, this means that the device failed configuration (i.e. the server could not establish communication with it and it was booted from the system). The table will list

- `Device Type` - this is the type of device that the instrument is communicating with. This might be something like `alicat` for the flow controllers or `ppt` for the pressure transducers.
- `ID` - value found under the `IDs` key for the device section in the INI file
- `Label` - label in the INI file used for UI purposes associated with the device
- `Active?` - indicator displaying whether the device is present (green) or not (red)

Interacting with Plots

Plots on the user interface are based on the canvas plotter dygraphs. As such, the plots utilize many of the same interactive means that the dygraphs' API defines.

Zooming

Zooming in on points is accomplished via left mouse button. To zoom in on y-values (i.e. to decrease the y-range), simply left click and drag straight down. To zoom in on particular x-values, left click on the plot and drag right or left. To return the view to the original setup (i.e. autoscaled to the current view maximum and minimum), double click using the left mouse.

Right-click Menus

Each plot will contain a right click menu (or context menu). The right click menu is unique to each implementation of the plot, often containing a list of variables that may be selected for plotting.
In addition to the variables available for plotting, the right-click menu will contain three selections in common: `Autoscale`, `Autoscale 1x` and `Clear Data`. `Autoscale` sets an adjustable window for plotting along the y-axis. As the maximum and minimum within the plot window change, the window will adjust automatically to the new maximum and minimum values.

> **Code Relation**
>
> Within the code, this will set the plot limits for the y-axis to the values of `[null, null]`.

`Autoscale 1x` forces the plot to scale to the current maximum and minimum. It will do this 1x. `Clear Data` will empty all data arrays associated with a plot thus starting the process of recording over again.

> **Code Relation**
>
> Data is stored in AngularJS services. When the user selects clear data, this will empty the data in the arrays within the service. A service is a **singleton** so this means that any other entity using that data that was effected will also be effected. Some plot directives (such as the Alicat, PAS and CRDS) are reused on the interface, so this applies to them also.

Calibration

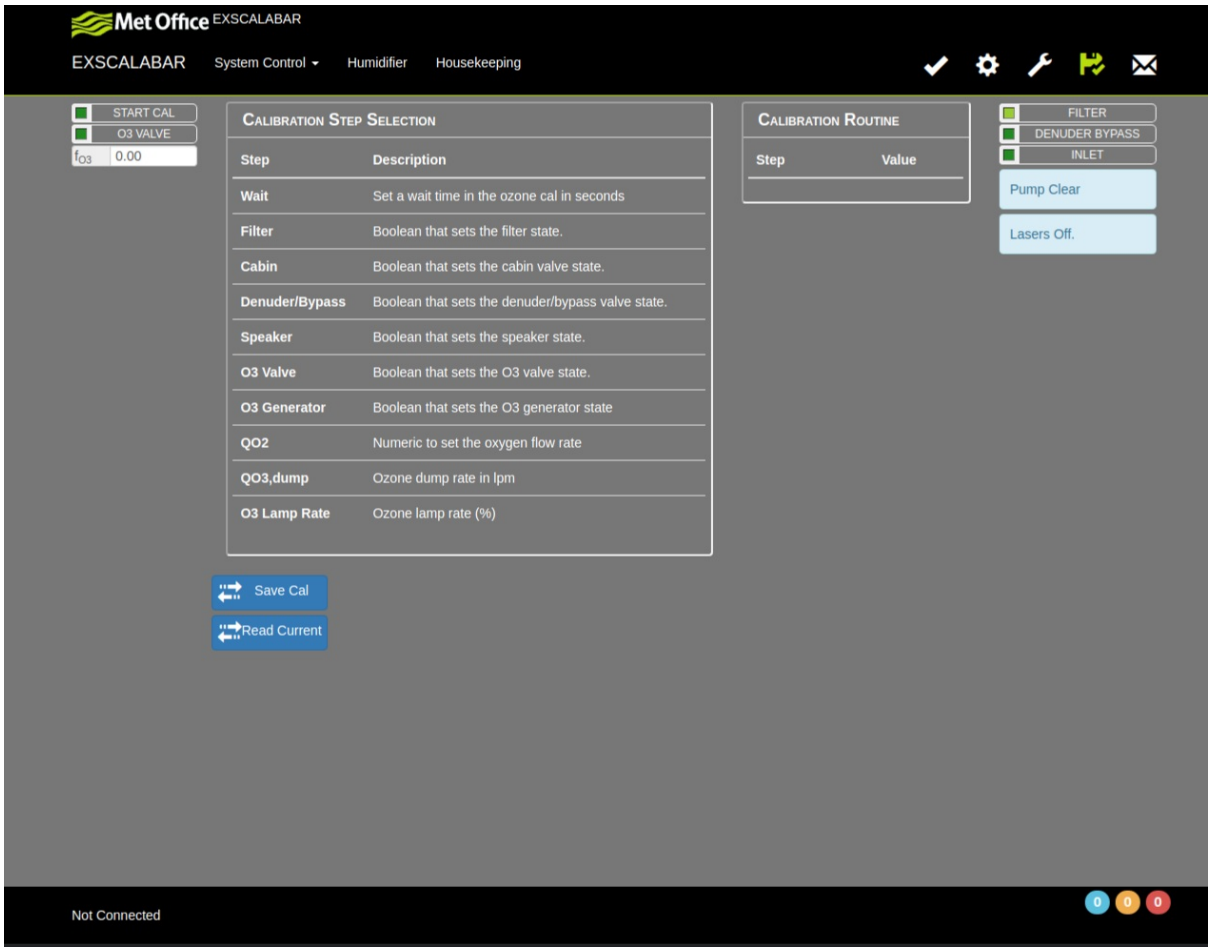
The calibration page is shown in the following image. This page contains all information required for running ozone or pressure calibrations. There are two controls on the page to manual change the state of the system:

- `O3 Valve` - controls the O3 valve. A lit indicator tells the user that the valve is open.
- `fO3` - adjust the ozone lamp frequency in Hz.

Generating a new calibration file

The two primary features of the page are the `Calibration Step Selection` and `Calibration Routine`. The first table is used to build a calibration routine; when the user double clicks on one of the steps, the step will appear in the latter table with a value that can be changed but contains the default value. The steps that the user can choose are:

- `Wait` - set a wait period in seconds
- `Filter` - boolean representing whether the path will be set to filter (`TRUE`) or sample (`FALSE`)
- `Inlet` - boolean representing whether the path will be through the aircraft inlet (`TRUE`) or from the cabin (`FALSE`)
- `Denuder/Bypass` - boolean that sets the path through the denuder (`TRUE`) or bypass (`FALSE`)
- `Speaker` - boolean representing whether the speakers on the PAS are on (`TRUE`) or off (`FALSE`)
- `Pressure Control` - number representing the pressure setpoint on the Alicat pressure controller
- `O3 Valve` - boolean turning the O3 flow on (`TRUE`) or off ('FALSE')
- `O3 Generator` - boolean turning the O3 generator on (`TRUE`) or off (`FALSE`)
- `QO2` - number representing the oxygen flow rate in the native units of the controller
- `QO3,dump` - number setting the ozone flow dump rate in lpm
- `O3 Lamp Rate` - number setting the ozone lamp rate in Hz



9: The calibration page.

As stated above, when a value is double clicked, the step is shown in as the next step in the `Calibration Routine` table. If any step is not desired, the user can double click the undesired step and this will remove the step from the sequence in the current routine.

Storing the generated calibration file

When the user is ready to save the calibration routine, they will click `Save Cal`. This will produce and transmit an XML file that is then sent to the server. The XML file will look something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<OZONE>
    <Wait>20</Wait>
    <Filter>FALSE</Filter>
    <Speaker>20</Speaker>
    <O2-Valve>FALSE</O2-Valve>
    <O3-Valve>FALSE</O3-Valve>
    <O3-Generator-Power>FALSE</O3-Generator-Power>
    <O2-Flow-Rate>100</O2-Flow-Rate>
    <Wait>20</Wait>
</OZONE>
```

The XML file is stored on the server in the folder `c:\exscalabar\cals` using the file name `o3.xml`. When a new file is generated by the client, the server will save a date stamped file with the same contents as a backup. The file will be saved in the same folder on the server and will look like `o3_yyymmdd_HHMMSS.xml` where the letters following `o3_` represent the current date and time that the file was generated.

> NOTE: The default file that is cached and run when the user hits the button `Calibrate` is `o3.xml`. The user can modify the text in this file or rename another file with the `o3.xml` name. Be careful that the file is the correct format though or it will not properly load!

Reading a calibration file from the server

When the user first opens the calibration page, no calibration sequence is loaded. To load the current file labeled `o3.xml` on the server, the user must push the button with the label `Read Current`. This will read the file in the file on the server located at `c:\exscalabar\cals\o3.xml` and populate the table labeled `Calibration Routine`.

Running a calibration

If the indicator is not lit on the button labeled `Calibrate`, simply hit the button to start the calibration sequence. The calibration sequence on the server will produce messages with the tag `[cal]` that are colored aqua.

Stopping a calibration

If the button is lit (indicating a running calibration), then hit the button again to stop a calibration. Regardless of how the calibration is stopped, the server will send messages to do the following:

- Close the O3 valve
- Set the speaker to the initial state
- Set the filter to the intial state
- Turn the speaker cycle off
- Turn the filter cycle off
- Set the O2 flow rate to 0

Development

For development of the user interface, we use several different tools. The first of these tools is git, described previously. This is the repository system used throughout this project.

In addition, the code uses the AngularJS framework for developing single page applications. A branch of this code is actually contained in the repository so there is no need to download this from the AngularJS repository.

Because AngularJS generates cross-origin requests (a security issue in the real world), AngularJS code must be hosted on a server. This is not a problem as the code will be hosted on the controller web server. But, for development and testing purposes, the user will likely want to be able to run it on any desktop they are developing on. For this purpose, the user will require a local web server.

nodejs

A local web-server can be generated from a variety of languages (Python being an excellent choice), but for the purposes of this project a better choice is nodejs. The application is currently not particularly opinionated about the version of node, but it is recommended to go with at least version 6.11.3, the current long-term service version of the code.

The current version of the Windows binaries for node can be found here for 64-bit systems. For other versions, see the website for more information.

gulpjs

Once the developer has acquired the tool, it is time to install the streaming build system used for development. This is gulpjs. To install gulp, simply use the following command in your git bash

```
npm install --global gulp-cli
```

Once you have installed gulpjs, you will need to install gulp into the project devDependencies. To do this, run the following command in the project folder from the command line

```
npm install --save-dev gulp
```

Since we have developed a gulp file, you can now theoretically run that file by simply running the command `gulp` . Configurations for gulp are defined in `gulpfile.js` . Typing `gulp` will run the `gulp.task` defined as `default` . To run an alternative task, you will type `gulp [taskname]` where that task name is defined in the code

```
gulp.task('taskname', [...])
```

Despite the fact that the user has now installed gulp, they will not be able to run the default task defined in `gulpfile.js` asthere are several depedencies that have to be handled. This is discussed in a following section.

Use of AngularJS

The user interface makes *extensive* use of AngularJS. AngularJS is a framework that utilizes data-biniding to develop dynamic views in web-applications. AngularJS is a powerful but complex set of Javascript files developed by Google. Most of the framing for the desired functionality is found in the files currently developed for the EXSCALABAR application and is readily extensible. However, for development of new components for interaction with the existing components, it is recommended that the developer familiarize themselves with the basics before proceeding. This application uses version 1.6.x; this is not compatible with version 2.x. Below are outlined in a very basic manner the components that are used in the EXSCALABAR application. For more extensive descriptions of these components, see the AngularJS documentation.

When looking at an AngularJS application, one must understand two aspects. The first is directly associated with the "static" HTML files. In an AngularJS application, there is markup associated with the AngularJS backend. This additional markup will perform one of several actions.

Controller

The first function of AngularJS is to bind a controller variable to a particular element. This will utilize the `ng-model` directive and will tie directly to a controller variable defined under `$scope`. An example of this can be found in `crds.html`. Here we have a `div` element that utilizes the `ng-controller` directive to bind the element to the data and functionality defined by the controller called `ExCrdCtl` (more on the controller below). This looks like

```
<div ng-controller="ExCrdCtl">
    <div class="row">
        <div class="col-md-3 col-lg-2">
            <div class="panel panel-default">
                <div class="panel-heading">Laser Control</div>
                <div class="panel-body">
                    <cui-numeric ng-model="rate" width="45" dtype="int" evnt-func="setRate(rate)">f (Hz)
                    </cui-numeric>
                    <cui-numeric ng-model="dc" width="45" evnt-func="setDC(dc)">DC (%)</cui-numeric>
                </div>
            </div>
...
</div>
```

In this snippet, we have an element defined by the directive `cui-numeric` (more on directives below) which ties the variable `rate` in the controller defined by `ExCrdCtl` to the numeric input that controls the laser repetition rate. The variable is bound both ways (two-way data binding) - if the variable is changed by the code found in the controller defined by `ExCrdCtl`, then the displayed input value will change; if the variable is changed by the user, then the variable `rate` will be updated.

The Angular controller is used to augment the scope of the AngularJS application. When the controller is attached via the `ng-controller` directive (as shown above), the controller object specified will be instantiated and a new child scope defined within the controller using the injectable parameter `$scope`. In the above example, the controller which is instantiated is defined in the file `ex.crd.ctl.js`.

> The naming should be somewhat consistent across the application. All files that the application specifically depends upon are prefixed with `ex`. The string that follows usually defines what the portion of the application the script is associated with - in the is case the CRD. And finally, this string is followed by a descriptor that defines role the script plays in the AngularJS application - `ctl` for controller, `dir` for directive and `svc` for service.

If we look in the script defining the controller, we will find the following code at the beginning:

```
angular.module('main').controller('ExCrdCtl', ['$scope', 'cvt', 'ExCrdSvc','Data',
```

```
        function ($scope, cvt, ExCrdSvc,Data) {...}
```

This defines a new `controller` object that will be called `ExCrdCtl` and will utilize the `$scope`, `cvt`, `ExcrdSvc` and `Data` services. These services are "injected" (this is dependency injection) into the controller. The variable `rate` is defined as

```
$scope.rate = cvt.crd.fblue;
```

When the controller is instantiated, the value will be populated with the corresponding value found in the the `cvt` service `crd.fblue`. On the backend, the `rate` is directly changed by the UI, but the *state* of the control is not stored in the controller itself. When the value is changed, not only will the variable in the controller be updated, a series of calls will be kicked off. The directive `cui-numeric` allows the binding of a function to the input - in this case, the function is called `setRate` which takes one input (the variable `rate`). When the value is changed the following code will be called

```
$scope.setRate = function () {
                var rate = arguments[0];
                cvt.crd.setLaserRate(rate);

            };
```

This code will call another function defined by the service `cvt` and found in the `crd` object (this happens to store properties and methods directly associated with the CRD operation). The value in turn will be sent to the server (not discussed here) via http calls and the current value table will be updated with the new value and updated again on the client side.

The client will reflect this new value in the `cvt` and broadcast an event called `cvtUpdated`. The controller contains a listener for this event as defined below.

```
$scope.$on('cvtUpdated', handleCVTUpdate);
```

The function `handleCVTUpdate` will be fired when this event is broadcasted. This function will update the variable `rate` from the CVT (which should be the same as what the user entered unless the server had to coerce the value because the entered value was not valid).

Services

Implicit in the above discussion is the presence of services. Services are used to organize and share code across the application. They are singletons - everytime the service is injected into a controller, it is a reference to the service. Each service will exist for the life of the application (i.e. until the window is closed) and therefore can be used to maintain the state of the application as long as it is open. Services are instantiated via the `factory` method as with the service `ExCrdSvc` which is defined in the file `ex.crd.svc.js` and shown below

```
angular.module('main').factory('ExCrdSvc', crdSvc);
crdSvc.$inject = ['$rootScope', 'Data'];
```

In the above instance, the service is defined by the function in the file called `crdSvc` and depends on two other services - the native AngularJS service `$rootscope` and the application service `Data`. In the case of this service, the primary function is to listen for the event produced by the `Data` service called `dataAvailable`. When this event is fired, the CRD data will be properly populated and the service will broadcast the event `crdDataAvailable` to any other modules that depend on that data.

Throughout the application, there are two services that are used extensively - these are the `Data` service (defined in `data-service.js`) and the `cvt` service (defined in `cvt-service.js`). Functions in both of these services are called in `ExMainCtl` (defined in `ex.main.ctl.js`), which is the main application controller at 100 ms intervals. This controller is defined in the file `index.html` as shown below:

```
<html lang="en"  ng-app="main" ng-controller="ExMainCtl">
```

In `ExMainCtl`, the functions for checking for new data and current value table updates is as follows:

```
$interval(function () {
                Data.getData();
                cvt.checkCvt();
                if (!ExDevStatusSvc.ini_complete)
                {
                    ExDevStatusSvc.get_ini();}
                    if (i >10){
                        i = 0;
                        ExDevStatusSvc.listenForComm();
                }
                else {i +=1;}
            }, 100);
```

The service `$interval` is a wrapper for the function `window.setInterval` and will execute every `delay` milliseconds (as defined by the second argument to the function).

In addition to checking for current value table and data updates, this function will check to see if the INI status has been updated. This calls the service `ExDeviceStatusSvc` and checks to to see

1. If the INI status has been updated and
2. if it hasn't, it will check for an updated INI entry.

The INI entry in this case will define the what devices the user should *expect* to see updating based on the configuration file on the server.

In addition to the two services outlined above, there are other services concerned with either taking data generated by the server and storing it on the client based on updates from the main data service or storing the status of controls updated on the client (as with the `net` service as defined in `network-service.js` file which maintains and handles the current network settings state).

Directives

The final AngularJS component that is used extensively throughout this application is the directive. Directives are a set of code that will define markers which will attach certain behavior to a DOM element to which that directive is attached. In most cases, the directive here is used to define:

1. the structure of HTML that is to be displayed and
2. the behavior that is associated with the DOM for that defined HTML.

Directives are apparent throughout the HTML. `ng-model` and `ng-controller` are two AngularJS directives that are used extensively by the application as shown in the first code snippet on this page. In addition, there are custom directives used throughtout the application. Some of these wrap graphical functionality such as that for `dygraphs` while others are used to provide the look and feel of key application elements (such as `cui-numeric`).

Directives are complex pieces of code can wrap some complex behavior into a small package (see `inumeric` in the folder `../assets/cui/inumeric` for an example). Before working with directives, the developer is encouraged to take advantage of the tutorials available and read the extensive documentation provided by the AngularJS team. That being said, many of the directives can be simply dropped in to add new numeric controls, indicators and buttons by following the example provided in the initial code snippet.

The Build System

For development of the UI, we have chosen to use gulp to aid in builds and debugging. gulp is a streaming build system that is developed in javascript and built on node.js. There are many existing plugins for running various tasks using gulp and most are available http://gulpjs.com/plugins.

As described in the introduction of this section, there is a single gulp file used for configuration of the build called `gulpfile.js`. Multiple tasks are configured for this system. These tasks are defined throughout the file and can be identified by the function call `gulp.task`. Tasks can be run individually or as batches.

When the user runs a task, they will type in the command `gulp [taskname]`; if `[taskname]` is blank, then the default task is run. This task is identified by the string 'default'.

In addition to the individual tasks defined in the file, batch tasks can be composed from the individual tasks. The batch tasks have an identifier string followed by an array of previously defined tasks. In the case of the default task this looks like

```
gulp.task('default', ['lint', 'scripts', 'ext_assets', 'int_assets',  'ngdocs', 'connect', 'connect2docs', 'ope
n', 'watch']);
```

An individual task looks like

```
gulp.task('bump-major', function () {
    gulp.src(['./package.json', './ui.json'])
        .pipe(bump({
            type: 'major'
        }))
        .pipe(gulp.dest('./'));
```

Dependencies

As previously stated, the build system can be run with the command `gulp`. But, in the case of the gulp file provided, the user must satisfy the dependencies of the build

For this build, we are using the following plugins:

- gulp-jshint - JSHint is a linter that flags suspicious usage of JS when working with the code. Issues can be caught before trying to execute the code in a page. JSHint will flag not only errors (which will prevent any attempt to connect or open a page in the build system) but will also provide the user with important feedback regarding JS usage. The individual task that uses this is `lint`.
- gulp-concat - This will concatenate individual files in the build. Given the lack of no formal way of including files as in C, this provides the developer with the ability to place all of the code into a single file so that they will have to reference only one JS file instead of many. For ease of development, the EXSCALABAR UI is split into *many* individual Java Script files, so this tool is especially important as we can place all of these files into a single file. The individual task that uses this is `scripts`, `ext_assets` and `int_assets`.
- gulp-uglify - This plugin is used to minify the codebase using UglifyJS2. **This plugin is no longer used and can be removed.**
- gulp-rename - This plugin is used to rename files. **Rename is no longer used by the system.**
- gulp-connect - Runs a simple web server. Pages built with AngularJS must be served or they will not display. This plugin is used by the individual task `connect` and `connect2docs`. When this task is run, the user will get output that looks like `Server started http://localhost:8000...`
- gulp-open - Open is used to open the initial page at port 8000 in the default browser. This plugin is used in the individual task `open`.

- gulp-ngdocs - Builds and serves documentation that is AngularJS-like. When `gulp` is run, the output will contain something that looks like `Opening http://localhost:8080 using the default OS app...` which indicates that the page is being served on 127.0.0.1 at port 8080. Not all documentation is present in this generated page. This plugin is used by the individual task `connect2docs`.
- gulp-bump - Bumps the semantic version of the application. When this is run, the version in `package.json` will be changed as specified. This is called in `bump`, `bump-minor` and `bump-major`, each of which will bump the version as defined in the name (except `bump` which will bump the patch version.

gulpfile Structure

Dependencies

The gulp build system relies on `require.js`. Require statements indicate files that will be required for the application to run properly and are located at the top of `gulpfile.js`. The files that are required for the build system are discssed above.

Defining Input File Lists

The files required for the client side AngularJS application are defined in several arrays following the require statements. The arrays are grouped logically and correspond to several different js files that are produced. In each of these lists, the order mattters as each list will be compiled into a single concatenated file, so those that reference others will have to come after the ones they reference. These lists are defined as follows:

- `watch_list` - an array of files that are specific to the EXSCALABAR application itself. These files are compiled to a single file called `exscalabar.js` (and `exscalabar.min.js` but this file is never referenced so it may be removed).
- `ext_assets` - an array of external assets required for the operation of the application. These files will rarely if ever change and are not actively developed or maintained by MSR Consulting. These files are compiled into a single concatenated file called `assets.js`.
- `int_assets` - these files are not necessarily specific to the application itself but may change periodically and are developed and maintained by MSR Consulting. These file include the custom dygraphs implementation (`cirrus-dygraphs-dev-js`) as well as the Angular hook for dygraphs and some custom UI elements. These files are compiled into a file called `int_assets.js`

Each of these concatenated files are placed into the folder `js`. These files are referenced at the bottom of the application entrance point `index.html` as follows:

