**ExxonMobil Chemical Company**

**Lab Data**

**Design Guide**

**Version 1.0**

**May 30, 2020**

**Prepared by:**

**ILS Automation Inc.**

**TABLE OF CONTENTS**

[1 Introduction 4](#_Toc41579753)

[2 Architecture 4](#_Toc41579754)

[2.1 Context Diagram 4](#_Toc41579755)

[2.2 Data Flow Diagram 5](#_Toc41579756)

[2.3 Ignition Resources 5](#_Toc41579757)

[2.3.1 Database Tables 6](#_Toc41579758)

[2.3.2 UDTs 7](#_Toc41579759)

[2.3.3 Python 7](#_Toc41579760)

[2.3.4 Windows 8](#_Toc41579761)

[3 Installation, Configuration, and Initialization 8](#_Toc41579762)

[3.1 Installation 8](#_Toc41579763)

[3.2 HDA Server 9](#_Toc41579764)

[3.3 Configuration Preferences 10](#_Toc41579765)

[3.4 Initialization 12](#_Toc41579766)

[4 Lab Values 12](#_Toc41579767)

[4.1 Lab Value UDT 12](#_Toc41579768)

[5 Lab Limits 13](#_Toc41579769)

[5.1 Validity Limits 14](#_Toc41579770)

[5.2 SQC Limits 16](#_Toc41579771)

[5.2.1 SQC Lab Limit Configuration 16](#_Toc41579772)

[5.3 Release Limits 18](#_Toc41579773)

[5.4 Custom Validation 18](#_Toc41579774)

[6 Lab Value Selectors 19](#_Toc41579775)

[6.1 Basic Selector Design 20](#_Toc41579776)

[6.2 Selector Configuration API 21](#_Toc41579777)

[6.3 Automatic Selector Configuration 21](#_Toc41579778)

[6.4 Storing Selector Value History 22](#_Toc41579779)

[7 Unit Parameter Classes 24](#_Toc41579780)

[7.1 Repeat Sample Value Handling 25](#_Toc41579781)

[7.2 Data Consistency Due to Grade Change or State Change 25](#_Toc41579782)

[7.3 Data Consistency 26](#_Toc41579783)

[8 Lab Value Manual Entry / “Local” Lab Data 26](#_Toc41579784)

[9 Derived Value Processing 28](#_Toc41579785)

[10 Lab Value Processing 29](#_Toc41579786)

[10.1 Lab Value Processing Overview 30](#_Toc41579787)

[11 User Interface 30](#_Toc41579788)

[11.1 Operator User Interface 30](#_Toc41579789)

[11.1.1 Lab Data Table Screen 30](#_Toc41579790)

[11.1.2 SQC Plot Screen 32](#_Toc41579791)

[11.2 Administrator User Interface 34](#_Toc41579792)

[11.2.1 Configure Lab Data 35](#_Toc41579793)

[11.2.2 Configure Derived Lab Data 36](#_Toc41579794)

[11.2.3 Configure Selector Lab Data 37](#_Toc41579795)

[11.2.4 Configuring Lab Data Display Tables 38](#_Toc41579796)

[12 Troubleshooting Tips 38](#_Toc41579797)

[12.1 OPCHDA Status 39](#_Toc41579798)

[12.2 Message Queue 39](#_Toc41579799)

[12.3 Loggers 39](#_Toc41579800)

[13 Lab Feedback Control – Bias Calculations 39](#_Toc41579801)

[13.1 Theory 39](#_Toc41579802)

[13.2 Ignition Implementation 40](#_Toc41579803)

[13.2.1 UDTs 40](#_Toc41579804)

[13.2.2 External Python 41](#_Toc41579805)

[13.2.3 Initialization 42](#_Toc41579806)

[13.2.4 Inhibiting I/O 42](#_Toc41579807)

[13.2.5 Configuration 43](#_Toc41579808)

[13.2.6 Migration 44](#_Toc41579809)

[13.2.7 User Interface 44](#_Toc41579810)

[13.2.8 Troubleshooting 45](#_Toc41579811)

[13.3 Obsolete 46](#_Toc41579812)

[Appendix A Legacy System Information 50](#_Toc41579813)

[Appendix B Meeting Notes 52](#_Toc41579814)

# Introduction

**THIS DOCUMENT HAS A LOT OF REPITION FROM THE USER’S GUIDE. THIS IS MEANT AS AN INTERNAL: ILS DOCUMENT!**

This document describes the design of the Lab Data Toolkit that was developed for the Baton Rouge Chemical Plant (BRCP) using Ignition from Inductive Automation.

The toolkit manages reporting laboratory, or off-line, data between an external laboratory data management system and Ignition. Laboratory data differs from most data used in Ignition in that there is a significant delay between the time that the sample is taken and the results are reported. Laboratory data may be used in many ways, but it typically is used by SQC applications that monitor and control product quality. The toolkit provides support for laboratory measurements from a DCS, a PHD historian, or locally entered values. It provides a robust scheme for validating the value and for handling it if the value does not appear valid. The toolkit also calculates the lab value bias and provides feedback to the laboratory measurement system.

# Architecture

This section describes the Lab Data Toolkit Architecture

## Context Diagram

This diagram shows the systems that are involved in getting measurement data from the Laboratory system. Over time, the interfaces have become more reliable between the systems, but the diagram points out that there are numerous points of failure.



Figure 1 - Lab Data Context Diagram

## Data Flow Diagram

This diagram shows the data stores, processes, and interfaces that are involved in the lab data system. Understanding the data flow is crucial to troubleshooting the system.



Figure 2 - Lab Data Context Diagram

## Ignition Resources

The toolkit consists of User Defined Templates (UDTs), windows, and Python that executes both in gateway and client scope. The Python is contained in an external package *ils.labdata* in the *user-lib/pylib* folder under the Ignition folder in *Program Files*.

Although the toolkit is described in a stand-alone manor, it will reside in your project with the other toolkits (Sequential Control, Diagnostic, Recipe Data, etc). The resources for this toolkit will be organized using folders in the resource and tag trees. There will be a “Lab Data” folder for UDTs, windows, templates, and tags.

### Database Tables

The tables shown below are defined for this module:



Figure 3 - Lab Data Database Tables

### UDTs

The following UDTs are defined expressly for this module. The purpose of these UDTs is to expose the Lab Data values processed by the Lab Data toolkit for use by other toolkits. Even though the tags are memory tags, they should not be edited in order to coerce a new lab data value into the system.

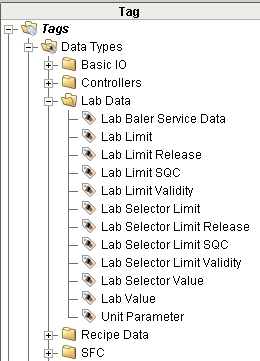


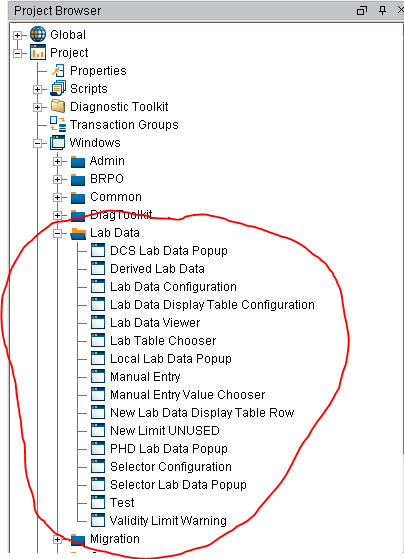
Figure 4 - Lab Data User Defined Types (UDTs)

### Python

The python used to implement Lab Data is contained in external Python in the ils.labData package

### Windows

The windows are contained in the XOM project in the Lab Data folder.



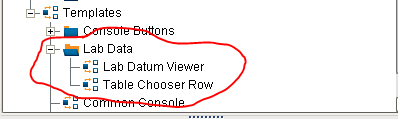


Figure 5 - Ignition Windows for Lab Data

# Installation, Configuration, and Initialization

This section provides a description of the required software for installation and the start-up configuration options for the Lab Data toolkit.

## Installation

There are no special installation steps required for this toolkit; all screens, UDT’s, and Python scripts are delivered as part of the normal installation.

## HDA Server

The lab data module uses the OPC Historical Data Access protocol to access lab data in Exxon’s PHD historian. In Ignition, from the gateway web page, a special historical tag provider needs to be configured as shown below.

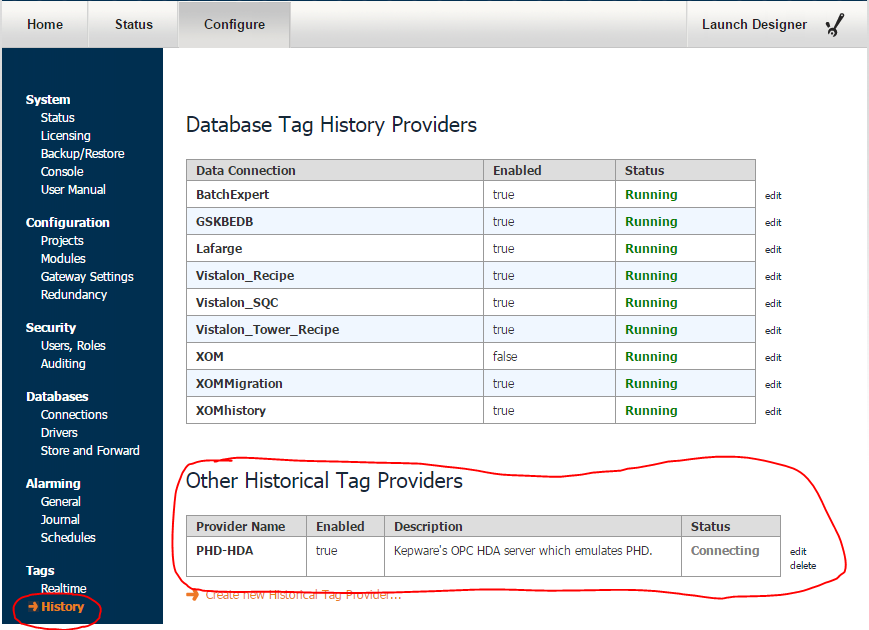


Figure 6 - HDA Server Configuration

## Configuration Preferences

The Lab Data module uses a set of memory tags to customize the behaviour of the module. The following characteristics of the Lab Data Toolkit can be customized using the Ignition Designer.

