

ExxonMobil Chemical Company

Common Facilities

User's Guide

Version 1.4

January 20, 2021

Prepared by:

ILS Automation Inc.

TABLE OF CONTENTS

1	<i>Introduction</i>	4
2	<i>Window Layers</i>	4
3	<i>Consoles</i>	5
4	<i>Message Queues</i>	7
4.1	Database Tables	8
4.2	Scripting Interface	8
4.3	User Interface	9
4.4	Creating a new Queue	12
5	<i>Operator Logbook</i>	13
5.1	Database Tables	13
5.2	Scripting Interface	15
5.3	Client User Interface	15
5.4	Report Configuration	16
5.4.1	Archiving and E-Mailing Daily reports	16
5.5	Usage Conventions	18
5.6	Creating a new Logbook	19
6	<i>Operator Console Alert</i>	20
6.1	Scripting Interface	22
6.2	User Interface	23
6.3	Usage Conventions	24
6.4	Custom Callback	24
7	<i>Engineering Units</i>	25
7.1	Steps to Create a New Type of Recipe Data	26
8	<i>Grade UDT and Grade Handling</i>	27
9	<i>Batch Tracking</i>	27
10	<i>Message-Reply Utility</i>	28
10.1	Implementation	28
10.2	Scripting Interface	29
11	<i>Watchdog Utility</i>	30
11.1	Watchdog Theory of Operation	30
11.1.1	OPC Read Watchdog	30
11.1.2	OPC Write Watchdog	30

11.2	Watchdog Configuration	31
11.2.1	OPC Read Watchdog Configuration	32
11.2.2	Monitor Strategies	33
11.3	Watchdog Alarm Configuration	34
11.4	Watchdog Alarm Pipeline	35
11.5	Watchdog Alarm User Interface	36
11.6	Watchdog Evaluation	38
12	<i>Security / Logon Facilities</i>	38
12.1	User Mode / User Group	38
12.2	Single Sign-On	38
12.3	Console / Post Determination	39
13	<i>Miscellaneous Features</i>	39
13.1	Warm Boot Support	39
13.2	Ignition Uptime Tag	40
13.3	Start-up Considerations	41
13.4	XY Chart Customizations	41
13.5	Tag Replication	42
13.6	Data Pump	45
14	<i>Developing on the Production System</i>	46
14.1	Tags	46
14.2	External Python	46
14.3	Client Event Scripts	46
14.4	Project Resources (Windows, Diagnostic Diagrams, Templates, Reports)	46
14.5	Global Resources (Alarm Pipelines, SFCs, Shared Scripts)	46

1 Introduction

This document describes the common facilities that are part of the Ignition based automation platform. This document is intended for site engineers responsible for developing, supporting, and extending applications built using the toolkits. It is not a User's Guide intended for console operators. The facilities described here are part of the system developed by ILS Automation, they are not provided by the Ignition software.

Some of the overarching design principals used in the design of the platform are:

- A single Ignition project is used for all of the Baton Rouge sites. The project contains both common and site specific window definitions. This project will be installed on each sites Ignition gateway. Over time, site specific screens and logic may be added to each site's project in such a way that it is isolated from the common core. Modifications to common screens and logic must be standardized across sites.
- A common database schema is used for all Baton Rouge sites but each site configures their database with site specific data.
- All of the sites share a set of User Defined Types (UDTs) which are tag templates for defining things like lab data or PKS controllers.
- Each site has a set of site specific tags.

2 Window Layers

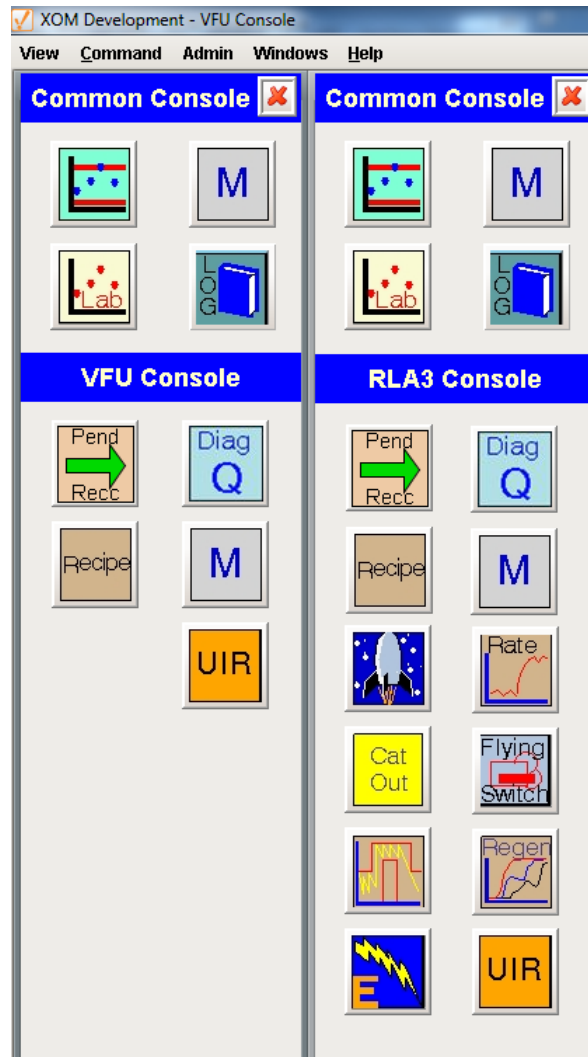
Ignition uses layers to determine the visibility of overlapping windows. Oftentimes the simple rule that the last window opened should be on top is the best policy but this only works if all windows are on the same layer. The higher the layer number the higher the importance of a window. A lower layer window can never cover a window with a higher priority. The default layer for windows is a 0. Ignition does not provide the concept of a modal window, this can be approximated by assigning a high layer number to the window. The following table describes the scheme used throughout the toolkits.

Layer	Description
0	All windows, help screens
1	Not used at this time
2	Choosers (Lab Data, SQC), Setpoint spreadsheet
3	Not used at this time
4	Warnings, OC alert (loud workspace)

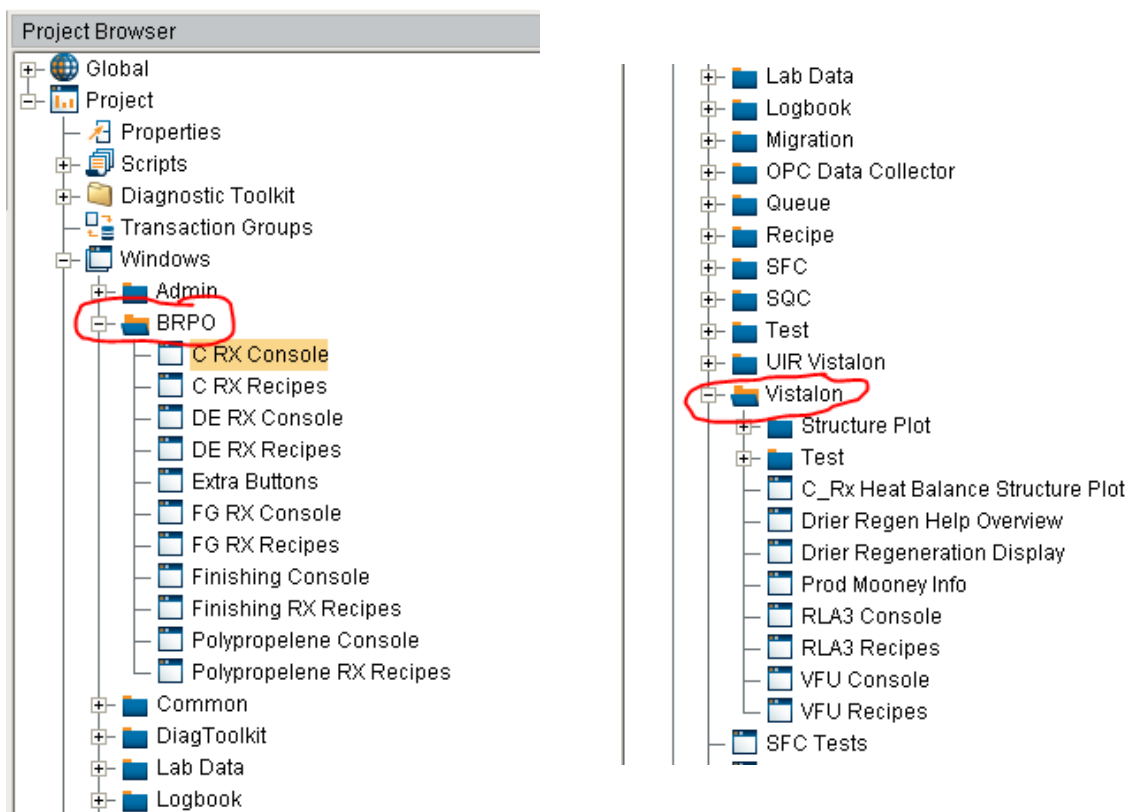
3 Consoles

The console operator accesses various aspects of the system from one or more console windows. Whereas the previous toolkit has a separate workspace for the common console and for the actual console, the new toolkit uses a template for the common console and reuses it on each console window. Custom console windows are designed for each site. Console windows are “docked” so that other windows cannot cover them.

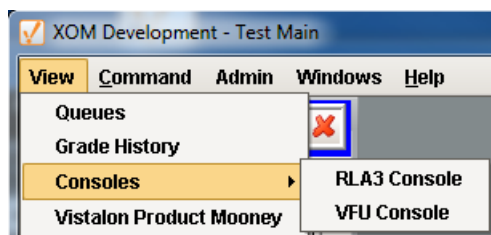
For example, Vistalon has two consoles: VFU and RLA3, which are shown below:



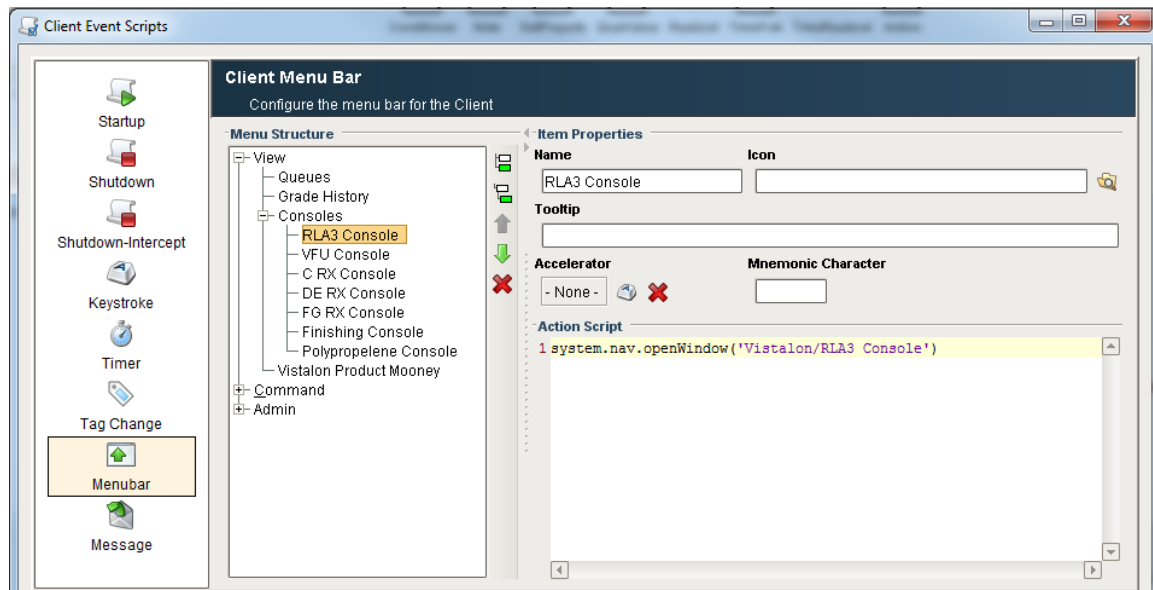
In Designer, the Project Browser shows how site specific windows are organized. Each site will have a folder containing the windows for that site. As mentioned earlier, a single Ignition project will be used by all sites.



Console windows are accessed from the main menu:



The menu is a project resource in client scripts and is configured as shown below. In order to provide a single project to all of the Baton Rouge sites, the menu contains all of the consoles for all of the sites. The action script for each of the console menu choices specifies the path to the console window as shown above in the designer view.



The extra choices are removed by the client start up script by querying the TkConsole table in the (site specific) database, which contains a list of consoles for the site. Note: The approach of configuring the entire set of consoles and removing unwanted ones was taken because there isn't a way to programmatically add and configure menus.

To add a new console for a site:

- 1) Create a new window.
- 2) Add a new choice to the consoles menu.
- 3) Add a new record to the TkConsole table.

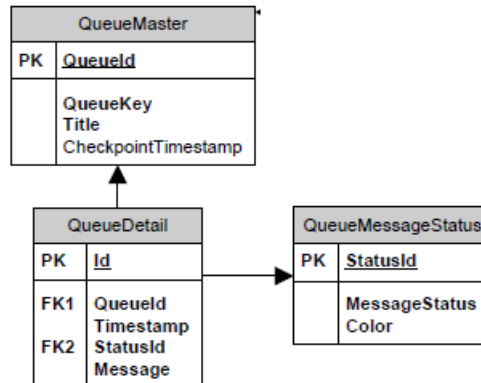
4 Message Queues

Message Queues are a shared facility for the various higher level toolkits such as Lab Data, Sequential Control, and the Diagnostic Toolkit. Message queues are used for troubleshooting and provides a simple mechanism for operator interactions. A control room environment typically has long periods of inactivity punctuated by periods of high activity. The operator or engineer may have trouble keeping up with high activity periods. The message queue system provides a utility for storing and reviewing messages in the sequence that they were posted.