```
<!--Import libraries -->
<!--TODO: Make sure that you have the min libs for production!!! -->
<script src="js/assets.js"></script>
<script src="js/int_assets.js"></script>
<script src="js/exscalabar.js"></script>
```

> Note that paths are relative so the folder `js` will just have to be placed in the same folder as `index.html`.

Another list follows these arrays called `docList` - this provides a list of modules that are used to compile the AngularJS documentation for this application. This is called in the task with the id `ngdocs`. This documentation will also be compiled and run a server at port 8000.

> Currently, this documentation has limited usability as not all of the files have been properly documented in the ng-docs format. For more information on the ng-docs format see this page.

Tasks

Following the arrays of file names are the definition of the gulp tasks. As explained above, the tasks may be run individually by calling any of the tasks on the command line using the syntax `gulp [taskname]`. However, serveral tasks have been defined which are composed of an array of individual tasks. These are defined at the bottom and are as follows

- `default` - this is the task that will be run when the user types `gulp` into the command line. The js files will be linted, files will be concatenated as described above, ng-docs will be built and served, the application will be served and the application page will be opened at `http://localhost:8080`.
- `no-lint` - same as `default` but no linting will be done (the task `lint` will not be run). This simply means that no hints will be provided to improve the javascript code.
- `no-browse` - same as `default` except the application will not be served.

Server

The server refers to the portion of the software that resides on the PXI controller that is used to control how data is acquired and what happens to it. This code is written in LabVIEW 2014 but may be upgraded as necessary (and it is recommended that the maintainer upgrade when possible to avoid deprecation issues).

Aside from the code found in the project directories, the following modules will be required for operation:

- OpenG Toolkit; specifically the following libraries in the toolkit are used by this project:
  - OpenG LabVIEW Data Library
  - OpenG Error Library
  - OpenG String Library
  - OpenG Time Library
  - OpenG Variant Configuration Library
- NI String Tools Library
- ESF - Extensible Session Framework

All of these libraries except the ESF are available via the JKI VI Package Manager. When the system is moved or upgraded, the will have to be redownloaded and applied to the correct LabVIEW version. There is no need to download the individual OpenG pacakges - these are all available in the OpenG Toolkit.

The ESF package is not available through the package manager. Instead, the developer may find this here

> https://forums.ni.com/t5/Reference-Design-Content/Extensible-Session-Framework-ESF-Library/ta-p/3522019.

Unpacking the ESF will place it in the `user.lib` folder. This is not consistent with current standards and can cause some issues. It is suggested that in the future, this package be repackaged for a local build and the deployment set to `vi.lib`; unlike `user.lib`, `vi.lib` is searched for packages so the maintainer will not have the compile errors that they might have if the package is placed in `user.lib` (the system will not be able to find it unless told specifically to look in `user.lib` and the maintainer will be forced to manually repoint the packages to this implementation).

Running the Server

The server may be operated in one of two manners:

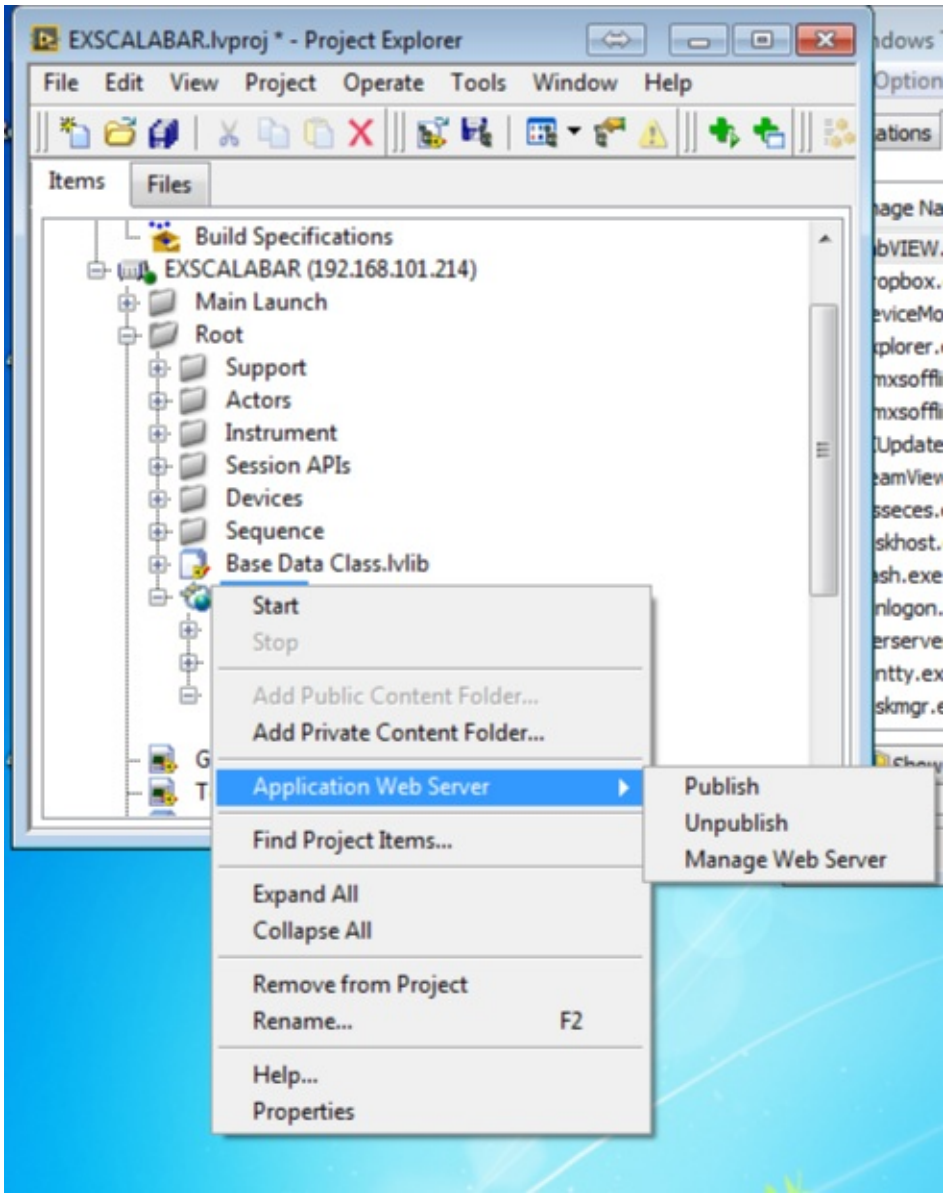
- from the LabVIEW environment for development and debugging purposes
- via a published web service

All code is accessed through the web service startup VI called (originally) `Startup`.

Running via the Published Web Service

I start here with the published web service as this method of running has implicaitons for development mode. This is the simplest manner of running but the code is not directly available for debugging if the service is published. If the service is not published, the user will simply:

1. open the project in LabVIEW
2. expand the code under the real-time target `EXSCALABAR`
3. right-click on the service `xService` and select `Application Web Server >> Publish`
4. reboot



10: Publishing a web service

Once the web service is published, it will launch on boot. To prevent the application from running at boot:

1. open the project in LabVIEW as abvove
2. expand the code under the real-time target `EXSCALABAR`
3. right-click on the service `xService` and select `Application Web Server >> Unpublish`
4. reboot

When the system is rebooted, the application will not launch.

Running in Development Mode

If the developer wishes to debug or develop the application further, they will want to run in development mode. Before starting, it is good practice to go ahead and unpublish the web service as described above. Once the user is certain that the application is not set to run on boot, they will proceed as follows:

1. open the project in LabVIEW

2. expand the code under the RT target
3. right-click on the target and select `Connect`. On successful connection, the LED beside the target image will light green.
4. right-click on the web service `xService` and select `Start`

A Third Way?

A final possible way for launching the server application is to build an executable. This was the original path, but the system had some startup issues. Using this method in LabVIEW 2014, the startup application took several minutes to begin execution. After much discussion with National Instruments Application Engineers, this method of launching the program was ditched for the first method described above - publishing the web service.

Operational Overview

This system relies heavily on the Actor Framework. Applications utilizing the Actor Framework are composed of actors - asynchronous objects that maintain their own internal state and communicate outside of their bounds via discrete messages of which the delivery is not reliably determined in time. In this application, a single actor `Controller` maintains lists of active actors and as well as maintains the data for the system. For the most part, messages are funnelled through the `Controller` object. Each actor will determine when to send data regardless of when other actors send data. The `Controller` is in charge of aggregating data and routing it to the appropriate entities (i.e. the web service and file actors). More is discussed concerning the architecture further along the document. This section focuses primarily on launching the application and processes associated with the actor in charge - the `Controller` object.

Below is an activity diagram demonstrating the launch sequence of the system. When running in the LabVIEW environment for debugging purposes, the user will start the program as described in the previous section by connecting and starting the web service. The startup VI for the web service will run `Launcher` found in the `Main Launch` folder (the only file in the folder) in the project. The `Launcher` VI has one thing that it does - it simply launches the `Controller` actor object. **The `Controller` is the entry point of the application and is the top level object that controls the operation of the application; this page strictly covers the operation of the system with respect to the `Controller`.**

> For the purposes of operation, this VI `Launcher` could be removed altogether and replaced with the contents - `Actor::Launch Root Actor` with `Controller` as the actor to be launched.

As with every `Actor` object, the `Controller` has a set of actions that it will perform. Each `Actor` in the system will run the `Actor::Pre Launch Init` when launched. In the EXSCALABAR DAQ, this is where all configuration will occur. If an error is thrown here, then the `Actor` *should* shutdown without sending a last ack to the actor in which it is nested. In the case of the `Controller`, this means that a fail at launch will result in a failure to launch. The error is fatal because the `Controller` is the hub of all activity - most messages will pass through this `Actor` at some point.

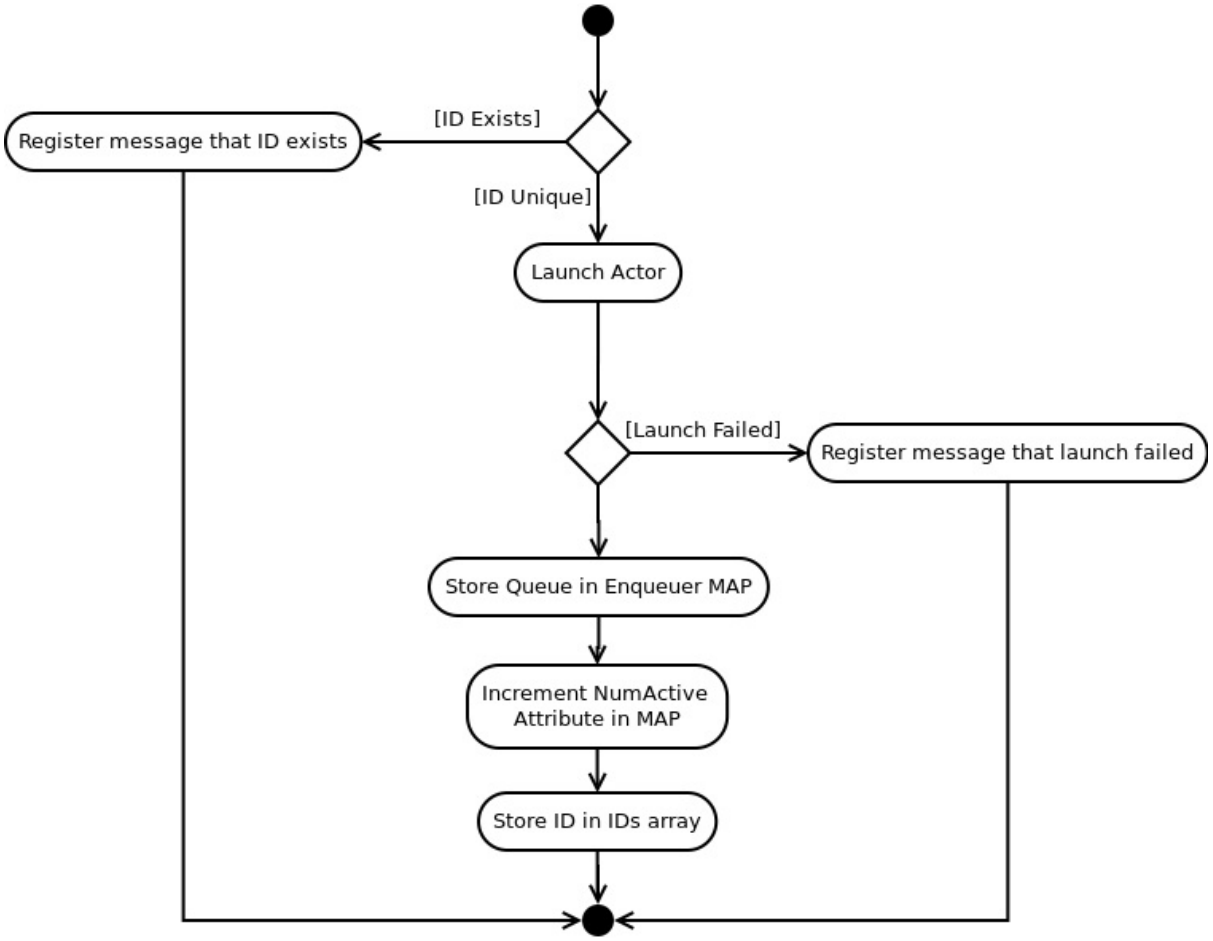
Launching Nested Actors

If the launch is successful, the `Actor::Actor Core` process will kick off and continue until a stop message is sent. A stop message may be sent by the user via a command to halt the system *or* if an error is thrown and not handled properly in the core itself. The `Actor Core` is where all messages will be handled by the `Actor`. Processes not required for startup run in parallel to the main `Actor Core` implementation.

11: Launch Activity

In the EXSCALABAR DAQ, the `Controller` will kick off the `Controller::Actor Core` operations by launching all nested actors as shown in the figure above. Actors are asynchronous meaning that there is no guarantee as to when an action associated with the actor will occur. But, when the `Controller::Actor Core` is launched, a series of events occur *before* the parent method is called. As can be seen from the diagram, the two instruments - PAS and CRDS - are launched followed by all device objects (as read from the configuration file). The actor that monitors the aircraft network is launched as well as the main and mirror file writing actors. Failure of any of these processes to launch will result in a fatal error which will cause the actor in question to not start and may cause the system to not boot.

The launching process is wrapped in the `Controller::Launch Nested` method. This sequence of events in this method is shown in the figure below. This method accepts an input called `ID` which will be used to directly identify the nested actor in the case where the `Controller` needs to send a message



12: Launching nested actors

> One thing to note for future development - this method `Launch Nested` as well as any code handling nested actors should be placed in the `SuperActor` object. This is a general method for an `Actor` object to launch, track and handle nested actors. The built-in methods for handling nested actors are of limited use. The primary issue that an application faces is that 1) there is no way for the application to concretely identify a particular nested actor and 2) issuing a shutdown command for an actor with nested actors will simply send a `Send Normal Stop` to the nested actors and shutdown the top-level actor thus resulting in often unpredictable results.

In this method, the `Controller::Enqueuer MAP` property will be queried to determine if another `Actor` with this `ID` was already launched; if it was, it will write a message to the log indicating this to be the case.

> **`Enqueuer MAP`**
>
> This map is one of several that occurs throughout the code. Each MAP object is simply a variant with properties attributed to it that are accessed through the `Get Variant Attributes` or `Set Variant Attributes`. The MAP allows wrapped access to key-value pairs of data (similar to Python's dictionary structure).
>
> In the case of the `Controller::Enqueuer MAP` property, the variant will contain the following attributes:
>
> - `IDs` - an array of strings that is a simple list of IDs of nested actors that have been registered with the MAP.
> - `NumActive` - this is the number of nested actors that are represented in the MAP.
> - a variant attribute with each corresponding to each ID in the `IDs` attribute that contains the `Enqueuer` for that nested actor.

If it was not, then this method will attempt to launch the nested actor by calling the `Actor::Launch Nested Actor`. If it is successful, then this method will return with an `Enqueuer` for that actor. That `Enqueuer` will be stored in the `Controller::Enqueuer MAP` under it's respective ID, the number of nested actors represented in the map will be incremented and the ID will be added to the `IDs` attribute in the map. If the launch fails, then an error *should* be returned and the `Controller::Enqueuer MAP` is never touched. The error message is logged in the message stream.

> What is actually stored is a type-def that contains a boolean as well as the enqueuer; for future development, the boolean may be removed. It remains simply because this structure was in place throughout the development before it was determined that the boolean was no useful.

Well defined `IDs` are as follows:

- `PAS`: photoacoustic spectrometer instrument actor
- `CRDS`: cavity ringdown spectrometer instrument actor
- `AC Network`: actor for communicating with the aircraft network
- `Main`: actor that controls file access to the main file that is written to
- `Mirror`: actor that controls file access to the mirror file that is written to

Some well defined `IDs` that are not launched every time:

- `cal`: calibration actor that is started when the user requests a calibration
- `paswvfm`: file actor associated with the writing the PAS waveform data
- `crdtaus`: file actor for writing all of the CRD τ data
- `crdrd`: file actor for writing ringdown data

Additional Processes

The `Controller::Actor Core` process also handles a variety of other actions that do not result in the launching of an actor. One of the most important actions is to set the power up state via the method `Controller::Init Power`. In this method, the parameters associated with power up are read from the configuration file and the appropriate commands are send to the boards via a DAQ write in the VI `Controller::Handle Power Switch`.

Once all of the nested actors have been launched, the controller will check to see if there were nested actors launched. If there were not, then the controller will send a shutdown message.

Finally, there are a series of processes that are kicked off and run in parallel with the main `Actor Core`. These are as follows:

File Writing Messaging

A process defined in `Controller::Send Write Main MSG` sends a message to write to file any files that are open at regular intervals.

Filter Cycling

A process that monitors the filter and filter cycle states and changes the filter state as necessary as defined in `Controller::Handle Filter Automation`.

Monitor Common Functionality

A process that monitors core functionality that can not be claimed by discrete actors in the system is defined by the method `Controller::Common`. This method monitors the state of some variables that are maintained in the CVT whose state are controlled by messages sent to the controller. These variables may be written to file or used to control the visible state of the system via LEDs.

In this VI, several channels and CVT variables are used to toggle LEDs on the box for the laser power. Each LED is controlled by whether 1) the lid is off, 2) the laser is enabled and 3) the lasers have power.

In addition, the following variables are monitored and written to the primary data structure:

- Blocked impactor voltage
- Interlock state
- Pump blocked state
- Denuder position
- Calibration state
- Inlet valve position
- Filter valve position
- Message array
- Main data collection time stamp

Finally, CRD data required for real-time calculation is routed and stored via the method `Controller::Store CRD Data`. This method checks the Map `CRD Data Route` and determines what relevant data (flow rate, temperature and pressure) to associated with a particular cell.

Configuration

Configuring the System

In software, the access to the configuration file discussed below is controlled through a session (discussed in a later section). The session opens and closes the reference to the configuration file and no particular object has control over the reference directly. **There is not a discreet configuration action - all actors which are children of the `SuperActor` have a copy of the configuration session and generally perform configuration actions in their individual `Pre Launch Init` implementations as required.** In some instances, the objects owned by nested actors will require access to the configuration files and a copy of the session will be generated for these - this includes the `Instr Actor` as well as `Device` objects.

The configuration file path is specified in the global variable `xGlobal Data` under the variable `Cfg Path`. To make changes to the file, the user can ftp into the server using a tool such as FileZilla and navigate to the location defined in the global variable. **Although this variable is not programatically changed, it can be changed by the user at development time - be sure to update records to reflect the chane in location!!!** Currently, the file may be found at `c:\exscalabar\cfg\exscalabar.ini`. It is critical to save the current copy if the user intends to make changes to the file. Refer to the working copy for formatting or to the guide to the variables below.

> **Future work:** While not currently a task that is performed by the system, it is advisable that in future work the system be changed to write a copy of the configuration file to the data location to prevent a loss of a functional copy of the configuration file.

The Configuration File

The server is configured via an INI file located on the server under the folder `c:\exscalabar\cfg` called `exscalabar.ini`. The file is written in the standard INI format; that is, a file broken out into sections (denoted by square braces) containing a series of key-value pairs. Below is a description of the functionality defined in the INI file.

Serial Device Configuration

Serial devices are those that are communicating on a serial port, either via RS232 or RS485. Devices are configured for a common serial port. Devices defined in the file are:

- `Alicat` - Alicat flow controllers and meters
- `PPT` - Honeywell pressure transducers
- `Vaisala` - Vaisala hygrometers
- `Meerstetter` - Meerstetter TECs
- `TEC` - TE Tech TEC

Each serial device contains common parameters for configuring communication on the serial port. These parameters are:

- `Port` - This is the key for the serial port that the device or device cluster will communicate on. The value usually is the string *COM* followed by a number.
- `Serial Config.baud rate` - Defines the baud rate for device communication.
- `Serial Config.data bits` - Defines the number of data bits in the communication.
- `Serial Config.stop bits` - Defines the number of stop bits. This value will be 0 for 0 bits, 10 for 1 bit and 20 for 2 bits.
- `Serial Config.parity` - Defines the parity of the communication
- `Serial Config.flow control` = 0
- `Serial Config.endModeforReads` - Integer defining how the end is determined for reading from the port. These values are:
  - 0 = `None`

    - 1 = `Last Bit`
    - 2 = `Termination Char`
- `Serial Config.endModeforWrites` Similar to `endModeforReads`. Integers are defined in same manner.
- `Msg Config.sendEndEn` - Boolean indicating whether a terminating character is sent with a write to the port.
- `Msg Config.suppEnEnRd` - Boolean indicating whether to supress the end of a read with a termination character (i.e. reads will terminate on timeout or when a specific number of bytes is read from the port).
- `Msg Config.termChar` - Hexadecimal value indicating what the termination character for communication is. 10 will be a new line character and 13 will be a carriage return.
- `Msg Config.EnTermChar` - Boolean determining whether a termination character is used or not.

If there are multiple devices communicating on the same line (port), then each device will be found under the same section using a unique ID. IDs are defined in the key `IDs` which contains a set of comma delimited strings (no spaces) representing each device on that line. These IDs are used to identify the device in the `Enqueuer MAP` found in the `Controller` and also provide the ability to configure entries in the INI file directly related to that device instance. An example is the `[id].Q0` which represents the initial setpoint for devices under the `Alicat` section.

All devices are children of the superclass `Configure Device`. This object has a single key associated with it called `[id].CalEquation`. This value is used for calibrating outputs and the default is `1*x`.

Device Specific Configuration

The following sections define functionality outlined in the INI file unique to each serial device.

Alicat

- `[id].Q0` - initial setpoint in the native units
- `[id].label` - label used by the UI for display purposes
- `[id[.Address` - alphabet character address of the Alicat device. This is configured in the hardware.

PPT

All parameters below (aside from `label` and `address`) are directly related to commands that can be sent via serial communication. More extensive definitions of the functionality can be found in the PPT user manual.

- `twait`
- `Units` - set the units that the pressure measurement is returned in. Possible settings for this are:
    - `atm`
    - `bar`
    - `cmwc`
    - `ftwc`
    - `inhg`
    - `kgcm`
    - `kpa`
    - `mbar`
    - `mmhg`
    - `mpa`
    - `mwc`
    - `psi`
    - `user`
    - `lcom`
    - `pfs`
- `Pmin` - **THIS KEY IS UNUSED**
- `[id].label` - label used by the UI for display purposes
- `[id].address` - numeric character address of the PPT device. This is configured in the hardware.

- `Nsteps` - **THIS KEY IS UNUSED**

TEC

This section defines parameters associated with the TE Cooler. The following define the control parameters that are used by the TEC. These values are sent to the TEC at initialization. For more information regarding how these parameters effect TEC performance, refer to the user manual.

- `P` - proportional gain
- `I` - integral gain
- `D` - derivative gain
- `T0` - initial setpoint in degrees Celsius

The next set of keys is associated with the maximum power for heating and cooling and values run from 0 to 1 where 0 represents no heating or cooling and 1 represents maximum power applied:

- `clx` - cooling side multiplication factor; value is 0-1
- `htx` - heating side multiplication factor; value is 0-1

These final keys are used internally by the software:

- `IDs` - only one id will be present in this list as there is only one TEC
- `address` - address used for communication
- `label` - label used for plotting purposes

Meerstetter

This section defines configurable settings associated with the Meerstetter TECs. These first keys are used internally by the system:

- `IDs` - comma separated list of strings representing the individual TECs communicating on the serial port
- `[id].address` - numeric representing the address of the device (01-99)
- `[id].label` - label used in the display on the client

The following keys impact how the device is controlled. All of these keys represent settings that are sent down to the controller.

- `[id].ctl` - determines what the control input is, either `Temperature` or `Power`
- `[id].I` - live current for the controller
- `[id].Imax` - maximum current for the device
- `[id].V` - control voltage (if using the `Power` setting)
- `[id].T` - control temperature (if using the `Temperature` setting)

Vaisala

This section defines parameters for communicating with the Vaisala probes. Similar to some of the other serial devices defined above, multiple probes may communicate on the same line.

The following are used internally.

- `IDs` - comma separated list of strings representing the individual TECs communicating on the serial port
- `[id].address` - numeric representing the address of the device (01-99)
- `[id].label` - label used in the display on the clienti

Devices communicating on the line have a unique address. This address is defined in `[id].Address`. This value is 0 to 99.

Filter

In EXSCALABAR, the sample line has two states - sample and filter. In the former, the system pulls in air through the sample line for measuring the optical properties of particulate matter. The latter provides background for calculating key properties (extinction and absorption). In addition to the two states of the sample line, the controller can be configured to automatically cycle between the two. The `Filter` section provides configuration parameters for state of the filter at boot as well as the state of the automatic cycling (both of which can be changed on the user interface). The corresponding keys are:

- `fpos` - boolean indicating the initial state of the system
- `period` - integer indicating the period of the filter cycle in seconds
- `length` - length of the filter portion of the cycle in seconds
- `auto_cycle?` - boolean indicating whether the system starts in a mode where the controller automatically cycles between filter and sample

The latter three keys are used exclusively for auto-cycling. If `auto_cycle?` is `TRUE`, then system will use the `period` and `length` keys to determine when the system is filtering or sampling.

CRDS

This section defines the initial configuration of parameters associated with the cavity ringdown spectrometer. There are four cells in the CRD, so entries that can provide configuration specific to individual cells will be displayed as a comma-separated list (**no spaces**). The exception to this is when the configuration is specific to the individual lasers - in this case, there are two and there will be two values. The keys in this section are defined as follows:

- `flaser` - two integers that define each laser repetition rate in Hz
- `dc` - two floats that range from 0-1 that define the duty cycle of the lasers
- `Coadd` - four numbers defining how many ringdowns to coadd for the calculations of the tau values
- `lambda` - wavelength of the laser in each cell
- `IDs` - unique labels for the cells used in the file writing to define data associated with a particular cell
- `ncells` - number of cells; **not used**
- `RH` - initial RH setpoint for two of the cells; **not used - replaced by humidifier values**
- `lcell` - length of cavity in meters
- `expFit` - enumerated value representing the exponential fit method; appropriate values are DFT1, DFT5 and LRS
- `kpmt` - PMT gain for each cell
- `kred` - red laser gain
- `kblue0` - first blue laser gain
- `kblue1` - second blue laser gain
- `enred` - boolean representing initial red laser enable state
- `enblue0` - boolean representing initial first blue laser enable state
- `enblue1` - boolean representing initial second blue laser enable state

In addition to these key values, the `CRDS` section contains information on the routing of signals for determining conditions within the cells. These values are used in the calculation of extinction and are the temperature (`T`), pressure (`P`) and flow rate (`Q`). Each cell is identified by the `IDs` value followed by the variable of interest (for instance if there is an ID `crds_405` and we want to determine the device associated with pressure we would use the key `crds_405.P`). The value for each of these keys will be a device ID (as defined in each device section under the key `IDs`).

PAS

Similar to the `CRDS` section, the configuration file also contains a section for the photoacoustic spectrometer labeled `PAS`. The keys are defined as follows:

General parameters

- `ncells` - integer representing the number of cells in the system

- `m` , 'b' - slope and offset used to calculate the absorption from integrated area ($sigma = mx + b$)
- `TEC.T0` - floats representing initial setpoints for the TECs associated with each cell; **not used - replaced by TEC configuration section** `Meerstetter`
- `TEC.En` - booleans representing initial enable state for TECs associated with each cell; **not used - replaced by TEC configuration section 'Meerstetter'**

Speaker Operation

The following parameters define how the speaker is operated.

- `chirp.center` , `chirp.band` - integers defining the center and bandwidth of the chirp in Hz used to determine the resonant frequency of the cavity
- `spk.cycle` - boolean determining whether the speaker automatically cycles
- `spk.period` - period in seconds of speaker cycle
- `spk.length` - length in seconds that speaker will be on
- `spk.enabled` - boolean which determines whether all speakers will be enabled
- `spk.ienabled` - boolean associated with each individual cell that determines whether these speaker will be on
- `spk.vrange` - float representing the speaker voltage range
- `spk.voffset` - float representing the voltage offset of the speaker signal

Laser Operation

- `las.vrange` - voltage range for each laser in Volts
- `las.offset` - voltage offset for each laser in Volts
- `las.f0` - initial modulation frequency of the lasers in Hz; generally not used as the speaker is initially on at startup
- `las.mod` - initial modulation pattern of lasers; enumeration that has two values - `Sine` and `Square`
- `las.en` - booleans determining initial enable state of individual lasers
- `serializeWF` - boolean indicating whether to save the PAS waveforms at boot
- `connect_filter` - boolean that determines whether the speaker and filter cycles are connected at boot (i.e. a speaker calibration is started when the system switches to filter)

Humidifier

The section `Humidifier` defines parameters for the two humidified flow streams delineated by the labels `Med` and `High` that are prefixed to the keys. Each one has several parameters associated with the operation of the humidifiers. These are defined as follows (prefix excluded):

- `RHsp` - this is the initial setpoint of the system
- `TEC` - Meerstetter thermoelectric cooler ID associated with the humidifier. This is used to send signals to the humidifier when the software is actively controlling the RH
- `Vaisala` - Vaisala ID associated with this humidified line. This determines which device will provide the process varialbe for the control scheme.
- `P` , `I` and `D` - control parameters for the PID loop found in the `Humidification` VI for controlling the RH.
- `Ctl` - boolean indicating whether the humidity is being actively controlled within the line.
- `mfc` - string ID of the Alicat associated with this sample line

File

Several files are generated by the system. These files are:

- a main file that contains all 1 Hz data ('MainFile`)
- a mirror file that mirrors the contents of the main file (key found under `MainFile` called `Mirror.Path` that indicates the path for the mirrored data)

- a file containing the PAS waveform data collected every second (`PAS Waveform`)
- a file containing all τ's collected every second that are used to calculate the average τ found in the main data file (`CRD Taus`)
- a file containing ringdown data used to calculate τ (`CRD RD`)

Each file entry contains the following skeys

- `max_file_size` - maximum file size before the current file is closed and another is opened in MB
- `ext` - file extension (always include the `.`)
- `prefix` - prefix that will begin the file name
- `[file].Path` - top level path to which the system will write files. The prefix is an id that defines which file object the entry refers to. This is necessary because it is possible to have more than one entry.
- `folder` - folder in the path that the files will be written to

Using the configuration file input, all files will be written to the path defined as

```
[[file].path][folder]
```

The file will be defined using the date and time of the file start so that the file name looks like