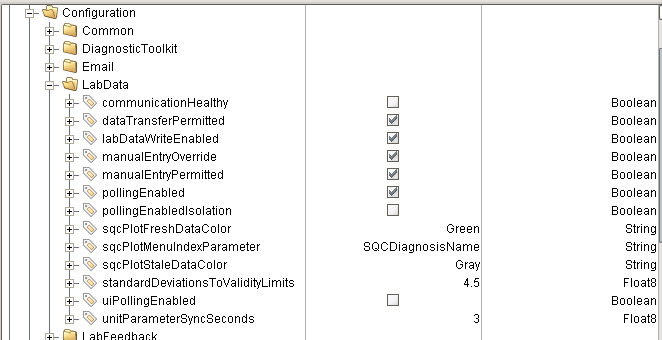


Figure 7 - Configuration Tags used for Lab Data

| Tag | Data Type | Description |
| --- | --- | --- |
| Communication Healthy | Boolean | This is a dynamic configuration parameter that is set by yet to be determined watchdog logic. When True, then only lab data which is designated as local may be manually entered (unless the following override flag is set). Initial value is True |
| dataTransfer Permitted | Boolen | *Is this used?* |
| labDataWrite Enabled | Boolean | Used to enable / disable writing values to external systems. This does NOT inhibit reading lab data from various sources nor from writing the processed values to local memory tags. It is used when writed derived lab values out to PHD. |
| manualEntry Override | Boolean | This is set by the watchdogs that monitor the health of the communication to the Lab Data system. If the watchdog determines that the communication is bad, ten this is set to True, thereby allowing entry of lab data manually. *I’m not sure if this is actually used! Look at labData.manualEntry.py* |
| manualEntry Permitted | Boolean | A flag that is used to override the manual lab data entry policy of only allowing local data to be entered manually. This may be used when communication to the lab data system is down, but the watchdogs have not detected it. |
| pollingEnabled | Boolean | This must be True for the module to automatically retrieve values from the OPC-HDA server |
| pollingEnabled Isolation | Boolean | This must be True for the module to automatically retrieve values from the OPC-HDA server for ISOLATION lad data tables and tags |
| sqcPlotFresh DataColor | String | Used to configure the chart pen for the fresh data. |
| sqcPlotMenu IndexParameter | String | Specifies the property that will be used as the button labels in the SQC Plot chooser (section 11.1.2). The options are: “SQCDiagnosisName”, “LabValueName”, or “LabValueDescription”. |
| sqcPlotStale DataColor | String | Used to configure the chart pen for the stale data. |
| standardDeviations ToValidityLimits | Float | The number of standard deviations from the target value to the upper and lower SQC limits defined in the RtSQCLimit tables. This is used to calculate the target and standard deviation. |
| uiPolling Enabled |  | Specifies if the Lab Data Table chooser and the SQC plot chooser table update automatically via a timer to reflect change in status. |
| unitParameter SyncSeconds | Float | Amount of time to wait, when processing unit parameters, to synchronize the lab value and the lab sample time. |

## Initialization

The Lab Data module has its own initialization logic in *ils.labData.startup.gateway*. Its only purpose is to create required configuration tags and assign default values if they do not already exist. It is generally a requirement to restore lab data history on startup to recover data that has arrived while the system was down or if this is an initial installation to restore data for the recent past. However, it is generally necessary to perform site specific selector configuration first. Therefore, there is no built-in history restoration as part of the lab data module. The module provides a function *ils.labData.startup.restoreHistory(tagProvider, daysToRestore).* The *daysToRestore* argument is an optional argument with a default value of 7.

# Lab Values

There are three sources of lab data: **PHD**, **DCS,** and **local.** The most common source of lab data is from PHD. The normal use case is that the lab takes a sample, analyses it, and inserts the result into the PHD Historian with a timestamp from when the sample was taken, not when the measurement was completed. DCS lab data is received directly from the DCS via an OPC tag. Local lab data is data entered by the operator.

## Lab Value UDT

Regardless of the source of the lab data, the same “Lab Value” UDT is used. The UDT defines the public interface to lab data values. Any of the tags that are available in the UDT are available to other toolkits. The two tags of particular interest are *value* and *sampleTime*. The last update time of the value tag is the time that the value was reported as opposed to the time that the sample was taken, which is what the toolkits are generally interested in. The *badValue*, *rawValue*, and *status* tags are present only for troubleshooting. It is unclear why any toolkit would ever want to use the *rawValue* since it is not validated.

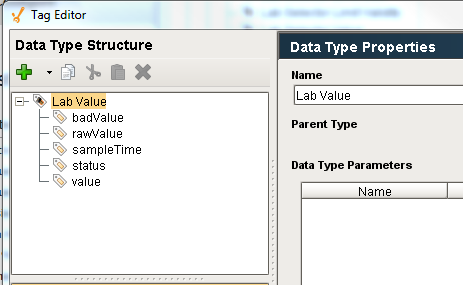


Figure 8 - Lab Value UDT

# Lab Limits

There are three types of limits that can apply to any lab datam. The limits are modelled in the database using the LtLimit table which is related to the LtValue table via the ValueId. Whenever a new lab value is entered, from PHD, OPC or local, the LtLimit table is searched for a limit record. The table would allow all three sets of limits to be defined for the same lab datum, although this is not generally the practice.



Figure 9 - Lab Limit Database Tables

## Validity Limits

Validity limits have an upper and lower limit. They may be manually configured or loaded from a database. The toolkit provides automatic validation of any **lab-data** for which a validity limit has been defined. The validation compares the raw value with a high limit and a low limit. Values that are outside of the acceptable limits are a serious problem and it is critical that the operator be notified immediately. If the raw value is outside the configured limits, then the responsible console/post will be notified of the lab data error by sending an “ocAlert” message to the appropriate post. The display shown below is large and the background toggles between purple and red.

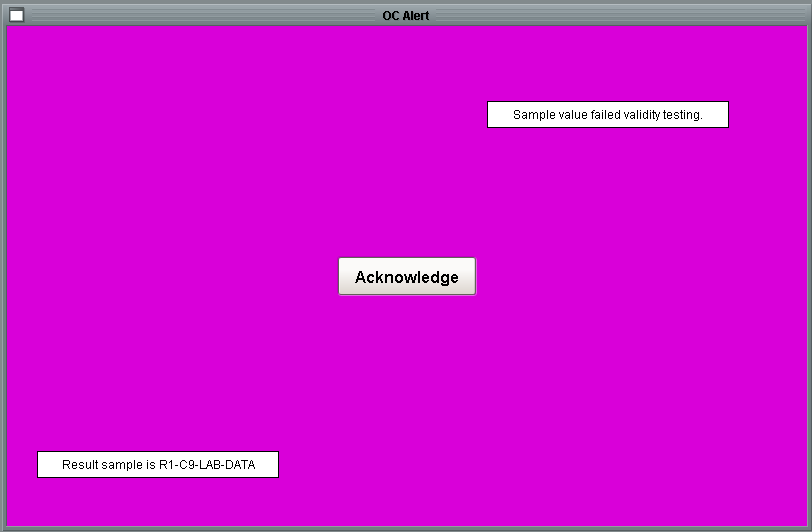


Figure 10 - OC Alert Screen for Validity Violation

When the user presses the ‘Acknowledge’ button, then the screen shown below is displayed.

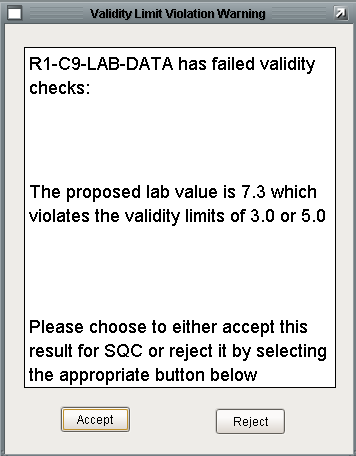


Figure 11 - Validity Limit Violation Warning Screen

The operator needs to determine if the raw value is believable or if the product attribute being measured is really out-of-spec. If the operator determines that the measurement is not believable due to an operator error, a measurement error, or a data communication error, then the operator can press “Reject” to reject the measurement. If the operator believes the measurement then the operator should press “Accept”.

If the operator presses “Reject” then no further action is taken and the lab value is effectively discarded. If the operator presses “Accept” or if the timeout period elapses, then the value and sample time are written to the UDT and the value is stored in history.

The action taken by the operator is recorded to the LABDATA queue. The message contains details about the lab data parameter including parameter name, value and limits.

Additional Considerations:

* If the console is not connected when the message is sent, then the value is automatically accepted – the presumption is that the measurement is valid.
* I have not implemented any logic for a new value arriving while I am processing the current value. Depending on how long the timeout is for the operator response to a bad value it could happen – not sure if this would cause a problem…
* I have not added any logic to protect against the window becoming disconnected while the window is displayed – If this is a problem, I can automatically accept the value if the window is closed.
* There is a timeout on the violation warning screen but not on the OC notice screen – so that screen could flash for hours and the lab value would never be accepted.

## SQC Limits

SQC limits imply that there are also validity limits. Therefore there are two sets of limits: SQC and validity. The SQC limits are loaded from the recipe data base and are entered about the process target and are +/- 3 standard deviations from the target. The validity limits are +/- 4.5 standard deviations from the target. (The default value is +/- 4.5 but may be customized using the *standardDeviationsToValidityLimits* configuration parameter described in section 3.3. In order to avoid inconsistent data, only the SQC limits are stored in the database. The target, standard deviation, and validity limits are calculated from the SQC limits.

There is no additional processing of lab data for SQC limits, other than the standard processing defined for validity limits.

### SQC Lab Limit Configuration

SQC lab limits are generally loaded from the recipe toolkit automatically when a grade change is detected. The recipe toolkit tables that model SQC limits and some typical data are shown below. While generally applying to SQC limits, the recipe toolkit may also configure release limits. If either the upper or lower limit is NULL then the LtLimit will be updated with NULL and NAN will be written to tags. Release and validity limit may be one-side although this is not typical.