The message queue utility is comprised of database tables, Python scripts, and a Vision user interface.

4.1 Database Tables

Message queues are implemented in a SQLServer database. One of the important advantages of using SQLServer is that the messages are permanent. The Message Queue utility uses three database tables: QueueMaster, QueueDetail, and QueueMessageStatus; the entity relationship diagram is shown below. Refer to the *EMC Database Design* specification for details.



4.2 Scripting Interface

There are three scripting interfaces provided which are shown below. They are all in *ils.queue.message*. The first interface, *insert*, inserts the message into the queue specified by the *queueKey*. The second interface, *insertPostMessage*, inserts the message into the queue that is designated for the specified *post*. Both interfaces require a status, which is used to color code the messages. While the queue status values are configurable, the default values are *Info*, *Warning*, and *Error*. The last interface, *clear*, sets the checkpoint timestamp of the queue. This simulates clearing the contents of a queue, but rather than deleting the records, the checkpoint is set and then the view will only display newer messages.

```

# Expected status are Info, Warning, or Error
def insert(queueKey, status, message, db = ''):
    from ils.queue.common import getQueueId
    queueId = getQueueId(queueKey, db)

    _insert(queueId, status, message, db)

def insertPostMessage(post, status, message, db=''):
    from ils.queue.common import getQueueForPost
    queueKey=getQueueForPost(post)

    insert(queueKey, status, message, db)

def clear(queueKey, db = ''):
    from ils.queue.common import getQueueId
    queueId = getQueueId(queueKey, db)

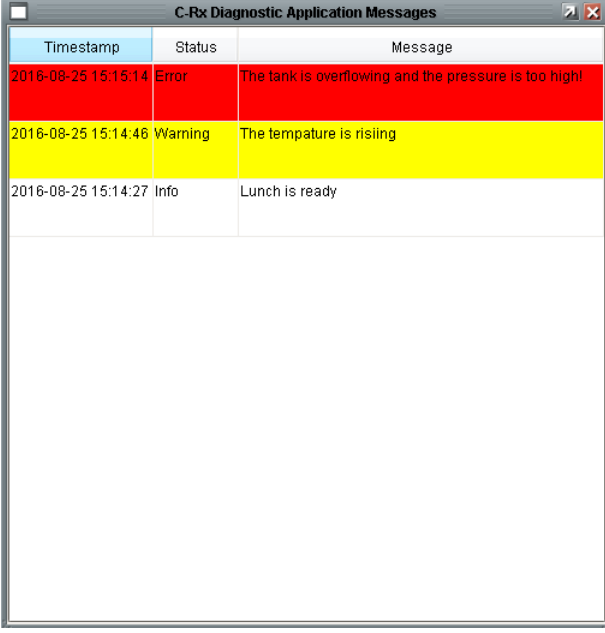
    SQL = "update QueueMaster set CheckpointTimestamp = getdate() where QueueId = %i" % (queueId)

    system.db.runUpdateQuery(SQL, db)

```

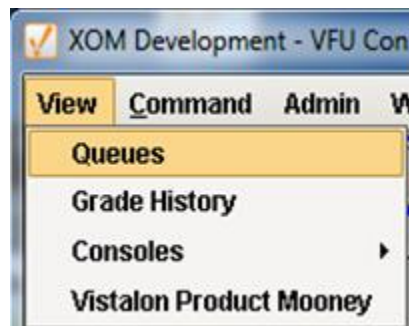

4.3 User Interface

There are a number of windows for using and managing message queues. The basic window for viewing the contents of a queue, which is accessed a number of ways, is shown below. The window shows all of the messages in a single queue since the last checkpoint with the newest message at the top. The messages are color coded based on their status.

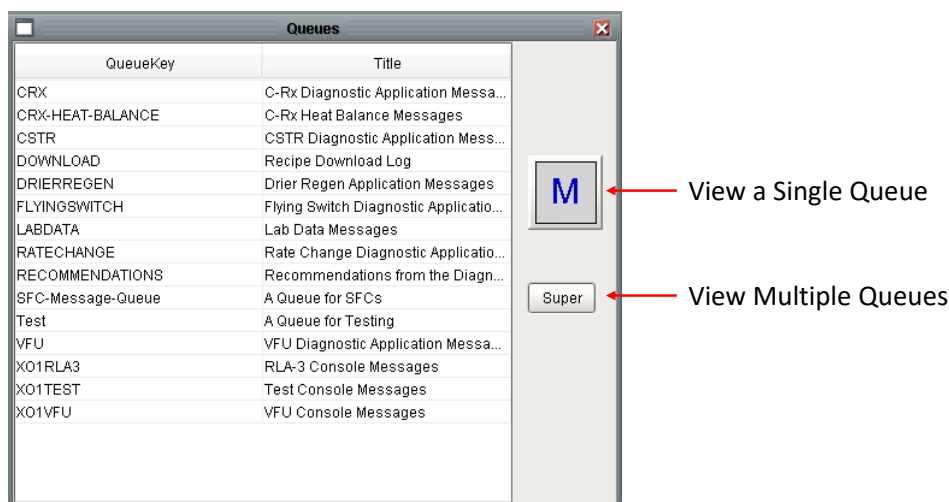


Timestamp	Status	Message
2016-08-25 15:15:14	Error	The tank is overflowing and the pressure is too high!
2016-08-25 15:14:46	Warning	The temperature is rising
2016-08-25 15:14:27	Info	Lunch is ready

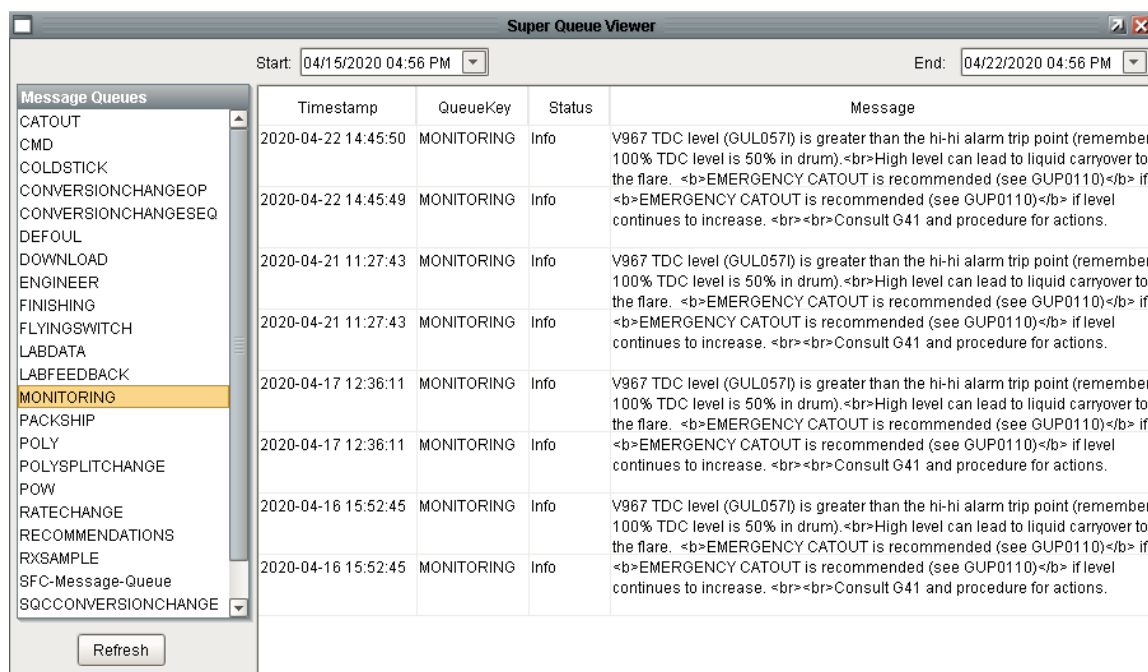
From the main menu, the View -> Queues menu is used to select a queue to view.



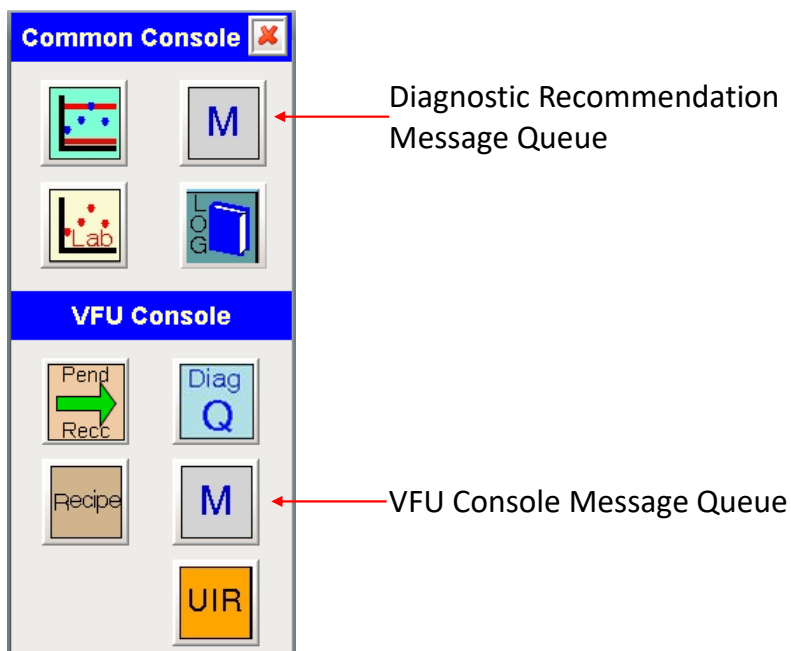
It displays a window from which the contents of any queue can be viewed:



The Queue Viewer button brings up the window shown at the beginning of this section. The “Super” button brings up the window shown below. It allows you to quickly change from one queue to the other. It also allows you to set specific start and end times without regard to checkpoints. This may be useful if you know the time of an event and want to see what was happening in several of the toolkits around that time. The data shown in this view is exactly the same as in the other view because all of the queue messages are stored in the SQL*Server database.

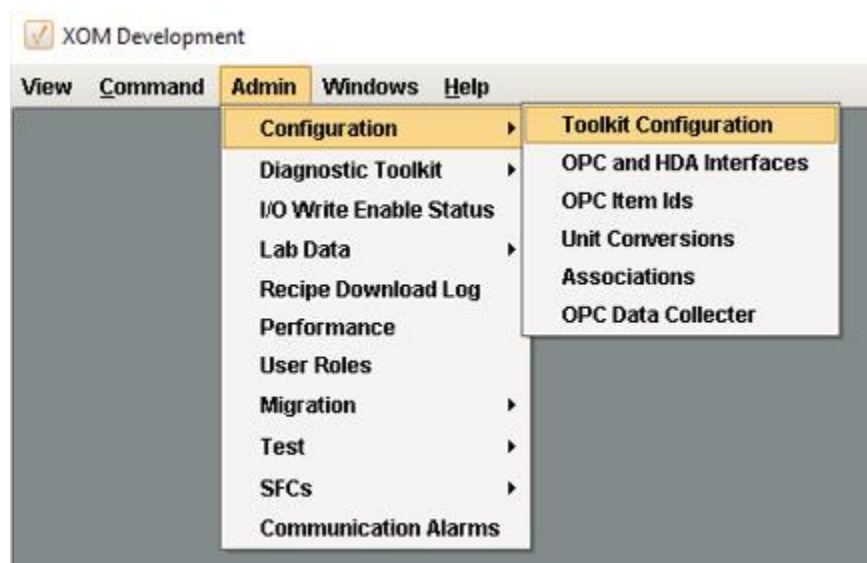


Additionally, buttons are provided on the console window for quick single click access to specific queues. The Show Queue button is widely used on other screens to make the relevant queue easily accessible.



4.4 Creating a new Queue

Queues are managed by selecting the *Configuration -> Toolkit Configuration* choice in the *Admin* menu.

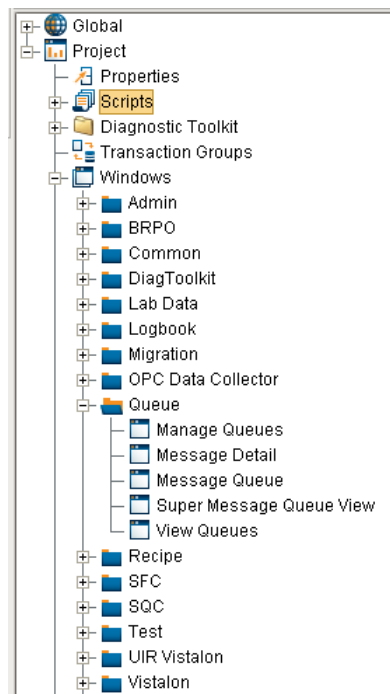


This opens a window tabs for configuring various aspects of the platform. The first tab is for Queues.

The screenshot shows the 'Toolkit Configuration' window with the 'Queues' tab selected. The window contains a table with columns: Key, Title, Checkpoint, Position, Threshold, Admin, AE, and Operator. The table lists various system messages and their configurations.