```
[prefix]YYYYMMDD_HHmmss[ext]
```

AC Network

This section defines key parameters for communicating with the aircraft network. The keys are defined as follows:

- `address` - IP address to listen for broadcast data on and to send data to
- `port` - port that the broadcast data communicates on
- `active?` - controller will listen to the broadcast port on launch if this is `TRUE`
- `file` - a LabVIEW based XML file that will define what the packet looks like that the system is looking for

General

This section is intended to store general configuration parameters that can not be readily associated with another device or subsystem. Currently this section contains only the key `inlet` which is a boolean indicating whether the system should sample from the inlet (`TRUE`) or the cabin at startup.

Power

This section contains a set of boolean based keys for configuring the initial power up state of different subsystems. These include

- `denuder` - power for the thermodenuder
- `laser` - laser power
- `ozone` - power for the ozone lamp
- `pump` - power for the sampling pump
- `TEC` - power for the TECs, the Meerstetters and the TE Technology

Log

This section is intended for configuring how the system logs information, warnings and errors. The key `Path` determines the folder to which the log files will be written. `Log Warnings` is a boolean indicating whether the system will track warnings. `Ignore Codes` is a list of comma separated integers of error codes that the system should ignore (i.e. not log).

Task Configuration

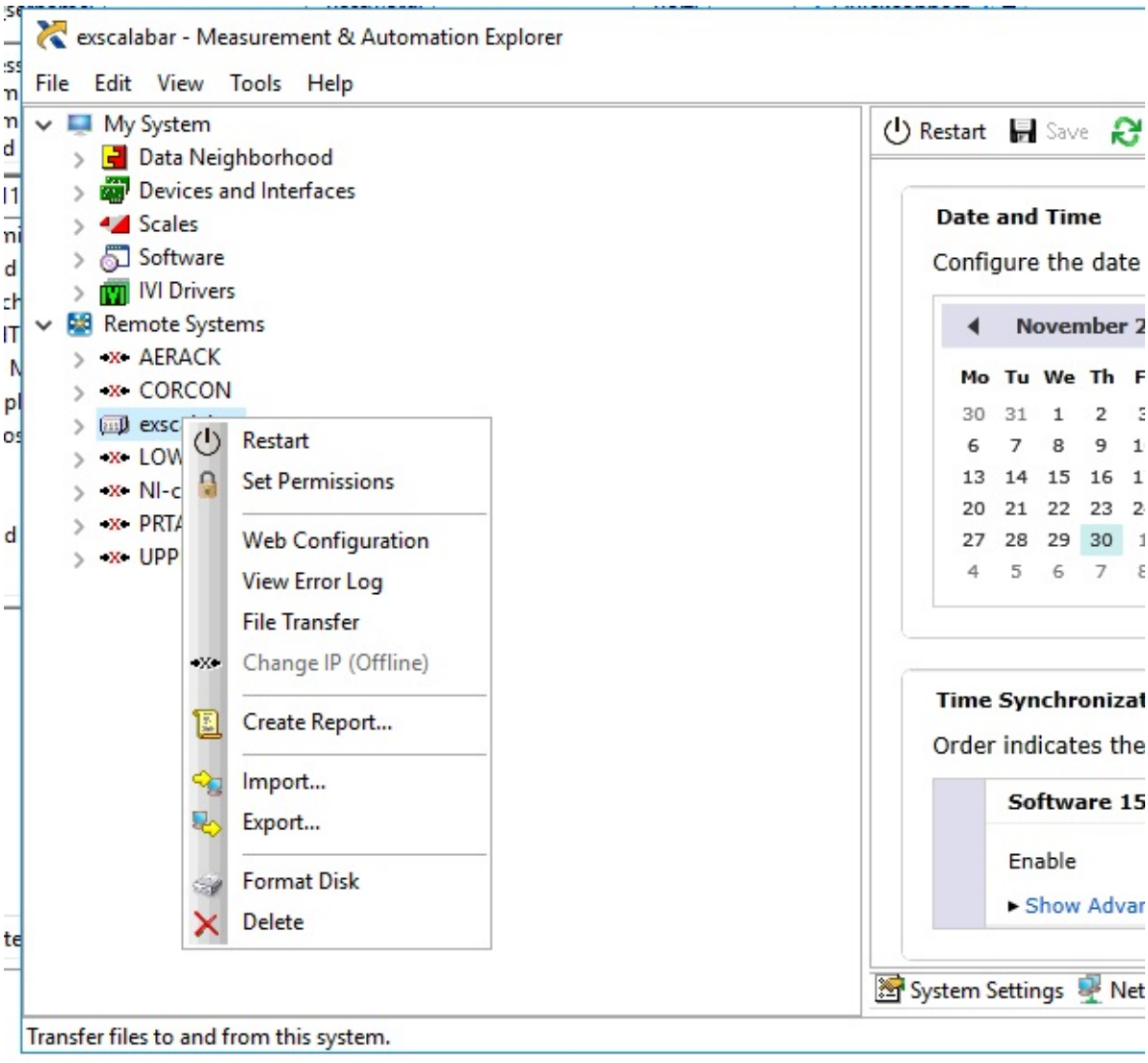
Hardware related configuration is done prior to the operation of the application in the Measurement and Automation Explorer (MAX). In MAX, the hardware cards can be given specific names and the IO can be grouped into tasks. A task defines the IO configuration for a group of channels. This may include (depending on the type of IO)

- voltage range
- timing
- output scale

Channels in tasks may be named to provide a clearer description of the intended use of the channel (but is not necessary for proper function).

Backing up the Configuration

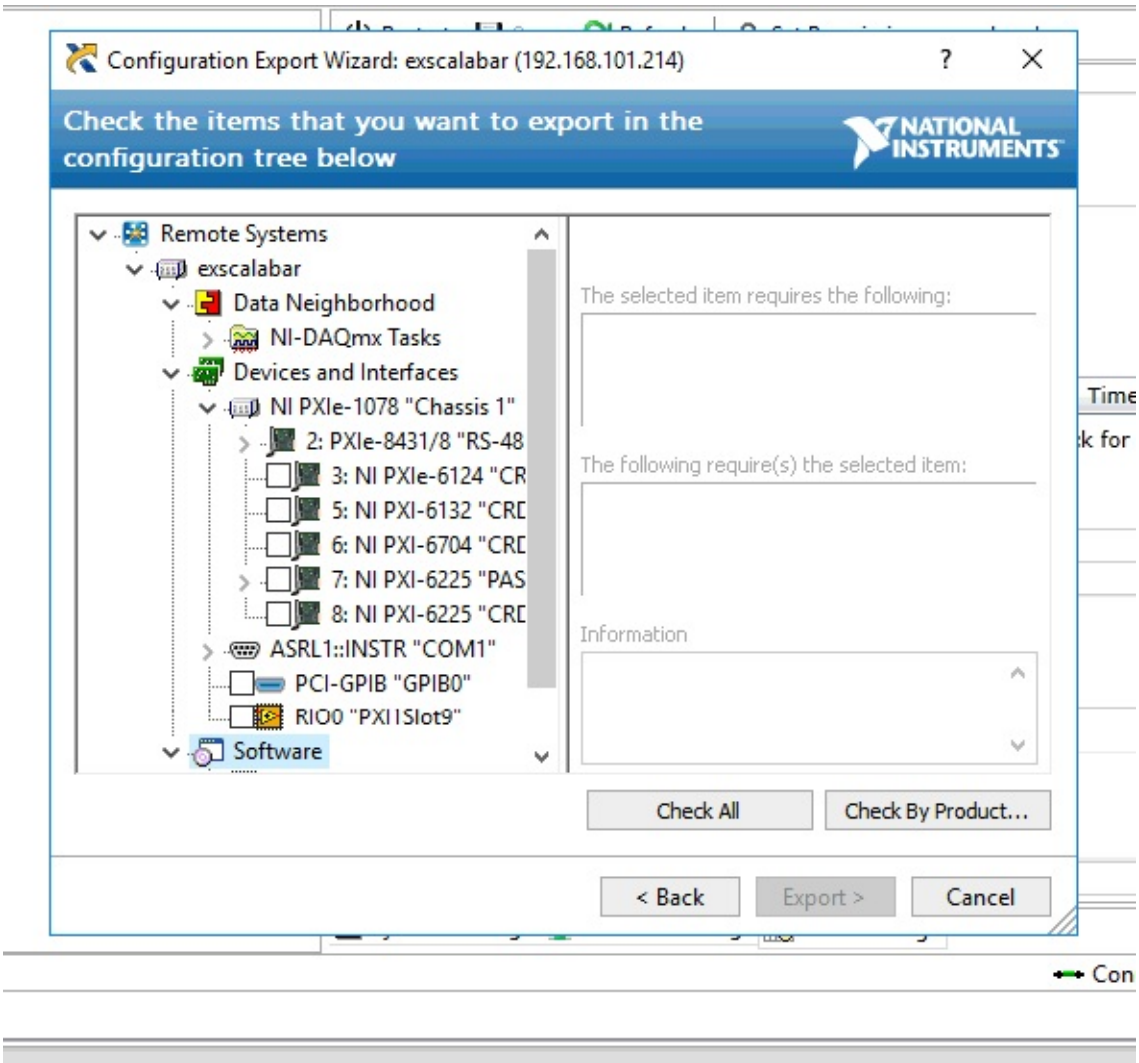
The MAX configuration for EXSCALABAR is crucial to the application operation and as such should be backed up on a regular basis. Consider backing up the configuration before and after campaigns and when any work is performed on the system.



13: Backing up the MAX configuration

To back up the configuration, open MAX and expand `Remote Systems`. As in the image above, right click on the system labeled `exscalabar` and select `Export...`. This will bring up a set of dialogues for the `Configuration Export Wizard`:

1. **Choose a file type and location.** For this step, select `.nce` as the file type. This will allow the user to reimport directly into MAX if necessary. The location is up to the user, just remember where it is stored.
2. Once the location and file type is chosen and the user selects `Next >`, the system will search for exportable items. When the dialog returns, it will contain the information concerning the configuration that can be exported. At this point, no boxes are checked as shown in the snapshot below



   You will want to check the boxes for
   - *all* of the tasks defined under `Data Neighborhood -> NI-DAQmx Tasks
   - the cards listed under `Devices and Interfaces -> NI PXIe-1078 "Chassis 1"` as well as the information associated with `COM1` and the `RIO0 "PXI Slot9" (this is the FPGA) It will be up to the user to determine whether they wish to export the software configuration as this can be rebuilt easily when a system is formatted or replaced (and it may be that the user wishes to use a newer version of LabVIEW)
3. Once the user has made the selections they wish to backup, then click the button `Export >` This will kick off the generation of a report in the next window indicating that the configuration is being saved.
4. If desired, save the report by selecting the button `Save Report..` (this is not necessary as the report does not contain any information that is particularly informative).
5. Close the window by clicking `Finish` and verify that the file is where you intended it to be.

Generating a Report

In addition to saving the configuration, it is also useful to generate an html report. With this report, the user can see a description of each and every task as well as the cards in the system. To create a report, simply right click on the `exscalabar` remote system and select `Create Report..` . You can choose where and under what name you wish to save the report, but the format will always be html. Follow the dialogues to retrieve a report tailored to your needs (although it is recommended that you generate a report for the information that you expect to be in the configuration file). The report will contain a top level html file with support files in a separate folder.

MAX Hardware Description

Below is outlined a description of the hardware as defined in MAX under `Devices and Interfaces` . These cards are used in the following section to define tasks. The original purpose of the cards is defined in the Phase I document outlining recommendations regarding the DAQ system. The cards are defined briefly below in the order they reside in the chassis:

Slot	Card	Name	Description
1	PXIe-8135	PXISlot1	Controller
2	PXIe-8431/8	RS-485 SN:0199CA6A	8 channel RS-485 card with 10 position modular jacks
3	PXIe-6124	CRD_Blue	4 channel card that can capture analog inputs at 4 MS/s/ch. Intended for use with the blue laser.
5	PXI-6132	CRD_Red	4 channel card that can capture analog inputs at 2.5 MS/s/ch. Intended for use with the red laser.
6	PXI-6704	CRD_AO	16 channel DC analog output.
7	PXI-6225	PAS_HK	General purpose multifunction DAQ card.
8	PXI-6225	CRD_HK	General purpose multifunction DAQ card.
9	PXI-7842R	PXISlot9	Multi-purpose reconfigurable IO module (FPGA). Used primarily for handling PAS laser modulation.

Task Description

The following constitutes the list of Tasks as defined in MAX and used by the data acquisition system:

Task Name	Card	Type	Rate	Line	Description
PAS_TEC_En	CRD_AO	DO	On Demand	p0.0:5	Intended to control the power to the PAS cell TECs. **This Task appears to no longer be used.**
Filte_Valve_RTN	CRD_HK	DI	On Demand	p1.0:1	Determine the position of the valve that handles the sample/filter line. **This task is currently not used.**
Power Supply	PAS_HK	DO	On Demand	-	This task is used to toggle power using a series of digital outputs. Interactions with this task are handled by the VI `Controller::Handle Power Switch` and the user interacts with this method through

					the `Power Supply MSG` in the `Controller` library.
				p0.4	Sample pump power
				p0.5	O3 Generator power
				p0.6	Denuder power
				p0.7	Laser power
				p1.0	TEC power
Inlet LED	PAS_HK	DO	On Demand	p1.4	Indicates the current position of the of the inlet valve. If this value is `TRUE`, then the system is sampling from the inlet. If `FALSE`, it is sampling from the cabin. If sampling from the inlet, then the inlet LED will be lit. This Task is handled in the `Change Inlet MSG::Do` method.
Cabin_Valve_RTN	PAS_HK	DO	On Demand	p1.5:6	Intended to retrieve the position of the cabin valve. **This task is no longer used by the syste.**
Blocked Impactor	CRD_HK	AI	1 kHz	ai0	Provides a voltage for indicating the differential pressure across the impactor. This value is averaged over a second and is handled in the `Controller::Common` method.
Filter Valve CMD	CRD_HK	DO	On Demand	p0.2:3	Changes the state of the filter valve. This task is handled in the `Change Filter State::Wrapper` message method. When both values go high, the system is set to send flow through the filter. Otherwise the system will be on sample.
PAS_HK	PAS_HK	AI	10 kHz	-	Retrieve a variety of data associated with the PAS. This task is handled in `ePAS::Get Data`. All measurements are RSE unless otherwise stated.
				ai0:ai4	Photodiode data associated with the photoacoustic cells. This data is bundled into the individual cells as raw waveforms. These measurements are differential.
				ai5:6, ai16:18	Laser RMS data for the PAS photodiodes. This data is averaged over the 10k samples and is placed in the variable `PAS::lRMS`.
				ai19, ai21, ai23, ai33, ai35	Thermistor measurements from each of the cells. The 10k samples are averaged and the voltage is converted to a temperature. The resulting value is stored in `ePAS::T`.
Denuder Valve RTN	CRD_HK	DI	On demand	p0.5:6	Digital inputs that measure the current position of the valve associated with the denuder path. **These lines are currently not used by the software.**
General Input	CRD_HK	DI	On demand	-	General purpose digital inputs. Inputs associated with this task are handled in `Controller::Common`.
				p1.4	Digital input indicating whether the TTL line associated with whether the pump is blocked is active.

				p1.6	Input for interlock.
CRDS Blue Clk	CRD_Blue	CO	4 MHz	ctr0	Data clock for the CRDS blue channels. This clock is used to configure acquisition in `eCRDS::Configure Blue`.
CRDS Blue AI	CRD_Blue	AI	-	ai0:3	Analog input acquired from the PMTs associated with the blue laser channels in the CRDS. The clock for the acquisition is defined by the task above and this task is configured in the same method as the clock.
Filter LED	PAS_HK	DO	On Demand	p2.1	Controls the physical LED on the instrument front panel. This value is handled in `Change Filter State::Wrapper`.
CRD HF Laser Output	CRD_HK	CO	2 kHz	ctr0	This value is used as a *default* value in the `eCRDS` object and is configured for operation in `eCRDS::Configure Blue`. The value can be found in `Blue Tasks -> DO`. The rate is determined by the MAX task.
Acquire Ctr	CRD_HK	AI	-	ai0	**This task doesn't appear to be used in the code.**
O3 LED	PAS_HK	DO	On demand	p2.0	Sets physical front panel LED to indicate that the ozone system is active.
O3 Lamp Output	PAS_HK	DO	On Demand	p2.0	Used to generate a TTL pulse train. This task is called in `O3 Frequency Train Actor::Generate Pulse Train`. This VI runs parallel to the `Actor Core` and generates a continuous pulse train by toggling the output.
Generate Pulse	CRD_HK	CO	50 Hz	ctr1	**Not currently used**
CRD Laser Gain	CRD_AO	AO	On demand	-	This task controls the gain for the three lasers. The task itself is explicitly called by the command object `Change Laser Gain CMD`. The command is sent from the message `Change Laser Gain MSG`.
				ao0	Gain for the red laser.
				ao1	Gain for the first blue laser.
				ao2	Gain for the second blue laser.
PMT Gain	CRD_AO	AO	On demand	ao5:9	Handles requests for changes to the gains the PMT for each of the cells in the CRDS. This task is explicitly called in the command object `Change PMT Gain CMD` which is send via the message `Update PMT Gains MSG'.
405 LED	PAS_HK	DO	On demand	p2.5	Toggles the LED on the physical front panel associated with the 405 nm lasers. This task is handled in the method `Controller::Common`.
Cabin Valve CMD	PAS_HK	DO	On demand	p0.0:1	Toggles the Hanbay valve assocated with the sample line which directs the flow from the cabin or the main sampling line. This task is handled in the `Change Inlet MSG`.
Spk LED	PAS_HK	DO	On demand	p2.4	Toggles the LED on the physical front panel associated with the PAS speaker. This task is called in the command object `Change Speaker State`.

Purge Switch	PAS_HK	DO	On demand	p1.3	Toggles the solenoid valve associated with the purge flow. This task is handled in the message `Purge Switch MSG'.
515 LED	PAS_HK	DO	On demand	p2.6	Toggles the LED on the physical front panel when the 515 LED state changes. This task is handled in the method `Controller::Common`.
O3 Valve	CRD_HK	DO	On demand	p0.4	Controls the state of the solenoid valve associated with the O3 line. This task is handled in the message `O3 Valve MSG`.
Cabin Valve RTN	PAS_HK	DI	On demand	p1.5:6	This input is intended to register the position of the Hanbay valve associated with the cabin/inlet valve. **This task is currently not used in the software.**
CRDS Red Clk	CRD_Red	CO	2.5 MHz	ctr0	Clock for acquiring data from the PMT associated with the red CRDS channel. This tasks is explicitly defined as one of the properties of the `eCRDS` object - `Red Tasks.Clk'.
Red Enable	CRD_HK	DO	On demand	p1.5	Toggles the state of the red laser associated with the CRDS. This task is explicitly called in the command object `Laser Enable CMD` which is in turn called by the message `Change Laser Enable State MSG`.
CRDS Red AI	CRD_Red	AI	2.5 MHz	ai0	This is the task that handles the analog input for the CRDS PMT associated with the red channel. It is explicitly defined as a property of the `eCRDS` object - `Red Tasks.AI`.
Heartbeat	PAS_HK	DO	On demand	p0.3	This task is used in `Controller::Common` to indicate the status of the `Controller`. The digital output is toggled at 1 Hz while the `Controller` is active resulting in a blinking LED on the physical front panel.
Denuder LED	PAS_HK	DO	On demand	p2.2	Digital output that toggles an LED on the physical front panel whenever the state of the valve associated with the denuder is changed. This task is handled in the message `Change Denuder State MSG`.
Blue Enable	CRD_HK	DO	On demand	p1.6:7	Toggles the enable state of the two blue lasers associated with the blue CRDS lasers. As with the `Red Enable` task above, this task is explicitly called in the `Laser Enable CMD` command object.
Denuder Valve CMD	CRD_HK	DO	On demand	p0.0:1	Digital outlines required for toggling the Hanbay valve associated with the denuder. This task as with the `Denuder LED` task is handled in the message `Change Denuder State MSG`.
CRD Red Laser Output	CRD_HK	CO	1 kHz	ctr1	Counter controlling the red CRDS laser rep rate. This task is explicitly one of the properties of the `eCRDS` object. The rep rate can be changed in the command object `Reset Red Laser Rep Rate`.
Error LED	PAS_HK	DO	ON demand	p0.2	Toggles an LED on the front panel to indicate an error state in the instrument. **This is currently not used by the software.**
					Data acquisition clock for the CRDS blue

CRD Blue Laser Output	CRD_HK	CO	4 MHz	ctr0	analog inputs. This task is explictly a property of the `eCRDS` class - `Blue Tasks.Clk`
O3 Lamp DO	PAS_HK	DO	On demand	p2.0	**This task is not used by the system.**
660 LED	PAS_HK	DO	On demand	p2.7	Toggles an LED on the physical front panel to indicate whether *any* red laser is enabled in the system. The value of the output is set in `Controller::Common` .

Web Service

Web Service Resources

Web service methods are found under the main web service under the folder `Web Resources`. They are further found under nested resources. The nesting aids in clarifying the functionality of the individual methods. The methods are divided into following nested resources:

- `General` - defines general functionality that does not fall under the auspice of a particular sub-system
- `CRDS_CMD` - CRDS specific functionality
- `PAS_CMD` - PAS specific functionality
- `Calibration` - functionality associated with calibrations
- `Humidity` - functionality for controlling the humidity control loops found in the controller
- `tetech` - TE Technology TEC related functionality
- `meerstetter` - Meerstetter TEC related functionality

Web Service Component Locations

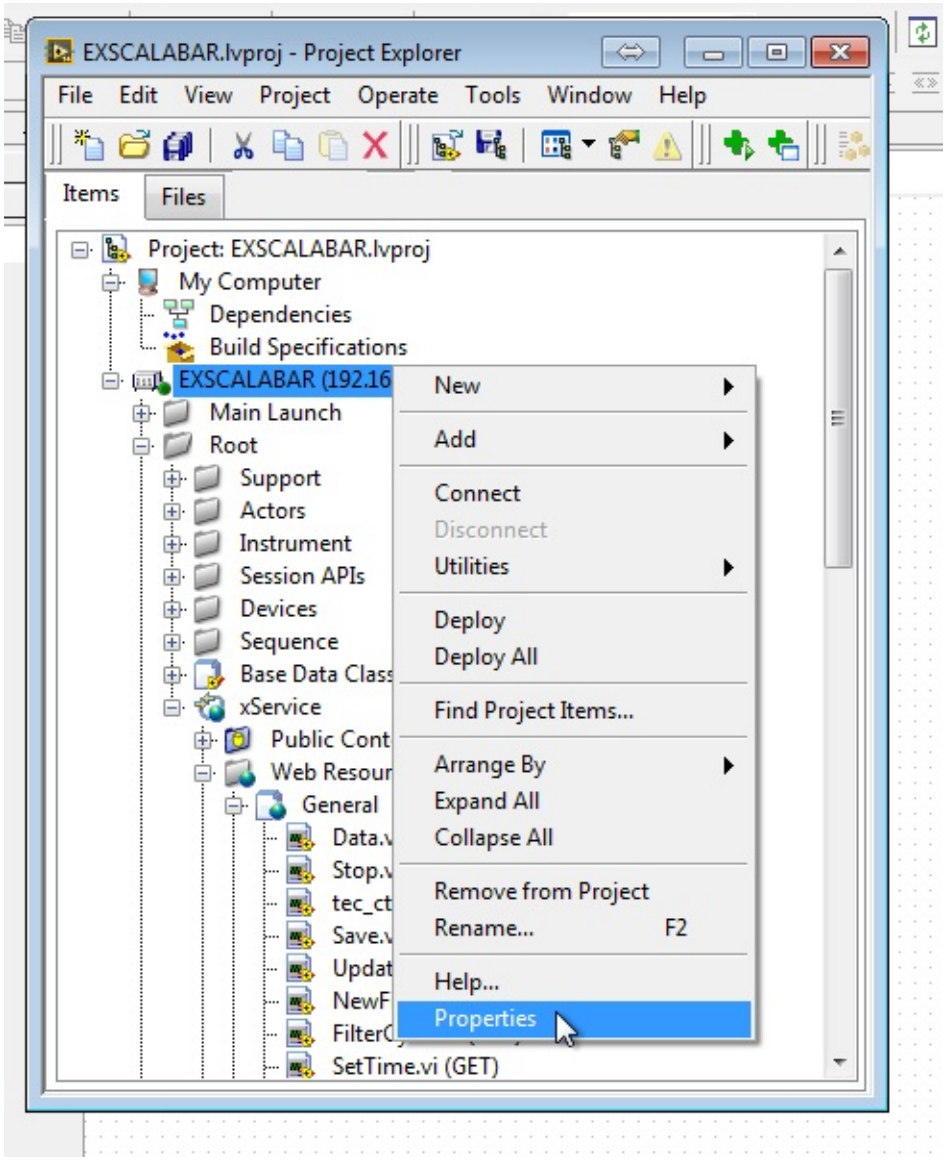
All requests to the web service will appear as follows:

```
http://[server-address]:[service-port]/xService/[nested-resource]/[method]
```

The request for data is found in the `Data` method under the resource `General`, so the request for data will look like

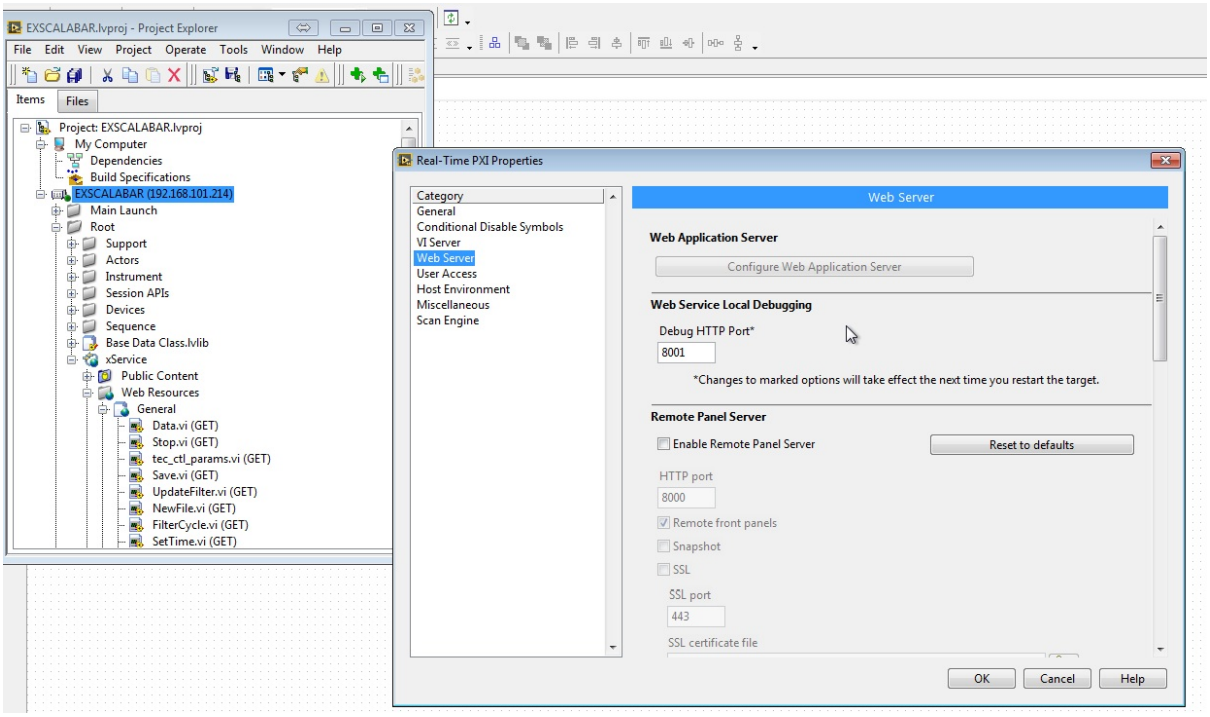
```
http://[server-address]:[service-port]/xService/General/Data
```

The web server port can be configured through the project properties for the remote target by right clicking on the target in the project and selecting properties as shown below.



14: Project properties

In the properties dialog, select the `Web Server` tab.



15: Web server properties

In this dialog, you can select the `Debug Http Port` which is the port that is used when running the system from the development environment (default is 8001). When the service is published, the port will be 8000 for contacting the web service.

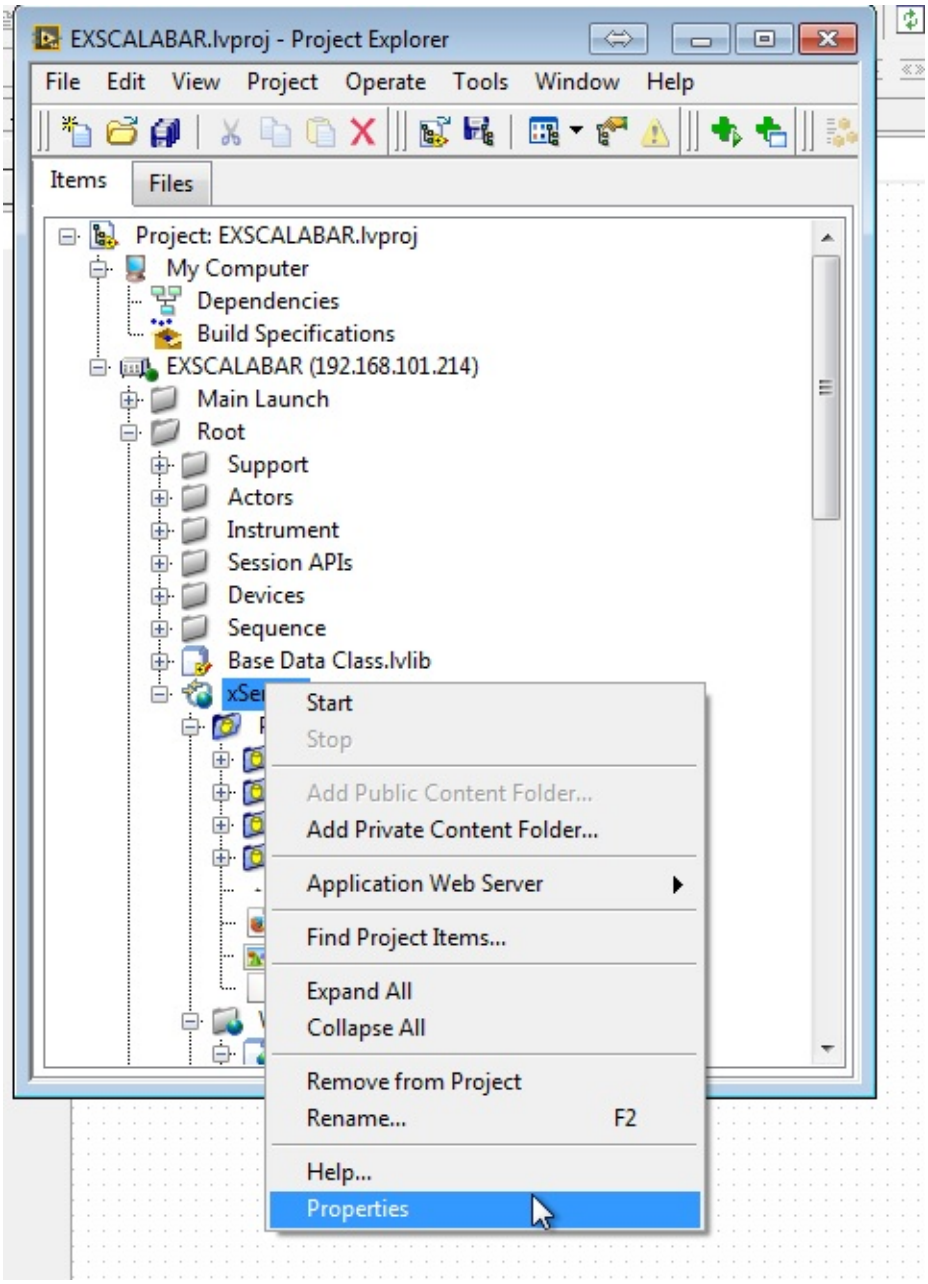
This port is also the port that the user will use to access the instrument page that is hosted on the server. To access the main page, the user will go to the following address:

```
http://[server-address]:[service-port/xService/index.html]
```

This will bring up the most current version of the instrument page stored on the server.

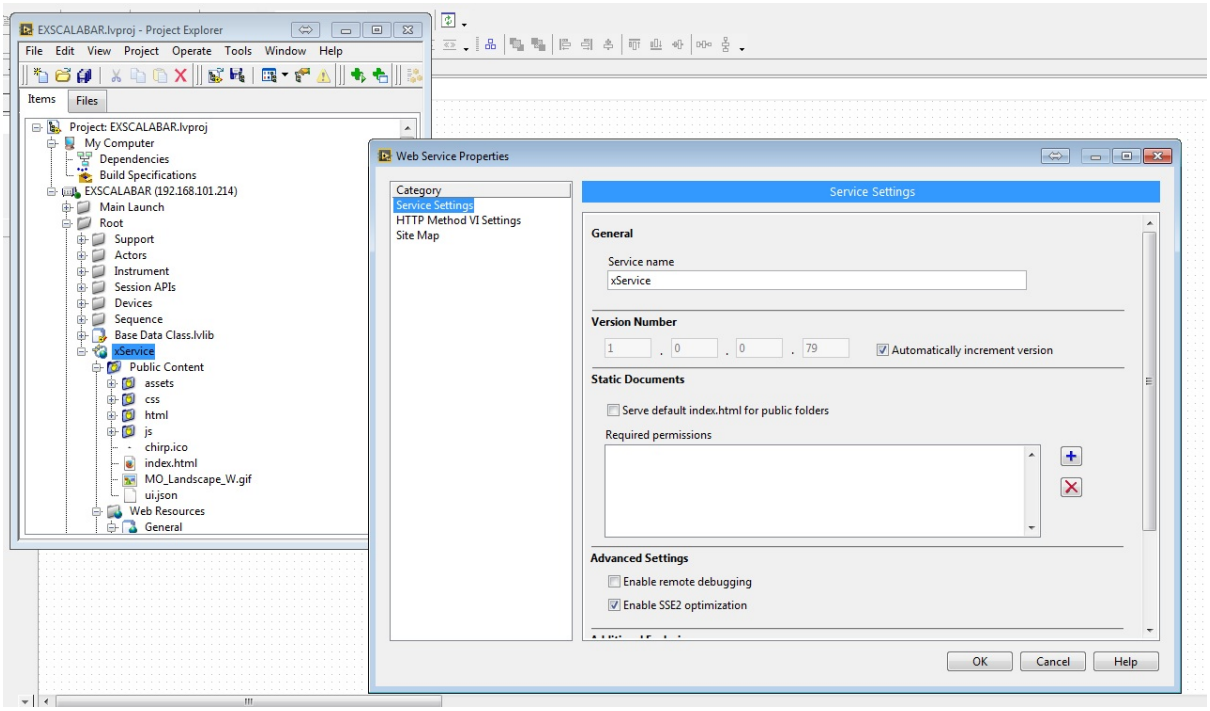
Web Service Method Configuration

Web service methods are configured via by right clicking on the web service in the project and selecting properties.



16: Configuring web methods.

This will bring up the window below. Select the `HTTP Method VI Settings` tab to configure the methods.



17: HTTP method settings

The `HTTP Method VI Settings` will contain a list of all methods in the project. Selecting a method will allow you to configure that method. All web service methods in this project use the `GET` http method except the command `saveCalFile` which uses the `POST` http method. In the latter case, we are sending an XML calibration file to the server for storage.

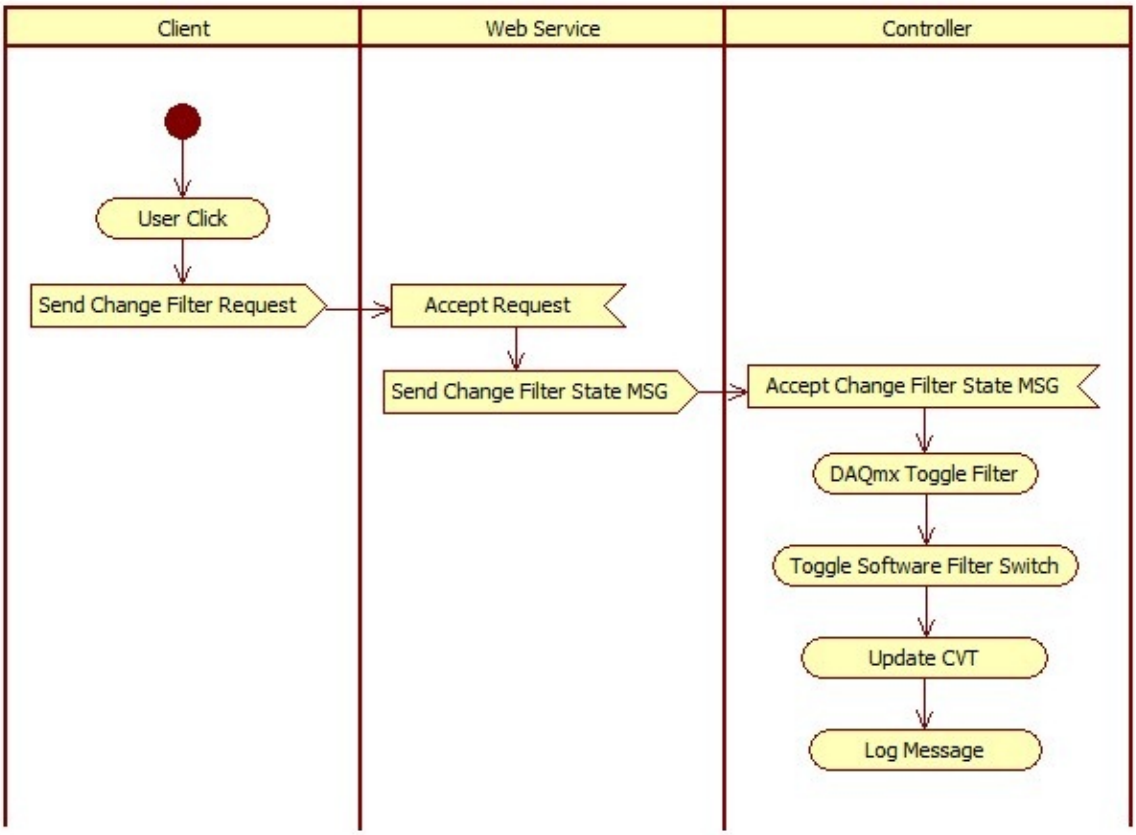
For the most part, the web methods that are configured with `GET` do not require further attention. The `Output Type` in the bottom panel under `Web Service VI Properties` is not used. These are commands that are sent *to* the server. The exception to this are two VIs which are used to *retrieve* data from the server. These two VIs are `Data` and `cvt`; the former is used to retrieve the current data from the server while the latter is used to retrieve an updated CVT. In both of these cases, the `Output Type` is set to `Stream` and both `Use headers` and `Buffered` are checked. `Use Headers` is selected because we are configuring the header to accept a cross-origin request.

Communication Errors

There are two kinds of communication errors that can occur. The first occurs because the web service is not available. In this case, the client will throw an error (in the console) to the effect of `ERR_CONNECTION_TIMED_OUT`. In the second case, the web service may be available, but the message queues are not good. This may be due to a real problem with the `Controller` or simply because the `Controller` is not running. In either scenario, attempts to access the queue will result in an error and the server will return the code 503.

Communication with the Server

When an http message is sent from the client to the server, a series of messages are triggered. The http message is sent from the client to the web service. The web service then forwards the message to the `Controller`. If the message is to be handled by the `Controller`, then the message will stop there and be processed in the `Controller::Actor Core`. An example of this is shown below for request to change the filter state.



18: Changing the current filter state.

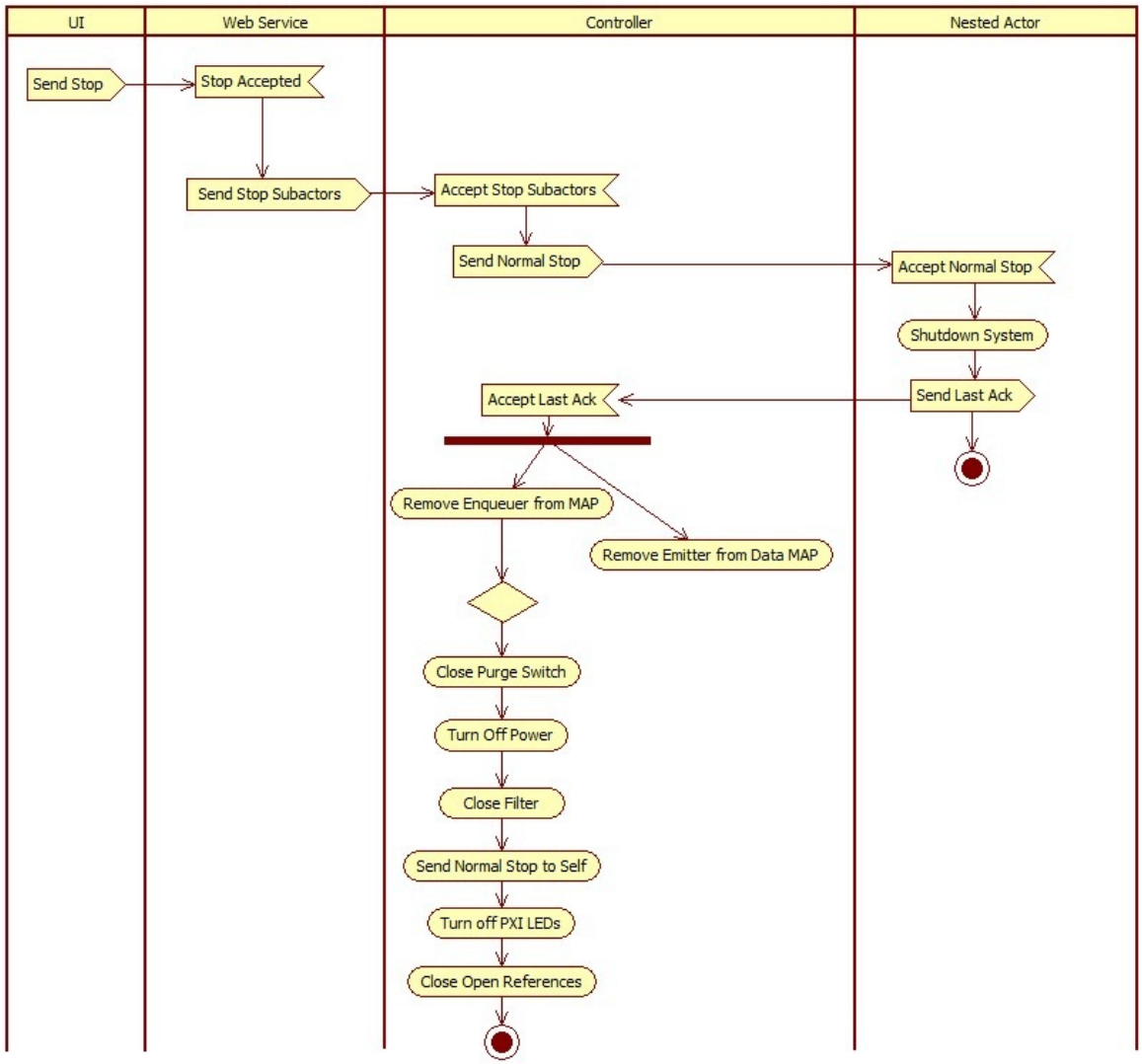
Shutdown

The process of shutting the system down is depicted in the image below. When the user requests the system to shutdown, the user interface will communicate this through the `stop` method under the `General` web resource in the web service. This method in turn will send a message to the `Controller` called `Stop Subactors`. The purpose of this message is to shut the system down in an orderly fashion.

The `Stop Subactors` message will trigger the `Controller` to send a `Normal Stop` message to *all* nested actors (depicted in this image as a generic lane called `Nested Actor`). The details of the shutdown process for each nested actor are unique to the particular actor, but once the actor has completed shutdown, it will send a `Last Ack` message to the calling actor (in this case the `Controller`).

The `Last Ack` message is handled by the method `Controller::Handle Last Ack Core`. In this method, the ID associated with the nested actor will be removed from the `Enqueuer MAP` in the `Controller` and decrement the number of enqueuers registered with the MAP object. In addition, if an emitter with that ID exists, it will also be removed from the data MAP (`MAP DVR`).

When the number of enqueuers registered reaches 0, then the `Controller` itself will shutdown. This process is depicted in the `Controller` lane of the image after the decision point. The `Controller` will close the purge flow switch, turn off power to subsystems, set the flow to filter and then send a `Normal Stop` message to itself. The `Normal Stop` is handled by `Controller::Stop Core`. In this method, the PXI LEDs are shut off and open references are closed.



19: Shutting the system down

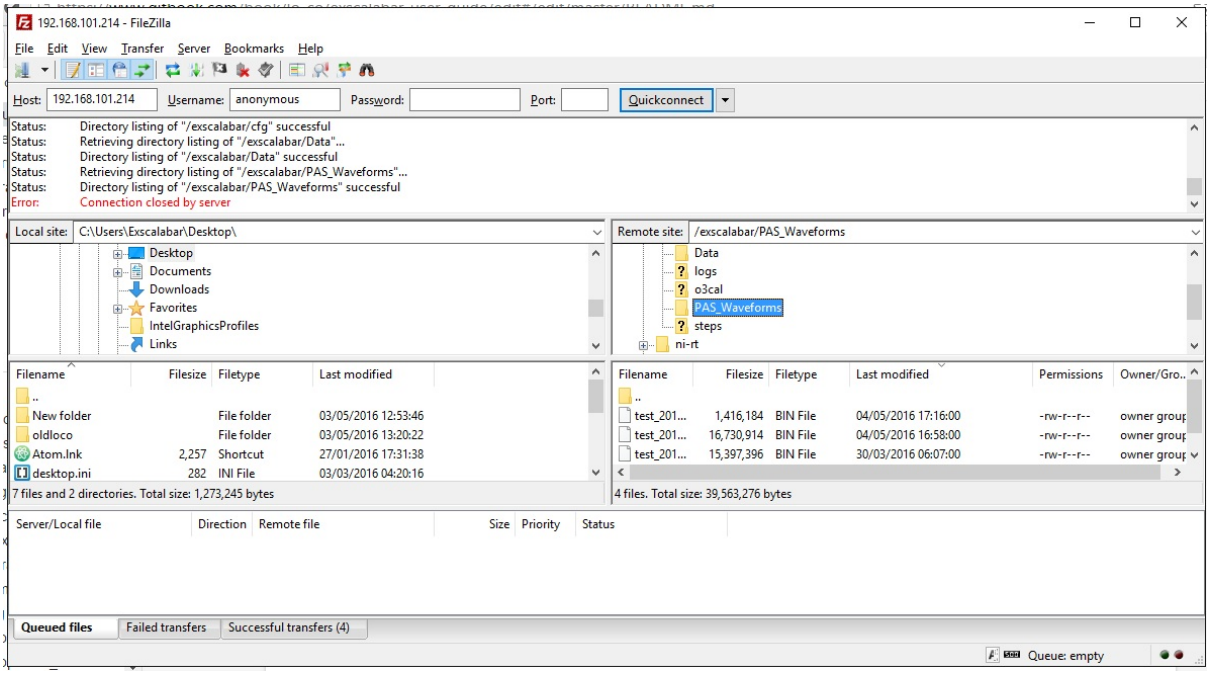
It is important to note that when the `Controller` is shutdown, not all other processes are stopped. The web server continues to run and can only be stopped depending on the mode. If the web server is started from the development environment, then the user simply right clicks on the service and selects stop from the options. If the web server is started as a boot process, then it will not be stopped unless unpublished (i.e. right click on the web server and select Unpublish).

In the development phase, the maintainer can determine if the system is shutting down properly by what happens when the web server is stopped. If the system is properly shutdown and the server stopped, then ***no*** VI, class or library should be locked. If any of these are locked, this indicates that the shutdown was not complete and there is a dangling process still open.

Access

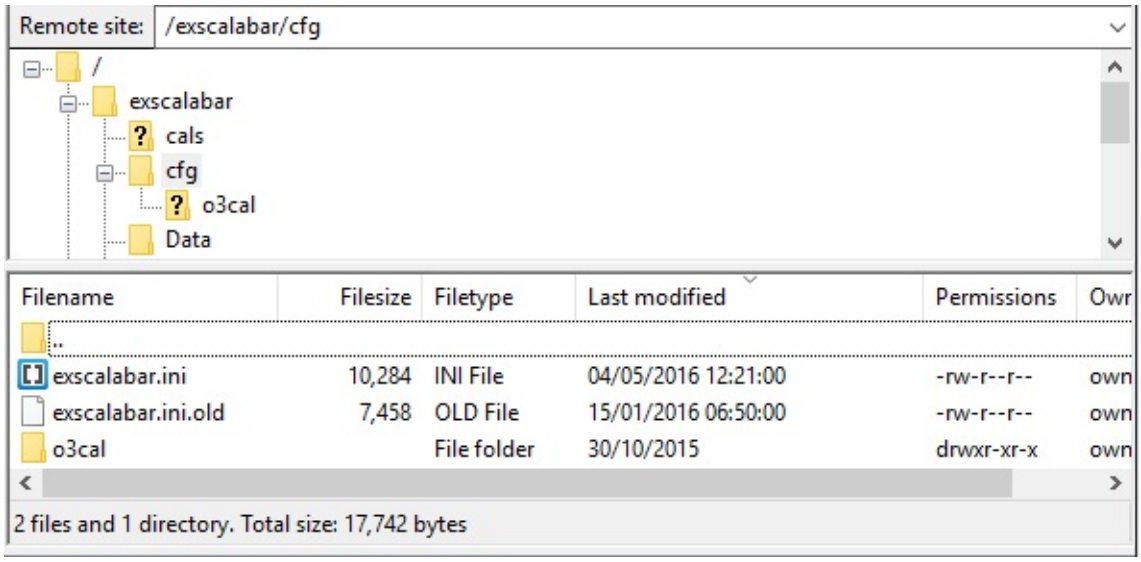
The configuration files reside on main drive of the server. This drive is accessible via FTP or an http get. The following examples use FileZilla to access these files via ftp, but any ftp utility should be sufficient. In order to access the server drive, the user must have the IP address of the server available. **There is no user name or password required at this time to access the server**, but the server may be configured to utilize this functionality.

In the image below, we have logged on to gain access to the server drive using FileZilla. The main directory in which files are placed by default is the `exscalabar` directory.



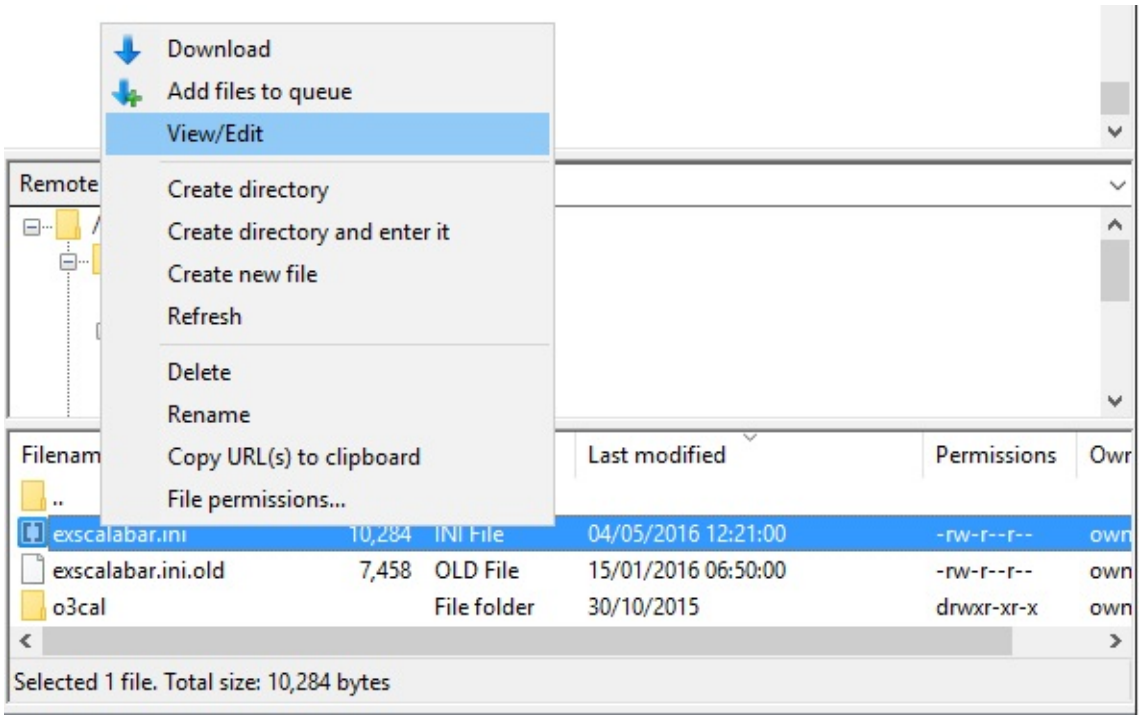
20: Filezilla

One in the `exscalabar` directory, navigate to the `cfg` directory. This directory contains the INI file for the system configuration. This file is called **`exscalabar.ini`** . Located within this directory is also the location of the calibration directory (currently called `o3cal`).



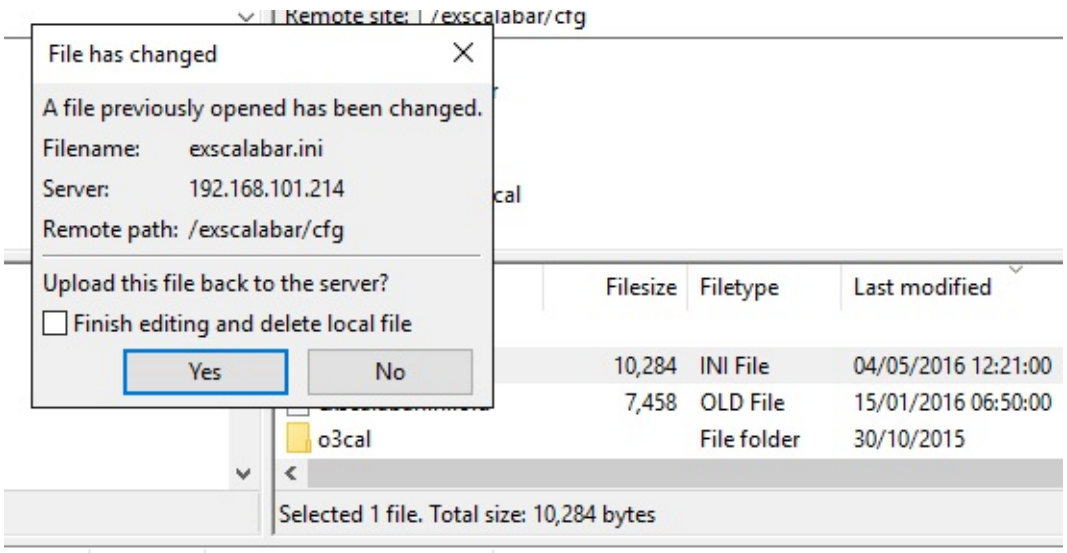
21: Navigating to the configuration directory

If the user wishes to edit the file that is on the server, there are two choices on how to do this. The user may right click on the file and select `View/Edit` (if using FileZilla). This will open a file in whatever application the user has configured the client to open an INI file (the default is Notepad on Windows machines).



22: Viewing and editing the configuration file

If the user opts to use this method for accessing the INI file, FileZilla will prompt the user to upload this file back up to the server. If the user is finished editing the configuration file, they may choose to upload the updated file and delete the local copy or simply just upload the current file.



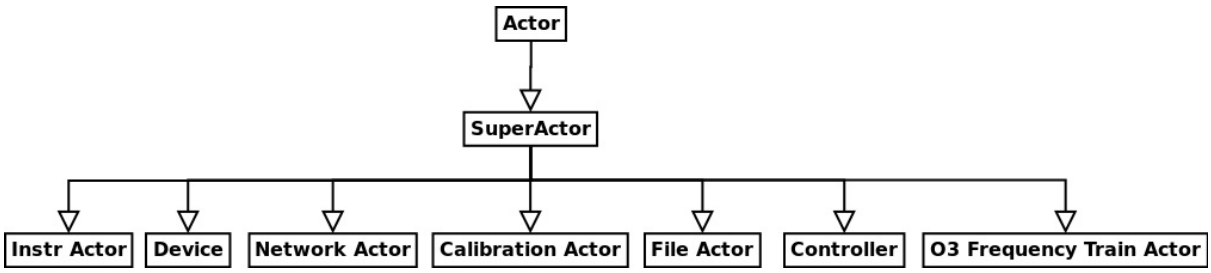
23: Reuploading the configuration file

A safer alternative to this method is to download the file to somewhere that is backed up and edit the file and upload the edited file. **Best practice: before altering the current file, save that file on the server as something that is not** `exsalabar.ini` **(such as** `exscalabar.ini.old` **).** This will preserve the current copy on the server for reuse if needed.

Software Functionality

A Brief Overview

As stated in the *Launch* section, the application is based on the `Actor Framework`. In this system, their are 7 specific actors. All of these actors are children of an object called `SuperActor` which in turn is a child of `Actor`. `SuperActor` is an abstract class that provides a common data structure (contains the ID as well as the CVT, Log and Configuration Sessions) as well as an abstract method for initializing the CVT. In addition, it implements two key `Actor` methods - `Pre Launch Init` and `Stop Core` - that are used to open and close the sessions appropriately. The general hierarchy of the actors implemented in this application is shown below.



As noted in the *Launch* section, this diagram does not describe the operational architecture, it simply notes the inheritance structure. While actors such as `Controller` and `Calibration Actor` have discrete functionality, other actors are generic and utilized multiple times to encapsulate behavior of different objects in the system (i.e. `Instr Actor` and `Device`). More will be said about these different objects in the following sections.

General Patterns and Data Structures

The system consists of some core functionality that most of the key features of the system rely on. These common structures and patterns are described below. The more complex structures that invoke these patterns are defined in following sections.

Data

The `Data` object is a simple abstract class that contains three properties:

- `measTime` - a `Time Stamp` intended to provide information regarding *when* the measurement was taken
- `Data ID` - string providing a unique ID for the data object
- `write time` - boolean indicating whether the time should be included in written or transmitted output; if this is `TRUE` then the time stamp is added to the output string
- `ValidFor` - float providing a time in seconds that the data is valid.

This class is intended to serve as a wrapper that can contain any structure of data. The base object itself contains three abstract methods for different writing operations that are intended to be overwritten (they are all abstract so invoking the base methods will do nothing). Two of these methods are for serializing the data to strings while another is intended to provide a unique header to associate with file data for each piece of data that is written to file. The methods are:

- `Serialize-JSON`: provide the code to serialize the data to the proper json format for transmission over the network. To validate an json, you can use any IDE that has a json linter included (atom has a json package that will lint the structure for you) or you can copy and paste produced json to this json validator. The invoking the base method will write the time *if* `write time` is `TRUE`, otherwise the method will write nothing and all information concerning the data should be written in the child class implementation. **All json serialization processes expect the final caller to provide the bounding brackets.**

- `Serialize-Text` : intended to produce a tab delimited ascii stream for writing to file. Similar to the `Serialize-JSON` method, if `write time` property is `TRUE` , then the base method will produce the time. Otherwise, it produces nothing.
- `Generate Unique Header` : produce an array of strings that are intended to be written at the top of the output file. If `write time` is `TRUE` , then this will provide a header for the time that will be `[ID].Time` where `ID` is the `Data ID` .

> For the two serialization methods described above, if the difference between the current time and the measurement time (`measTime`) is greater than `ValidFor` , then both methods will throw the error 408, a custom error that indicates the data is stale.

The object `Data` is found in the library `Base Data Class` along with some basic data wrappers. The functionality for the basic data wrappers has not been entirely implemented and should be filled out as needed.

Sessions

Sessions are used throughout the project. Sessions are intended to invert control so that the session itself is in control of initialization and shutdown. This is important for processes in an asynchronous system such as the EXSCALBAR software that are shared across actor boundaries. These processes might include file writing, network communication or serial communication.

The session object used in this system is called `Session-ROOT` and was developed by others. The actual session is found in `user.lib` . When building a new development system, the developer will have to download and install the package. This package can be found currently here.

The API for the session framework contains two methods that are used throughout the application:

- `ROOT-Obtain Session` - a session is populated with the proper fields and digested by this method when the session is first invoked in the caller. The output of this is a DVR that wraps the session object. This method will check to see if a session exists with the current handle - if it does, then the object will be initialized via `initialize Session Data` and a DVR containing the session object will be passed out. If the session has already been initialized, the stored reference will be passed out and the number of live handles implemented.
- `ROOT-Release Session` - this method is called to release the reference. Each time this method is called, the number of live handles is decremented. Once this value reaches 0, then the method `clear Session Data` is called and the session officially is closed. The live count requirement can be overriden by setting `force destroy?` to `TRUE` . This will immediately invalidate the reference and any other holders of that reference will get errors if they attempt to access the session located within the reference.

Session implementations that override this class will need to provide specific functionality for initialization and shutting down. This logic is implemented in the two protected VIs `intialize Session Data` and `clear Session Data` respectively.

All implementations of this object are found in the folder in the project `Session APIs` . These are

- `CVT` - session for performing operations on the current value table maintained by the server
- `System Log` - handles the system log for writing data to the log
- `Exscalabar FPGA` - maintains the handle for the FPGA
- `Config Data` - provides methods for interacting with the configuration file
- `Serial Session` - handles serial communication interactions. This session is used generally by the devices.

Common Patterns

The Command Pattern

The command pattern is a common software pattern often found in scripted programming languages. The command pattern consists of a root object which contains a single common method. This method in other implementations might be called `Execute` . An invoking module will the instantiate and have a concrete method for handling the command.

The command pattern in LabVIEW is similar to the queued message handler. In the queued message handler, one might find

Implementation

The implementation of the `Command` object can be located in the EXSCALABAR LabVIEW project under the folder `Support`. The class is abstract in the sense that it is not intended to be called without a concrete implementation. The `Command` has the following properties and methods:

The `response` property

The response property is a queue containing a boolean value. This property is utilized in the `Send Command and Wait for Response` method to determine whether the command was successfully implemented.

The `Handle Command` method

This method is the primary method of the `Command` class. This method *must* be overridden (a requirement). This is the method in which the code to be executed will be contained. This method is **protected** and called by the **public** function `Execute`.

The `Execute` method

This is the outward facing method that will be called by the invoker and is therefore **public**. In this method, the `Handle Command` method is called followed by the `Command Reply` method which will be used to indicate that the command was successful to the sender. This method is static and reentrant.

The `Send Command and Wait for Response` method

This method can be called by the sender of the command to 1) send the command to be handled and 2) wait for a response to determine whether the command was successful. This method:

1. Sets up the response queue (calls `Obtain Queue` and stuffs the reference into the `response` property).
2. Sends the command via the command queue.
3. Waits for a response as to whether the command was correctly executed
4. Destroys the response queue via `Release Queue`.

Maps

In software engineering, a map structure is an associative container that stores elements as a combination of a key value and a mapped value. The key values are used to identify elements and the mapped value stores the content associated with the key. There are two map structures that are used extensively by the application and both are found in the `Controller` - `MAP DVR` and `Enqueuer MAP`.

In all cases, map structures are built using a variant as a container. The variant itself contains no data but the portions of the map are stored as attributes of the variant. Each map will have two attributes plus the mapped values. The attributes are:

- `IDs` - this is an array of strings that contains all of the IDs in the structure. The mapped values are a one-to-one correspondence with these keys.
- `NumActive` - this is an integer that indicates the number of mapped values present

Handling the Enqueuers

The `Enqueuer MAP` is intended to store the enqueuers of all nested actors in the `Controller`. Nested actors are launched from the method `Controller::Launch Nested`. The very first thing this VI does is to check to see if the ID of the actor that the `Controller` is attempting to launch already exists - if it does, it will not launch it again. This is important for future work - the maintainer must make sure not to duplicate the existing IDs if another nested actor is launched.

If the nested actor does not exist, then the system will call `Actor::Launch Nested` with the nested actor as input. If the launch is successful, it will place the enqueuer into an attribute with the given ID, increment the number of active enqueuers and add the ID to the `IDs`. This ID will be used to move messages around the system.

When the system is shutdown (in this case, the web service method `Stop` is called), the message `Stop Subactors` will be called. This in turn calls `Controller::Stop Nested` which will run through the list of enqueuers in the `Enqueuer MAP` and send a normal stop to all of the nested actors. At this point, the `Controller` is not shutting down. When each nested actor exits, it will send a `Last Ack` message to its caller (the `Controller`). This `Last Ack` message is handled by the `Controller::Handle Last Ack Core`.

As each message is processed, the nested actor enqueuer will be removed from the map and the `NumActive` attribute is decremented. When this value gets to 0 (i.e. no enqueuers remain), then the `Controller` will run through its shutdown by sending a normal stop message to itself.

Handling Data

The `MAP DVR` handles the data that is used by the system. This map structure is very similar to the enqueuer map. But it does not keep track of the number of mapped values in the structure. In addition, it contains an array of booleans called `Registered`.

On startup, the system will need to register all data emitters with the `MAP DVR`. The registration is performed in `Controller::Register Emitters` and tells the system *what* data to expect. It will register the ID with the `IDs` attribute described above and then provide a `Registered` flag (set to false to indicate that the map has not received good data yet). The mapped value associated with each ID is a `Data` object (described in a following section).

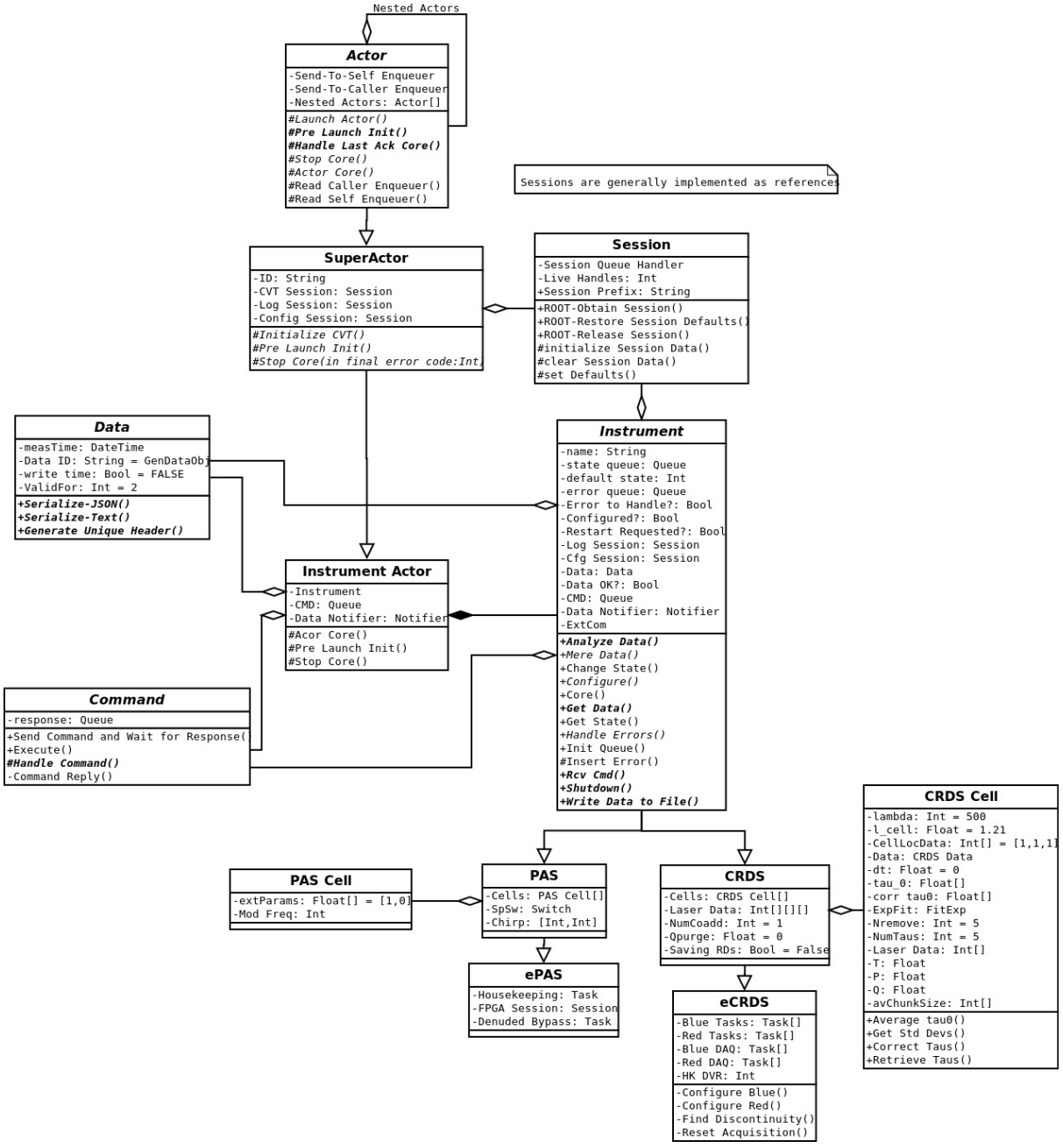
The purpose of the `Registered` flag is to provide flexibility. While the data structure is not entirely important when the system is communicating with the client, it is of the utmost importance when writing to file. Before the system writes to file, it will need to know the exact structure of the data that is to be written. The array of flags in `Registered` tells the system that all data that we expect has been received. No data will be written to file *until* all flags are true.

Flags are set to true when a genuine data packet has been received. At this point the data object stored in the mapped values is concrete (i.e. it is no longer the generic `Data` object). This is checked in the `Exe Write Main::Wrapper` method. Once all data objects are registered, a `Registration Complete` flag is added to indicate that the registration is complete and we are no longer waiting for good data to write to the file.

When writing to the main data file, the data contained in the `Data` object will be serialized to text using the method `Data::Serialize - Text`. The use of the map structure guarantees the order by using the `IDs` attribute. Similarly, the data is written on request by the web service using the same method but calling the `Serialize - JSON` method.

Instrument Software Functionality

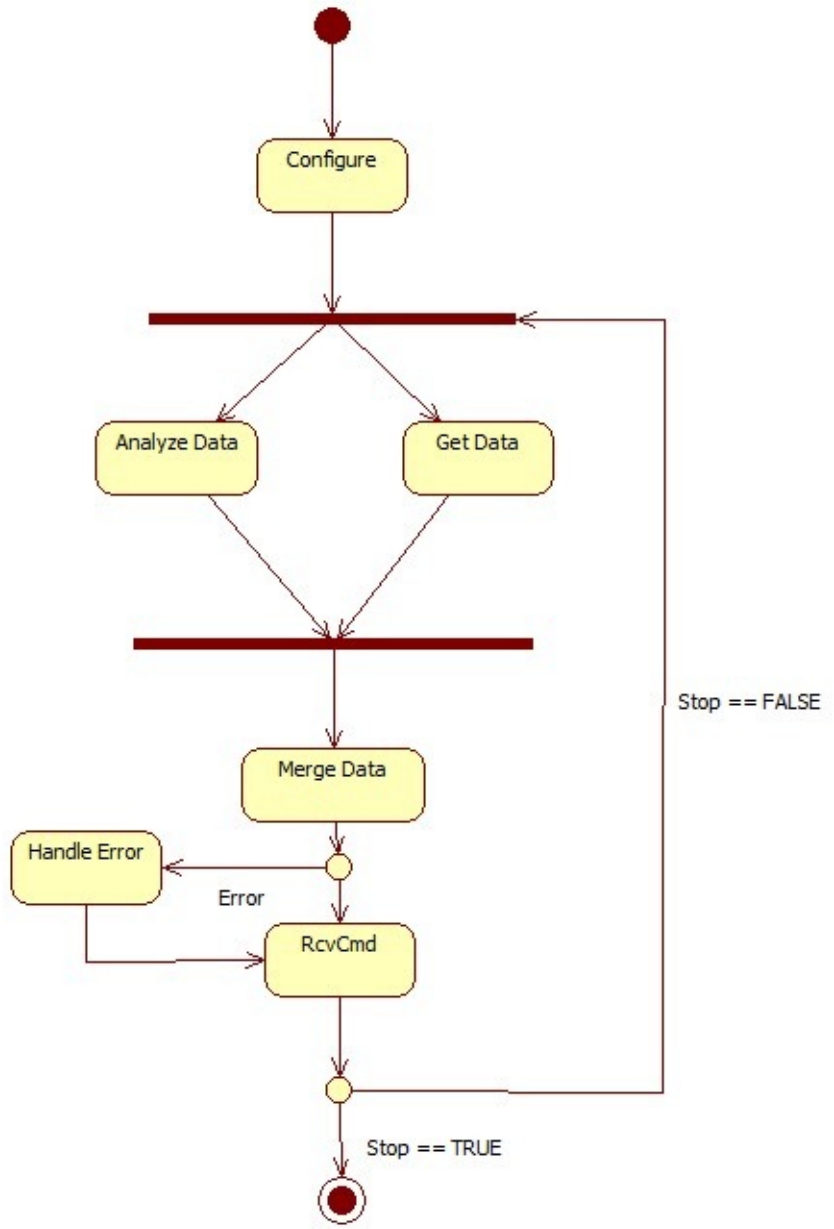
Instruments are coherent objects that represent some large scale functionality. Instruments use data from devices and generally require some further analysis to produce a final data product. In this project, the two instruments that are clearly defined are the PAS and the CRD whose functionality are described by the `CRDS` and it's child `eCRDS` and the `PAS` and it's child `ePAS` which are described below. The object hierarchy are described in the diagram below.



24: Instrument hierarchy.

Instrument

The `Instrument` object is the parent of all instruments. This object is abstract - it does nothing in and of itself but sketch out some basic information and operations that are used by each of the instruments. General functionality is defined in the method `Instrument::Core` and the state transitions are shown below (although `Instrument::Configure` is not included in this method as it is expected that the user may wish to attempt to initialize elsewhere as with the `Instrument Actor` described below). It is important to note that the use of the method `Core` is not required to use the `Instrument` object; this method strictly encapsulates the intended use of the object.



25: Instrument states.

Functionality of the `Instrument::Core` picks up after configuration. This method is just a state machine with the following states defined:

- `Acquire`
- `Wait`

- `Handle Error`
- `Quit`

In `Instrument::Core`, the states are driven by a queue which uses a enumerated type to strictly define the states. If the queue is empty, then the machine will choose a default state that should be defined when the class is instantiated. Generally, the default state is `Acquire`.

`Acquire` encapsulates several methods:

- `Get Data` - this method is used to retrieve data
- `Analyze Data` - performs analysis on retrieved data
- `Merge Data` - merges analyzed data into the main object stream as necessary

To fully utilize available processing power, the `Instrument::Core` method uses pipelining to analyze the previous iterations data while obtaining the current data. The reason that this technique is utilized is that there is generally little computational overhead to retrieving data; all of the computational power is generally put toward the analysis. Using this technique prevents overruns, taking advantage of CPU down time during the retrieval.

Once the data is acquired, the analyzed data is merged into the main object stream as necessary. In some cases, data is required for future calculations. For instance, the CRD will require the previous τ to determine how far out the system will fit the data.

After merging the data, the state machine will handle any errors that were thrown during the whole acquire phase (including analysis). This is performed in the method `Instrument::Handle Errors`; this method should be overridden to handle errors specific to the implementation of the instrument object.