Figure 12 - Recipe Toolkit Database Tables and Data

Data from the recipe tables shown above are loaded into the active lab data tables by the Python script *ils.labData.limits.updateSQCLimitsFromRecipe(grade)*. The lab data limits must be carefully configured so that the LtLimit.RecipeParameterName = RtSQCParemeter.Parameter. A portion of the LtLimit data for Vistalon is shown below.

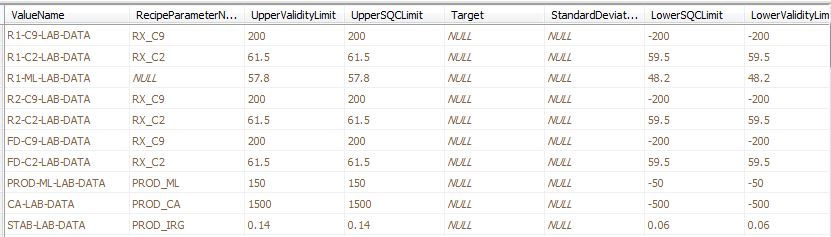


Figure 13 - Typical Lab Data Limit Data

## Release Limits

Release limits implement product performance limits that are based on product specifications rather than on process capability. Violation of release limits mean that the product is probably off-spec and may require special treatment in order to be released. Similar to validity limits, the toolkit provides automatic validation of any **lab-data** for which release limits have been defined. The validation uses the flashing notification screen to the operator just the validity limit processing. Once the notification is acknowledged, the following dialog it posted. It provides a convenient mechanism for automatically generating a UIR.

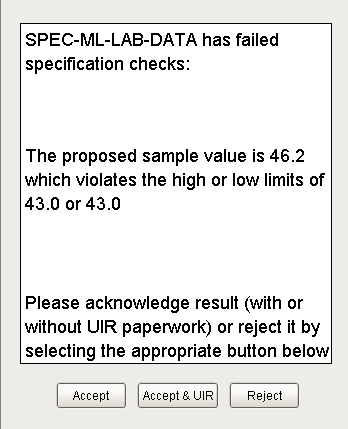
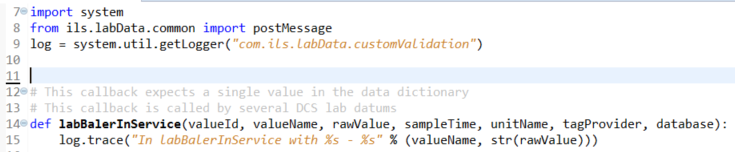


Figure 14 - Release Limit Notification Dialog

## Custom Validation

Custom validation allows a totally custom validation strategy to be implemented by writing a Python script. The Python can be located in extern Python, if using an external IDE such as Eclipse or in the global shared scope. If using External Python then it should be placed in the module *xom.site.labData.validationCallbacks.py* where site is your site name such as Vistalon. If in shared scope it should be placed in *shared.site.labData.validationCallbacks*. The script cannot be placed in project scope because it is called from a global timer script which is not attached to any project. The custom validation script takes seven arguments as shown below:



It returns three values as shown below:

|  |  |  |
| --- | --- | --- |
| Variable | Type | Description |
| isValid | Boolean | Specifies if the lab data value is valid. If it is valid then the lab value handling framework will promote the lab value to the lab data UDT and to the lab value history database. If the value is not valid then the handling depends on the notifyConsole setting below. |
| notifyConsole | Boolean | This value is only relevant if isValid = False. If notifyConsole is True then the framework will attempt to notify the appropriate console to allow the operator to accept the value in spite of it failing validation. If no console is available, then the value will be accepted. The notification is similar to the Validity Limit notification described above. If notifyConsole is False then the value will be rejected without notifying the operator. |
| statusMessage | String | If the value is not valid, then this will be written to the status tag of the lab value UDT. |

If the custom validation chooses to implement some totally custom notification / override then it is up to the custom method to save the values to the Lab Data tags and Lab Data history tables. The custom validation procedure runs in the gateway and should not have wait states. Any user interaction runs in a client so communication between the validation procedure in the gateway and the client notification will require a message.

The custom validation callback is specified in the Lab Data Configuration screen described in section 11.2.1. It is stored in the database in table *LtValue* in attribute *ValidationProcedure*.

# Lab Value Selectors

Lab Value Selectors provide a mechanism whereby a generic piece of lab data may come from different tags based on the configuration of the unit. This allows other toolkits within the application framework to reference the generic piece of lab data and does not need to be concerned with the ultimate source or the management of the source.

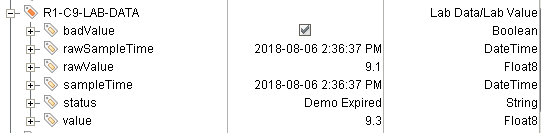
## Basic Selector Design

The configuration of selectors in the new system is similar to the old system. Lab Value tags/UDTs are configured for each of the possible data sources and then the appropriate value is promoted to generic selector object which would then be propagated to the appropriate unit parameter if appropriate. So in a scenario where a lab value could come from three different sources, there would be three tags and a selector.

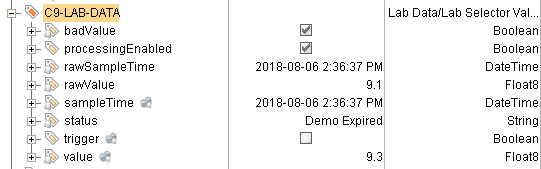
A typical implementation is shown below. The lab datum objects, R1-C9-LAB-DATA, R2-C9-LAB-DATA, and FD-C9-LAB-DATA are configured and update automatically. Based on the configuration of the unit, one of the lab datum objects are designated to promote their values to the lab selector C9-LAB-DATA.



The Lab Value UDTs have a set of memory tags that are explicitly updated from the Lab Data module.



The selector UDT has a set of expression tags that refer to the appropriate Lab Value UDT instance.



The expression tags are configured from properties on the UDT. The figure shows that the source of the selector at this moment is R1-C9-LAB-DATA:

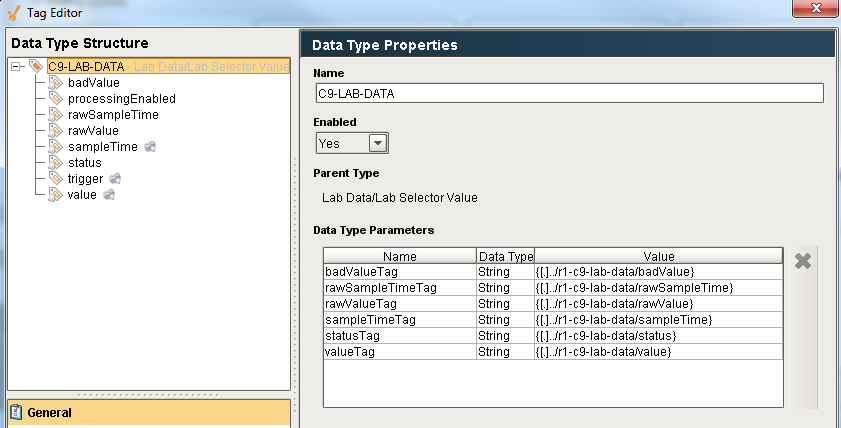


Figure 15 - Selector Instance Configuration

## Selector Configuration API

Selectors such as the one shown in the previous section can be conveniently configured via a Python scripting interface based on whatever logic is appropriate. The Vistalon selector configuration considers the unit configuration and the flash drum configuration for example. A portion of the Vistalon script demonstrating the API is shown below:

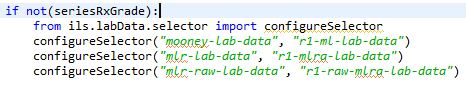
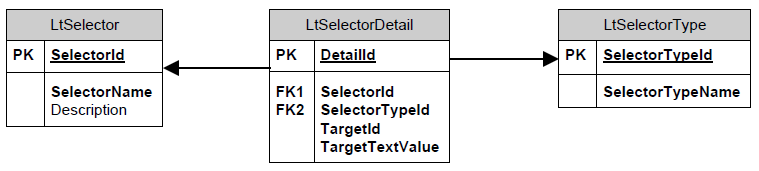


Figure 16 - Selector Caption API Example

## Automatic Selector Configuration

*This is just an idea with the goal being a data driven way of configuring selectors. I don’t think it would work for Vistalon, but maybe it would work for other sites. At this time it will not be implemented, but it remains in the specification for future consideration.*

The idea behind automatic configuration is that an event would be defined and then a set of detail configurations would be defined for that event. The processing of the details would be provided by the Lab Data module. Detection of the event is left to the site-specific logic. An API would be provided to implement the details for the event in a single call. The table structure to support this is shown below.



## Storing Selector Value History

The history for selector lab data is stored in an entirely different manner than normal lab value handling. Lab Data history is stored in the *LtHistory* database table. Normal lab data inserts data into this table by labData.scanner.py in the same thread as the data is read from PHD or the DCS. As described above, lab data selectors are implemented entirely by tags. The Lab Selector Value UDT shown below defines value changed handlers that insert the lab values into the *LtHistory* table. The UDT contains six expression tags that generally all update at the same time. The value and sampleTime need to be stored in the *LtHistory* table. In order to synchronize the insert for near simultaneous update the UDT contains a trigger tag. The “Value Changed” handler for the value and sample Time is identical and shown below:

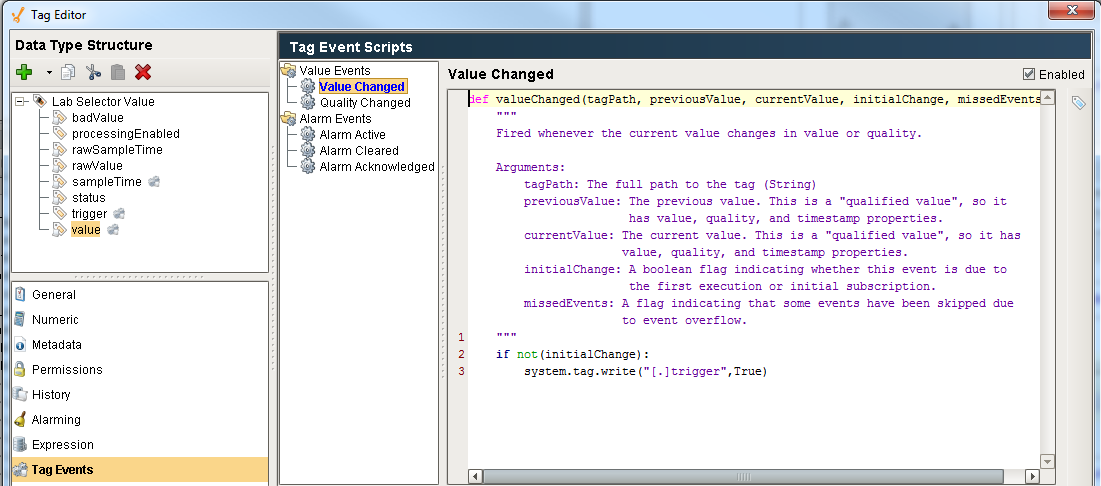


Figure 17 - Selector Value Change Script

The trigger tag change scripts is shown below. It calls a Python script which inserts the data from the value and sampleTime tags and inserts it into *LtHistory*.