Key	Title	Checkpoint	Position	Threshold	Admin	AE	Operator
CATOUT	Cat Out Sequence Messages	04/26/19 2...	bottomLeft	1.5	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
COLDSTICK	Cold-Stick Sequence Messages	02/09/18 0...	bottomCen...	2.5	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
CRX	C-Rx Diagnostic Application Messages		center	10	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
CRX-HEAT-BALANCE	C-Rx Heat Balance Messages		center	10	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
CSTR	CSTR Diagnostic Application Messages		center	10	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
DOWNLOAD	Recipe Download LogX		bottomLeft	2.5	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
DRIERREGEN	Drier Regen Application Messages	09/05/17 2...	center	1.5	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
LABDATA	Lab Data Messages		center	10	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
LABFEEDBACK	Lab Feedback Messages		center	10	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
RATECHANGE	Rate Change Sequence Messages	12/13/17 1...	center	1.5	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
RECOMMENDATIONS	Recommendations from the Diagnostic Toolkit	08/19/19 2...	center	10	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
SFC-Message-Queue	A Queue for SFCs	03/09/20 1...	center	1.5	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
STRUCTURE	Winapp Messages for Structure Plot		center	10	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Test	A Queue for Testing	08/20/19 1...	topLeft	2.5	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
VFU	VFU Diagnostic Application Messages		topRight	2.5	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
XO1RLA3	RLA-3 Console Messages		center	2.5	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
XO1TEST	Test Console Messages	12/18/19 1...	topLeft	1.5	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
XO1VFU	VFU Console Messages		center	2.5	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

The queue windows are defined as shown below:

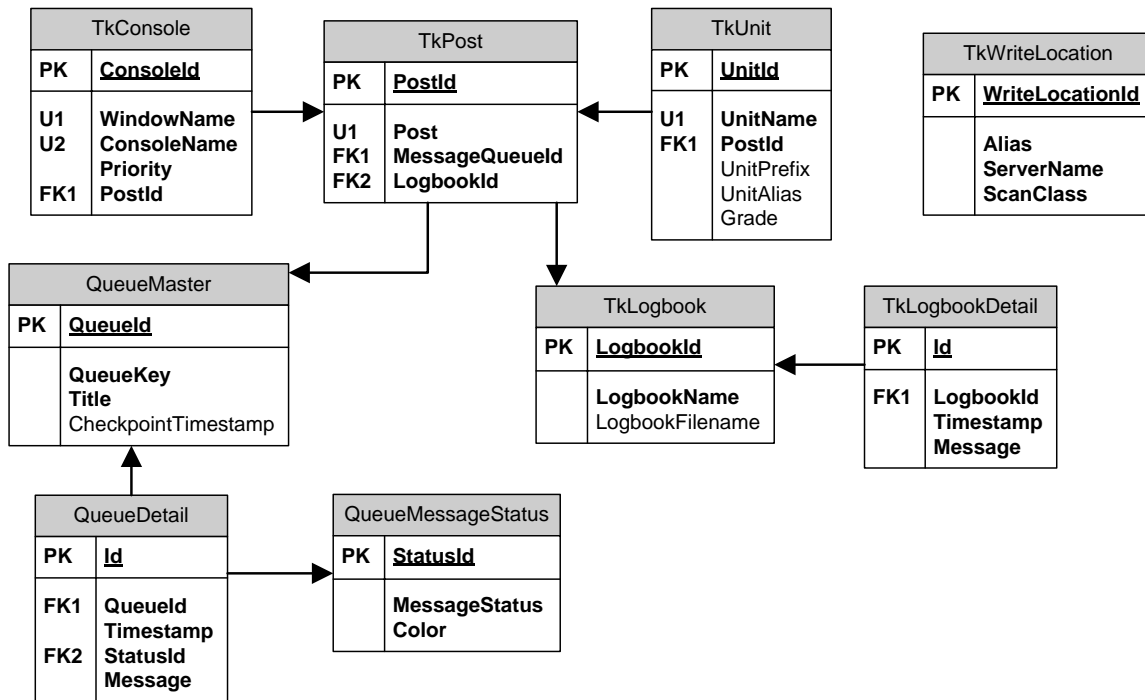


5 Operator Logbook

The Operator Logbook facility aims to fulfill the goal of eliminating paper logbooks by providing an electronic logbook. The intent is to provide logging functionality of operator-initiated changes or operator-approved changes through the various toolkit applications. The provides a means for recording automated and ad hoc comments to a logbook, archiving logbooks to the filesystem, browsing logbooks from a client, and emailing logbooks to concerned parties

5.1 Database Tables

The master copy of the logbook is maintained in the SQLServer database. The database provides reliable permanent storage for the logbooks. (Daily reports of logbook activity may also be written to the file system, but the master data is in the database). The Entity Relation diagram for the logbook is shown below:



The tables of interest are TkLogbook, which defines a logbook, TkLogbookDetail which stores the contents of a logbook, and TkPost which defines the logbook that is used by a Post.

5.2 Scripting Interface

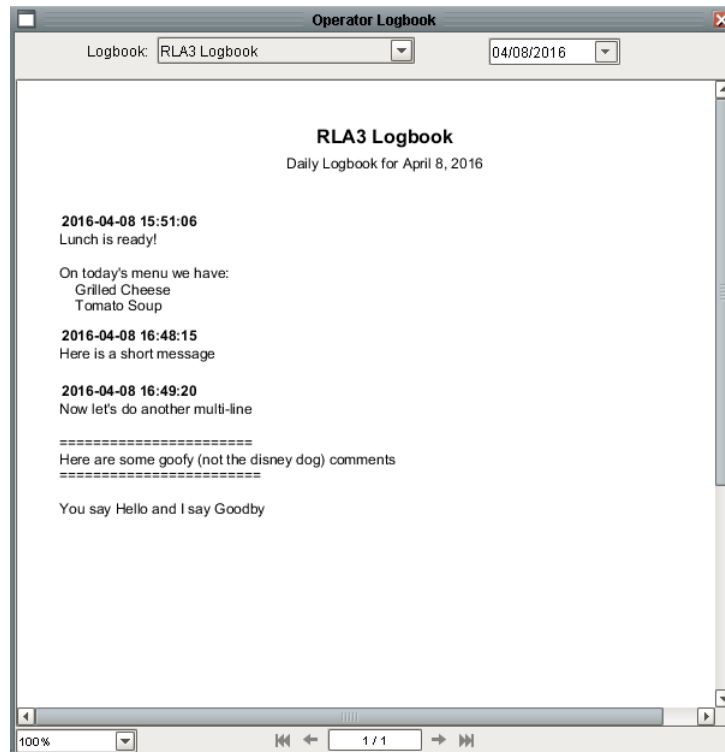
Python scripts are provided for conveniently posting messages to an operator logbook. There are two scripting interfaces provided which are shown below. Both interfaces are in *ils.common.operatorLogbook*. The third argument, database, is optional unless called in global scope (from an SFC or a tag change script). If omitted then the default database is used.

```
def insert(logbook, message, database=""):
    logbookId = getLogbookId(logbook, database)
    _insert(logbookId, message, database)

def insertForPost(post, message, database=""):
    logbookId = getLogbookIdForPost(post, database)
    _insert(logbookId, message, database)
```

5.3 Client User Interface

Operator logbooks are accessed by the Logbook icon on the common console. The button will open today's logbook for the operators post as shown below for the RLA3 post.



The window contains controls at the top for viewing other logbooks and for selecting a different day. The contents of the report change automatically when either are changed. The window has controls at the bottom for paging through multi-page logbooks and for scaling the report. Right clicking on the report displays a popup menu for saving or printing the report.

5.4 Report Configuration

The logbook viewer described above is implemented using Ignition's reporting module. The module contains facilities for saving and emailing reports on a schedule.

5.4.1 Archiving and E-Mailing Daily reports

The final step of designing a report in Ignition is to specify of schedule for actions for the report. The current system writes the daily report to a file. Note: Because the current system keeps a permanent record of the logbook in a database the need to archive should be reviewed, but for now the system will be configured to archive the daily report.

The following example is for a site with three logbooks named: "RLA3 Logbook", "VFU Logbook", and "Engineer Logbook". The first step is to specify the schedule. The daily logbook report is scheduled to run one minute before midnight.

The screenshot displays the Ignition Report Configuration window with the 'Schedule' tab selected. The window has a top navigation bar with icons for Report Overview, Data, Design, Preview, and Schedule. Below this is a table with two columns: 'Schedule' and 'Actions'. The table contains three rows, each with 'At 11:59 PM' in the 'Schedule' column and 'Save File' in the 'Actions' column. Below the table are three sub-tabs: 'Schedule', 'Parameters', and 'Actions'. The 'Schedule' sub-tab is active, showing a 'Common Settings' dropdown set to 'Custom'. Below this are five rows of time and frequency settings, each with a text input, a green checkmark, and a dropdown menu. The settings are: Minutes (59, :59 (59)), Hours (23, 11 p.m. (23)), Days (*, Every Day (*)), Months (*, Every Month (*)), and Weekdays (*, Every Day (*)). At the bottom, under the 'Options' section, there is a checkbox labeled 'Enabled' which is checked.

Schedule	Actions
At 11:59 PM	Save File
At 11:59 PM	Save File
At 11:59 PM	Save File

Schedule

Common Settings: Custom

Minutes: 59 ✓ :59 (59)

Hours: 23 ✓ 11 p.m. (23)

Days: * ✓ Every Day (*)

Months: * ✓ Every Month (*)

Weekdays: * ✓ Every Day (*)

Options

☒ Enabled

The next step is to specify the parameters. Because the daily report runs at one minute before midnight, today's date with time of 00:00:00 is the start date and 23:59:59 is the end date. The name of the logbook is the final parameter.

The screenshot shows the 'Parameters' tab of a configuration window. It contains three main sections, each with a 'Default' checkbox and a text input field with a multi-line editor icon (top-left) and a function icon (bottom-right).

- StartDate**: The text field contains `1 dateFormat(now(), "MM/dd/YYYY 00:00:00")`.
- EndDate**: The text field contains `1 dateFormat(now(), "MM/dd/YYYY 23:59:59")`.
- Logbook**: The text field contains `1 "RLA3 Logbook"`.

Below the main 'Logbook' section, there are two more nested 'Logbook' sections, each with its own 'Default' checkbox and text field:

- The first nested 'Logbook' section contains `1 "VFU Logbook"`.
- The second nested 'Logbook' section contains `1 "Engineer Logbook"`.

The final step is to specify the actions. A single action to save the report to a file has been configured. It requires the folder path and an expression for naming the file. The site has devised a naming scheme so that a year's worth of files will be kept.

The screenshot shows three instances of the 'Actions' tab, each with a 'Save File' action configured. Each instance has a 'Folder Path' field and a 'FileName' multi-line editor.

- Instance 1 (Top)**:
 - Folder Path**: `E:\Vistalon\event_logs\Operating_Logbooks\`
 - Format**: PDF
 - FileName**:


```
1 "RLA3_logbook_m"
2 + dateFormat(now(), "MM")
3 + "_d"
4 + dateFormat(now(), "dd")
```
- Instance 2 (Middle)**:
 - Folder Path**: `E:\Vistalon\event_logs\Operating_Logbooks\`
 - Format**: PDF
 - FileName**:


```
1 "VFU_logbook_m"
2 + dateFormat(now(), "MM")
3 + "_d"
4 + dateFormat(now(), "dd")
5
```
- Instance 3 (Bottom)**:
 - Folder Path**: `E:\Vistalon\event_logs\Operating_Logbooks\`
 - Format**: PDF
 - FileName**:


```
1 "Engineer_logbook_m"
2 + dateFormat(now(), "MM")
3 + "_d"
4 + dateFormat(now(), "dd")
5
```

An e-mail action can be configured which will email the report as a PDF attachment. If the e-mail distribution list is dynamic then a group defined in the e-mail server can be used rather than maintaining the distribution list in the report.