If there were no errors or the errors have been handled, the system will transition to the `Wait` state. Here, the machine will handle an interaction with the client. This state calls the method `Instrument::Rcv Cmd`. This method can be overridden by the implementation, but also can be used as is. The base implementation will check for any commands that are coming through on the `CMD` queue owned by the class. If there are any commands in the queue, they are dequeued and the `Command::Handle Command` method is called. In this application, the base method is used to push messages from the owning `Instrument Actor` down to the `Instrument` implementation itself as described below.

In the `Rcv Cmd` method, the stop signal will be handled. If the signal is received, then the state will be changed to `Quit` and the `Instrument` will exit. If the state is not set to `Quit`, then the system will return to it's default state, generally `Acquire`.

PAS Functionality

PAS functionality is defined over several classes:

- `PAS` - a child of the `Instrument` class that defines PAS functionality that is generic to the operation of the PAS. This instance will not implement `Instrument::Get Data` as this is particular to the implementation. This implementation will perform basic configuration (such as cell number and description), analyze data using procedures defined below and merge some of that analyzed data for future analysis. In addition, it will handle the speaker cycling if the mode is automatic (in `PAS::Rcv Cmd`).
- `PAS Cell` - defines some basic parameters for the individual PAS cells in the instrument. There are no methods defined on this object.
- `ePAS` - a child of the `PAS` object that defines methods and properties specific to the EXSCALABAR implementation of the PAS. The `ePAS::Get Data` implementation retrieves data from 1) the microphone via the FPGA and 2) housekeeping data from the DAQ card. The housekeeping data is the laser photodiode data and the thermistor data. **It is important to note that the `Get Data` routine expects there to be 5 channels - if future development requires a change, this should be accounted for.** Ideally, this should be flexible. In addition to the `Get Data` method, this object also implements other methods such as `Configuration` and `Rcv Cmd`. The former executes configuration routines specific to the implementation while the latter handles commands sent to the PAS by the system. The implementation of the `Shutdown` routine is used to close the DAQ housekeeping task as well as release the FPGA session held by the `ePAS` object.

Calculations

All calculations and analysis occur within the general `PAS` method `Analyze Data. Waveform data retrieved from the previous iteration using the` Get Data` method is handled cell by cell.

If the speaker is on:

1. The method takes the microphone and photodiode data and returns the magnitude and phase of the time domain signal using an FFT with no window.
2. The phase at the resonant frequency is returned for both the microphone and the photodiode and stored in the `PAS Cell Data` properties `phi_pd` and `phi_mic`.
3. The spectrum is stored in the `PAS Cell Data` property `freqData` as a waveform.
4. The micrphone spectrum is then fit with Lorentzian distribution across the speaker signal domain (specified by the user input variables `PAS::Chirp`). This isa fit using the `Nonlinear Curve Fit` routine in the VI `Get Q Factor`. The fit uses the template `Lorentzian Curve Fit` and looks like

$$f(x) = a_0 + \frac{a_1}{(x - a_2)^2 + a_3}$$

5. Q and f_0 are retrieved from the values of a_i. $f_0 = a_2$ and

$$Q = \frac{a_2}{2\sqrt{\frac{a_1}{0.5 - a_0} - a_3}}$$

6. These values are placed in the `PAS Cell Data` properties `f0` for the resonant frequency and `Q` for the width. The fitted waveform data is placed in the property `Fitted Data`. When fitting, the integrated area `IA` is set to NaN.

If the speaker is off:

1. The microphone frequency spectrum is retrieved via FFT.
2. The magnitude of that spectrum is converted to power by squaring the magnitude.
3. The integrated area is retrieved from the spectrum in the method `PAS Cell Data::Get Integrated Area`. This value is simply the magnitude of the power spectrum at the *modulated* frequency.
4. The noise level is then subtracted from the integrated area. The noise level is based on the spectrum itself and uses an algorithm devised by [1]. This value is an *electronic noise* level and essentially removes values of IA that may be attributed to white noise in the spectrum.
5. A background integrated area that is captured during the filter is then subtracted from the current integrated area and the new value is then adjusted by the fit parameters provided in the INI file by the user.

It is important to note that the value of absorption calculated in (5) is strictly an estimate. It is expected that the user of the instrument will provide further offline analysis.

> It is not clear if there is any value in the use of the routine to find the white noise level. This has been removed from the NOAA AOP instrument as of 2017.

CRD Functionality

Similar to the PAS, CRD functionality is defined over several objects. These objects are described as follows:

- `CRDS` - this is the base class that defines the core functionality. This object will perform the base configuration, provide a routine for coadding in `Get Data`, and provide the analysis of the ringdown waveforms in `Analyze Data`.
- `CRDS Cell` - this object is a representation of the individual cells. It contains the routines for *all* of the analysis that is performed in `CRDS::Analyze Data`.
- `eCRDS` - this object is a child of `CRDS` and contains the specifics of user interaction (`Rcv Cmd`), configuration (`Configure`) and data retrieval (`Get Data`).

Calculations

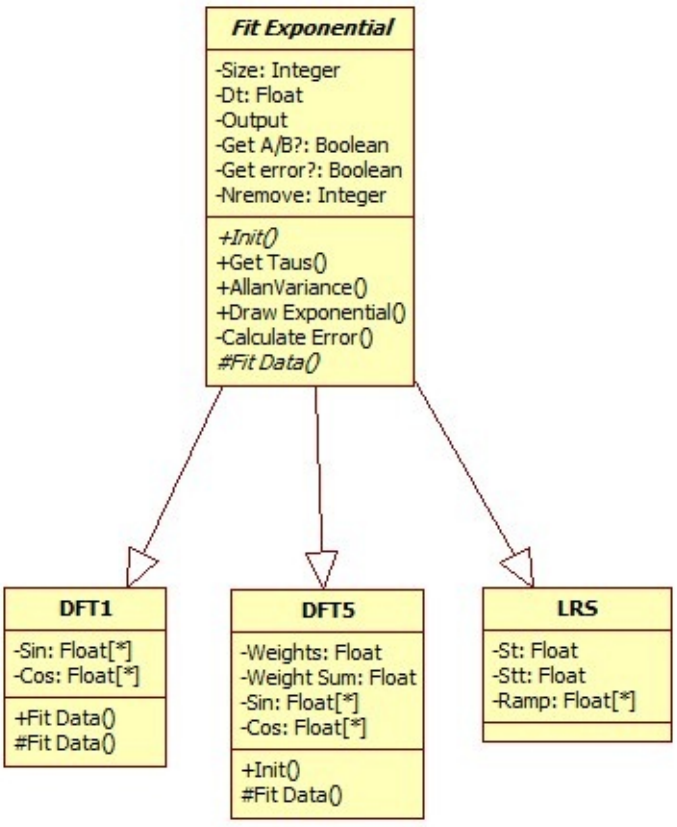
Several CRD calculations are performed in order to retrieve the an extinction, the primary measurement of the CRDS. All calculations are performed in the `CRDS Cell` object.

1. In `CRDS Cell::Retrieve Taus`, the method takes the raw ringdown data and calculates a τ.

    i. Prior to the operation, the method first removes the first N points specified by the variable `Nremove`. This value is specified by the user in the configuration file and allows the user to specify a number of points that might not be valid at the beginning of the operation due to known latencies in the system.

    ii. The remaining points in the raw data are truncated to a value of $N * \tau_{i-1}$, where N is specified by the variable in `CRDS Cell` called `NumTaus`. The default for this value is 5. **Currently, there is no hook in the code for setting this value by the user so unless the code is changed the system will use the default value.** The value of τ_{i-1} is the value of τ at the previous iteration. The value returned from this equation will be a time which will be converted to a number of points based upon the time step of the acquisition routine. If this value exceeds the total number of points available then the max will simply be the maximum number of points available. So, for example, if the ringdown time constant τ from the previous period is 150 μs and the total sample time is 500 μs, then the routine will only take up to 500 μs rather than the 750 μs from the calculation.

    iii. Once the range of the waveforms for fitting have been determined, the method will loop through the array of retrieved data and fit each of the waveforms. The fit routine is defined in the `Exponential Fit Library`. This library defines four objects as shown in the figure below:



This routine uses the template method to determine how the data is fitted. The method is determined by the user via the configuration file and is stored in the `CRDS Cell` property `ExpFit`. The method `CRDS Cell::Retrieve Taus` calls the public facing method of the generic `Exponential Fit` class `Get Tau`. This method in turn calls the `Fit Data` class which is implemented in one of the three concrete implementations: `DFT1`, `DFT5` and `LRS`. The details of the calculations for each of these implementations can be found in the references [3], [4], and [2] respectively.

iv. While the τ values are calculated, the waveforms are summed.
v. Once the τ values have been calculated, the outputs from the exponential fit routine are averaged and an average waveform is calculated for display purposes. The value `max` is taken as the maximum of *the average waveform*. If the calcualted τ is outside the bounds of 2 and 300 μs, the value will be coerced to either of these values.
vi. A fitted waveform is calculated from the output of the fitted data (this includes A, B and τ). This fitted waveform is for display purposes only.
vii. All τ related values are converted to microseconds.

2. Once the τ values have been calculated, they will be corrected for Rayleigh scattering. This will be

$$\tau` = (c\sigma(N_{ref} - N) + 1/\tau)^{-1}$$

where *c* is the speed of light (2.99792×10^8 m/s), $Nref$ is the reference molecular density (2.50362×10^{25} m^{-3}), N is the molecular density at cell conditions and σ is the scattering cross section. The molecular density is calculated as

$$N = \frac{P * A}{T * R}$$

where P is the pressure in Pascals, A is Avogadro's number (6.0225×10^{23} molecules/mol), T is the temperature in Kelvin and R is the ideal gas constant.

3. The extinction is then calculated as

$$\sigma = \frac{R_l * f_{dil}}{c}\left(\frac{1}{\tau} - \frac{1}{\tau_0}\right)$$

where f_{dil} is the dilution factor that accounts for the mirror purge flows, R_l is the effective cavity length in meters and *c* is the speed of light as defined above. The raw extinction is calculated using the raw values for τ and τ_0 while the corrected extinction corrects both values for Rayleigh scattering.

The τ_0 value is determined while the system is filtering. When the valve is changed from filter to sample, the method `CRDS Cell::Average tau0` will average the *last 5* τ value recorded and store these in the variables `Tau0` and `Tau0cor`.

Instrument Actor

The `Instrument Actor` is the object that wraps instrument functionality. This is the actor that will communicate instrument information with the `Controller`. Instruments are launched after the `Controller::Pre Launch Init` returns successfully. Both the PAS and CRD objects are called via two separate instances of the `Instrument Actor`. These two instances are launched from the `Controller::Launch Instruments` in `Controller::Actor Core`.

The `Instrument Actor` requires two key pieces of information defining the functionality and how the actor will communicate with the `Controller`. These two pieces of information are defined by the properties `Instr` which is an `Instrument` object and `Update Data MSG`. `Update Data MSG` is a generic message. The use of a generic message allows us to decouple the `Instrument Actor` from the `Controller`. The generic type is the `Generic Update Data Message` implemented as `Update Controller Data MSG` as described in the chapter covering the `Data` structure.

The `Instrument Actor` performs several functions:

- In `Pre Launch Init`, the `Instrument` object contained by the `Instrument Actor` is configured. If configuration fails, the current instance of the `Instrument Actor` will throw an error and fail resulting in a removal of the instrument from the system.
- In `Instrument Actor::Actor Core` The parent implementation of `Actor Core` runs in parallel two three other loops:
  - The implementation of the `Instrument::Core`. This runs the state machine for the instrument as described above.
  - A loop that listens for the data notifier that is sent every time the data is updated in the `Instrument::Core`. When a

data packet is received from the instrument, the message is shipped to the `Controller` via the implementation of the `Generic Update Data MSG` specified in the `Instrument Actor`. This loop will terminate on shutdown of the `Instrument` object wherein the notifier is released.

    - A loop that listens for any changes to the CVT via the queue specified by the queue in `Instrument` called `ExtCom`. If the `Instrument::Core` signals an update via this queue, the `Instrument Actor::Actor Core` will directly reference the CVT session and calle `CVT::Insert Data` to update the CVT. Upon shutdown, the `Instrument::Core` will release the `ExtCom` queue and trigger this loop to terminate.

- The parent implementation of the `Actor Core` will listen for requests from the user interface and forward those requests to the `Instrument Core` via the command queue specified by the property `CMD`. The `CMD` queue, though owned by the `Instrument` object defined by `Instrument Actor::Instr`, is released by the `Instrument Actor::Stop Core`.

Data

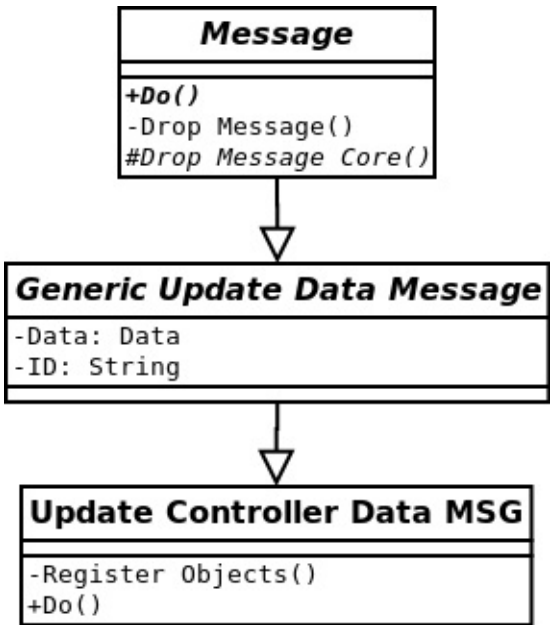
Data retrieved from the various entities are registered at startup as *emitters* with a data value reference (DVR) that is maintained as a property of the `Controller` object called `MAP DVR`. When data is broadcast by the various emitters, then the DVR will capture these broadcasts and make them accessible to the various processes that require them (network communication and file writing).

Registering Data for File Operations

Emitters are generally initially registered with the controller at launch time. In most cases, when an actor handling an object that may generate data is launched using the `Controller::Launch Nested` method, the method `Controller::Register Emitters` will run immediately following if 1) the actor contains memebers that may supply data for file writing and 2) the launch of that actor was successful. The purpose of registering these objects that will be stored in the map is related directly to writing files and used to provide flexibility - by monitoring the registration status, the software will determine when it is acceptable to being writing to file (it will wait until the registration is complete to being writing the file).

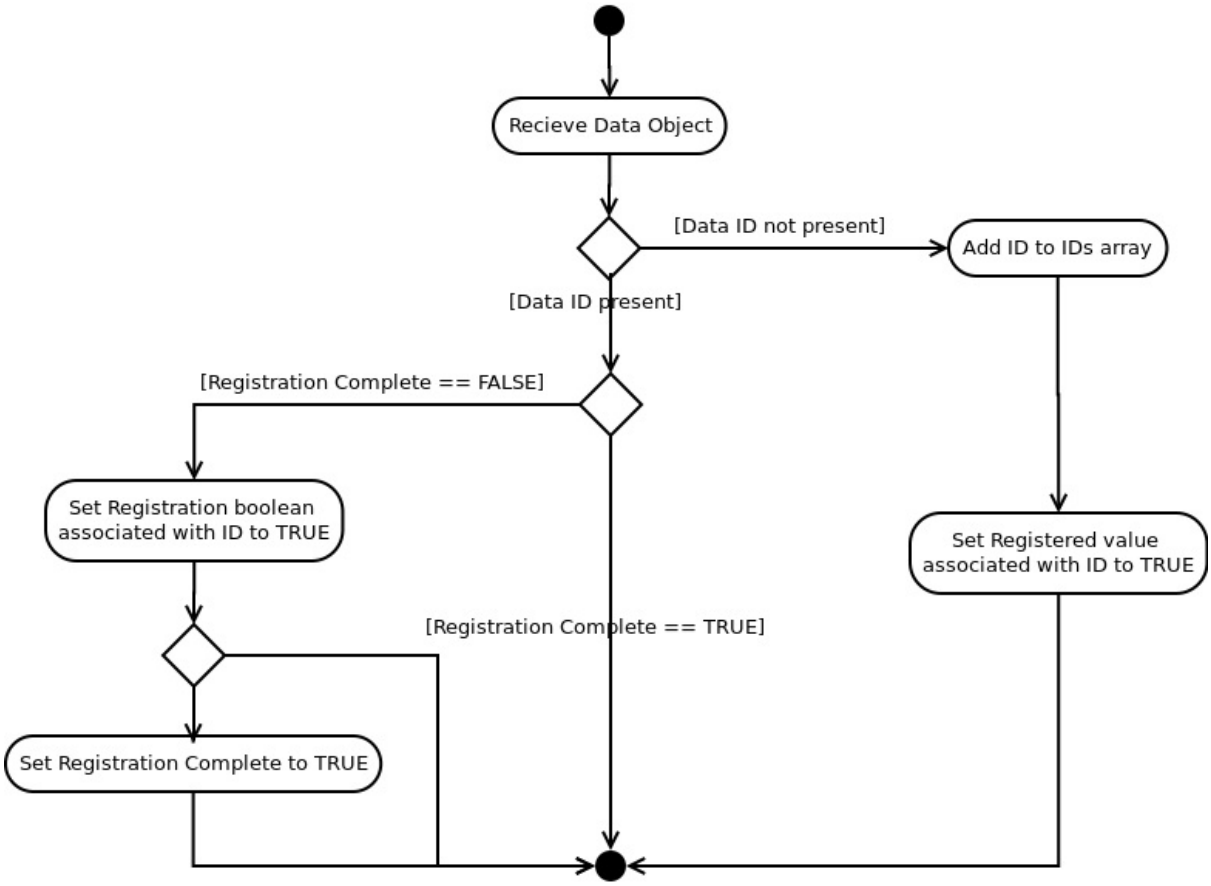
The method `Controller::Register Emitters` does exactly what the name implies - it registers objects that will supply data by `ID` with the `Controller` property `MAP DVR`. `MAP DVR` is a map similar to the `Enqueuer MAP` previously described. It contains two well defined attributes - `IDs` which is a list of IDs belonging to data objects that will be contained by `MAP DVR` and an array of booleans called `Registered` corresponding to the array `IDs` which indicates whether the first bit of data has been received by `MAP DVR`. In addition, there is a corresponding attribute with the titles found in the array each containing a `Data` object (defined below). The initial value of all entries in the `Registered` array is `FALSE` (not registered).

Registration will only tell the system *what* data to expect (it does not anticipate how that data is structured). An data object is considered "registered" when the `Controller` recieves that first data packet with a valid ID. Data is recieved by the `Controller` via the message `Update Controller Data MSG`. This is a child of the `Message` object used in the `Actor Framework` that provides the only path to send data to the `Controller`. The class hiearchy of the update message is shown below. Generic message contains two properties - `Data` which is a `Data` object as defined below and `ID` which is a unique string defining the object. The latter is the ID that should be found int the `IDs` array of the `MAP DVR` property of the `Controller`.



26: Update controller data message hiearchy

The `Message::Do` method is implemented by the implementation `Update Controller Data MSG`. The `Do` method implements the details of handling the data specific to the `Controller` and will call another method called `RegisterObjects`. This method will handle the registration of the objects, determine when registration is considered complete and the actual storage of the data objects.



27: Registering Data

The `Data` Object

All emitters broadcast an object with a common parent. That parent is called `Data`. The `Data` object may be found in the library `Base Data Class` which contains a lot of basic data object as well as the root `Data` object. This object has the following properties:

- `measTime` - a timestamp representing when the data was collected.
- `Data ID` - unique string to represent the object.
- `write time` - boolean indicating whether the time is to be sent with any data string.

The `Data` object also contains some methods that are intended to be implemented by the child. There are two serialization methods intended to ease sharing of data throughout the system:

- `Serialize - JSON` - generates a JSON string that may be used to be broadcast to the client in response to http requests. Any changes or additions to any of the `Data::Serialize-JSON` methods should adhere to the JSON Data Interchange Format.
- `Serialize - Text` - generates a string array that may be used to generate a string for writing to an ASCII file.

A companion abstract method to the method `Serialize to Text` is the method

- `Generate Unique Header` - called before data is written to file to generate a unique ASCII header.

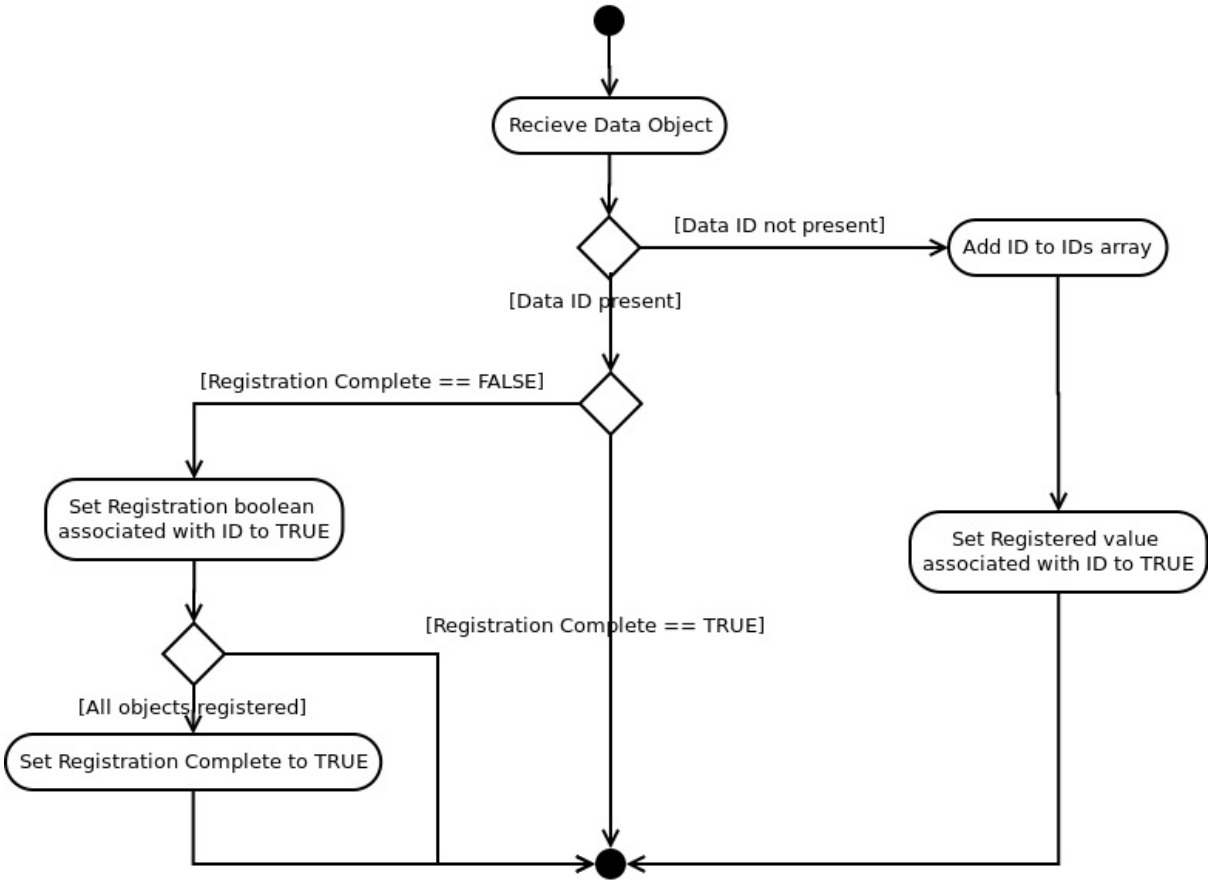
`Controller` Data Storage

Relies on a generic data message to decouple the nested objects from the `Controller` itself. The realization of the generic data object is specified at the creation of the different objects that are expected to send data back to the `Controller`

MAP Structure

Populating the MAP

Registration



Structure of MAP.

How it is populated

Devices

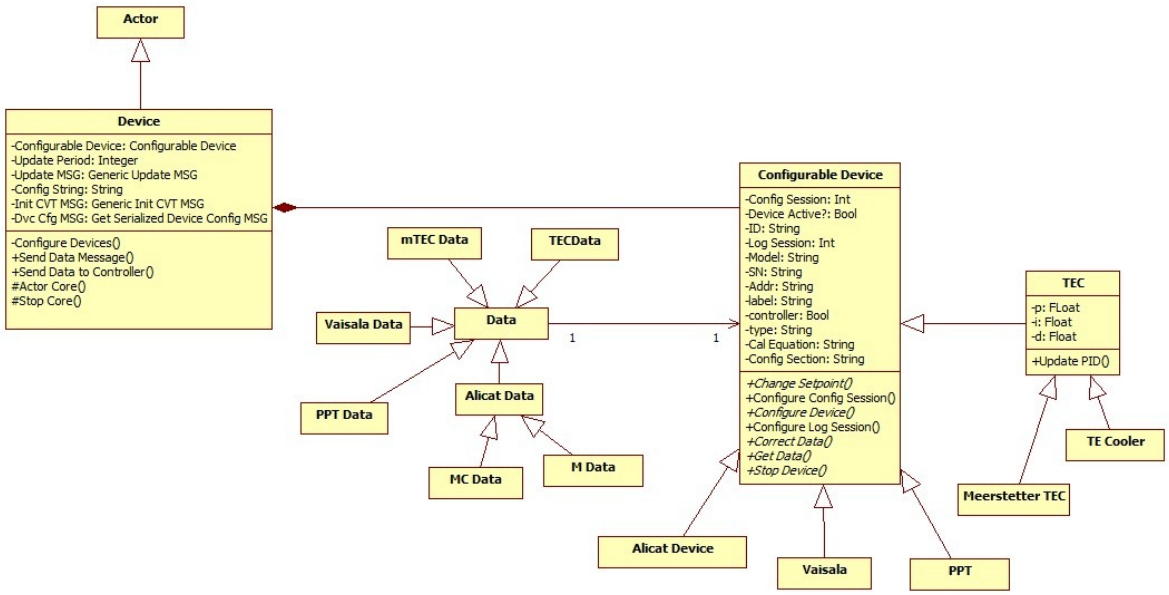
The EXSCALABAR system consists of a large array of devices. A device here is considered a piece of hardware that supports a core measurement but does not directly make a key measurement and can be run by other systems or by itself. As such, devices tend to be reusable and extensible. In this system, the physical objects that we can call devices are

- Alicat flow measurement and control devices
- Honeywell precision pressure transducers
- Vaisala hygrometers
- Meerstetter TECs
- TE Tech TEC

Each of these devices

- Makes a measurement necessary to the analysis of the data
- Is configurable
- *May* accept some input (if the device is a controller) but does not require much interaction

Below shows a uml diagram showing the software architecture of the device. The functionality in this system is shared by both a `Configurable Device` object and a `Device` actor. Both are described in more detail below.



28: Device architecture.

`Configurable Device`

The `Configurable Device` object describes the core functionality of each of the devices. It defines the key methods and properties that are used by *all* children. Some properties are necessary to the functionality of the device as software while other pieces are simply provided to further differentiate the object.

It is worth noting here that each device is intended to return a core data set. This is packaged as a `Data` object and is returned by the method `Get Data`. While this data is intended to be the core product, it may not be the only data that is returned by the device. For example, the TECs often return a myriad of data that indicate the health of the instrument.

The core properties defined by the `Configurable Device` object are:

- `Config Session` - this is a session of type `Config Data`. This is the session that is shared with other parts of the EXSCALABAR application used for configuring the application.
- `Device Active?` - boolean that indicates whether the current device has been configured properly and is in use. **This property is not generally used and is purely informational.**
- `ID` - a *unique* string that identifies the current device object.
- `Log Session* - contains the` System Log` session for the application.
- `Model` - this is the model as returned by the device. This is purely informational and is returned from the device itself. This information is passed back to the client through the CVT. The information is not necessary for proper functioning.
-
  - `SN` - string containing the device serial number. As with the model number, this is returned by the device itself if available and is not necessary for the proper functioning of the software. It is returned to the client via the CVT.
- `Addr` - this is the string based addressed used for communication with the device. This *is* required by the device for communicaiton. This value will appear in the CVT and is defined in the INI file.
- `label` - string used by the user interface for displaying data for the device. This value is defined in the INI file and returned in the CVT.
- `controller` - boolean indicating whether the device will accept an input. This is hardcoded for devices that serve solely as meters (such as the Vaisala or PPT probes) and is determined at run time by the software for those that can be either/or such as the Alicat flow devices.
- `type` - a string that indicates what *kind* of device this object is. The types that are defined are:
  - `alicat`
  - `ppt`
  - `vaisala`
  - `tec`
  - `mtec`
- `Cal Equation` - an implementation dependent string for adjusting key data that may be returned by the device. By default this is an identity, i.e. data retrieved is the data that is returned.
- `Config Section` - string that defines the section of the configuration file that the configuration information is under.

In addition to the properties defined above, the `Configurable Device` class defines some basic methods for interaction:

- `Change Setpoint` - method for changing the setpoint of the device if the device is a controller. Accepts a float and does nothing if the device is not a controller. This method is *not* a must override.
- `Configure Config Session` - this is a top level function that simply obtains a `Config Data` session and stores it in the property `Config Session`.
- `Configure Device` - this is an abstract method that doe one thing - attempts to retrieve the string for the `Cal Equation`. If this value is not present, it will default to `1*x`. This method is a must override.
- `Configure Log Session` - this is the same as the `Configure Config Session` method except that it will configure the `System Log` session.
- `Correct Data` - method used to correct core data. This method is abstract but children are *not* required to override this method.
- `Get Data` - retrieves the core data set. This is abstract and must be implemented by the children.
- `Stop Device` - method to be called on shutdown. This method is abstract and must be overridden.

The `Configurable Device` object is straight forward and only defines a basic API for how to interact with devices. There are no patterns directly associated with this object and in its most basic form, a device could be run using a state machine - configure, get data, stop device, etc. The more complex interaction is found in the actor implementation of the device structrue.

The `Device` Actor

The `Device` actor is composed of a `Configurable Device`. This composition is a 1:1 relationship - that is, each `Device` actor contains one `Configurable Device`.

This class contains some properties and methods that increase the complexity of the implementation. These properties and methods are used to decouple the implementation from other portions of the code (particularly the `Controller`). In general, this class uses the strategy pattern to achieve this. The properties of interest here are:

- `Updat MSG` - this is the message that is used to update the caller with the current `Data` object produced by the `Configurable Device`.
- `Init CVT MSG` - this is a `Message` that is used to update a setpoint at initialization. The concrete implementation of this is found in the `Controller` library and is called `Device Init CVT MSG`. This simply injects the initial setpoint into the CVT and readies the structure for transmission.