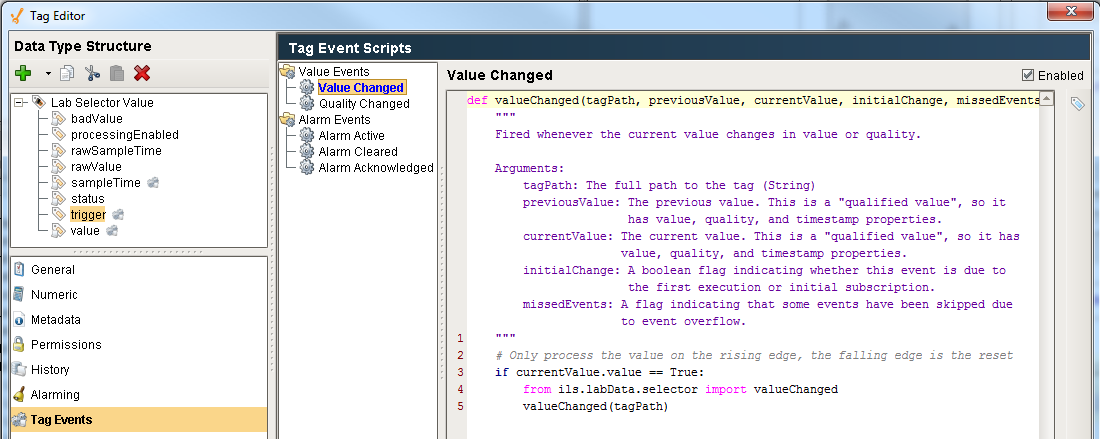


Figure 18 - Trigger Value Change Script

# Unit Parameter Classes

Unit parameters are a general-purpose class / UDT but since a typical lab data value may propagate to a unit parameter it is implemented as part of the lab data module. Unit parameters are also referred to as a filtered parameter. The unit parameter is configured with a sample size, n. Its value is calculated by averaging the last *n* values.

Unit Parameters are implemented in the UDT shown below. The buffer of values used to calculate the filtered value is stored in a database table, but there is no configuration required in the database. Unit parameters are created and configured manually in the designer. There is not a screen in the client for configuring unit parameters. There is nothing that requires configuration in the database when a new UnitParameter is created. The tables are maintained automatically as values are processed.

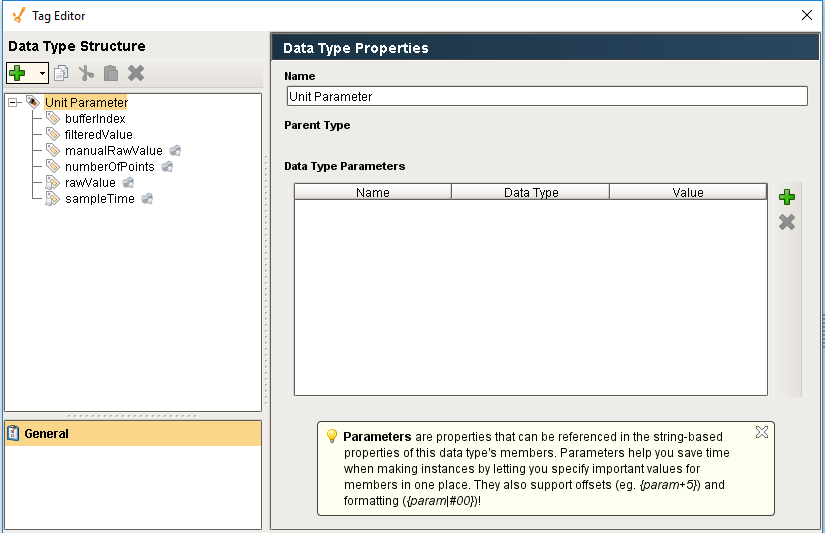


Figure 18 - Unit Parameter UDT

The *numberOfPoints* tag defines the size of the buffer used to calculate the mean. The buffer is maintained in the database. Changes to the *numberOfPoints* value causes the buffer to automatically resize.

The *bufferIndex* tag is an index into the circular buffer of values. It is an index into the last value that was written. This value is automatically incremented and should not be manually changed. Changes to the *numberOfPoints* value causes the buffer to automatically resize.

A Unit parameter has two modes of operation, automatic or manual. In automatic mode, the expression of the *rawValue* tag is configured to with an expression which typically refers to a tag such as a lab datum or OPC tag. In manual mode, something writes a value into the *manualRawValue* tag which is a simple memory tag. Both the *rawValue* and the *manualRawValue* trigger the same calculation and the filtered value is written to the *filteredValue* tag. Typically, the source of the unit parameter is either the *rawValue* or the *manualRawValue*, but not both, although nothing prevents this.

## Repeat Sample Value Handling

When a Unit Parameter is used with a Lab Data value input, special consideration must be given to handle repeat sample values. When used with Lab Data, the *sampleTime* expression tag should be configured to reference the *sampleTime* of the lab data UDT. The UDT has a value changed script on both the raw value and on the sample time. When configured in this manner there are three possible operating scenarios as shown below:



The first case is when a new different value with a new sample time is received. The second case is when two identical values are received, in this case, only the sample time is updated. The third case is when a lab value is updated, in this scenario, only the value is updated and the sample time remains unchanged.

In the figure above, each arrow represents a processing thread. In the first case there are two threads and the algorithm needs to be careful to prevent the value being placed into the value buffer twice. It is assumed that the *rawValue* and *sampleTime* are received at nearly the same time. The algorithm checks that the value and sampleTime were received at nearly the same time. If they were not, then it will wait a configurable dwell time. If only one of the two was updated then we are operating in case 2 or 3. Each thread reads both the raw value and the sample time and the time that those values were received as reflected in the *LastChange* attribute of these tags. If both threads detect that both the *rawValue* and the *sampleTime* have been updated, then the *sampleTime* thread exists, thereby preventing duplicating the value in the buffer.

Note: A unit procedure can operate in automatic mode with just the *rawValue* expression tag configured. The *sampleTime* tag is optional although it is necessary if consecutive identical lab datums are expected.

## Data Consistency Due to Grade Change or State Change

Unit Parameters are often used with Lab Data and Lab Data may have a significant delay between the time the lab sample is collected and when the lab sample is reported. It is important that data collected before a state change but reported after the state change should be ignored. There are two aspects to this: clearing the buffer of data has already been collected and preventing data yet to be reported but collected before the state change from being added to the data buffer. This is automatically accomplished by updating the *configurationChangeTime* tag of the Unit Parameter UDT. This is implemented in the *configurationChanged()* function in *ils.labData.unitParameter.py*

## Data Consistency

In certain situations, there is a concern that related data are consistent. For example, if the raw value for a unit parameter comes from lab data, then at a specific moment in time it may be important that the value from lab data and the filtered value of the unit parameter are consistent. This is relevant when using event based processing of the values in multiple threads. Although the need specifically arose related to lab data and unit parameters, there is a general need to determine if two tags that are related are consistent.

Therefore, utility function *checkConsistency()* has been provided in the Diagnostic Toolkit, refer to the **Diagnostic Toolkit Design Specification** for details.

# Lab Value Manual Entry / “Local” Lab Data

The majority of lab data is expected to be entered automatically via OPC or ODBC communication to external lab measurement systems. However, lab data can be manually entered when either of the following conditions is true:

* The lab data reporting is not automated and will always be reported manually. This is determined by an entry in the *LtLocalValue* table
* Communication to the lab system is temporarily not available OR the lab data configuration override tag named *“Configuration/LabData/manuallyEntryPermitted”* is set OR the user is an engineer AND the lab data is configured to allow manual entry by the *ManualEntryPermitted* attribute in the LtValue table AND the lab data is not a derived value.

Data is manually entered with the user-interface shown below, which is launched from the lab data pushbutton on the common console.

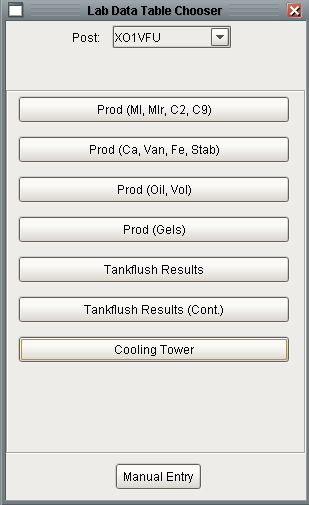


Figure 19 - Lab Data Table Chooser Screen

Pressing the “Manual Entry” button displays the dialog shown below which displays a choice of all of lab measurements for the post. The screen on the left shows the choices for an operator when communication is healthy and the manual override is not set. The screen on the right shows the choices when the user is an operator or communication is down or the manual override is set.