The screenshot displays the configuration interface for an email action. The top navigation bar includes 'Report Overview', 'Data', 'Design', 'Preview', and 'Schedule'. The main window is split into a top 'Schedule' section and a bottom 'Actions' section. The 'Schedule' section shows a task scheduled 'At 11:59 PM' with the action 'Save File, Email'. The 'Actions' section lists 'Save File' and 'Email', with 'Email' being the active configuration. The configuration fields for the 'Email' action are as follows:

- From Address:** phassler@ils-automation.com
- Subject:** 1 {Report.Name}
- Attachment Filename:** 1 {Report.Name}, 2 + " - ", 3 + dateFormat(now(), "M-d"), 4 + ".pdf"
- Body:** 1 "Report attached"
- Mail Server:** XOM Mail Server
- Format:** PDF
- Retries:** 0
- Recipients Source:** Email Addresses

A table of 'Recipient Emails' is also visible, listing two recipients:

Address	Method
phassler@ils-automation.com	To
michael.kurtz@exxonmobil.com	To

5.5 Usage Conventions

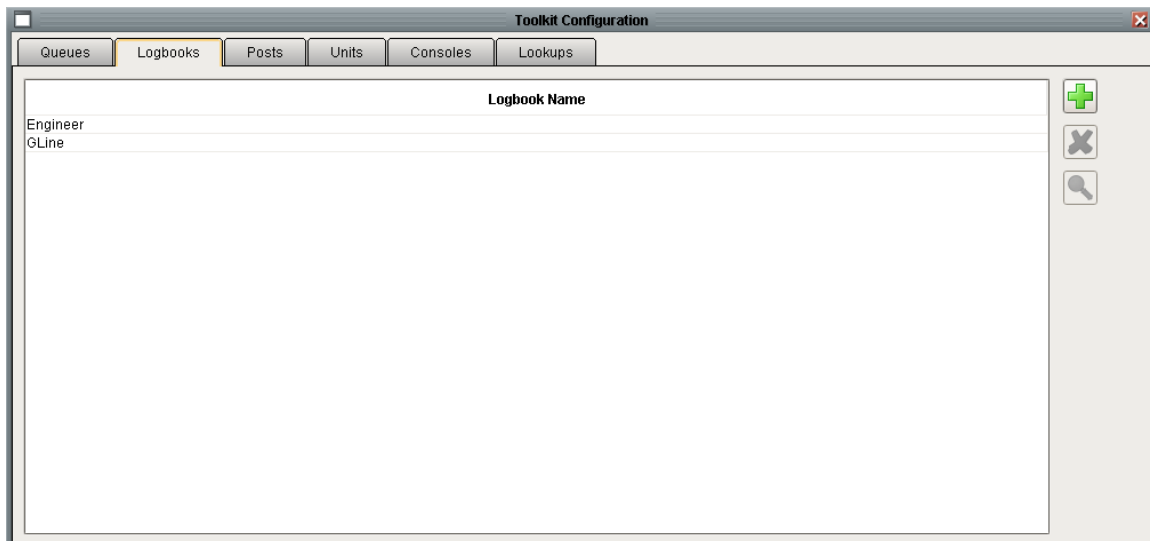
There is generally an Operator Logbook for each post, as defined by the configuration of the *TkPost* table. A logbook can be shared between several posts if desired. Additional logbooks unrelated to a post can also be created.

5.6 Creating a new Logbook

Queues are managed by selecting the *Configuration -> Toolkit Configuration* choice in the *Admin* menu.



This opens a window tabs for configuring various aspects of the platform. The second tab is for Logbooks. The standard is that there is a logbook for each console and one for the Engineer.



6 Operator Console Alert

An Operator Console alert, hereafter referred to as an OC alert, is used to notify the operator of some alert or event. In general, it is a mechanism for the server (gateway) to notify the operator (a client) that the gateway has detected something of interest. The gateway uses messaging to notify a client of the alert. The client then processes the message. An OC alert is not appropriate to notify a client of something detected at the client such as invalid data entry.

The Ignition-based platform does not use audible alerts to get the operator's attention. Instead, the OC can be alerted to abnormal conditions by displaying a window that makes itself as annoying as possible by changing color and animating messages across the screen. This window is sometimes referred to as "the loud window". The operator can then get more information by hitting the "Acknowledge" button. In order for the OC alert to be effective, the operator must be logged into the Ignition client with the appropriate console window open, the computer must not be in sleep, screen saver, or hibernated state, and the Ignition client cannot be minimized or covered by another Windows application.

The decision to keep the platform mute is based on the operating conditions rather than on technical limitations of the platform which is fully capable of playing sound files or interfacing with paging systems. This was mostly because the applications were typically viewed as one of the following cases:

1. Advisory in nature.
2. Considered slow to the point that immediate action by the operator was not important.
3. Sequence related such that any alerts do not need to be audible as the operator is physically viewing the screen while the action is taking place.

By providing the OC alert facility as part of the platform, we have created a consistent mechanism for alerting the operator. It is now only a requirement that developers utilize the given API within their applications.

Because the primary goal of the OC Alert utility is to notify the operator of an important event, there is a notification escalation policy as part of the utility. An alert is meant for all users is characterized by not specifying a post. However, if a post is specified then the following escalation will be followed:

1. Determine the console operator account for the specified post is logged in. The gateway will examine all connected clients to determine if the username matches the post name. An alert will be sent to all matching clients if more than one is found. If one or more is found then no further notification is necessary, otherwise proceed to 2.
2. Determine if there is a client that is not logged in as a console operator but is viewing the console window. The gateway does this by sending a "*listWindows*" message to each client. The alert will be sent to every window displaying the console window. If one or more is found then no further notification is necessary, otherwise proceed to 3.

3. Send an alert to every client.

There is no time delay mechanism that will send an alert a newly connected console operator in the event he was not connected at the time of the alert.

The OC Alert utility is comprised of Python scripts and user interface components.

6.1 Scripting Interface

Python scripts are provided for conveniently posting an OC alert. The scripting interface is in *ils.common.ocAlert*. The interface for posting the OC Alert window is shown below:

```
def sendAlert(project, post, topMessage, bottomMessage, mainMessage, buttonLabel, callback=None,
              callbackPayloadDictionary=None, timeoutEnabled=False, timeoutSeconds=0):
```

The arguments are:

Argument	Description
project	The name of the project, typically “XOM”
post	The post that the alert is intended for. See the discussion above regarding the notification escalation procedure.
topMessage	The text of the top message
bottomMessage	The text of the bottom message
mainMessage	The text of the main message – this supports HTML so line breaks can be added where desired. If an empty text string is supplied then the blue field is made invisible.
buttonLabel	The text of the button
Callback (optional)	Fully qualified path to a Python procedure that will be called when the user presses the button or the alert times out. The default is None
callbackPayloadDictionary (optional)	A dictionary of arguments that will be passed to the callback script. The default is None
timeoutEnabled (optional)	True or False, specifies if the alert will automatically be dismissed after the specified period of time. The default is False
timeoutSeconds (optional)	If timeoutEnabled is True, specifies the time period that the alert will be displayed before it is automatically hidden and the callback is called. The default is 0.

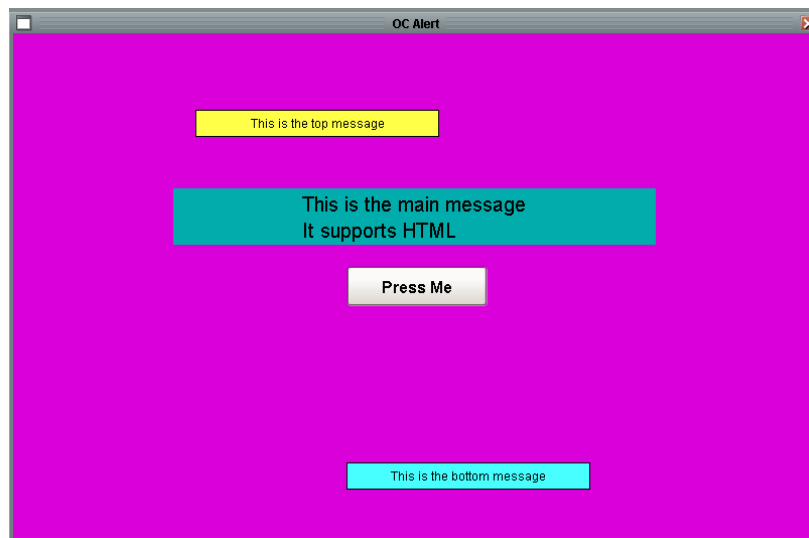
The user supplied callback may be internal or external Python. The callback takes two arguments: event and payload. The callback used by the Lab Data toolkit in response to a validity limit violation is shown below:

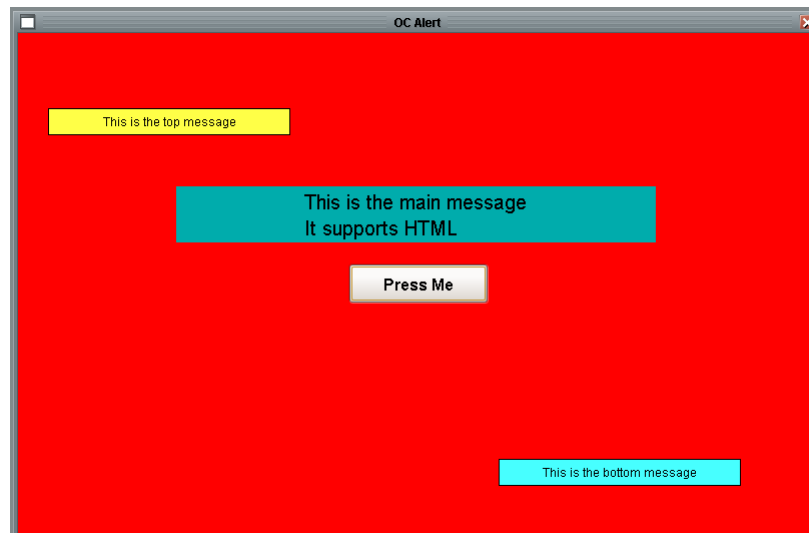
```
# This is a callback from the Acknowledge button in the middle of the loud workspace.  
def validityLimitActionLauncher(event, payload):  
    system.nav.closeParentWindow(event)  
    system.nav.openWindow("Lab Data/Validity Limit Warning", payload)
```

This callback posts a window, but the callback can do whatever action is desired. Keep in mind that the callback executes in the client. The event is the action pressed event on the button.

6.2 User Interface

The user interface is limited to the loud workspace that is shown below. The background color alternates between red and purple. The top and bottom messages scroll back and forth across the window. The standard action of the button is to dismiss the loud workspace. If additional processing is desired when the operator presses the button then a custom callback must be supplied.





When the window is displayed on multiple clients, pressing the “Acknowledge” button on one client does not hide the window on other windows.

6.3 Usage Conventions

Two common usages for the OC alert are:

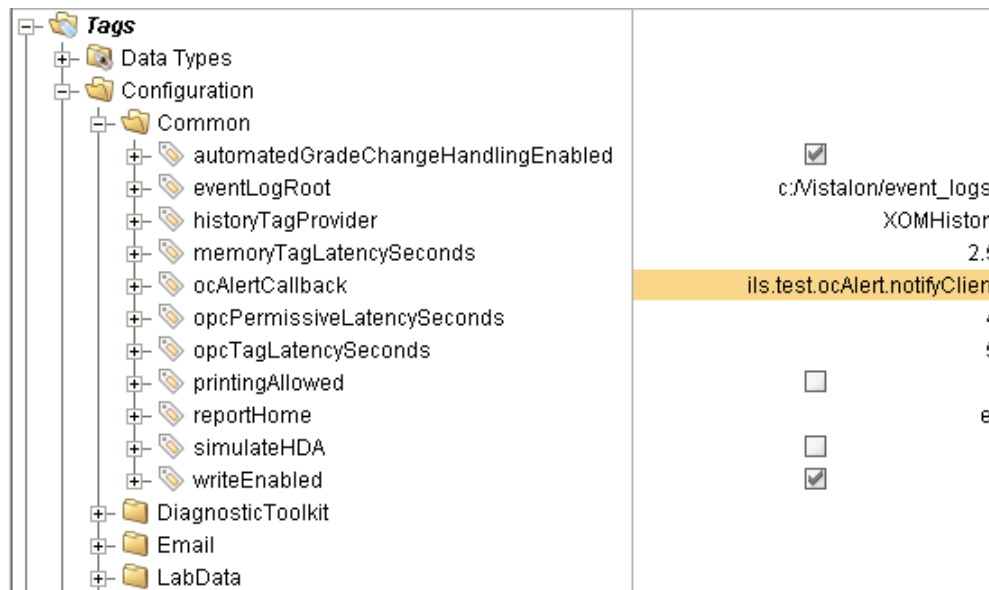
1. When the diagnostic toolkit detects a problem and recommends corrective action.
2. When lab data receives a raw value is outside the configured limits

Using the API described above, additional uses may be easily implemented.

6.4 Custom Callback

In addition to the standard behaviour described above, the notification strategy can be extended by specifying a user supplied Python callback that is called every time an OC alert is sent. A common usage is to execute a script that maximizes the Ignition client on the operator console. (There does not appear to be a generic way to do this within Ignition, but it is possible within the operator console automation.)

The callback is specified by configuring the configuration tag shown below:



The callback is called once by the API that sends the OC alert. The callback typically runs in the gateway for a diagnostic toolkit diagnosis or a SFC or a lab data limit violation. A sample callback is shown below. It takes a single argument, payload, a dictionary of values that is the same as is sent to the client for the standard notification.

```

1 '''
2 Created on Feb 17, 2020
3
4 @author: phass
5 '''
6
7 def notifyClient(payload):
8     print "-----"
9     print "---          MAXIMIZE THE CLIENT NOW          ---"
10    print "-----"
```

7 Engineering Units

Automatic unit conversion is implicit in recipe data and is based on the generic engineering unit definition module. Most types of recipe data support the designation of engineering units.