- `Dvc Cfg MSG` - a `Message` that injects basic device properties into the CVT. The concrete implementation of this is found in the `Controller` library and is called `Update Device Configuration MSG` which simply injects some values regarding the current device into the CVT and serializes the CVT to json.

Unlike other actors, there is no implementation of the `Pre Launch Init` method. In this case there is no operation required by the actor at this stage except to configure the `Configurable Device`. The configuration of the device though is performed at the start of the `Actor Core` in `Configure Device`. The reason is that for devices communicating on a digital port this configuration process may take some time (if the device happens to not be present it might take somewhere between 5-10 seconds to determine if the device is not there). To prevent the whole system from hanging in initialization mode, this configuration is performed in the `Actor Core` so initialization occurs in parallel.

Aside from the initialization occuring at the kick off of the `Actor Core`, a method called `Dend Data to Controller` runs parallel to the `Actor Core`. This method polls the device using the `Device::Get Data` method and returns that data to the `Controller` via the `Update MSG` provided when the `Device` actor is kicked off. This runs continuously at the rate supplied when the `Device` actor is instantiated and is defined in the property `Update Period`. While this property is flexible, it is generally set to 1 s as this is sufficient to keep data current.

When a shutdown is called on the `Device` actor, the main `Actor Core` will close. When this closes, the `Stop Core` method will fire. In addition to performing the taks associated with `SuperActor::Stop Core`, it will also call the concrete implementation for the device `Device::Stop Device`. When the `Actor Core` completes, a notifier will be closed indicating to the `Send Data to Controller` method that it should stop and send the `Last Ack` message to the `Controller`.

FPGA IO

The system contains a card called PXI-7842. This couples multifunction DAQ capability with a field programmable gate array (FPGA). The FPGA allows the developer to configure the IO itself and how it is handled. This makes the board considerably more flexible than a standard multifunction DAQ board that utilizes an asic.

This board is used primarily to service the PAS IO although it does have some available IO that may be used as needed. This card was chosen specifically for handling the PAS AO associated with the lasers. Each laser can and is likely to fire at a slightly different frequency. On most multifunction DAQ cards, the AO is synchronized in frequency if the available circular buffer is used. Waves for driving the AO can be generated in software, but this method is not reliable given the loads this system produces.

The main VI for the FPGA is called `FPGA Server`. In this method, the functionality of each loop is explicitly defined by the label. However, it is briefly worth noting how the AO waves for the lasers is produced. There are two loops that service the AO - the first is defined as **Analog Output**. This loop actually sends the current value to the board. In this loop, the sine wave is actually defined by the VI `Produce AO`.

The other loop is called **Square wave generation - SCTL**. This loop has one purpose - to produce the AO associated with a square wave. This loop runs continuously. When the user selects the output to be a square wave, the generation of the sine wave in `Produce AO` will be bypassed and the current value of the square wave as defined in `SquareOut` will be passed to the AO nodes. The reason this is necessary is that the square wave is produced by the built-in method `Square Wave Generator` (in this case wrapped in the method `Generate Square Wave`). This method requires the use of a single-cycle timed loop (SCTL). SCTL loops can not contain any IO nodes as these nodes are not guaranteed to execute within a single cycle.

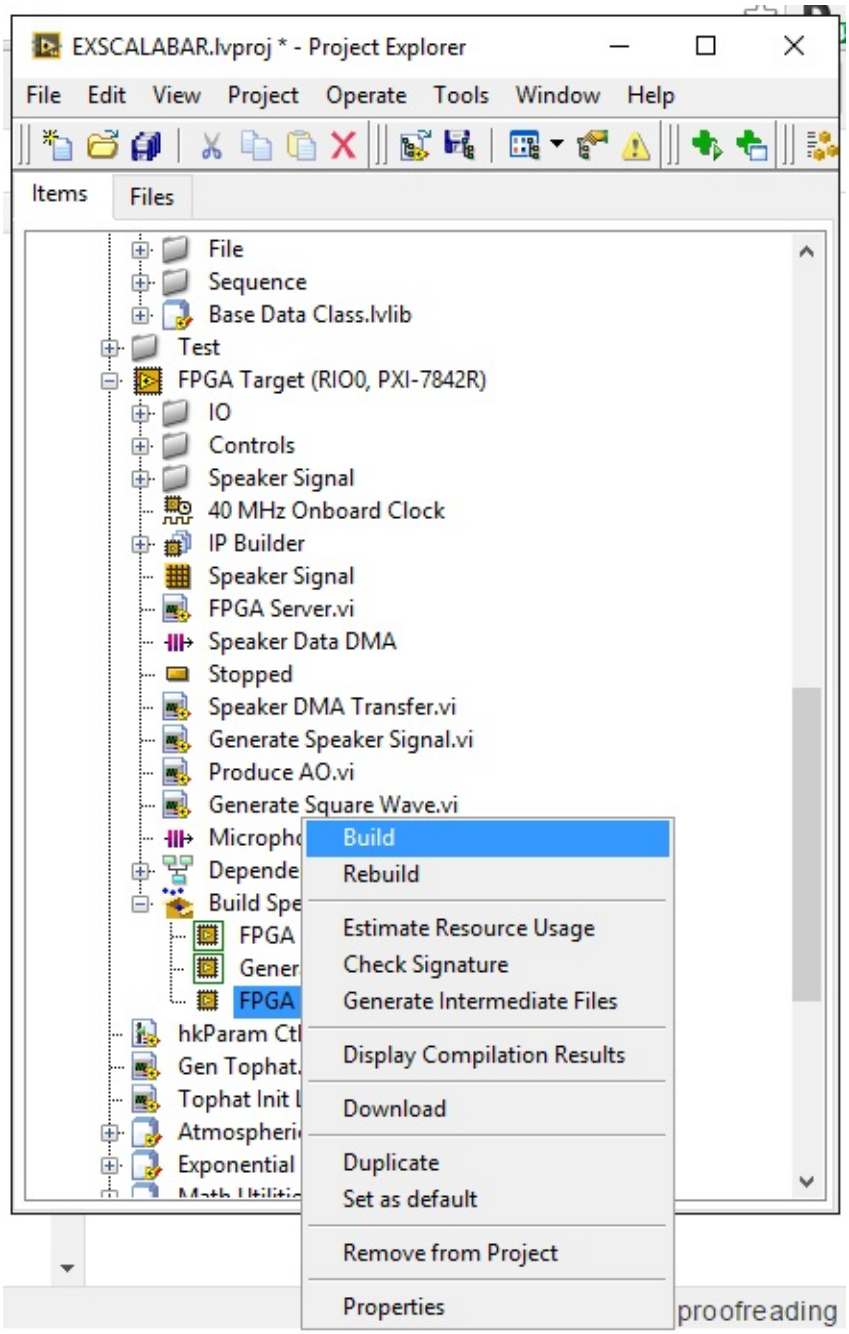
The `FPGA Server` contains two methods that run while the FPGA is active. The first one called `Speaker DMA Transfer` strictly does what it states. The user can request a new speaker signal that might encompass a different bandwidth or central frequency or both. When a new speaker signal is requested, the command `Load New Speaker Chirp` will execute. This will translate a frequency and a frequency band into a speaker signal via the method `Tophat Init LUT`. This data is transferred to the FPGA via DMA (`Speaker Data DMA`). The FPGA method `Speaker DMA Transfer` simply listens to the DMA FIFO for new elements and writes then to a memory location called `Speaker Signal` for use when the speaker is requested.

The second method called is related to the previous one discussed. This one is called `Generate Speaker Signal`. This simply takes the output of the `Speaker Signal` memory bank and runs back and forth across the waveform to produce the chirp. The utilization of the memory space is maximized by only storing half of the waveform. Since the speaker waveform is symmetric, the code simply reverses the direction it traverses in memory when it gets to 40000. This produces a chirp with very high resolution.

Microphone data is acquired in the top level VI in the loop labeled **PAS Data**. The data is acquired at a rate of 40 kHz, or every thousand ticks of the native clock. This frequency is well beyond the Nyquist frequency which would be approximately < 3 kHz. Laser data is acquired by one of the other multifunction DAQ boards as previously described. There is not enough space to acquire all of the PAS AI on this card.

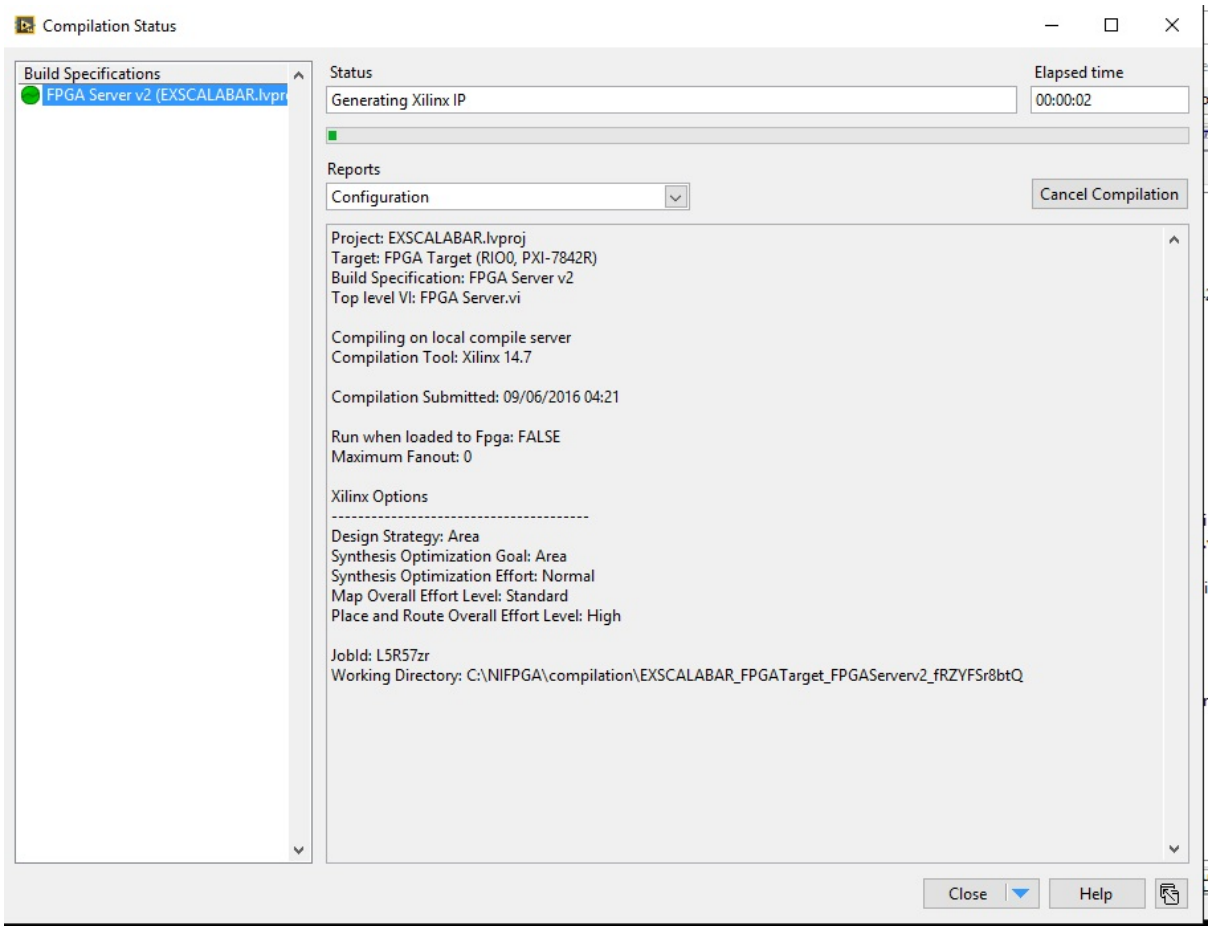
Building the FPGA

Building the FPGA bitfile is simple. Strictly right click on the build spec and click *Build* as shown below. The current build spec used is `FPGA Server v2`.



29: Building the FPGA Bitfile

Once the build process starts, the developer will be presented with the following dialog indicating the compilation status.



30: Compiling the bitfile

When the compilation is complete, the bitfile will be automatically deployed when the system is restarted. The bitfile is currently saved under `..\exscalabar\FPGA Bitfiles`. The bitfile that is currently written to is called `FPGAServerv2.lvbitx`.

> The developer may update this bitfile name, but if they do they will also have to adjust any controls that reference this. In this project, there is one control that is used to configure the FPGA session and this is called `R FPGA Ref`. This is a typedef that contains the FPGA reference. This can be configured by right clicking and selecting the new FPGA bitfile location.
>
> Another issue associated with the FPGA is when the location of the project changes for instance when it is worked on from a computer other than the current one that it is located on. In this case, the project will fail to find the typedef and the developer will have to point the project to the correct location. This file is located in the directory `../exscalabar_server/FPGA`.

Files

The server will generate several different files depending on configuration settings and user input. These are:

- the main file - this is the primary source of data and may be mirrored to two drives depending on the initial user settings.
- CRD tau data - this is a record of all taus calculated by the system.
- PAS frequency data - this is frequency data related to the PAS operation.
- a log file

In addition to these generated files, there are two files that relate to server operation that may be edited by the user but ultimately reside on the controller hard drive. These files are:

- the configuration file
- the calibration description file

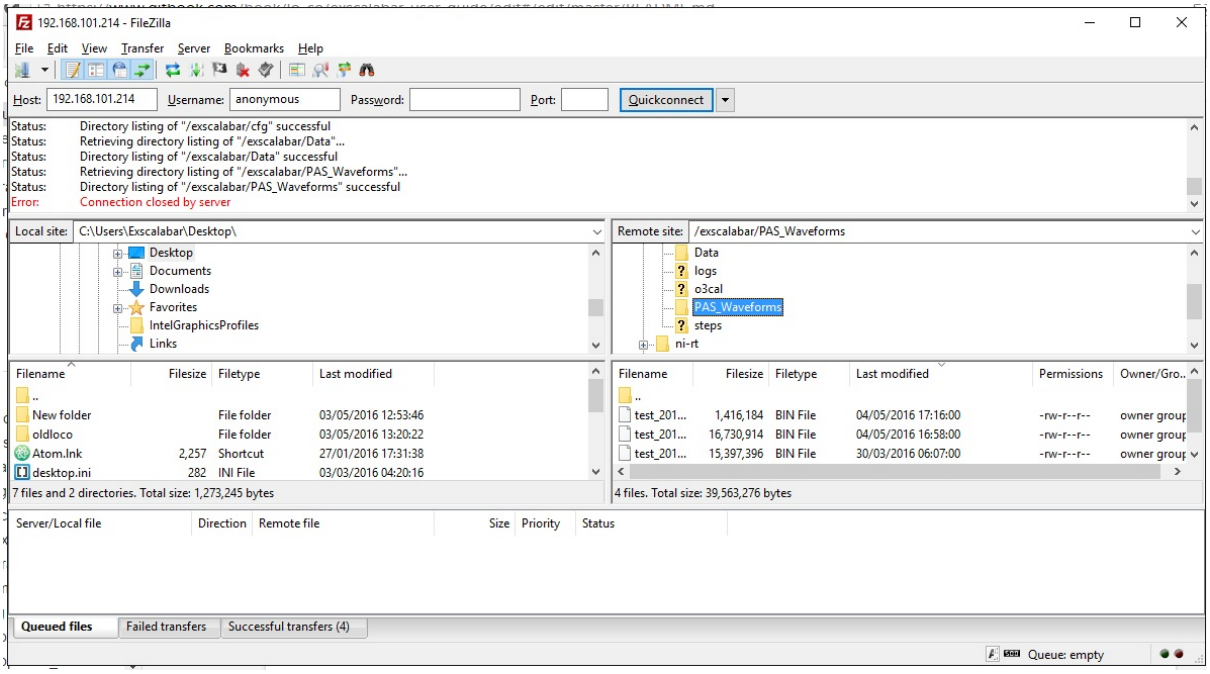
The main drive of the controller is labeled `c:` while drives that are plugged in via USB are labeled starting with `u:`.

Access

Files may reside on either the internal drive of the controller under the directory `c:` *or* on a USB drive that has been inserted into the controller. On the USB drive, files may be accessed one of two ways - via ftp or removal and reinsertion of drive into another computer. Files that reside on the controller itself may only be accessed via ftp.

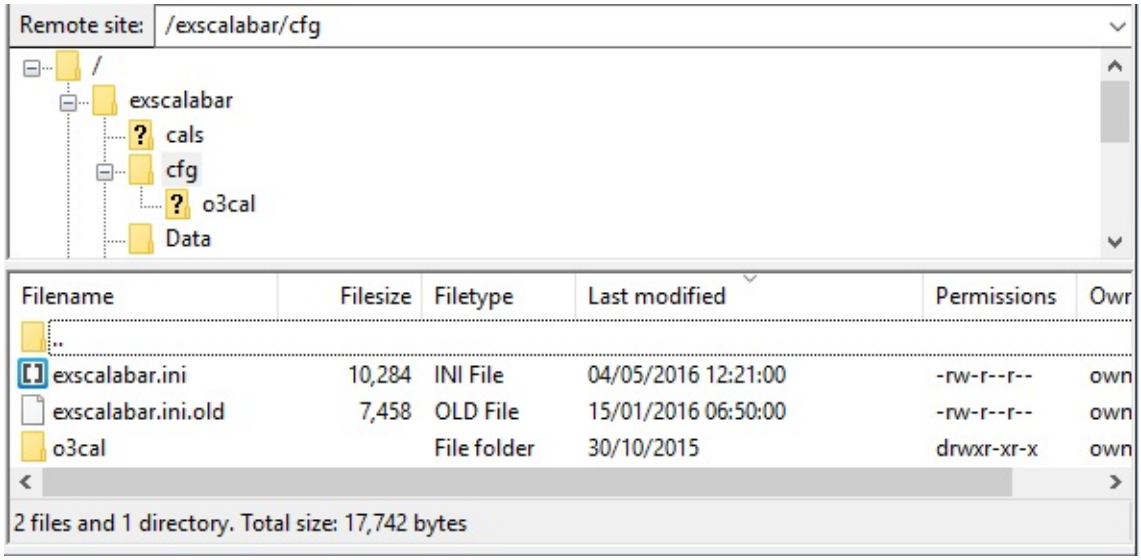
To access the internal drive, the user will require a program to ftp into the drive. The following examples use FileZilla to access these files via ftp, but any ftp utility should be sufficient. In order to access the server drive, the user must have the IP address of the server available. **There is no user name or password required at this time to access the server**, but the server may be configured to utilize this functionality.

In the image below, we have logged on to gain access to the server drive using FileZilla. The main directory in which files are placed by default is the `exscalabar` directory.



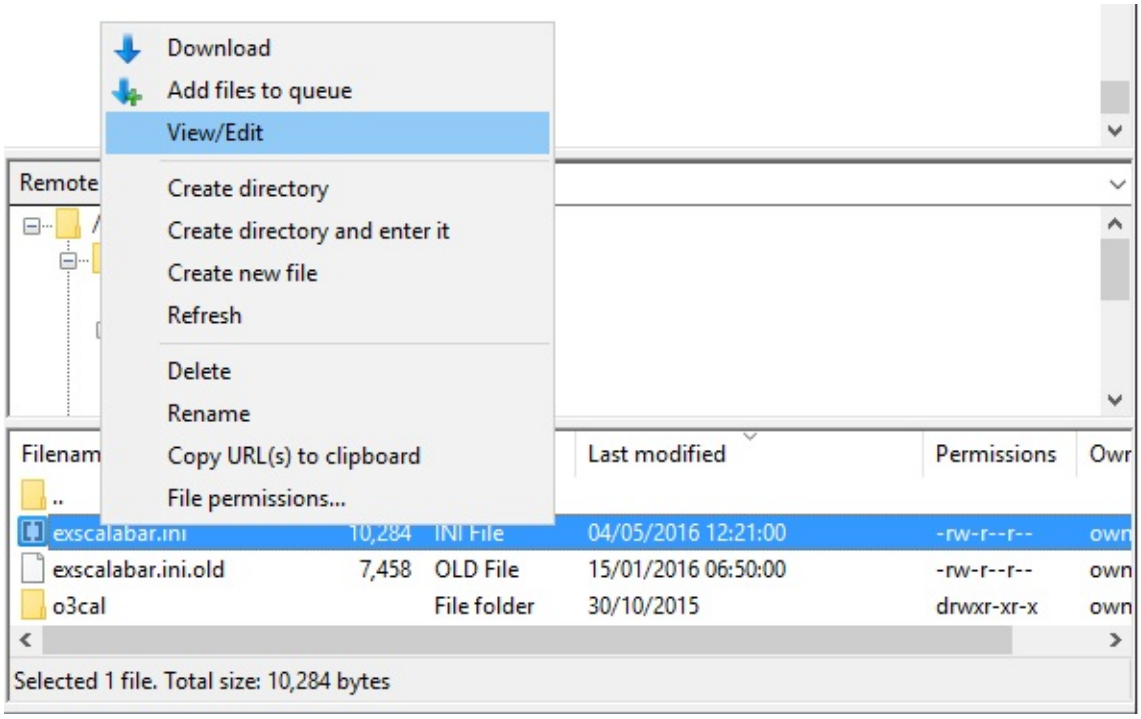
31: Filezilla

Two of the files found on the controller reside at a specific location under the folder `exscalabar`. The configuration file that is called from the software is called `exscalabar.ini` and is located under the director `c:\exscalabar\cfg`. Similarly, located within this directory is also the location of the calibration directory currently called `o3cal`.



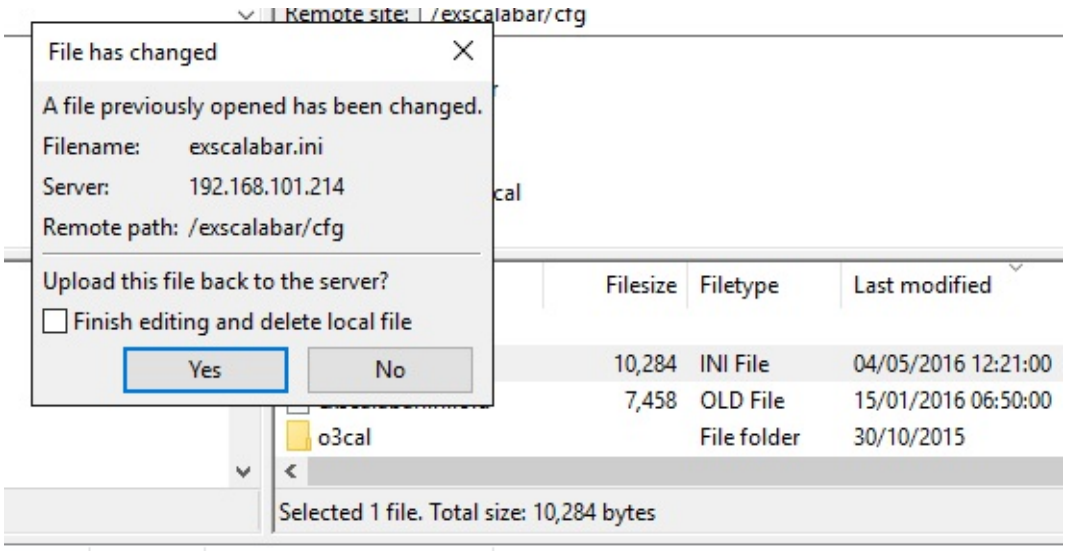
32: Configuration file location

If the user wishes to edit the file that is on the server, there are two choices on how to do this. The user may right click on the file and select `View/Edit` (if using FileZilla). This will open a file in whatever application the user has configured the client to open an INI file (the default is Notepad on Windows machines).



33: Viewing and editing files

If the user opts to use this method for accessing the INI file, FileZilla will prompt the user to upload this file back up to the server. If the user is finished editing the configuration file, they may choose to upload the updated file and delete the local copy or simply just upload the current file.



34: Uploading changes

A safer alternative to this method is to download the file to somewhere that is backed up and edit the file and upload the edited file. **Best practice: before altering the current file, save that file on the server as something that is not `exsalabar.ini` (such as `exscalabar.ini.old`).** This will preserve the current copy on the server for reuse if needed.

As described below, the system is highly configurable and most files that *are written* by the software can be configured to write to almost anywhere that is writable in the system. **Take note - when writing to the main drive, the user may fill the drive as the drive is not particularly large to begin with. It is probably better practice to write to an external drive.**

File Writing Functionality

All generated files are written using the same actor object - `File Actor` found in the `File Library`. The `File Actor` object is generated as needed except for the main data file and the log file which are both instantiated at system startup.

Aside from initializing the actor object with user defined data such as file location and whether the file is being saved, the `File Actor::Pre Launch Init` will also create a DVR to maintain the save state as well as a DVR to maintain the file reference. These DVRs allow the `Actor Core` to alter and share the state of these two values with the loop that actually writes the data to file (as discussed below). In addition to these two DVRs, the initialization routine will also obtain a queue that will hold a `CMD` object (discussed in a previous section) that is used to store up to 10 consecutive packets of data to be written.

Writing Data

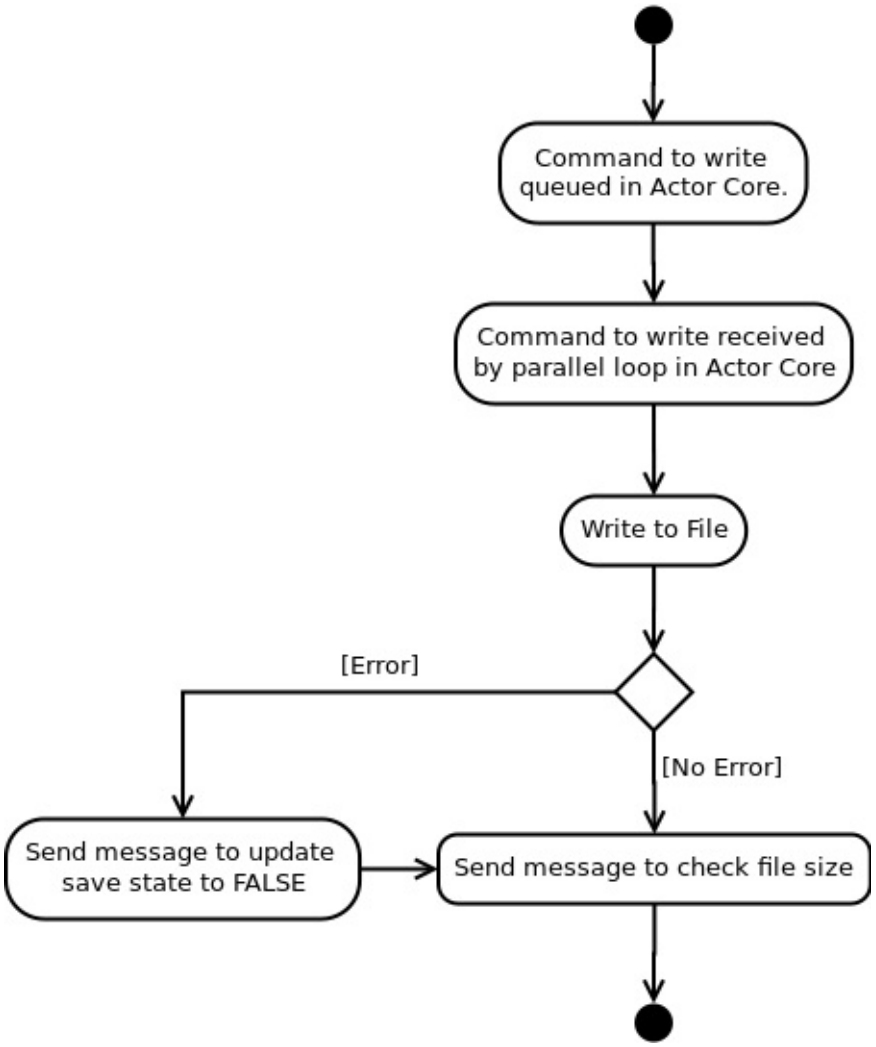
Data is written in `File Actor::Actor Core`. Aside from the log file, all files written to are sent a command via a message from the `Controller::Send Write Main MSG` found in the `Controller::Actor Core` at a regular interval (1 s); messages are sent for enqueuers that are found. These enqueuers are

- `Main`
- `Mirror`
- `paswvfm`
- `crdtaus`

- `crdrd`

The message that kicks off a file write operation is called `File Data MSG` and is found in the `File Library`. The loop that sends these messages is a timed loop that operates at 1 second intervals. The method `File Data MSG::Do` takes the data object to be written and pumps it into the command stored in `File Actor::Exe Write`. The instance of this command is defined when the `File Actor` object is created and is unique to the implementation of the `File Actor`. In order to keep the `Controller` decoupled from the `File Actor`, the command has a base class called `Exe Write`.

The object `Exe Write` is an abstract implementation of the `CMD` object and implements the method `Handle Command`. The class `Exe Write` contains a method called `Write to File`; this method is abstract and is implemented by the children. `Write to File` contains the concrete implementation of the actual file writing.



35: File writing sequence

The sequence for file writing is shown above (not shown is the message sent by the `Controller` described above). This sequence starts with the `File Actor::Actor Core` receiving the message from the `Controller` to execute a file write for the current instance of the `File Actor` object. The `Actor Core` queues up the instance of the `Exe Write` to write to file contained by the current instance of the `File Actor`.

The command is received by the loop that runs parallel to the `Actor Core` parent method to prevent locking up of the message processing core. And the parallel nature is the reason for the presence of the DVRs in the object itself - they allow the message processing portion of the `Actor Core` to share state with the file writing portion of the `Actor Core`.

The file save state is changed via one of two ways:

- by an error thrown during a file writing as shown in the above diagram
- or by a command from the user

In either case , a message is sent to the core to update this property.

Once the command is received to write to file, the `Handle Command` of the object `Exe Write` will call `Exe Write::Write to File` the implementation of which is dependent on the current instance of the `File Actor` . Once file writing is complete, the `Handle Command` method will check to see if there was an error during the process - if so, the method will send a message to the core to set the save flag to false.

Checking the File Size

Finally, when the command has been handled, the core will send the message to itself to check the current file size. The user can command in *all* files that the system generate a new file after so many bytes. The key for this in the configuration file is `max_file_size` . The user may wish to limit the file size for two reasons:

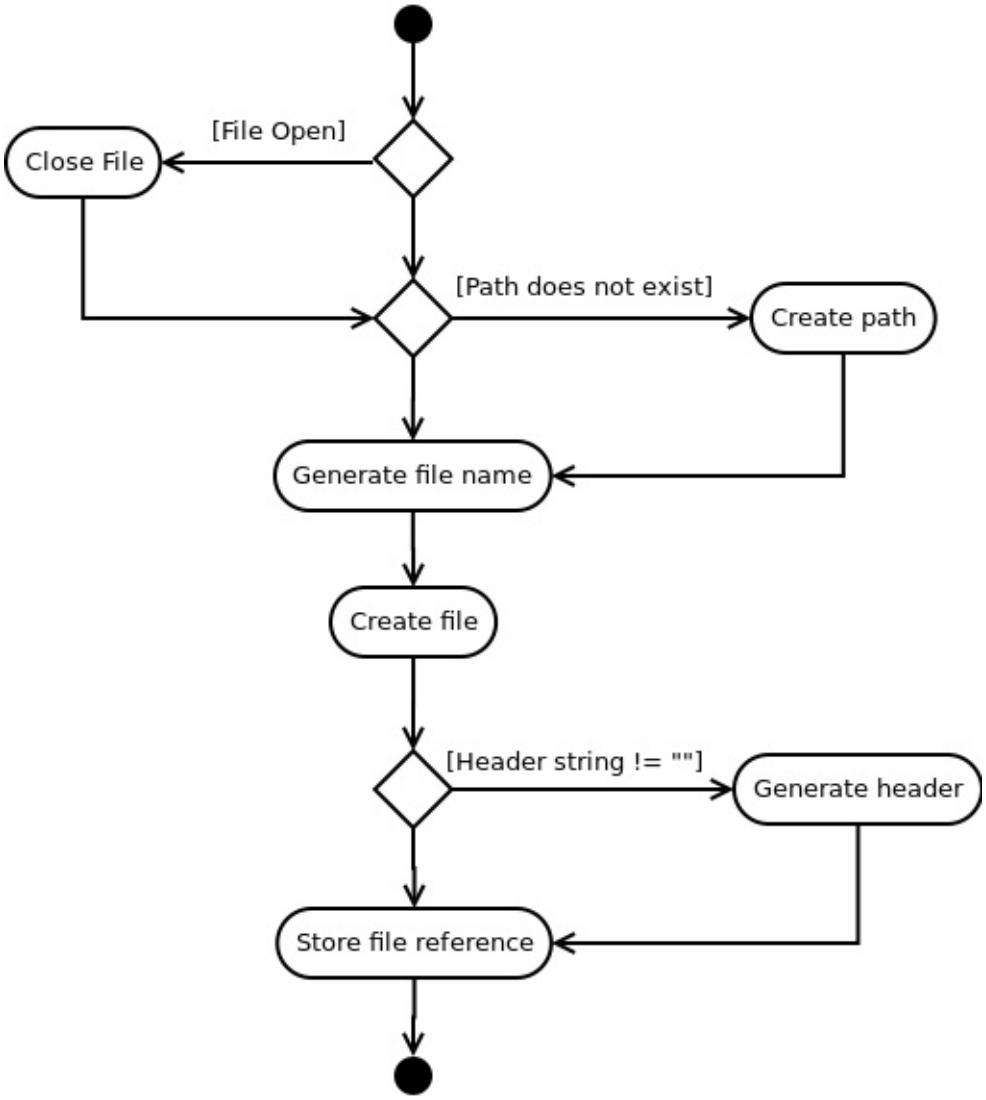
1. prevent data from being lost in the event of a crash
2. prevent problems with data analysis due to large file size

It is up to the user to determine what the appropriate values are for this, but defaults are generally 50 MB.

The message that is sent to the core is found in the `File Library` and is called `Check File Size MSG` . In this message processed by the `Actor Core` parent method in the `File Actor` , the method first checks to see if there is a file currently open. If there is not, then nothing will happen. If a file is open, then the method will check the current file size and compare that to the user defined one from the configuration file - if it is greater than the size specified, it will send a message to open a new file via the `Change File MSG` in the `File Library` .

Changing the Current File

When the message `Change File MSG` is processed by the `File Actor` . The sequence for creating a new file is shown below. When a new file is requested, a unique name based on the date and time will be generated. In addition, a prefix and extension specified by the configuration file will be added. Once the file is created, if there is a header is specified, that header will be written to the file. In most instances, there is no header written to file, the exception being the main files which record 1 second data as described in the following sections.



36: Generating a new file

The Main Data File

The main data file contains all of the data collected at 1 Hz by the server. This includes CRD values such as τ and PAS integrated area as well as data from devices and general system state.

PAS Waveform Data

One of the files that may be written is the PAS waveform data. A sketch of the activity associated with the user interaction and the actual file writing is shown below. PAS waveform data is not automatically written as this may produce a considerable amount of data.

The parameters for writing the file are defined in the INI file under the section `PAS Waveform`. The top level path that the file is written to is defined in the key `paswevfm.Path` and the directory that the data is written to is defined in `folder`. As with all files, the user may define the prefix and extension (as well as how much data to write to each file), but the format of the file will be binary regardless of the extension.

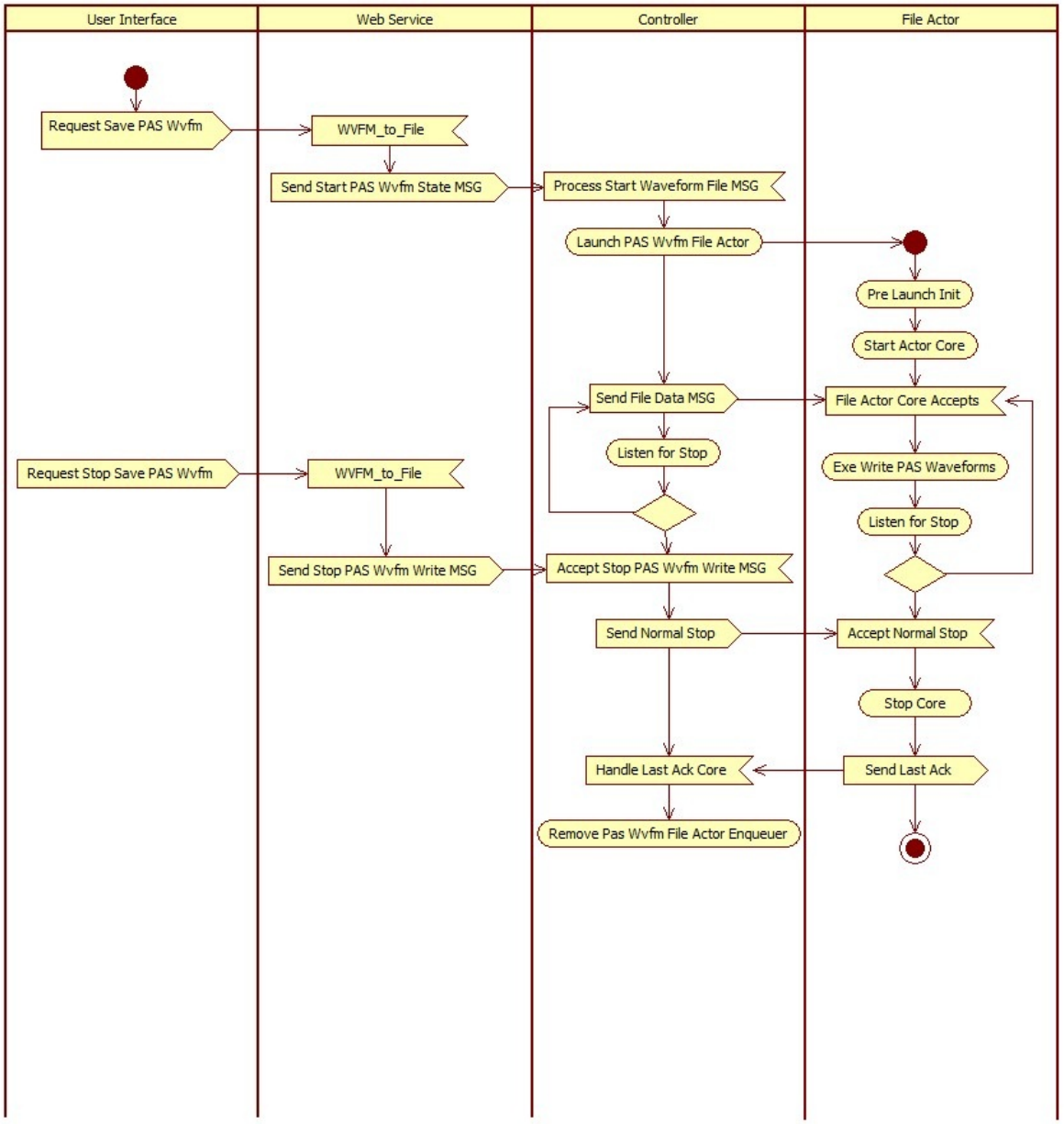
The process is kicked off when the user requests data to be written to file on the PAS page of the user interface. The request is packaged in the http message handled in the web service by the `WVFM_to_File` method in the `PAS` web resource. This in turn kicks off the message `Start PAS Wvfm State MSG` to the `Controller`. This message is processed in the `Controller::Actor Core`. This message kicks off a `File Actor` with the ID `paswvfm` loaded with the write message handler `Exe Write PAS Waveforms`.

If the actor is launched successfully, it will operate as described in the previous section, listening for the `File Data MSG`. As with all other instances, this message is sent from the `Controller::Send Write Main MSG` at 1 Hz. When this message is received by the `File Actor` handling the PAS waveform data, it will send the `Exe File Write` command implementation `Exe Write PAS Waveforms`.

The `Exe Write PAS Waveforms::Write to File` method will retrieve the waveform data from the data structure received and write that data to file. The file itself is a binary file and contains data written in the following format:

1. `Time` - a double precision float that is seconds since midnight 1904 Jan 1
2. `i` - 32-bit integer representing the first dimension of the mic data array
3. `j` - 32-bit integer representing the second dimension of the mic data array
4. `mic data array` - array of 16-bit integers representing the mic data. The array length is `i` * `j`.
5. `k` - 32-bit integer representing the first dimension of the photodiode data array
6. `l` - 32-bit integer representing the second dimension of the photodiode data array
7. `photodiode data array` - array of 16-bit integers representing the mic data. The array length is `i` * `j`.

Data is written to file until a signal is sent from the user interface. The same web service method handles this request as the one that handled the start of the writing. This method will in turn send a message to the `Controller` called `Stop PAS Wvfm Write MSG`. This message simply sends a `Normal Stop` message to the `File Actor` handling the PAS waveform data. The request will force the actor to shutdown and close everything in the `File Actor::Stop Core` method and send a `Last Ack` to the `Controller`. The `Controller` will remove the queue with the ID `PAS Wvfm` from the `Controller` enqueuer map.



37: Writing PAS waveform data

Writing Tau Data

As with the PAS waveform data, the τ data may also be written to file. The activity involved in terms of user and software interaction is the same as that shown in the figure describing the message passing for the PAS waveform data so it will not be shown or described here. The ID for the file actor associated with the τ data is `crdtaus` as defined in the method `Start Tau Write MSG::Do Wrapper` (the wrapper is provided strictly for debugging purposes and should be removed in future work).

The message that is responsible for the execution of the of the τ writing is `Exe Write Taus`. The message that kicks off the writing is `Start Tau Write MSG`. `Stop Writing Taus MSG`.

When the user requests the τ data to be written to file, then it will write to the location provided in the INI file. The parameters for the writing of the τ data are found under the section `CRD Taus` and the path for writing is given by the keys `crdtaus.Path` which provides the top level path and `folder` which tells the system which folder to dump the files in. As will all file keys in the INI file, the user may specify the prefix and extension, but the file will also contain a date-time string as described in the section discussing an overview of file writing.

The τ data will be written as binary data regardless of the extension chosen for the files. The format of the files will be as such:

1. `Time` - a double precision float that represents the time since midnight 1904 Jan 1.
2. `i` - 32-bit integer defining the number of cells for which τ data will be recorded.
3. `j` - 32-bit integer defining the number of shots (i.e. τ's) recorded for each cell. This should be the same as the rep rate of the laser.
4. `tau data` - array of double precision floats representing the τ data. The length of the array should be `i*j`.

Recording Ringdowns

Recording raw ringdowns is similar to recording PAS waveforms or τ data. As the process of interactions and software activity has been described elsewhere (in the section describing PAS waveforms), it will not be repeated here. Rather, this section will describe the actual data recorded and the messages used to start, stop and record that data.

A request for the system to write the ringdown data is kicked off with the message `Start CRD RD File MSG`. This will kick off a file actor with the ID `crdrd`. If the launch of the actor is successful, then it will send the `Update Write Ringdowns MSG` to the `Instrument Actor` handling the CRDS interactions. This send the command `Set Write Ringdowns` which will set flags in the CRDS related objects to indicate that the system is currently saving ringdown data. When this flag is raised, the `CRDS Instrument` object *will not* analyze the data and record τ's. The reason for this is that the process of writing the ringdown data is too intense to allow the analysis of data while the writing is occurring.

When the user wishes to stop recording ringdown data, the system will send the `Stop CRD RD Write MSG`. In addition to stopping the file actor associated with the writing of the ringdown data, it will also send the `Update Write Ringdowns MSG` to lower the flag and indicate to the `Instrument` object that the system is no longer writing ringdowns (so it may resume analysis of that data).

The message that handles the actual recording of the data is `Exe Write CRD RDs`. Similar to the other files where waveform data is written, the following data is written in the order given:

1. `Time` - a double precision float that represents seconds since midnight 1904 Jan 1
2. 'i' - 32-bit integer that represents the number of cells in the system
3. 'j' - 32-bit integer that represents the number of shots (repetition rate of the laser) acquired
4. 'k' - 32-bit integer that represents the number of points in each waveform
5. `ringdown data` - array of 16-bit integers representing ringdown data. The number of points in the array is given by `i*j*k`

Current Value Table

Control values are synchronized between the client and server side via a current value table (CVT). The CVT functionality is wrapped in an object called `CVT`. `CVT` is contained within a session so that it may be shared across the multiple structures that require access to the current value table. The `CVT Session` is a property of the superclass `Ex Nested`.

The CVT structure will look something like this:

```
├── Group_1
|   ├── tag_0: val
|   └── tag_1: val
├── ...
└── Group_2
    ├── tag_0: val
    ├── Nested_Group_1
    |   ├── tag_0: val
    |   └── tag_1: val
    └── tag_1:val
```

Object

The `CVT` object is a child of the `Session` object as defined in another section and contains the following methods unique to this implementation of the `Session` object:

`initialize Session Data`

This is called when the first `CVT` session is created. This function simply initializes the only property of the class `CVT` with a new variant.

`Insert Data`

This is one of the two primary public functions. Aside from the input `CVT Session In` and `Error In`, this method contains three inputs:

- `Group`: This is a string which will identify the group that this data belongs to.
- `tag`: This is a string and is the name of the data that will be associated with the value of interest.
- `Data`: This is a variant is the actual data that we wish to store.

This function takes these inputs and calls the following private method:

`Insert Data Core`

This is similar to the method `Insert Data` but it has one extra input. This input is called `Nested?`. If the `Group` input of `Insert Data` contains a `.`, this will indicate that the group is a nested group

Members

The following descriptions define the information that is stored in the CVT. The control data is generally written where a successful update is made. In most cases, this is in the message that is processed to change the control value.

Group	tag	Description

pas		Defines functionality specific to the operation of the PAS.
pas.spk	fcenter	Center frequency of the speaker chirp
	df	Bandwidth in Hz of the chirp
	cycle	Boolean indicating whether the speaker is set to automatically cycle
	length	Length of speaker activation if cycle is TRUE
	period	Period in seconds over which speaker cycles to ON
	enabled	Current speaker state. TRUE is ON.
	vrange	Voltage range of the waveform for the speaker
	voffset	Voltage offset of the speaker waveform.
	ienabled	Array of booleans indicating speaker state for individual cells
pas.las	vrange	Array of floats that define the laser modulation range
	voffset	Array of floats that define the laser modulation offset
	enabled	Array of booleans representing laser enable signals. The indices of the array are relative to the cell.
	modulation	Array of unsigned ints that define how the PAS lasers are modulated; 0 = sine, 1 = square
crd	klaser	Array that represents the laser gains for the red, blue 0 and blue 1 lasers respectively.
	kpmt	Array that represents the PMT gains for the red, blue 0 and blue 1 lasers respectively.
	enable	Boolean array containing the enable state of the lasers in this order - red, blue0, and blue1
crd.blue	f	Rate at which the blue lasers fire in Hz. **This will be removed in the future such that the value is not specific to a laser wavelength.**
	dc	Dutcy cycle of blue lasers (percentage of time that laser is ON). **This will be removed in the future such that the value is not specific to a laser wavelength.**
crd.red	f	Rate at which the red laser fires in Hz. **This will be removed in the future such that the value is not specific to a laser wavelength.**
	dc	Dutcy cycle of red laser (percentage of time that laser is ON). **This will be removed in the future such that the value is not specific to a laser wavelength.**
calibration	o3_valve	Boolean value - `TRUE` valve is open
	active	Boolean value indicating whether a calibration is active or not
Humidifier.Med		Values that define the control scheme of the lowere humidity humidifier.
	p	Proportional gain of humidifier control loop.
	i	Integral gain of humidifier control loop.
	d	Derivative gain of humidifier control loop.
	ctl	Boolean defining whether control is active.
	rhsp	Initial setpoint of the control loop.
Humidifier.High		Values that define the control scheme of the lowere humidity humidifier.
	p	Proportional gain of humidifier control loop.
	i	Integral gain of humidifier control loop.

	d	Derivative gain of humidifier control loop.
	ctl	Boolean defining whether control is active.
	rhsp	Initial setpoint of the control loop.
general		Defines some generic functionality not specific to a device or either instrument.
	filter_pos	Initial position of the filter - `true` = flow is filtered.
	inlet	Initial position of sample flow valve - `true` = flow is from the sample line rather than the cabin.
	power	Byte representing state of the power supplies; see below for a detailed description of this byte.
	purge	Boolean defining initial position of the CRD purge flow valve. `true` indicates that flow will be through the mass flow controllers.
Filter		Defines parameters specific to the filter operation. See below for a more detailed description of the filter operation.
	period	A period defined in seconds over which the filter cycles if `auto` is set to `true`.
	length	Time in seconds that filter is active in the filter cycle if `auto` is `true`.
	auto	Boolean defining the initial automatic cycle state of the filter. If `auto` is set to `true`, the filter will cycle at startup based on the `period` and `length` defined above.

The `device` Tag

The `device` entry in the CVT is a top level entry used to group all devices attached to the instrument. Under this top-level tag is an entry for every device that has been registered with the controller at startup. Those device that fail to start should not be registered with the CVT.

Each device under the `device` tag contains several guaranteed members as defined below. In addition, if the device is identified as a `controller`, the device will have an additional entry called `setpoint` which is a floating point number.

Example

An example of a device is given below:

```
"device": {
        "p1": {
            "type": "ppt",
            "label": "P<sub>1</sub>",
            "sn": "00089546",
            "controller": false,
            "address": "p1",
            "model": "PPT"
        },
        "AlicatA": {
            "type": "alicat",
            "label": "High RH",
            "sn": "104488A",
            "controller": true,
            "address": "A",
            "model": "104488A",
            "setpoint": 0.000000
        },
        ...
}
```

In the example above (taken from an actual CVT capture), there are two entries displayed in the `device` portion of the json file: an Alicat controller called *AlicatA* and a Honeywell PPT pressure transducer called *p1*. Both contain all guaranteed members while the device `AlicatA` contains the additional `setpoint` entry.

Guaranteed Members

- `type` - defines the type of the device. Used by the user interface to group arrays of devices.
  - `ppt` - Honeywell pressure transducer
  - `mTEC` - Meerstetter thermoelectric cooler
  - `TEC` - TE Technology thermoelectric cooler
  - `alicat` - Alicat flow device
  - `vaisala` - Vaisala hygrometer
- `label` - string that defines the identity of the device in plots and other user interface items
- `sn` - serial number of the device. Used strictly for device identification.
- `controller` - boolean indicating truthiness of whether the device is a controller.
- `address` - string address that defines how the server communicates with the device
- `model` - defines the model of the product. Used strictly for identification of the device.

In addition to the members defined above, those that are defined as a controller will have an additional entry called `setpoint` which is the current setpoint defined in the CVT for that device.

Calibration

Calibration on the server side is handled by two different actors - `Calibration Actor` and `O3 Frequency Train Actor`. The first is run on demand when a user requests a calibration and closes when the calibration sequence is finished. The second is started when the `Controller::Actor Core` runs and does not shutdown until the application shuts down.

The `Calibration Actor`

When the `Calibration Actor` starts, it will first parse the calibration file as defined in a previous section using the method `Calibration Actor::Read Calibration Routine`. This will load the sequence defined in the file into the property of the `Calibration Actor` class called `Sequence`. The object has a type `SEQ` which simply encapsulates a queue containing objects of type `STEP`.

When the `Pre Launch Init` routine is complete, the `Actor Core` will kick off. This will set a flag in the CVT to indicate that the system is calibrating and start the `Calibration Actor::Calibration Engine` method. This method is the heart of the `Calibration Actor`. This method will loop through the steps defined in the `SEQ` queue. Most steps will execute immediately, but one step `Wait STEP` will execute over multiple iterations. Each step provides a flag to indicate whether to proceed to the next step. Once all steps are complete, the main loop will close and normal stop message will be sent to itself.

`O3 Frequency Train Actor`

The `O3 Frequency Train Actor` has one function and that is to produce a pulse for driving the O3 generator. As previously stated, this actor is started when the EXSCALABAR application kicks off and does not complete until the application is closed. The actor uses a notifier that is obtained in `Pre Launch Init` called `FreqNotifier` to communicate the desired frequency of the pulse train to the loop handling the production of the pulse in the `Actor Core`.

This loop that produces the pulse train is encapsulated in the method `O3 Frequency Train Actor::Generate Pulse Train v2`. This pulse is generated strictly by toggling the digital output associated with the O3 generator input. If no pulse is desired, then the notifier will produce a default output of 0. This will shutdown the pulse train and set the DO to `FALSE`. A request to shutdown is communicated by the notifier. When the `Stop Core` is called, the notifier is released and the reference is returned as invalid.

General Tips and Tricks

Using Git

The version control system used at the time or writing is git. The repository is maintained on Github. Currently, there are two separate repositories for the project:

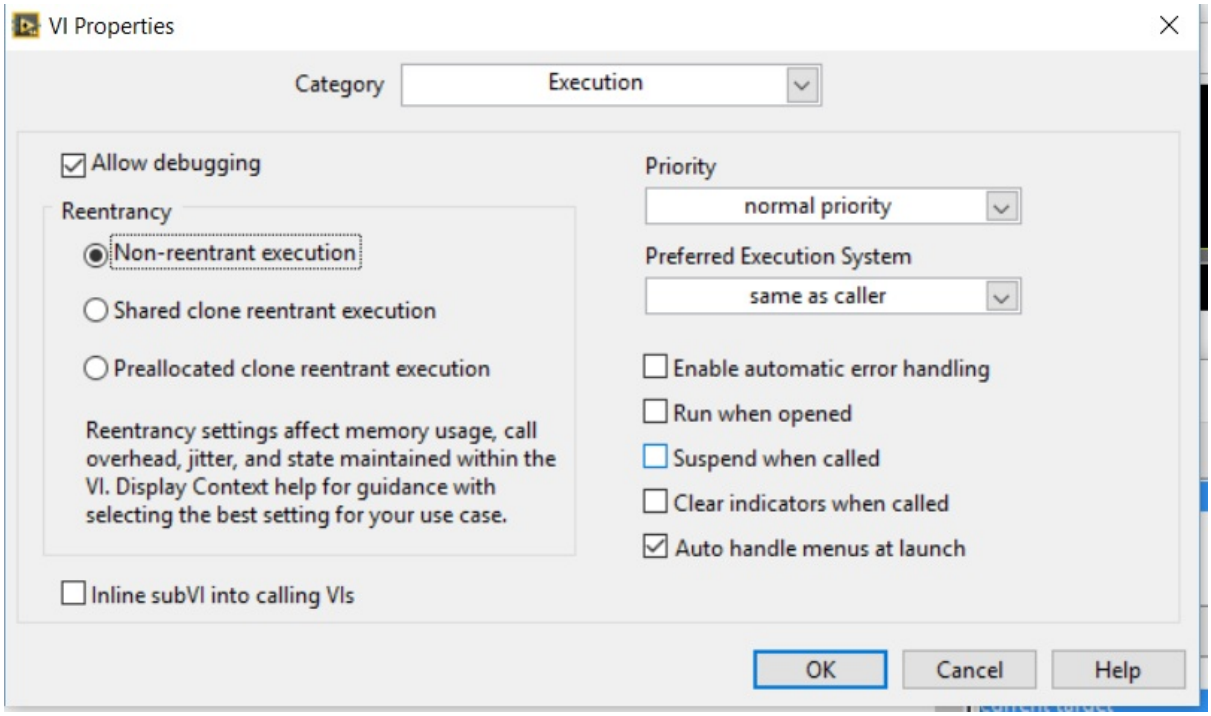
- exscalabar - this repository is used for storing user interface code.
- exscalabar_server

Debugging

Setting Reentrancy for VIs

The majority of the Actor Framework relies on VIs that are reentrant. This means that the VIs run in their own memory space and the state of one has no impact on another instance. This is a powerful tool for running tasks in parallel. The key problem with reentrancy on a real-time system is that the reentrant aspect relies on the instance having a unique front panel for debugging access. As there are no front panels in on a real-time system, this means that reentrant VIs are not directly accessible through traditional debugging techniques.

To handle this, the developer may employ a variety of techniques, but ultimately they will want to be able to access the portion of code unique the operation of interest directly. In order to do this, they will have to set the reentrancy of a VI of interest to **Non-reentrant execution** in the VI properties window. This window is accessed by either right clicking on the VI and selecting *Properties* or by opening the VI and selecting *File->Properties*. The developer may set the reentrancy properties by selecting the *Execution* category. By selecting **Non-reentrant execution** and checking the box marked **Allow debugging**, the developer will not be able to set breakpoints and probes to debug the VI of interest.



38: Setting the reentrancy

It is important to note that by setting the reentrant properties to **Non-reentrant execution**, only one instance of the VI may be accessed at a time. This means that VIs that run continuously will break the intended execution sequence. If the developer sets VIs such as `Actor Core` to non-reentrant, than the system will not function properly as all other instances of the `Actor Core` will wait for the completion of the execution of the first `Actor Core` to complete (in this system this is `Controller::Actor Core`. In the case of the `Message::Do`, the key to processing messages in the `Actor::Actor Core`, messages will not be executed in a timely manner if the execution is set to non-reentrant.

To avoid this, the developer should provide a wrapper VI for core functionality unique to the current instance of the VI of interest rather than setting the reentrancy of VIs such as `Actore::Actor Core` or `Message::Do`. This will allow the developer to debug as necessary while not impeding the execution of other portions of the system.

Clearing the Compiled Cache

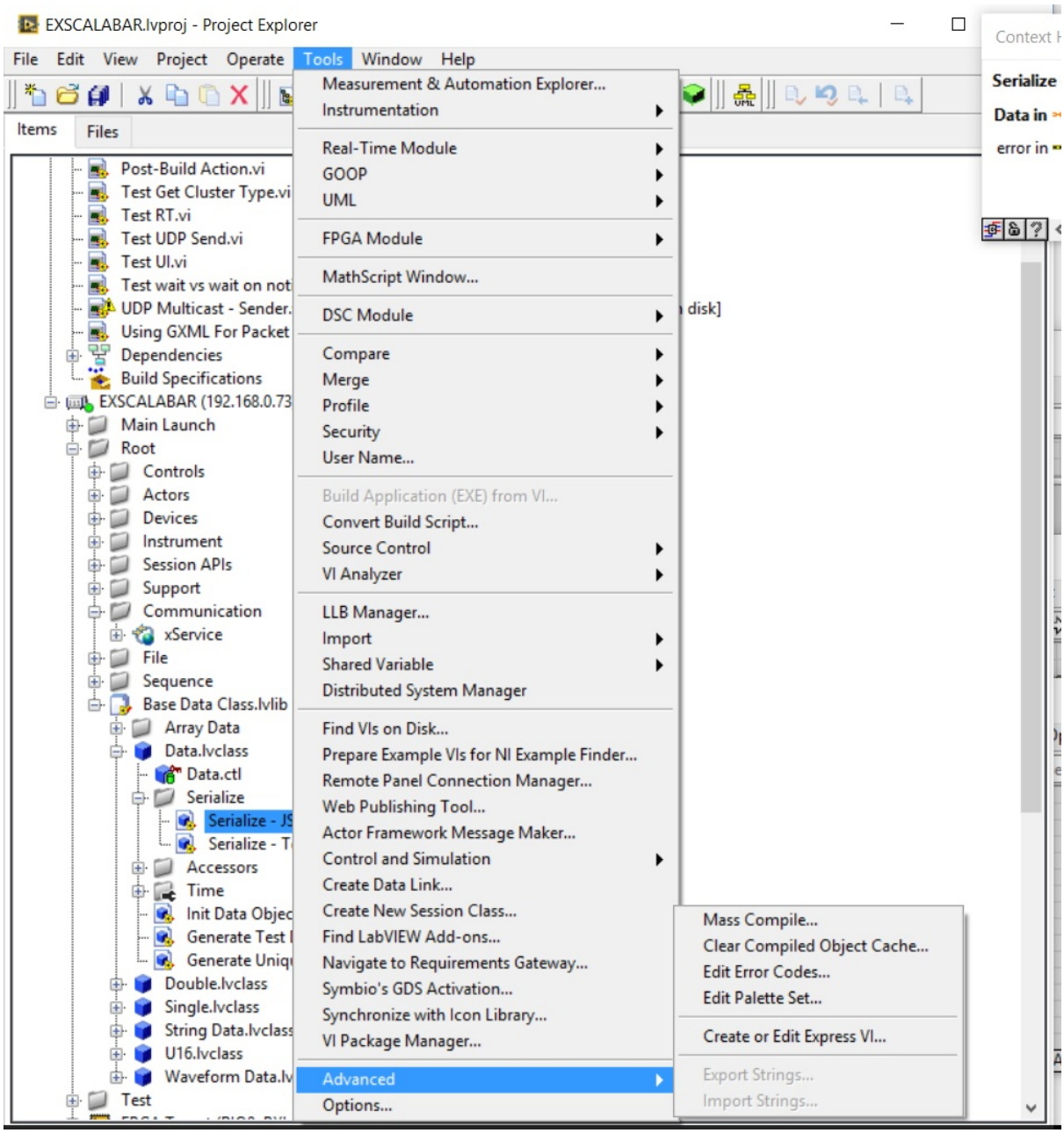
Occasionally, when debugging, the user may get an error on deployment (i.e. when they try to run a program or deploy the web service) that states there is an error with a VI. Upon opening the VI, the user will find that the VI is actually not broken. At this point, it would be best to clear the compiled cache (where the actual object code is stored), close the project and open it again. This *should* resolve these problems.

The user may clear the compiled cache one of two ways. They may clear it through the LabVIEW environment or by deleteing the object data folder. To clear the cache through the environment (see image below):

- navigate to the *Tools* menu on any screen in the LabVIEW environment (this is shown below within a project, but may also be done through a VI, library, etc.)
- Select *Advanced* from the *Tools* menu
- Click on *Clear Compiled Object Cache...*
- In the window that pops up, click the button *Delete*

This will remove the object file in the *LabVIEW Data->VIObjCache->[version]* folder.

Alternatively, the user may close the entire LabVIEW environment (i.e. shut it down) and simply delete the folder described above directly. The folder is usually found under the users *Documents* folder in Windows.

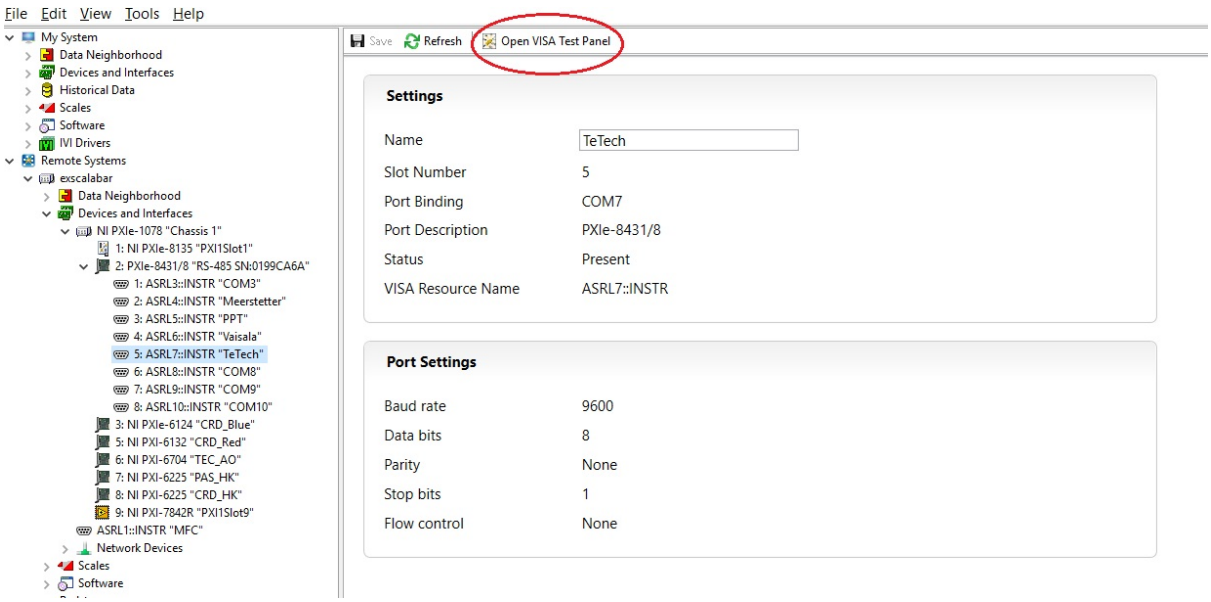


39: Clearing compiled cache

Debugging Serial Device Issues

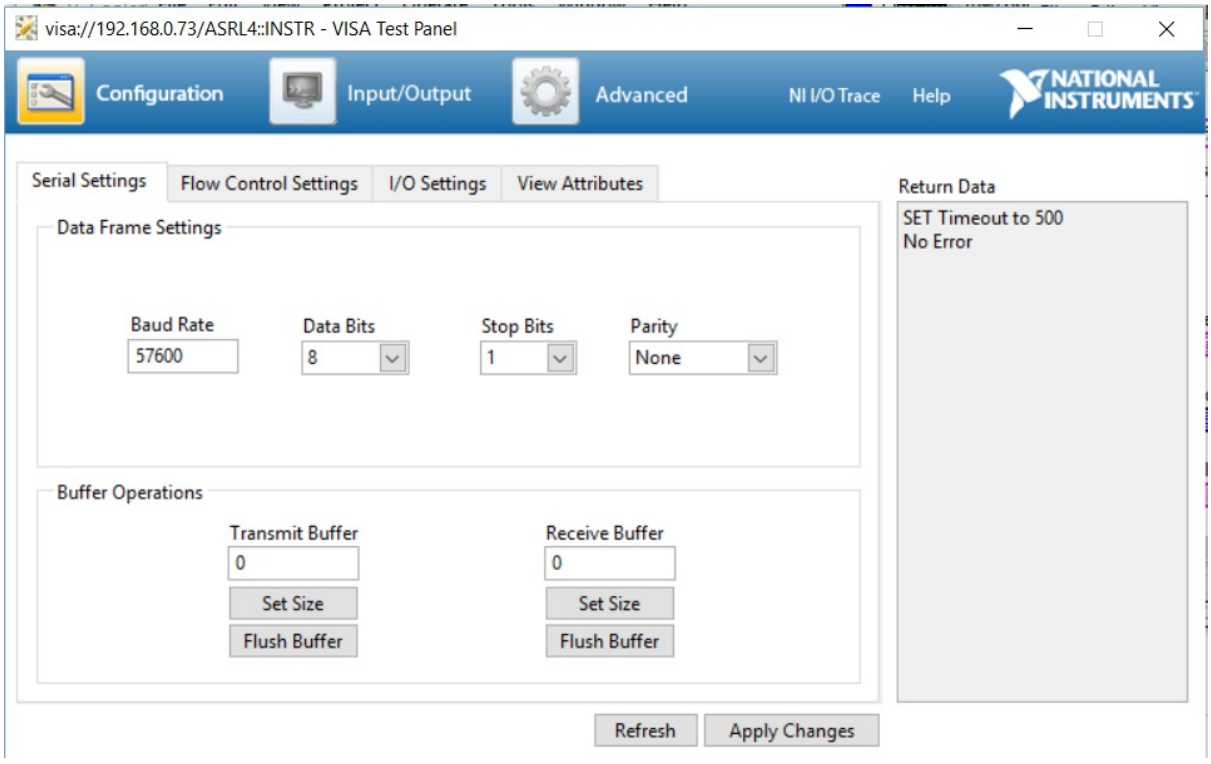
Occasionally, the user may have issues with serial communication with one or more sets of devices. In this case, MAX is a valuable tool. Settings for serial communication can be found under the specific heading of that device. This will include the port settings (such as baud rate, data bits and stop bit), message configuration and device addresses (if present).

To access a port in MAX, select the serial port of interest and click the button at the top of the panel `Open VISA Test Panel`.



40: Accessing serial devices through MAX

The test panel will look like below.

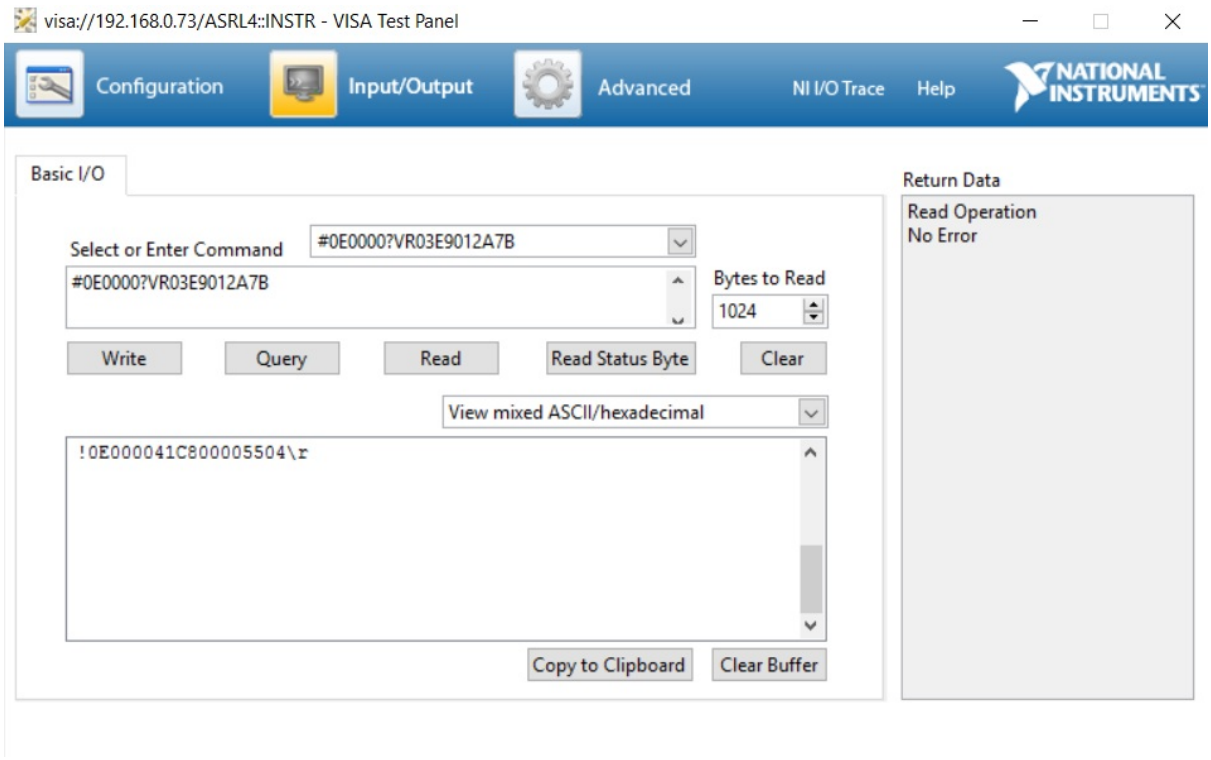


41: Serial data frame settings.

In this frame you can change the port settings as desired. Message settings can be found under `I/O Settings` (the user will generally not have to change `Flow Control Settings) as seen below.

42: Setting up serial IO.

Once the user has set the port up properly, they can communicate on that port through the `Input/Output` page.



43: Serial communication

The user can use this method to 1) check that devices are communicating as expected and 2) test new functionality of a device.

References

1	Peter H. Hildebrand and R. S. Sekhon, Objective Determination of the Noise Level in Doppler Spectra, 1974.
2	Michael A. Everest and Dean B. Atkinson, Discrete Sums for the Rapid Determination of Exponential Decay Constants, 2008.
3	Daniel Halmer *et al.*, Fast exponential fitting algorithm for real-time instrumental use, 2004.
4	Mazurenka, Mikhail *et al.*, Fast Fourier transform analysis in cavity ring-down spectroscopy: Application to an optical detector for atmospheric NO2, *Applied Physics B*, 2005.