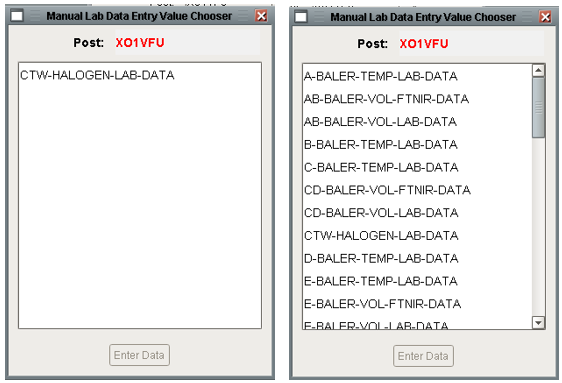


Figure 20 - Manual Lab Data Entry Value Chooser

Selecting a measurement and pressing “Enter Data” displays the screen shown below:

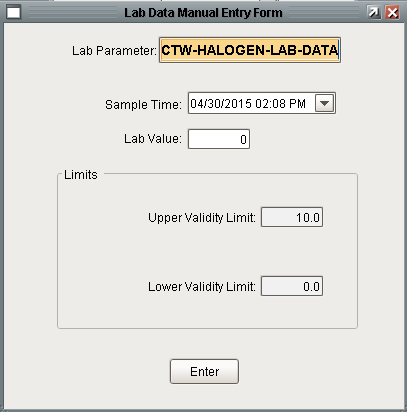


Figure 21 - Lab Data Manual Entry Screen

The sample time widget has a pulldown button that displays a nice calendar for specifying and validating the date and time. The limits container displays limits that are defined for the selected measurement. When the user presses “Enter” the value is validated using the normal validation logic. If the value fails validation the user will be notified via a pop up error message, but not the loud workspace which is used for automated lab data. Once the value passes validation it is stored to the local lab data history system. If the lab data is a “local” value then it is written to PHD via a normal OPC tag write.

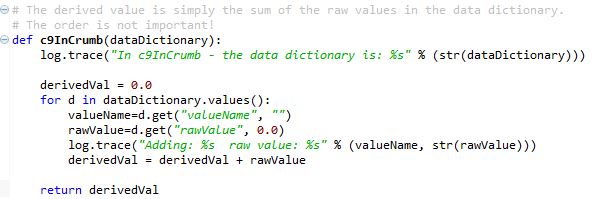
# Derived Value Processing

Derived values are values that are derived from one or more lab values. The derived value is calculated in a Python script using custom logic. The value that triggers the calculation is known as the trigger value. The logic generally involves one or more additional lab values, also known as related data. The toolkit ensures that all of the lab measurements are consistent by checking that the sample time of all the measurements are within a user specified time window **before** the user-written script is called. If they are not immediately consistent, the module will continue to check for a user specified amount of time. Once all of the lab values are consistent, then the Python calculation method will be called. The previous toolkit required that this bookkeeping be done in the user written script.

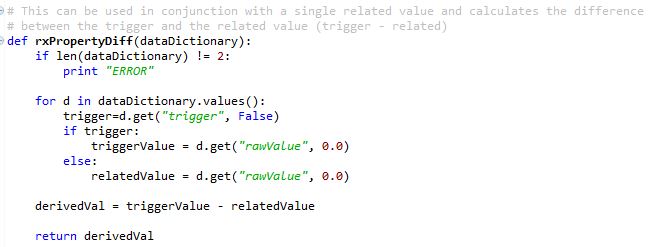
The values, including the trigger value, are passed to Python in a dictionary of dictionaries.



The following examples show actual Vistalon callback methods that illustrate how to access the data dictionary. The first example, c9InCrumb, iterates over all of the related data and simply sums its values. Because the toolkit validated the data before calling this method, the callback is quite straight forward. Similarly, the callback just needs to return the value, the toolkit will write the value to the UDT tag, the local history, and to OPC if the data is configured with a write target.



The second callback is written in a generic fashion but is dependent on there being exactly two values in the dictionary. The logic is to calculate the derived value as the trigger value minus the other value. This called from two different trigger values each with a different related data.



The related data is generally always lab data. If real-time data is needed in a calculation callback then it can be acquired directly via a tag read. Related data is generally only used when consistency is a concern.

The trigger value for derived data is generally a PHD lab value, but it could also be a DCS or local value. The derived value is configured in the LtDerivedValue table. The data that is related to the trigger value is defined in the LtRelatedData table.

# Lab Value Processing

Processing new lab values is entirely automatic. Lab data is polled on a regular interval for new lab data. The value that is read is compared to the last value processed. If they are the same then there is no further processing.

## Lab Value Processing Overview

The flowchart below shows the general processing that occurs when a new lab value is received.



Figure 22 - Lab Value Processing Flowchart

# User Interface

The Lab-Value-Handling module provides a user interface for viewing lab data. It provides a number of classes for configuring the user interface so that the data is presented in a logical, organized fashion.

## Operator User Interface

There are two main screens used by the operator related to lab data, the Lab Data Table screen and the SQC plot screen.

### Lab Data Table Screen

An application can contain many measurements; therefore the measurements are organized into logical groups or pages for viewing purposes. The user interface is accessed via the lab data pushbutton on the common console, shown below. Pressing the button opens the “Lab Data Table Chooser” screen. The screen provides a menu of the lab data tables available for the operator console that pressed the button. The workspace shows a maximum of eight lab table names. If more than eight lab tables are available for the selected post, then a strip of buttons is automatically added above the list to scroll through the pages of table names.



Figure 23 - Main User interface Context

Selecting any of the table names opens the screen in the top right that contains a row of tables each of which displays the most recent thirteen measurements for a lab value. The table updates automatically every 15 seconds. The previous application updated every ten minutes.

The “Get History” button is used to restore measurements that may have arrived when the OPC HDA interface was down. This should only be necessary if more than one measurement arrived while the interface was down as the regular 15 second poll will get the last measurement. When pressed, the last fourteen days of data will be fetched. Any data that is restored is not subject to validation, nor will it trigger a derived variable that may be dependent on the new data. Any new values that encountered will not be written to the lab data tags and therefore will not propagate to SQC diagnostic diagrams.

### SQC Plot Screen

The SQC plot screen combines lab data with diagnostic diagrams. It shows an SQC plot of lab data that is used by SQC blocks in a diagnostic diagram. Not all lab data are used in a diagnostic diagram involving SQC so not all lab data is viewable in this manner.

The following figure shows the screens involved in the SQC plot user interface.



*Note: these screen shots were taken from a test system and the data shown on the chart is not consistent, for example, the data points do not reflect a low 1 of 1 failure.*

The SQC plot user interface is accessed from the upper left button on the common console. It launches the SQC Plot Chooser screen. This screen shows all of the SQC diagnosis for the user’s post. SQC diagnosis for other consoles are available by choosing a different post from the post dropdown.

The list of SQC diagnosis used to populate the list comes from the DtSQCDiagnosis table. This table is initially populated during migration. It is updated every time the state of the Diagnosis changes. If a new SQC diagnosis is created, it will be added to the table automatically the first time the block processes a value. The list is animated to show SQC diagnosis that are currently active in red. Selecting a SQC plot will open the SQC plot window. The labels of the buttons in the list are configurable to show the name of the SQC Diagnosis, the name of the lab value that feeds the SQC diagnosis, or the description of the lab value that feeds the SQC diagnosis. This is configured aas described in section 3.3

The SQC plot window is passed the name of the SQC diagnosis and the internal id of the SQC diagnosis in the block language toolkit. Using the block id, the utility queries the block language toolkit to determine the target, standard deviation, SQC limits, violated rules, and source of the lab data. Specifically, the utility collects all of the SQC Observation blocks on the diagram and extracts the target, standard deviation, limit type (high or low), the number of standard deviations from the target, and the state. All of the observation blocks on the diagram will have the same target and standard deviation. The target is shown as the thick blue line in the plot. The limits are reflected in the red lines. Limits that are less than 0.5 standard deviations from the target are not shown. At most two upper and two lower limits will be shown on the plot – this is the typical configuration. The blocks that are currently active are listed in the violated rules list in the upper right. Neither the SQC diagnosis nor the SQC limit observation blocks contain the name of the lab data measurement used in the diagram. The only block on the diagram that contains the name of the lab data measurement used in the diagram is the Lab Data Entry block. Every diagram that contains a SQC diagnosis must also contain exactly one Lab Data Entry block upstream of the diagnosis.

The main component on the window is the EZ Chart. An EZ Chart was used rather than a simple chart because of the ease of drawing the blue and red lines using the calculated pens feature. The data for the chart are fetched from the database view named LtValueView. By querying the database for the values, the plots are not subject to losing data when the system is restarted. The chart is a real-time plot meaning that it updates periodically (every minute). The amount of time shown on the time axis is configurable from the widgets at the bottom of the chart. A popup tooltip is provided when you hover the mouse over a data point. The tooltip shows the time, value, and grade of the point.

Questions:

1. The current system’s tooltip shows the grade along with the time and value, currently we don’t record the grade with the lab data AND I don’t know how to bring in auxiliary data in the tooltip – I think we know how to do this but am still working through the details.
2. Currently, I think points are color coded to show if the point contributed to an SQC error. Is this necessary? Can’t they see that from the limits on the chart? I REALLY can’t change the color of individual points…

## Administrator User Interface

This section describes the user interface for configuring lab data instances. There are a number of screens for creating and configuring lab data which are accessed from the main Admin menu shown below:

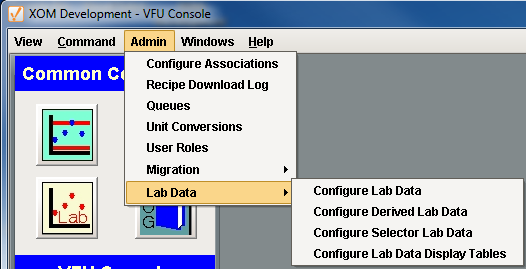


Figure 24 - Lab Data Administration Configuration

### Configure Lab Data

This screen is used to configure lab data that comes from PHD, the DCS, or locally entered. The screen shows lab data by unit and contains tabs for each different source. Lab data can be added, deleted, or edited via these screens. When a row is selected, the appropriate limits are displayed in the table at the bottom. Limits can also be added, deleted or edited.