Units are defined in the database. If the units table is empty, unit definitions may be loaded from a file in the XOM unit format. At time of writing, no standard UI for loading units exists, though the SFC Demo project does have a Unit Conversion window that can be used. Units can be loaded into the database with Python scripting, e.g.

```
import ils.common.units
```

```
file = system.gui.inputBox("Enter unit file", "")
if file != None:
    newUnits = ils.common.units.parseUnitFile(file)
    ils.common.units.Unit.addUnits(newUnits)
```

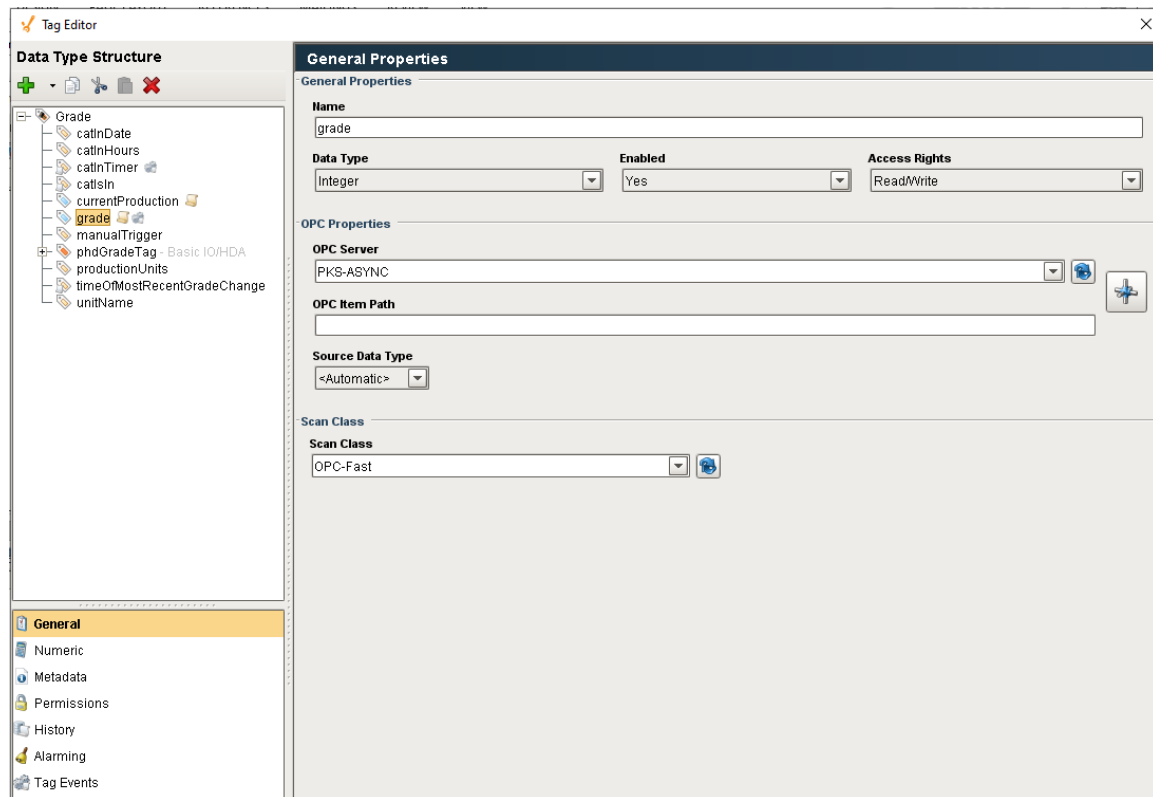
7.1 Steps to Create a New Type of Recipe Data

This provides an outline of the steps required to implement a new class of recipe data.

1. Insert a record into the SfcRecipeDataType for the new type.
2. Update the Visio Design Specification
3. Create database table(s)
4. Create database view
5. Test the Recipe Browser (Edit, Create, Delete)
6. Update the Recipe Data Editor window (Create New, Edit Existing).
7. Update the s88Get() and s88Set() APIs
8. Migration
- 9.
- 10.

8 Grade UDT and Grade Handling

A Grade UDT provides some basic grade and grade change handling. Because the action to take on a grade change is site specific the UDT does not implement any change logic. Typically a value change handler is written for the grade tag, which is an OPC tag. The UDT does provide some built-in support for keeping track of cat in hours, current production, and the time of the most recent grade change. The UDT assumes that the grade is an integer although the datatype of the grade tag can be overridden. The tag used for the grade in recipe download is a standalone OPC tag and is never the grade UDT.



9 Batch Tracking

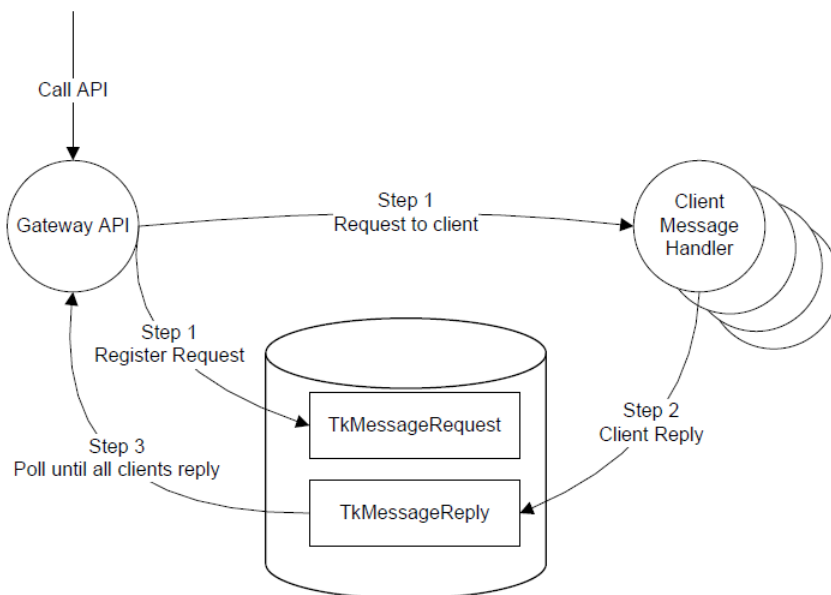
The Batch Tracking utility provides some rudimentary capabilities for tracking batches.

10 Message-Reply Utility

The standard mechanism for the gateway to notify a client of something is to send a message. Ignition has a well-defined message utility that includes client and gateway handlers and an API for sending a message. The purpose of this utility is to extend the capabilities to include a reply. The use case that prompted this utility was the OC alert notification where the gateway needs to notify every client that is displaying the console workspace. If no clients are displaying the console workspace then the alert is sent to all clients. Because there is no way for one client to know what another client is doing, the bookkeeping needs to be done in the gateway.

10.1 Implementation

The Message-Reply utility is implemented using standard Ignition building blocks: messages, message handlers, Python scripts, and database tables. The implementation is best described using the “*listWindows*” API as an example. This API allows the gateway to get a list of all of the Vision windows displayed on each client. The process begins by something in the gateway calling *ils.common.message.interface.listWindows()*. This API determines how many windows are connected, inserts a record into the TkMessageRequest table, and then sends the “*listWindows*” message, which includes the request id, to each of them. The client message handler, *ils.common.message.client.handle()* receives the message, obtains a list of its open windows, and inserts a record into the TkMessageReply table.



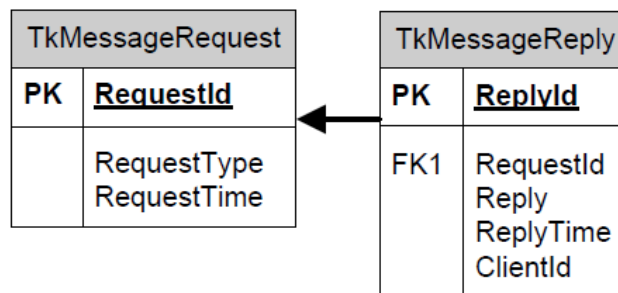
The listWindows API, which runs in the gateway:

```
--
41 # Send a request to all clients to return a list of the windows that are showing
42 # This will run until a response is received from every connected window
43 def listWindows(project="", db=""):
44     log.trace("%s - Listing windows..." % (__name__))
45     if project == "":
46         project = system.util.getProjectName()
47     from ils.common.message.gateway import sendAndReceive
48     pds=sendAndReceive('listWindows', project, db)
49
50     return pds
```

The message handler, *ils.common.message.client.handle()*, which runs in the client is shown below. This handler can be expanded to handle additional types of requests.

```
8 def handle(payload):
9     print "Received a message with payload: ", payload
10
11     requestId = payload.get("requestId", -1)
12     requestType = payload.get("requestType", "unknown")
13     clientId = system.util.getClientId()
14
15     # Get a list of all of the open windows on a client
16     if requestType == "listWindows":
17         windows=system.gui.getOpenedWindowNames()
18         reply = ",".join(map(str, windows))
19         SQL = "Insert into TkMessageReply (RequestId, Reply, ReplyTime, ClientId)" \
20             " values (%i, '%s', getdate(), '%s')"\
21             % (requestId, reply, clientId)
22         system.db.runUpdateQuery(SQL)
23
24
```

The database tables involved in the implementation are shown below:



10.2 Scripting Interface

There are currently three interfaces provided by the message-reply utility that can be called from Python. These are intended to be called from the gateway but also work from a client or the designer.

```

1 # Get the client ids of all clients logged in with a user name that matches a post name.
2 # This returns a list of client ids.
3 # Note: This does not use the message-reply utility because there is a system utility, getSessionInfo()
4 # that lists information about each client.
5 def getPostClientIds(post, project="", db=""):

```

```
# Get the client ids of all clients that are showing the console window for a specific console.
# This returns a list of client ids.
def getConsoleClientIds(console, project="", db=""):

># Get a list of windows that are currently displayed by each client.
# This uses the message reply utility to send a request to all clients to return a list of the windows
# that are showing. This will run until a response is received from every connected window.
# This returns a dataset with three columns: Reply, ReplyTime, ClientId. Reply is a comma
# separated string of the window names.
def listWindows(project="", db=""):
```

11 Watchdog Utility

Communication to external systems via OPC DA is critical. The Ignition system is only as robust as the communication to the external systems. The purpose of a watchdog is to monitor the communication with an external system. Different types of systems warrant different strategies but the goal is to determine if communication to a system has really failed even though no errors have been reported and the communication status appears healthy and to take corrective action if it is not healthy.

11.1 Watchdog Theory of Operation

The theory of each watchdog is described below.

11.1.1 OPC Read Watchdog

A read watchdog reads a OPC tag and can perform two tests. The *changeStrategy* test checks that the current value is different from the value the last time the watchdog ran. The assumption is that the DCS is changing the tag at a frequency faster than the watchdog is running. It does not try to synchronize the writing and reading, just that the value is changing. The *readAndCompareStrategy* test checks that the Ignition tag value matches the value from a direct OPC read and that the quality of both are good.

11.1.2 OPC Write Watchdog

A write watchdog is part of a system where the DCS increments a counter on some frequency and when an associated tag receives a value it clears the counter. The external system alarms if the counter ever reaches a certain threshold. For example, the external system may increment an integer counter every minute. Ignition writes to a different tag every five minutes which triggers the DCS to reset the counter. The DCS will implement an alarm if the value ever reached some threshold, 15 for instance. Ignition will trigger an alarm if a user configurable number of consecutive writes fail. It does not read the value that the DCS is incrementing.

11.2 Watchdog Configuration

Watchdogs are implemented using tag UDTs. There is a UDT for each strategy described in the previous section. The *OPC Watchdog* is a parent of both which defines common properties and member tags.

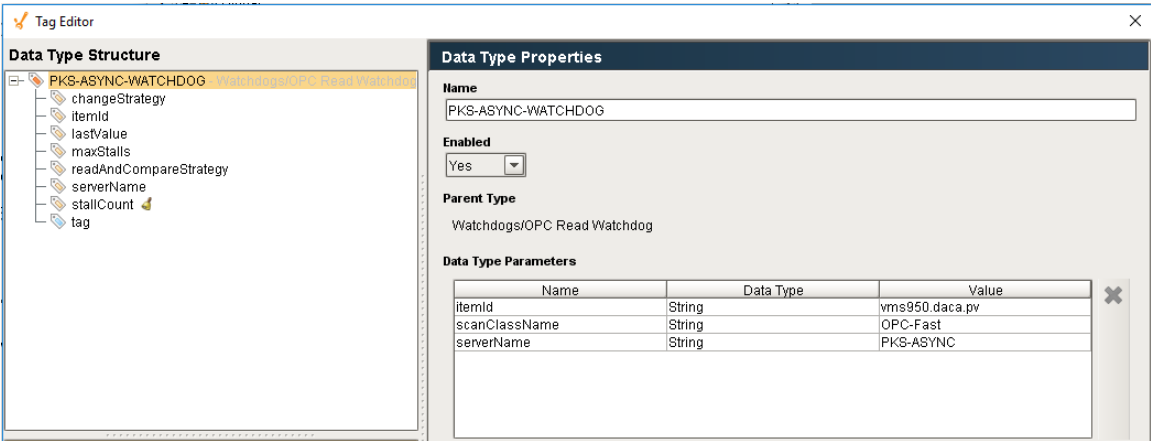
Tag	Value	Data Type
<div> <div>Tags</div> <div> <div>Data Types</div> <div>Basic IO</div> <div>Controllers</div> <div>Halobutyl</div> <div>Lab Bias</div> <div>Lab Data</div> <div>Recipe Data</div> <div>SFC</div> <div>Vistalon</div> <div>Watchdogs</div> <div> <div>OPC Read Watchdog</div> <div>OPC Watchdog</div> <div>OPC Write Watchdog</div> </div> <div>Grade</div> <div>OPC Dataset Collector</div> <div>OPC Derived Value</div> </div> </div>		
		Watchdogs/OPC Watchdog
		Watchdogs/OPC Watchdog

To robustly monitor the OPC communications used at a site, a read and a write watchdog is configured for each OPC connection. The watchdog UDTs are created and placed in the Site/Watchdogs folder. This is the only folder that is scanned for watchdogs to monitor.

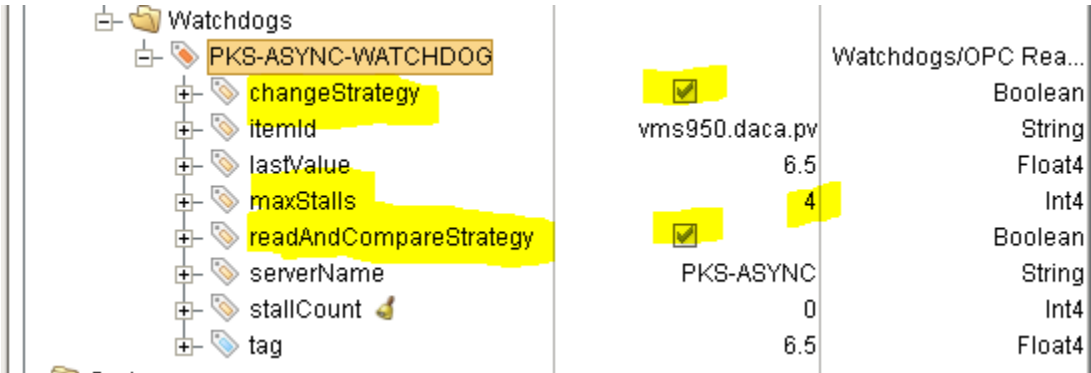
Tag	Value	Data Type
<div> <div>Tags</div> <div> <div>Data Types</div> <div>Configuration</div> <div>Data Pump</div> <div>DiagnosticToolkit</div> <div>LabData</div> <div>Recipe</div> <div>Sandbox</div> <div>SFC IO</div> <div>Site</div> <div> <div>CatOut</div> <div>Coldstick</div> <div>CRX</div> <div>CSTR</div> <div>Driers</div> <div>GC Analyzers</div> <div>RateChange</div> <div>RLA3</div> <div>VFU</div> <div>Watchdogs</div> <div> <div>OPC PKS-Async Read Watchdog</div> <div>OPC PKS-Async Write Watchdog</div> <div>OPC PKS-Sync Read Watchdog</div> <div>OPC PKS-Sync Write Watchdog</div> </div> </div> </div> </div>		
		Watchdogs/OPC Read Watchdog
		Watchdogs/OPC Write Watchdog
		Watchdogs/OPC Read Watchdog
		Watchdogs/OPC Write Watchdog

11.2.1 OPC Read Watchdog Configuration

The OPC read watchdog configuration is shown below.



Additionally, the *changeStrategy*, *maxStalls*, and *readAndCompareStrategy* tags need to be configured.



The behaviour of the Watch Dog UDT is determined by configuring the properties and tags in the UDT as describer below:

Type	Name	Description
Property	itemId	Used to configure the tag which is used in both types of checks. Also used to do an OPC Read in the “Read and Compare” check.
Property	scanClassName	Used to configure the tag which is used in both types of checks.

Type	Name	Description
Property	serverName	Used to configure the tag which is used in both types of checks. Also used to do an OPC Read in the "Read and Compare" check.
Tag Value	changeStrategy	If True, then the "Change" check will be evaluated.
Tag Value	maxStalls	Specifies the number of consecutive stalls that will trigger an alarm. Normally, an alarm will not be set until several consecutive stalls are detected.
Tag Value	readAndCompareStrategy	If True, then the "Read and Compare" check will be evaluated.

11.2.2 Monitor Strategies

The OPC watchdog utilizes several strategies as described in the

Strategy	Description
Change	This strategy monitors a read-only OPC tag that is constantly updated by the remote system. The watchdogs are all evaluated on a fixed schedule. The tag must update faster than the scan rate of the watchdogs. The watchdog compares the current value of the tag with the value that was read the last time. If ever the two match then the stall count of the watchdog is incremented. This test is performed when the <i>changeStrategy</i> bit is set.
Read and Compare	This strategy compares the value of the tag, which receives updates automatically, with the value from the <code>system.opc.read()</code> api which goes directly to the OPC server and reads the current value totally bypassing the scanClass read system. If they ever do not match then the stall count is incremented. This test is performed when the <i>readAndCompareStrategy</i> bit is set.

11.3 Watchdog Alarm Configuration

The standard Ignition alarming is used to detect when the *stallCount* exceeds the *maxStalls*. A stall is defined as any watchdog cycle where one or more of the individual tests fail. Normally, an alarm will not be set until several consecutive stalls are detected. The PKS-ASYNC-WATCHDOG shown above will alarm when there are 4 consecutive stalls. The configuration of the alarm is fully specified in the UDT and does not need to be altered for each instance.

The screenshot displays the Ignition Tag Editor interface for configuring an alarm. The left pane shows the 'Data Type Structure' for 'OPC Read Watchdog', with 'stallCount' highlighted. The bottom sidebar shows 'Alarming' selected. The main configuration pane is titled 'Alarm Configuration' and shows the 'OPC Read Watchdog - Above Setpoint, High' configuration.

OPC Read Watchdog - Above Setpoint, High

Main

Name	OPC Read Watchdog
Enabled	true
Priority	High
Timestamp Source	System
Display Path	{InstanceName}
Ack Mode	Manual
Notes	
Ack Notes Required	false
Shelving Allowed	true

Alarm Mode Settings

Mode	Above Setpoint
Setpoint	{maxStalls}
Inclusive	true
Any Change	false

Deadbands and Time Delays

Deadband	0
Deadband Mode	Absolute
Active delay (seconds)	0
Clear delay (seconds)	0

Notification

Ack Pipeline	
Active Pipeline	Watchdog
Clear Pipeline	

Email Notification Properties

Custom Message	
Custom Subject	

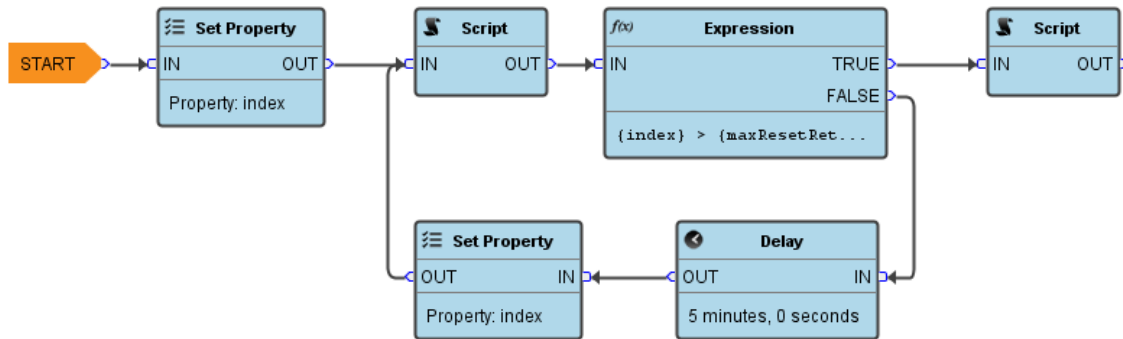
Associated Data

post	{post}
serverName	{serverName}
maxResetRetries	{maxResetRetries}

(Name)
(Description)

11.4 Watchdog Alarm Pipeline

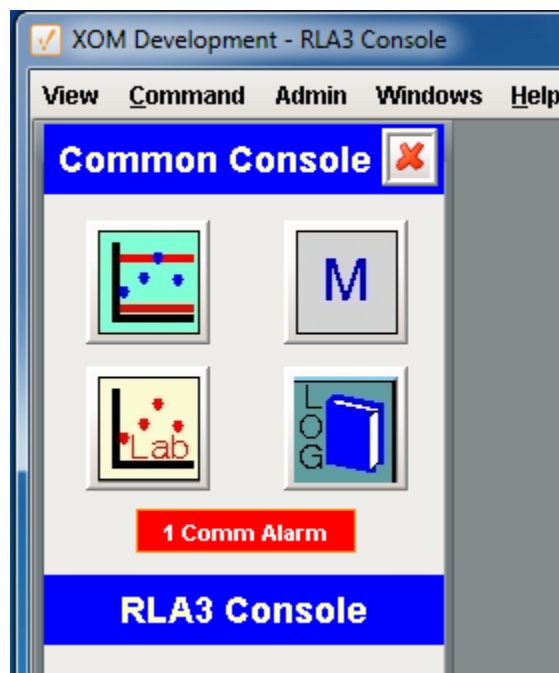
An Ignition Alarm Pipeline is used to define actions to take once the watchdog detects that there is a communication problem. There currently is a single pipeline for both watchdog types although additional pipelines can be developed.



The pipeline uses the magic Python from Travis that resets an OPC interface. It waits a while and then resets it again. It will try a user configurable number of times and then notify the operator via an OC alert that communication is bad and we were unable to get it working. If the alarm clears at any time then pipeline execution ends.

11.5 Watchdog Alarm User Interface

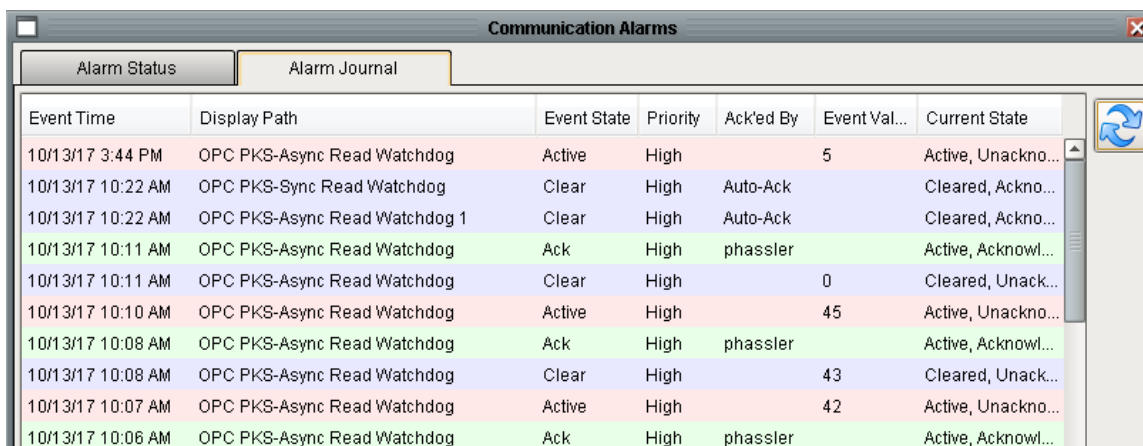
The standard Ignition alarming user interface is used to view the state of the watchdog alarms. The common console, which should always be visible, contains an alarm indicator which is transparent when there are no problems but flashes between red and yellow when there is any communication alarm.



The indicator is a button, when the button is pressed the “Communication Alarms” window is displayed. The “Alarm Status” tab shows the state of active and unacknowledged alarms.

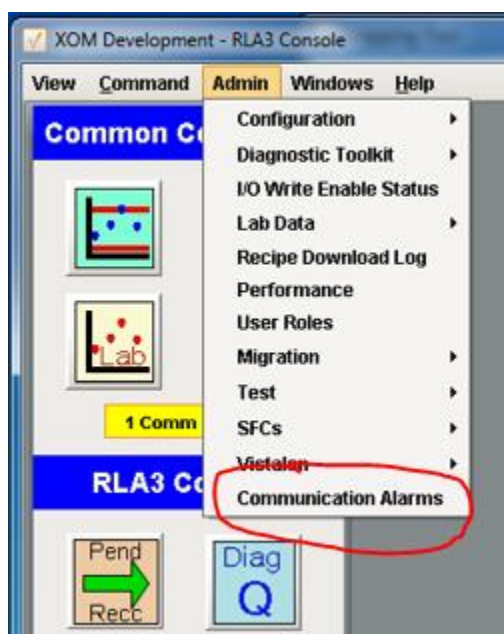


The “Alarm Journal” tab shows the history of all communication alarms including acknowledgment. This information can be used to determine how long the communication was down.



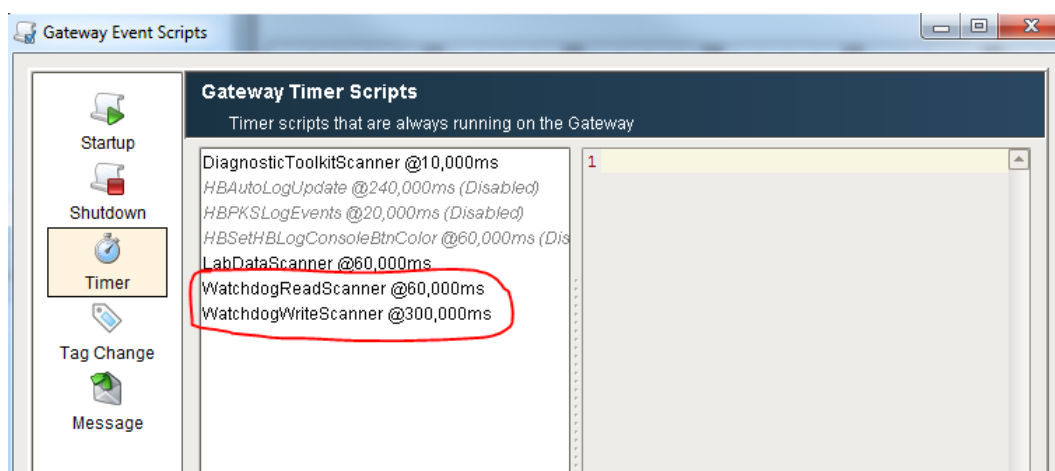
Event Time	Display Path	Event State	Priority	Ack'd By	Event Val...	Current State
10/13/17 3:44 PM	OPC PKS-Async Read Watchdog	Active	High		5	Active, Unackno...
10/13/17 10:22 AM	OPC PKS-Sync Read Watchdog	Clear	High	Auto-Ack		Cleared, Ackno...
10/13/17 10:22 AM	OPC PKS-Async Read Watchdog 1	Clear	High	Auto-Ack		Cleared, Ackno...
10/13/17 10:11 AM	OPC PKS-Async Read Watchdog	Ack	High	phassler		Active, Acknowl...
10/13/17 10:11 AM	OPC PKS-Async Read Watchdog	Clear	High		0	Cleared, Unack...
10/13/17 10:10 AM	OPC PKS-Async Read Watchdog	Active	High		45	Active, Unackno...
10/13/17 10:08 AM	OPC PKS-Async Read Watchdog	Ack	High	phassler		Active, Acknowl...
10/13/17 10:08 AM	OPC PKS-Async Read Watchdog	Clear	High		43	Cleared, Unack...
10/13/17 10:07 AM	OPC PKS-Async Read Watchdog	Active	High		42	Active, Unackno...
10/13/17 10:06 AM	OPC PKS-Async Read Watchdog	Ack	High	phassler		Active, Acknowl...