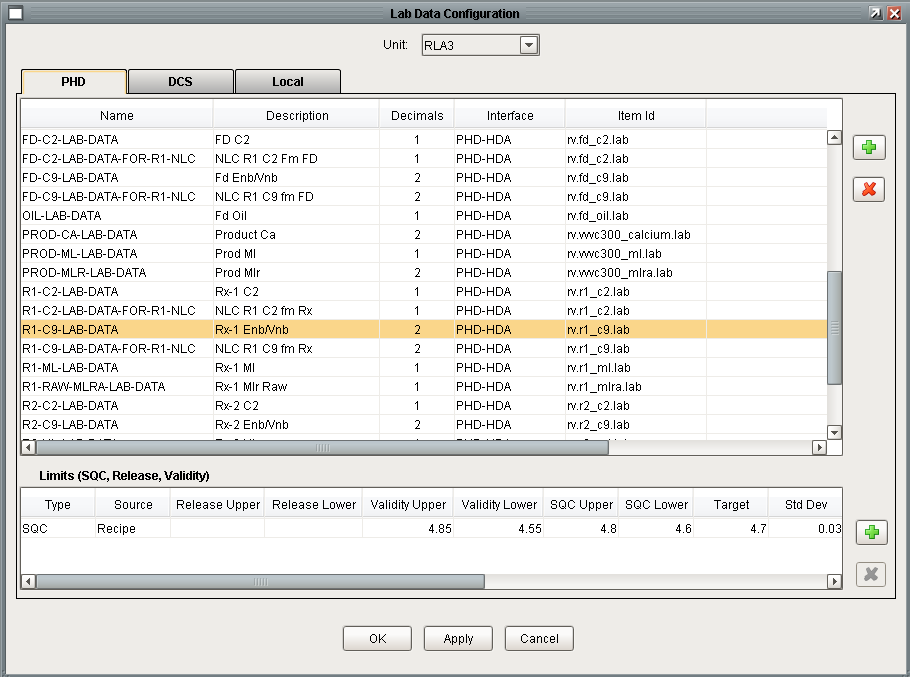


Figure 25 - Lab Data Configuration User Interface

### Configure Derived Lab Data

This screen is used to configure derived lab data including the trigger value, related data that is used in the calculation, the destination and the OPC tag where the result will be written.

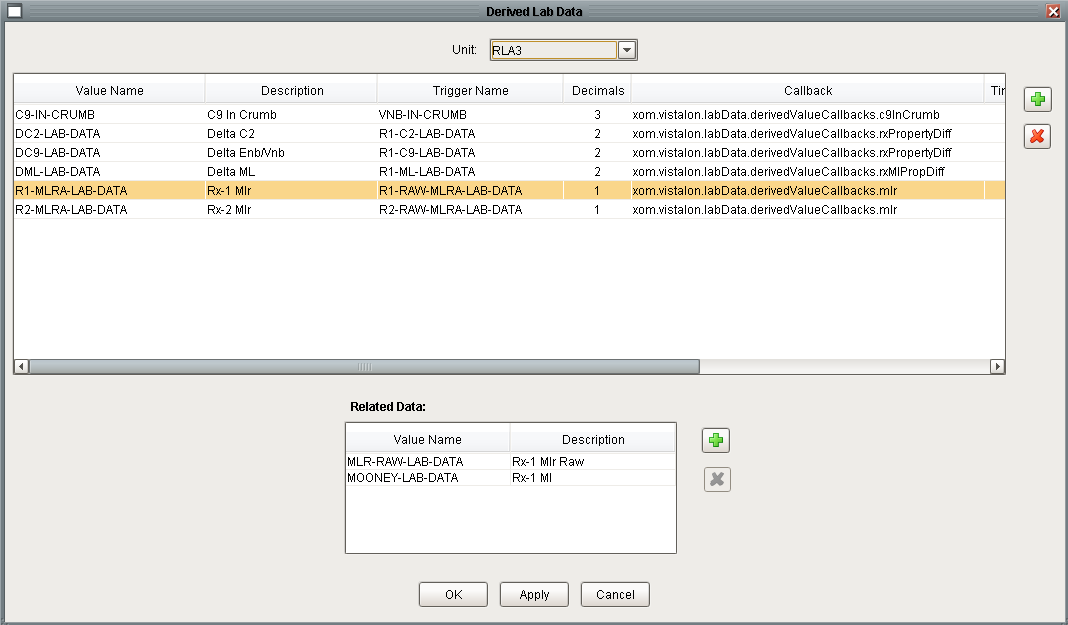


Figure 26 - Derived Lab Data Configuration

### Configure Selector Lab Data

This screen is used to configure selectors. The logic that determines the source for the selector is dynamic and outside the scope of the screen.

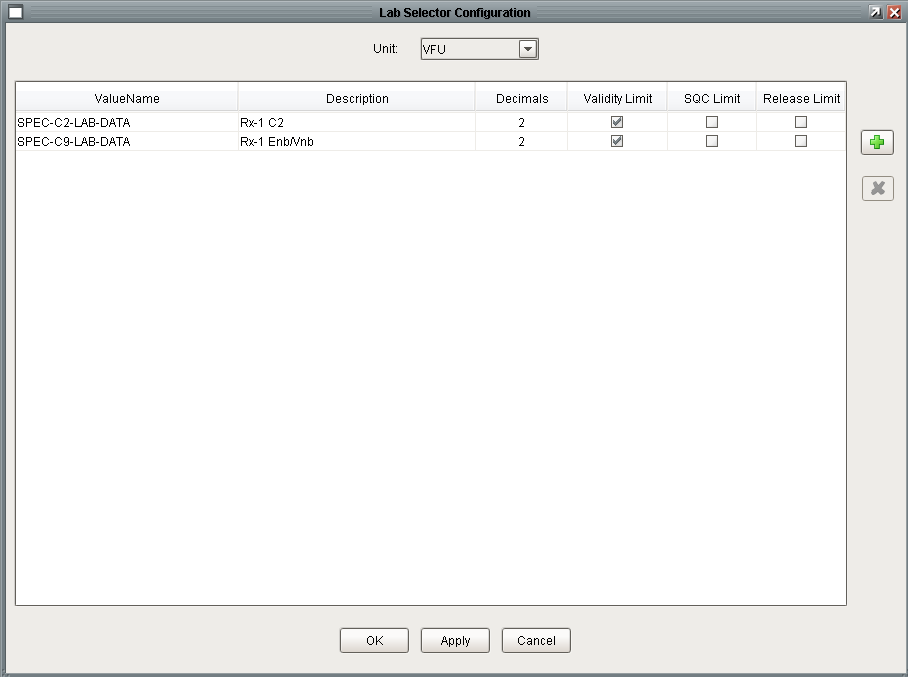


Figure 27 - Lab Data Selector Configuration

### Configuring Lab Data Display Tables

This screen is used to configure the operator user interface for displaying lab data (see the screen with the yellow background in Figure 22). The purpose of the display tables is to display lab data in logical groups and to only expose data that is useful to the operator.

The top table displays all of the screens for the post selected in the dropdown. There is no limit to the number of screens that can be defined. New screens are created by pressing the green plus button. The order that screens will be displayed on the user’s menu can be controlled by selecting a screen and using the up or down arrows. A screen is deleted using the red x. The bottom table shows the lab values that will be displayed on the screen selected in the top table. An individual lab value may not be displayed on more than one screen. Only existing lab data may be added to a screen, new lab data cannot be created from this screen.

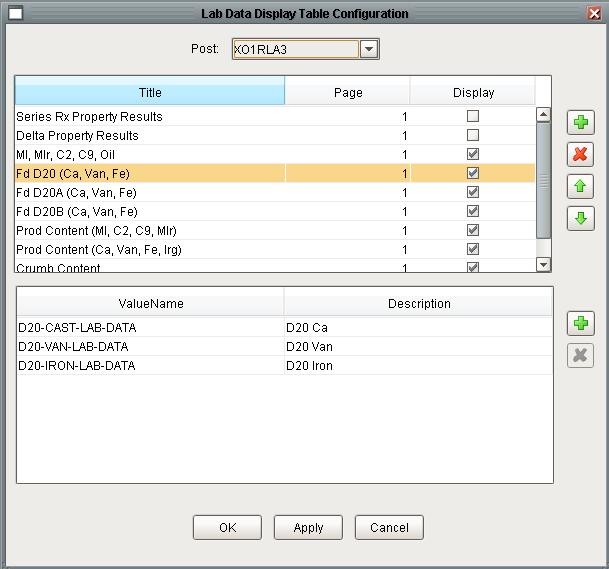


Figure 28 - Lab Data Display Table Configuration

# Troubleshooting Tips

The new toolkit is significantly different than the old toolkit and the troubleshooting strategy is also quite different.

## OPCHDA Status

The status of the OPC-HAD interface can be monitored from the gateway web page using the screen shown in Figure 5.

## Message Queue

The old application had two message queues associated with lab data. Their titles were “Lab Event Messages” and “Lab Data Debug Messages”. The former was used for logging high level lab data events. The latter was used for logging debug messages about detailed lab data performance.

The new platform will implement a single message queue with key “LABDATA”, which is titled “Lab Data Messages”. The debug functionality will utilize the loggers described in the next section.

## Loggers

Loggers are used to provide detailed logging information. Loggers are most useful for activity that executes in the gateway. Therefore the logger levels need to be set on the gateway and viewed either in the wrapper logfile or on the gateway console page. The following loggers are provided:

* com.ils.labData
* com.ils.labData.selector
* com.ils.labData.SQL

# Lab Feedback Control – Bias Calculations

## Theory

Bias calculations compares two values to create a bias to adjust a model based upon on a value that is assumed measured accurately. For example, a model calculates a value that is infrequently measured. A bias adjusts the calculated model value based upon the measured value. A simple bias is given by:

Bias = Lab Value – Model Value

The calculated value then becomes:

Calculated Lab Value = Model Value + Bias

There are different approaches to updating the bias. The above model is simply based upon the last measured lab value. This approach may cause issues if lab value changes dramatically from the previous update. In addition, statistically this may not be the proper approach. Another approach is to apply an exponential filter to the bias calculation.

The exponential approach is:

Raw Bias = Lab Value – Model Value

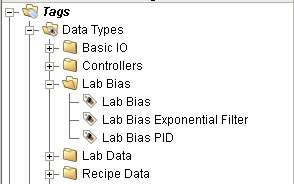
Bias = alpha \* Raw Bias + ( 1 – alpha) \* Bias

## Ignition Implementation

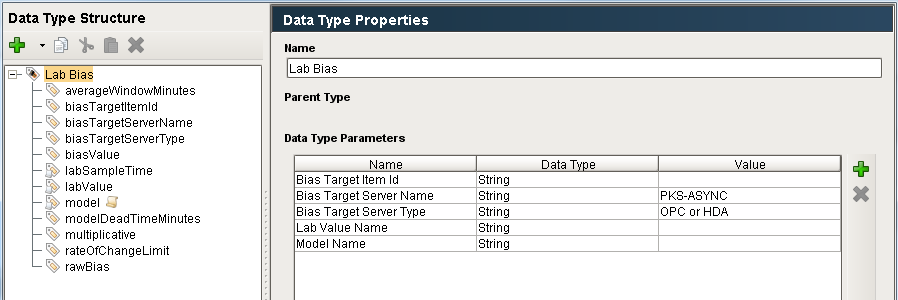
The Ignition implementation supports exponential filter and PID bias strategies. The Arima bias method is no longer supported.

### UDTs

There are three UDTs that implement the bias calculations. The three UDTS are shown below.



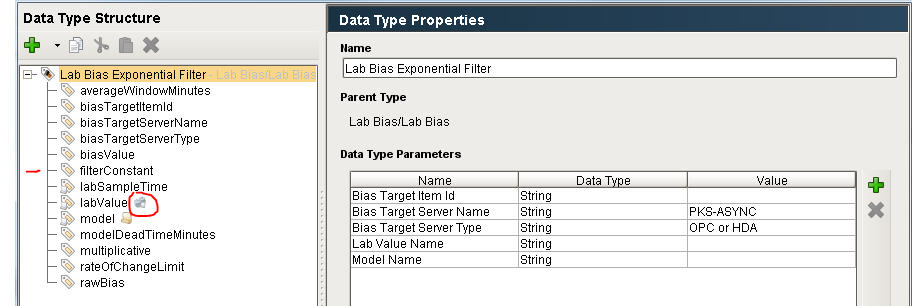
Lab Bias is the base definition that should not be instantiated. It defines properties that are common to both bias calculation strategies. Its definition is shown below:



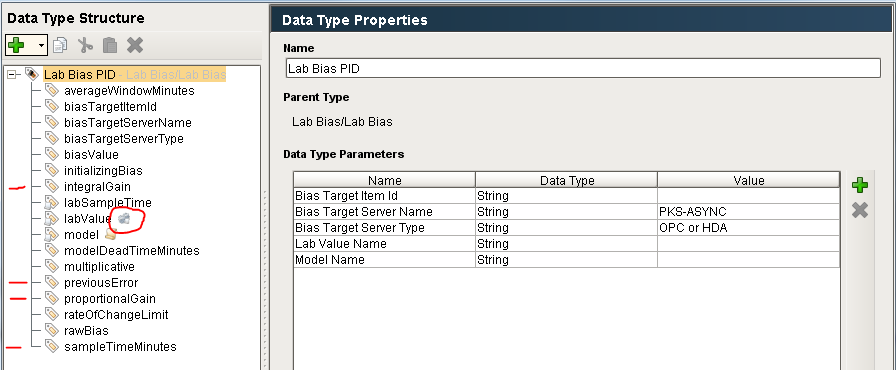
The base UDT defines the following properties which are used in the implementation of the bias.

| Parameter | Description |
| --- | --- |
| Bias Target Item Id | This specifies the OPC item-id where the calculated bias will be written. This is linked to the tag biasTargetItemId (1) |
| Bias Target Server Name | This specifies the OPC server where the calculated bias will be written. This is linked to the tag biasTargetServerName (1) |
| Bias Target Server Type | This specifies the type of the OPC server where the calculated bias will be written. The legal values are OPC or HAD. This is linked to the tag biasTargetServerType (1) |
| Lab Value Name | This specifies the lab data value that is the driving source of the bias calculation. This is linked to the labValue and labSampleTime tags of the UDT. |
| Model Name | This specifies the tag that provides the model value that will be used in the bias calculation. |

The exponential filter bias adds a filter constant tag; a script on the lab value tag, but no additional properties:



The PID bias UDT adds tags *integralGain*, *proportionalGain*, and *sampleTimeMinutes*. These are constants. It also adds a script on the *labValue* tag.



### External Python

The Python that implements the Lab Feedback handling is in the ils.labFeedback package. The logic is initiated when the labValue tag receives a new value. The logic executes in the gateway.

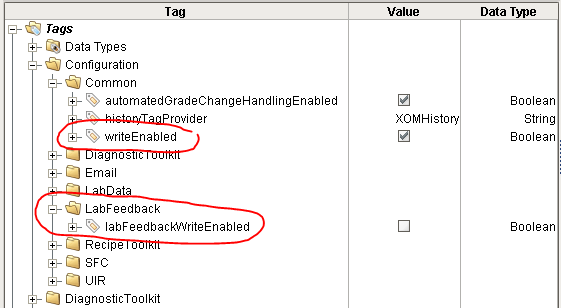
### Initialization

The initialization of lab bias is specified in the flowchart below. It is implemented in *ils.labFeedback.startup.gateway* which is called by the site specific gateway start-up script (for Vistalon this is *xom.vistalon.startup.gateway()*).

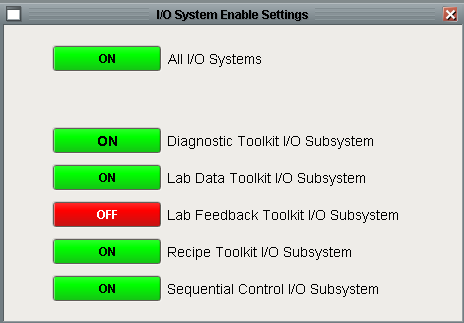


### Inhibiting I/O

The lab bias calculations run automatically once the UDTs are configured. The calculation is triggered when a new lab value is received. If a new bias value is calculated it is immediately written to the external system as long as writing I/O has not been inhibited. I/O must be enabled globally and for the Lab Feedback toolkit. This is implemented using configuration memory tags as shown below:

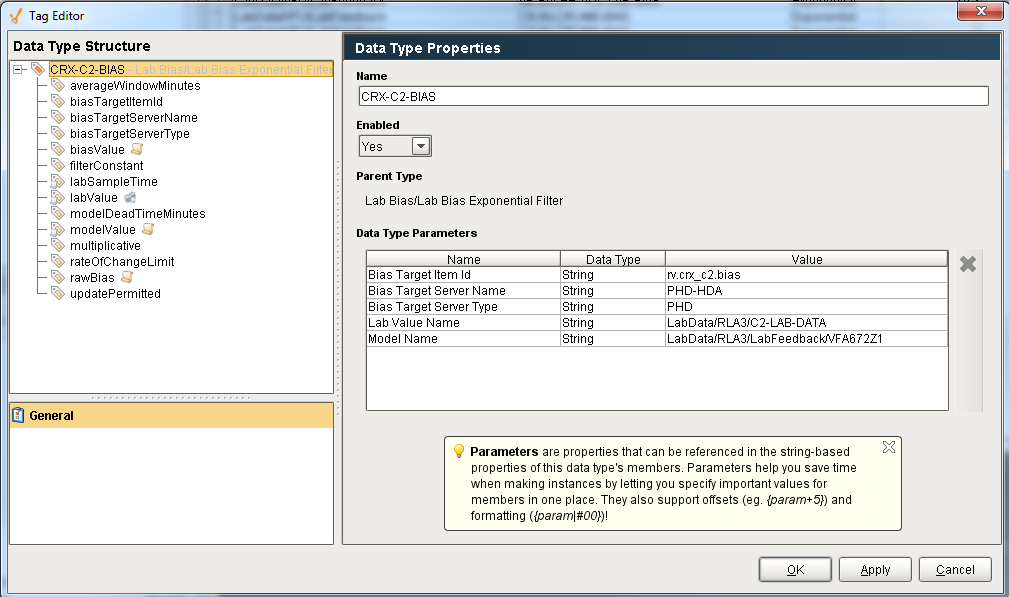


The I/O can also be inhibited in a client from the Admin menu as shown below:



### Configuration

Configuring a new lab bias object is done manually using the Ignition Designer. There is nothing in the database related to the lab bias feedback tags. A new lab bias is created by using the Tag Browser and creating a new tag instance of one of the two UDTs: *Lab Bias Exponential Filter* or *Lab Bias PID*. The instance may be placed anywhere in the *LabData* hierarchy of tags. To configure the new instance first configure the properties by clicking on the new UDT and selecting “Edit Tag”. Next, expand the UDT and configure the memory tags that do not shadow the UDT properties. The tags that need to be configured are: *averageWindowMinutes, filterConstant, modelDeadTimeMinutes, multiplicative, rateOfChangeLimit*.



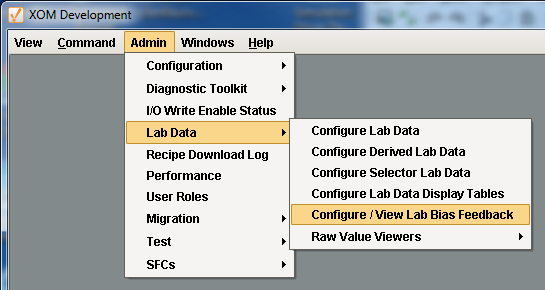
Note: The reason that there are memory tags that shadow the UDT properties is that properties are not accessible from scripting; therefore a memory tag is created that references the property. The reason that some configuration is done through the properties and others directly through the tags is that numeric tags cannot reference UDT properties.