The same “Communication Alarms” window can also be accessed from the “Admin” menu which is useful for viewing the journal when no alarms are active.



11.6 Watchdog Evaluation

The watchdogs are evaluated on a fixed interval from a gateway timer script:



12 Security / Logon Facilities

12.1 User Mode / User Group

The details of system access / system security have not been determined. But the Recipe Toolkit was designed to distinguish between three roles:

Role	Description
Operator	An operator. Cannot see OE recipe data and cannot edit AE recipe data.
AE	Can edit all recipe data
OE	Cannot access the recipe toolkit, but can access the recipe database application

12.2 Single Sign-On

Ignition supports Windows Active Directory and Single Sign On which means that once an operator logs in to their workstation, they will not need to log on to Ignition. The credentials used to log on to the workstation will be used by Ignition. This will require the assistance of ExxonMobil's systems group to set up the Windows groups to correspond with the roles defined above.

The Active Directory is configured with groups that vary from site to site and do not exactly match the roles that have been configured in Ignition. To facilitate the translation from the Active Directory group to the Ignition role is configured in the Role Translation table which is shown below for Vistalon.

	IgnitionRole	WindowsRole
1	AE	AE
2	AE	AE-ILS
3	AE	AE_PL_VI
4	Operator	OC_PE
5	Operator	Operator
6	Operator	RTA.Operators.GG
7	Operator	WPC.Operators.Vist
8	Operator	WPC.Operators.VistRLA3

This table is referenced when a client connects to Ignition to determine if the user is an operator and if so then the menus will be pruned to only display menus appropriate for operators.

There are two utility functions that are available that conveniently check a user's roles and these translations to determine if the client is an operator or an ae. These functions are: *isOperator()* and *isAE()* and are both contained in *ils.common.user*.

12.3 Console / Post Determination

At login, the client needs to determine its post / console. There are numerous ways this could be done, the exact mechanism has not been determined. Once the post is known, it will write the post to the client post tag. From there it will be available to any toolkit.

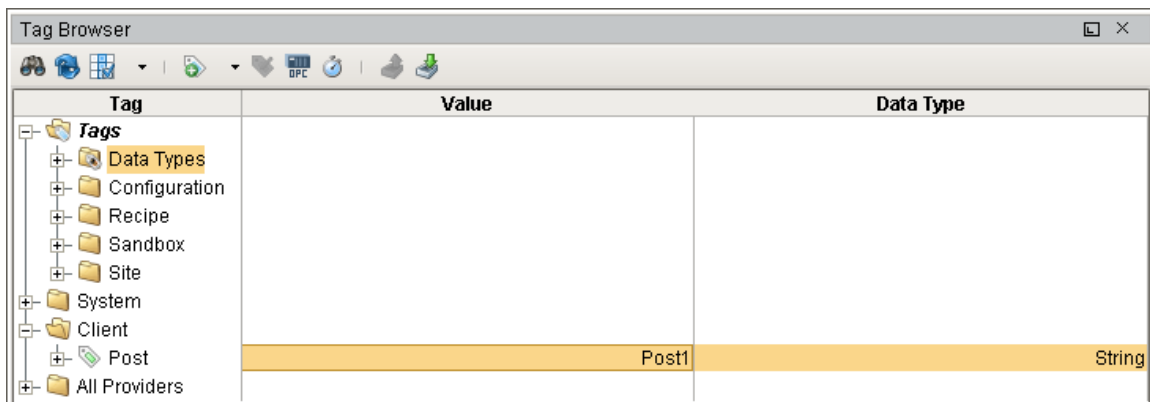


Figure 1 - Client Post Tag

13 Miscellaneous Features

This section describes miscellaneous small features that do not fit into any other category.

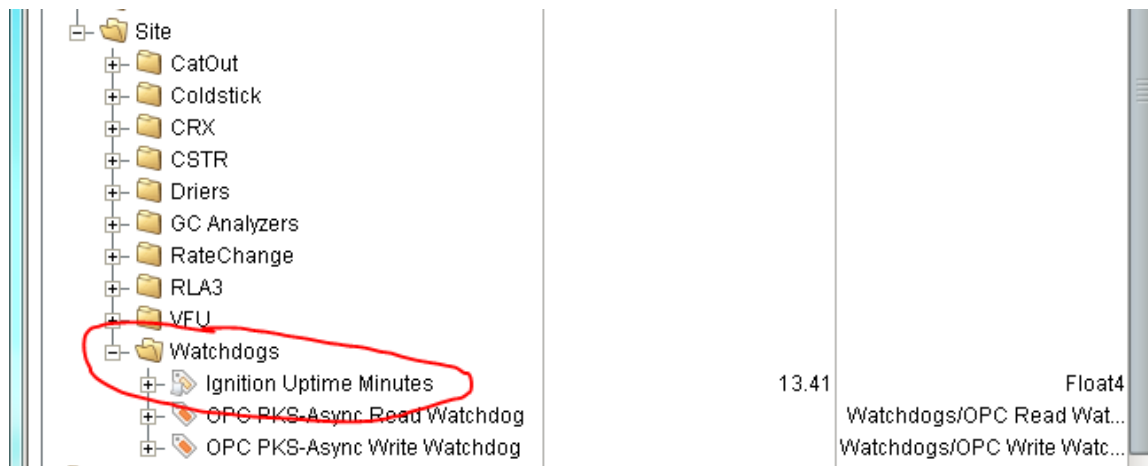
13.1 Warm Boot Support

Warm boot is not a well-defined process in Ignition. A cold boot is when Ignition is started after a computer reboot or when the service is stopped and then started. A warm boot refers to all of the other initializations that occur after various resources are edited

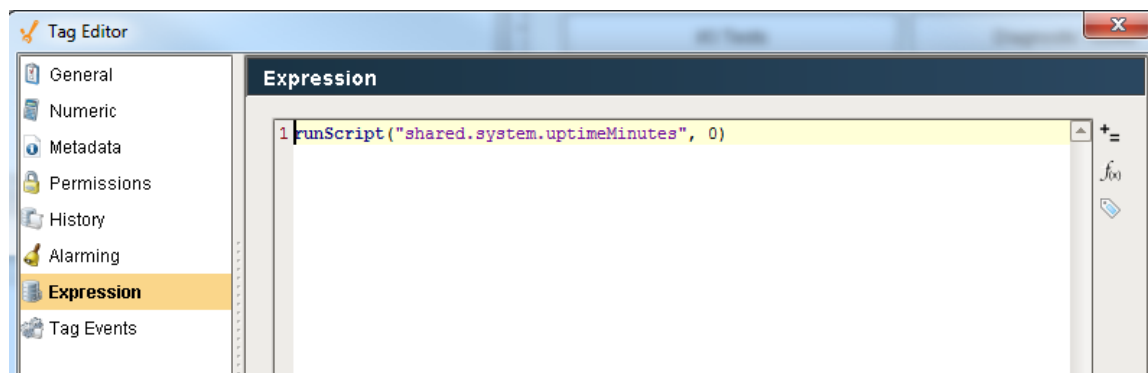
and saved. For example, editing the project's gateway event scripts and then saving the project will call the gateway start-up handler.

13.2 Ignition Uptime Tag

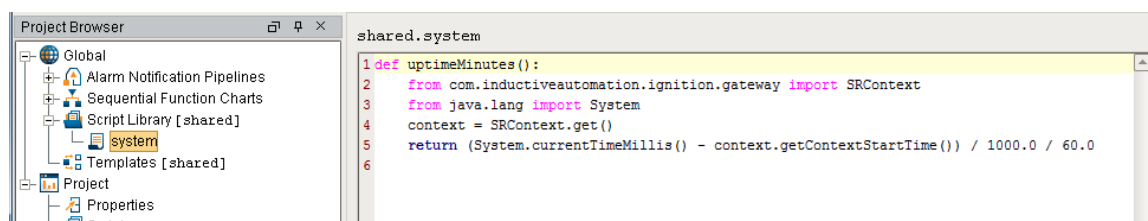
An expression tag has been created for each site in the Site/Watchdogs folder. It calculates the minutes that Ignition has been running. It does not get reset on a warm boot.



The tag is an expression tag with the following expression:



It calls a Python function which must be in shared Python:

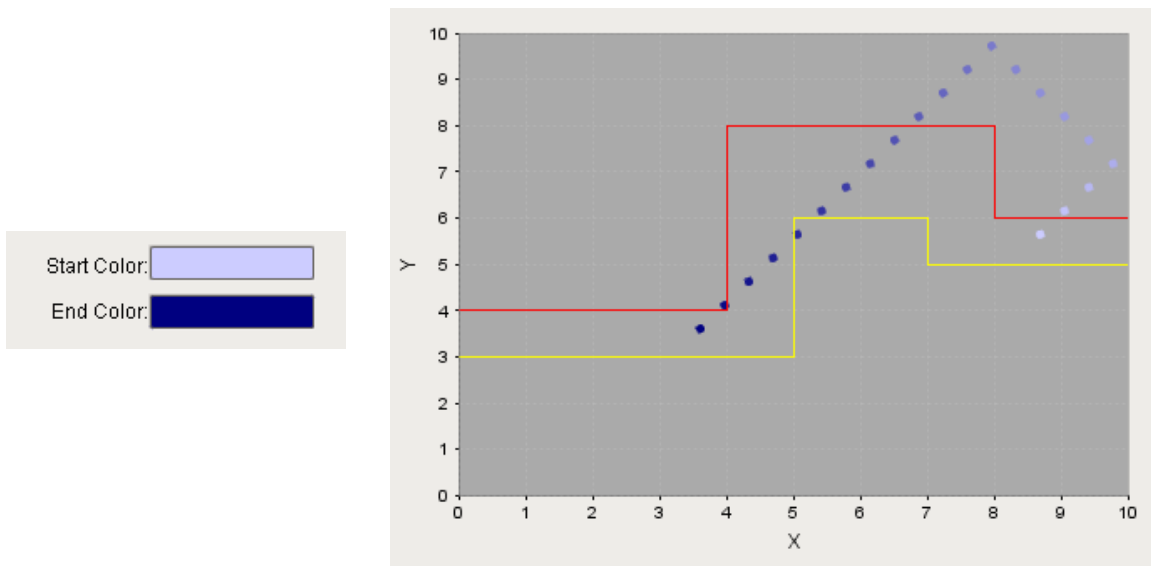


13.3 Start-up Considerations

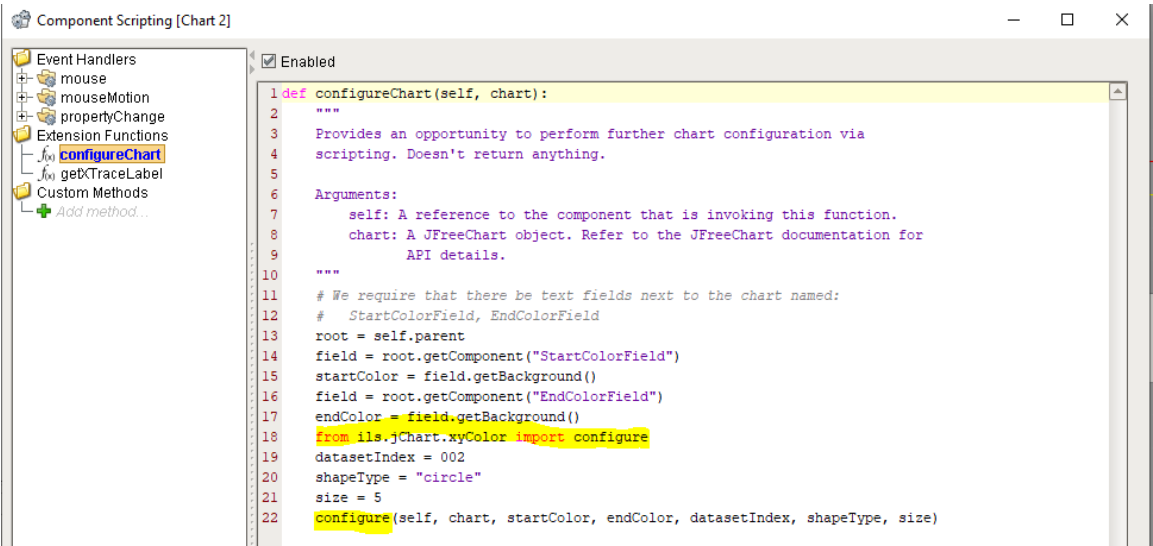
Each toolkit has a start-up script that performs necessary initializations that need to occur on a cold start. The start-up logic should check if it is being due to a warmboot and then skip any / all initialization that is not appropriate. There is a utility API `ils.common.util.isWarmboot()` that can be called. If the system has been running for more than 5 minutes then it will return **True**. On a cold start, the tag may not yet have a value depending on who wins the start-up race. If run minutes does not have a value then it is a cold start.