### Migration

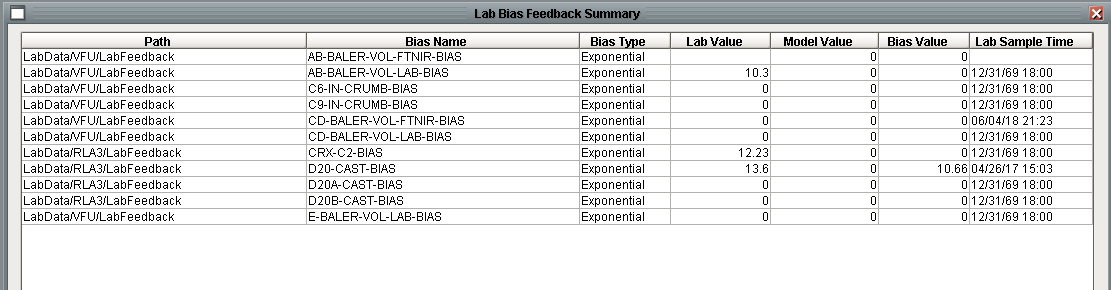
The migration of lab bias is semi-automatic. There is an export function in G2 that exports all lab bias objects and their configuration to an XML file. The data in this file is used to manually configure the new application.

### User Interface

There is a screen for viewing the current lab bias status. It is accessed from the main menu as shown below.



The screen shows every lab bias UDT. This screen is built dynamically by browsing for any of the lab feedback UDT. The tag browse system call is slow when searching over all tags; therefore it is constrained to only look for these UDTs in the LabData folder.



### Troubleshooting

The first step to troubleshooting is to observe the behaviour of the UDT in the Designer. All of the processing occurs in gateway scope. Detailed diagnostics can be obtained by putting the *com.ils.labFeedback* logger into trace mode. This will turn on detailed logging information that is written to the *wrapper.log* file. The bias calculation is initiated from the *ValueChanged* tag event script on the *labValue* tag.

## Obsolete

Initially at start-up Bias = Raw Bias and alpha between 0.001 and 0.99, smaller alpha more filtering.

There are other models to update the bias such as ARIMA that will not be considered.

The diagram functions automatically once the objects are correctly configured. The bias value is automatically calculated whenever the lab value receives a value. The OPC input on the left, which could be any subclass of a quantitative-variable, provides the model value for the corresponding lab value. The raw bias is calculated by the difference between the lab value and the model value.

The class hierarchy of **Bias-Value** is shown below. The class hierarchy uses abstract classes and multiple inheritance which makes the class hierarchy appear more complex than it really is. There are three types of bias calculations, **Bias-PID, Bias-EXP-Filt**, and **Bias-ARIMA**. Then there are two types of external systems: DCS and PHD.

The instantiable classes are:

|  |  |  |  |
| --- | --- | --- | --- |
|  | DCS | PHD | PHD-Hist |
| EXP-Filt | Bias-DCS-Exp-Filt | Bias-PHD-Exp-Filt | Bias-PHD-Hist-Exp-Filt |
| PID | Bias-DCS-PID | Bias-PHD-PID |  |
| ARIMA | Bias-DCS-ARIMA, Bias-DCS-ARIMA-Init, Bias-DCS-ARIMA-Init-Text | Bias-PHD-ARIMA, Bias-PHD-ARIMA-Init, Bias-PHD-ARIMA-Init-Text |  |

The class initially of interest is **Bias-PHD-Hist-Exp-Filt**. Basically this class updates the bias of based upon lab data sources from the PhD historian. The Ignition implementation will simplify the final hierarchy but will be based upon the base **Bias-Value** and **Bias-Exp-Filt** class. The bias-value attributes are described below.

Bias-Value Attributes – These attributes are inherited by all bias subclasses

|  |  |
| --- | --- |
| Attribute | Description |
| update-permitted | This attribute is set by a custom user-defined policy, either programmatically triggered by some event or a scanned rule, or manually by some user gesture. Bias will only be calculated when this attribute is true. |
| average-window | The time window during which the average of the model value will be calculated. |
| mdl-deadtime | The model deadtime - it should have correct time units appended to the value when stored (i.e., 6 seconds or 35 minutes). |
| mdl-deadtime-proc | A procedure for calculating model deadtime. A unique procedure can be specified for each instance. |
| multiplicative-bias | User configured attribute for determining how the raw-bias is calculated. If true then the raw-bias = the lab value / the model value. If false then raw-bias = lab-val - model-val. |
| initializing-bias | User specified initial value to be used for the bias when initializing the bias value which happens on a cold start or whenever a new value is entered. |
| update-bias-val-on-init | User configured attribute that when true specifies that the bias-value will be set to the initializing-bias value when the initialization method is called. If false, then the bias-value object will not be updated but the initializing-bias will be written to the external system. |
| raw-bias | The currently calculated linear bias without limit checks. Raw-Bias = the Lab Value - the model value. |
| rate-of-chg-lim | Constant used to define the maximum change in bias between successive calculations. Changes in bias that are greater than this limit will be flagged as suspicious. |

Bias-Exp-Filt Attributes – These attributes are inherited by the exponential filter subclasses

|  |  |
| --- | --- |
| Attribute | Description |
| exp-filter-constant | Constant used in the exponential filter calculation. |

Bias-PID Attributes – These attributes are inherited by the PID subclasses

|  |  |
| --- | --- |
| Attribute | Description |
| prop-gain | Constant used in the PID calculation. |
| int-gain | Constant used in the PID calculation. |
| sample-time | Constant used in the PID calculation. |
| previous-error | Used for saving the raw bias from the last calculation. Used to calculate the change in raw bias from the last calculation to the current calculation. |

Bias-ARIMA Attributes – These attributes are inherited by the ARIMA subclasses.

|  |  |
| --- | --- |
| Attribute | Description |
| filtered-bias-rate-of-chg-lim | Not used. |
| ar-parameter | Factor, between 0.0 and 1.0, used in the ARIMA filter calculation. If bias-model-adaptation is true, then this is recalculated each cycle. |
| weighting-on-old-mean-bias | Weighting for the exponential filter when initially calculating the mean-bias. |
| mean-bias | Bias using an exponential filter. |
| bias-model-adaptation | If true then the factors used to calculate the ARIMA filter are dynamically adjusted. |
| bias-model-adaptation-weighting | Weighting factor that gets recalcuated each time time the bias is calculated. |
| bias-model-ar-parameter-covariance | Factor used in the adaptation of the ARIMA model if Bias-Model-Adaptation is true. |
| bias-model-adaptation-forgetting-factor | Constant used in ARIMA calculation. |

**Bias-PHD-Hist-Exp-Filt** class inherits from both **Bias-Exp-Filt** and **opc-hist-float-output**. There are a few methods that reference that class **Bias-PHD-Hist** by name that need to be considered as the original hierarchy **Bias-PHD-Hist-Exp-Filt** subclassed from **Bias-PHD-Hist. Bias-PHD-Hist** is not required.

The exponential filter is fairly straightforward except that the class **opc-hist-float-output** supports a write-datum method to write data back to PHD server.

The procedures/methods and functions that need to be reproduced are:

bias-value::bias-update-private

bias-exp-filt::bias-update-private

bias-value::bias-initialization-private

bias-value::bias-control-mode-check-private

calc-exp-filter function

update-bias rule and initialize bias rule

# Legacy System Information

This section contains screen shots of pertinent screens in the old system. The figures are contained in an appendix so that once the old system is decommissioned these can easily be dropped from the specification.

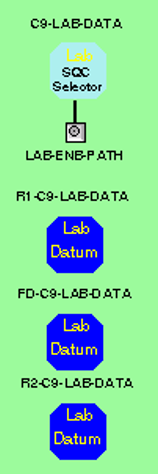


Figure 28 - Legacy Lab Data Selectors

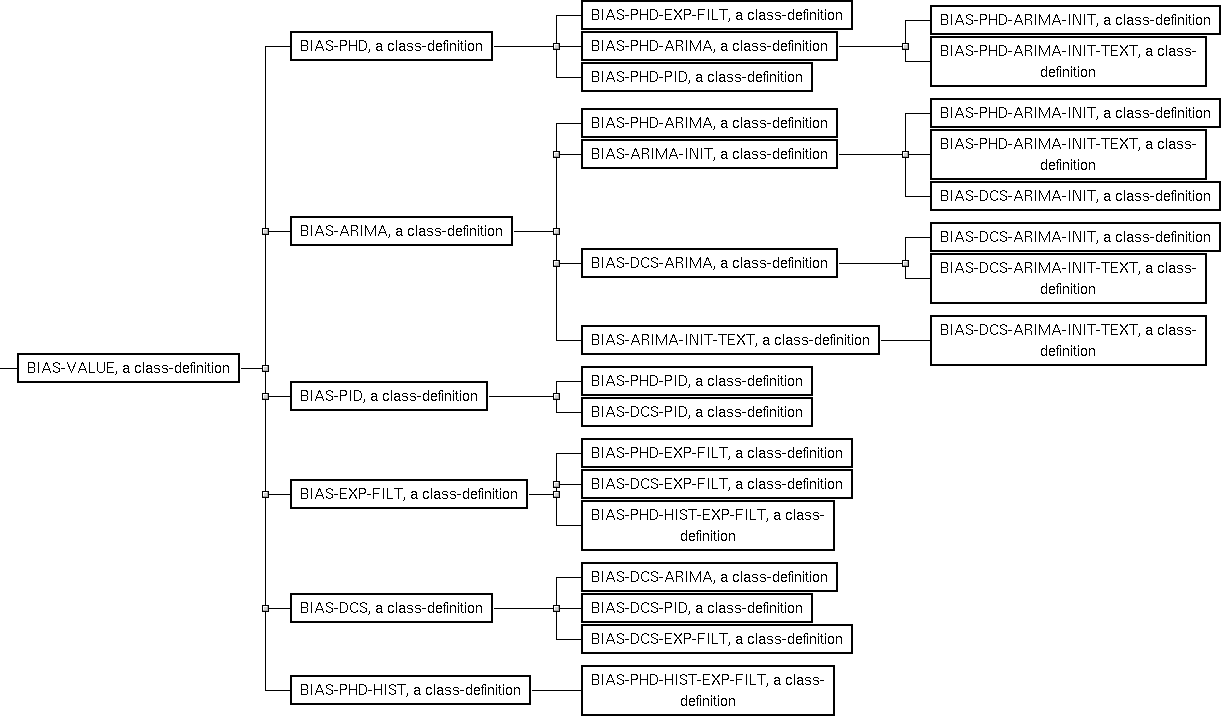


Figure 29 - Bias Value Class Hierarchy



Figure 30 - Typical Lab Bias Configuration

# Meeting Notes