13.4 XY Chart Customizations

There are several custom applications that use the XY chart. The chart plots one value vs another, possibly one tag versus another, for a specific timeframe. The chart may include an upper limit, lower limit, warning limit, error limit, and actual value. Each of these are stored in a dataset with two columns containing the x value and the y value. Generally, the limit data sets are a solid color. The value data set shows how the x and y values change over time. It is useful to indicate the beginning and the end of the dataset. There are several techniques that could be used. One way is to add an annotation at the start and end points. Another technique, which is discussed here, is to use a color gradient. The chart below has three data sets. A yellow warning limit, a red error limit, and the actual value. The actual value uses a color gradient from light purple to dark purple from the first point to the last point.



The customization to the chart is implemented in external Python. It is called from the *configure* extension function.



The screenshot shows a window titled "Component Scripting [Chart 2]" with a sidebar on the left and a main text area on the right. The sidebar contains a tree view with categories: Event Handlers (mouse, mouseMotion, propertyChange), Extension Functions (configureChart, getTraceLabel), and Custom Methods (Add method...). The main text area contains a Python script for the `configureChart` function. The script is as follows:

```
1 def configureChart(self, chart):
2     """
3     Provides an opportunity to perform further chart configuration via
4     scripting. Doesn't return anything.
5
6     Arguments:
7         self: A reference to the component that is invoking this function.
8         chart: A JFreeChart object. Refer to the JFreeChart documentation for
9             API details.
10
11     """
12     # We require that there be text fields next to the chart named:
13     # StartColorField, EndColorField
14     root = self.parent
15     field = root.getComponent("StartColorField")
16     startColor = field.getBackground()
17     field = root.getComponent("EndColorField")
18     endColor = field.getBackground()
19     from ils.jChart.xyColor import configure
20     datasetIndex = 002
21     shapeType = "circle"
22     size = 5
23     configure(self, chart, startColor, endColor, datasetIndex, shapeType, size)
```

The example shows two text fields on the root container for specifying the start and end colors. The colors could be specified as custom properties on the root container. Unfortunately, the chart component does not allow adding custom properties. There are seven arguments:

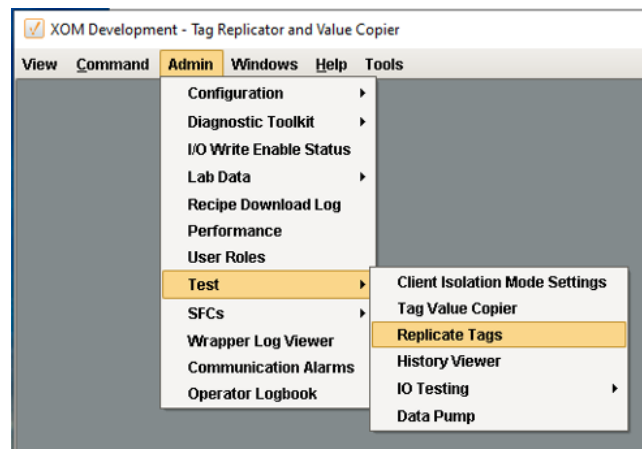
Argument	Description
self	The Ignition chart component
chart	The jChart object embedded in the chart component
startColor	The color for the first point.
endColor	The color for the last point
datasetIndex	The index into the chart's datasets that specifies which dataset the gradient gets applied to.
shapeType	The text string, circle or square
The size of the point	0 to 25, in pixels (?)

13.5 Tag Replication

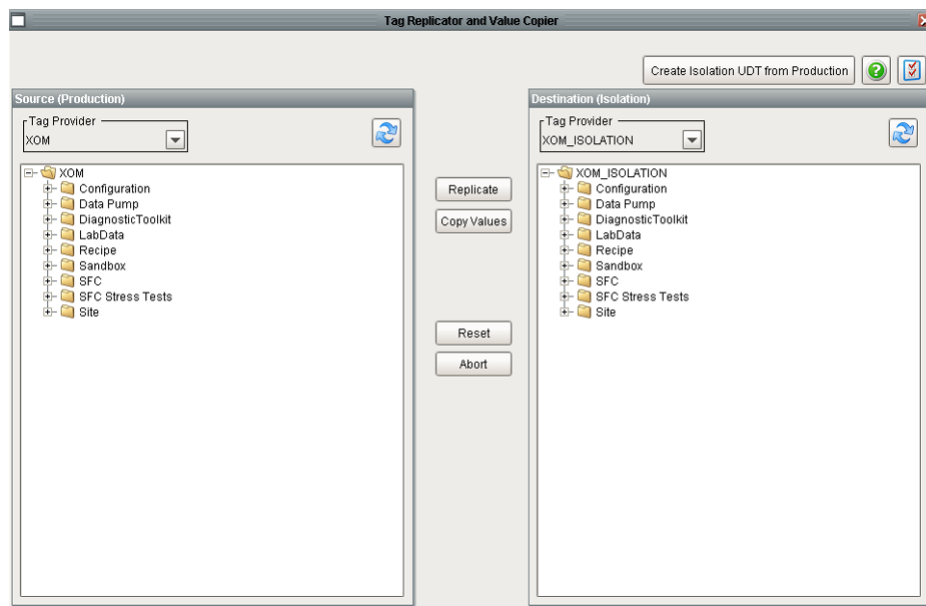
“Tag Replication” refers to the process of creating a shadow set of tags in the isolation tag provider using tags in the production tag provider. The shadow tags replace OPC tags with memory tags which guarantee isolation from actual external systems. This is the first step in setting up an environment for testing an application in an offline way. The isolation tags can be used in conjunction with the data pump to simulate an online system.

The original tag replication tool was implemented in Java as part of the Test Framework module.

The new tag replication tool uses a Vision window and Python along with a set of shadow UDTs in the main project. This can be run the designer by opening the window and putting the designer in run mode. It can also be run from a client from the main menu as shown below:



The window that is launched is shown below:



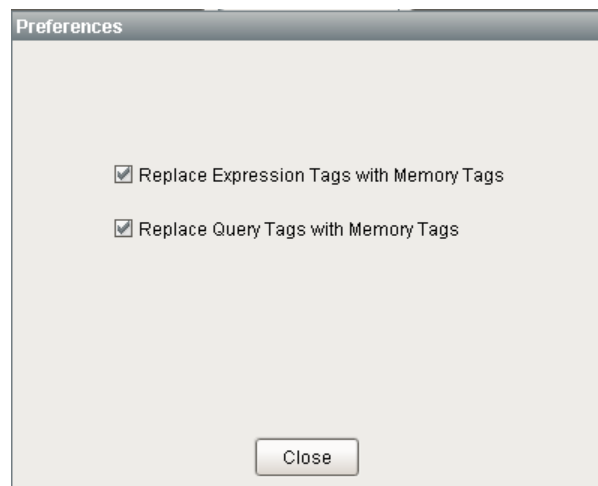
The window has two major functions: replicating tags and copying tag values. Both features require that: a source tag provider is selected, a destination tag provider is selected, and a node is selected in the “Source” tree. All of the tags in a tag provider can be replicated by selecting the root node. A single folder or a single tag may also be replicated. Once a tag has been replicated, it is not replaced, updated or deleted by subsequent attempts at replicating. Tags that have been deleted from the production tag

provider after it has been replicated in the isolation tag provider are not deleted by subsequent replications. Because the replicator is easy to use, if obsolete tags are a concern, it is recommended that the entire isolation tree be deleted and recreated.

The “Destination” tree is provided to provide confirmation that tags are created in the selected destination. The tree must be refreshed after the replication

Isolation tags differ from production tags by replacing OPC tags with memory tags. Production UDTs are replaced by instances of isolation UDTs with the same name. The isolation UDTs have the same structure as the production UDTs but the embedded OPC tags have been replaced with embedded memory tags.

There are two user-configurable preferences available under the preferences button.



1) Replace Expression Tags: Specifies if an EXPRESSION tag will be replaced with a memory tag or if an expression tag will be created with the same expression. If an expression tag is created in isolation then any references to the production tag provider will be replaced with the isolation tag provider.

2) Replace Query Tags: Specifies if a QUERY tag will be replaced with a memory tag or if a query tag will be created with the same SQL. Note that the definition of a query tag allows the data source to be specified. This is normally set to <Default> which specifies that the database specified in the definition of the tag provider will be used. So as long as the production tag being replicated uses the <Default> data source and that the production tag provider uses the production database and that the isolation tag provider specifies the isolation database.

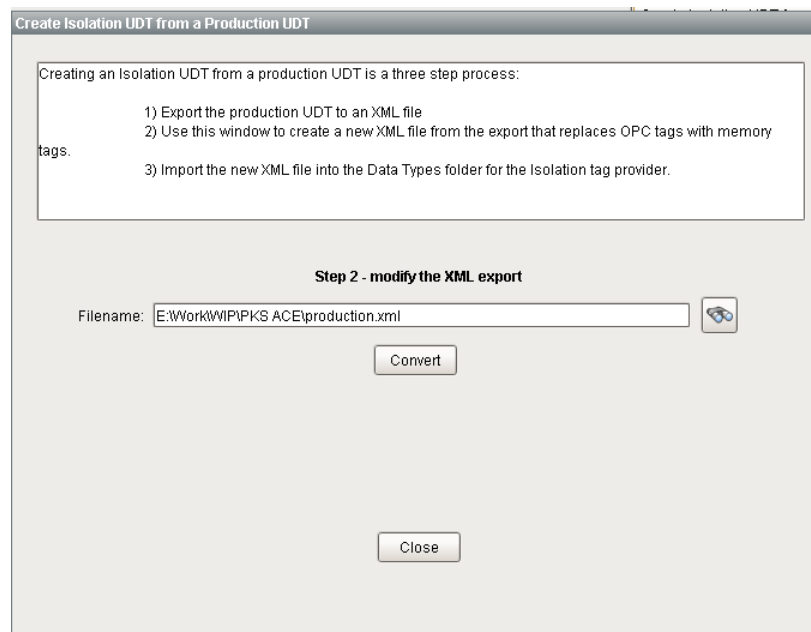
Derived tags are a seldom used feature in the original deployments. Derived tags are always replaced with memory tags when replicated. Tag change scripts will be applied to replicated tags without any modifications to the script. History keeping and alarms will NOT be replicated. If there are custom configurations to production tags that are desired to the isolation tags, then the configuration will need to be manually added to the

replicated tag. The custom configuration will not be affected by subsequent replication efforts.

The "Copy Values" button updates tags which have already been replicated with values from the source tag provider. The same rules apply for selecting source tags and for the treatment of SQL and expression tags.

The final capability of the window is to assist in the creation of isolation UDTs. Creating a UDT for isolation can be done manually by copying the UDT from production and replacing the OPC tags with memory using the tag editing facilities in the designer. Alternatively, you can:

1. Export the production UDT to an XML file.
2. Use the "Create Isolation UDT from Production" button which opens the window shown below. Select the XML file exported above and press convert. A new file will be written with the suffix `_isolation` added to the filename. This will automatically convert OPC tags to memory tags.
3. Import the XML file that is created above into the Tag Browser in Ignition.



13.6 Data Pump

The data pump is used to simulate a production environment using isolation (memory) tags. The data pump reads data from a CSV file and writes a record of values to a set of tags on a fixed interval.

14 Developing on the Production System

Ignition is both a development and a production platform. There are many different architectures that can be deployed. The main factors to consider are redundancy, robustness, and cost. The lowest cost architecture is a single license system that is used for development and production without redundancy. This is the most common architecture and it is important to understand the side effects of certain development activities.

14.1 Tags

Tag changes are immediate in all scopes as soon as the “OK” button is pressed. It does not require a project save. Changes to a UDT will automatically update all instances and will cause tag change scripts to run.

14.2 External Python

Changes to external Python are loaded immediately in client, designer, and gateway scope. It does not trigger gateway or client start-up scripts. It will cause client screens to reload and depending on the screen may cause unintended side effects.

14.3 Client Event Scripts

Any change to any type of client event script will update every client as soon as the project is saved. It does not run the client start-up script. This has the unintended side effect of exposing the full list of consoles for all of the Baton Rouge sites. This is because the Menubar definition contains all consoles and which are pruned to only the appropriate consoles by the client start-up script.

14.4 Project Resources (Windows, Diagnostic Diagrams, Templates, Reports)

At login, the client needs to determine its post / console. There are numerous ways this can be determined; the exact mechanism has not been determined. Once the post is known, it will write the post to the client post tag. From there it will be available to any toolkit.

14.5 Global Resources (Alarm Pipelines, SFCs, Shared Scripts)

Changes to global resources may trigger the gateway start up script to fire. However, as long as the warmboot protection is in place it will not have any unexpected side effects.