ExxonMobil Chemical Company

Sequential Control

User's Guide

Version 1.7
August 24, 2022

Table of Contents

1. INTRODUCTION

Sequential Function Charts (SFCs) are based on a graphical programming language in the IEC 61131 standard. This language may be familiar to PLC programmers, as it is one of the languages commonly available for programming PLCs.

1.1 About this Document

This document contains information about ILS Automation's Sequential Control Toolkit which extends the capabilities of Inductive Automation's (IA) Sequential Function Chart (SFC) module. This document:

- Describes the overall framework of the toolkit.
- Describes each block in detail.
- Describes the user interface.
- Describes best practices

1.2 Audience

This document is intended for personnel (engineers) whose responsibilities include developing new or supporting existing sequential function charts using the ILS Sequential Control module. It is not intended for end users (operators) that are responsible for executing the SFCs.

1.3 Related Documentation

Portions of the sequential control definitions are derived from the following two standards.

ANSI/ISA-S88.01-1995 Batch Control Part 1: Models and Terminology

ANSI/ISA-S88.00.02-2001 Batch Control Part 2: Data Structures and Guidelines for Languages

IEC 61131 – Programmable Controllers

Sequential Function Charts – Online documentation from Inductive Automation

Common Facilities User's Guide – ILS Automation

2. SEQUENTIAL CONTROL FRAMEWORK

A Sequential Function Chart (SFC) is a series of steps that are defined in a single location, a chart, and then called in sequential order. Additional elements in the chart can determine where the flow of the chart will lead. Charts can loop around indefinitely, or execute a set number of times before ending.

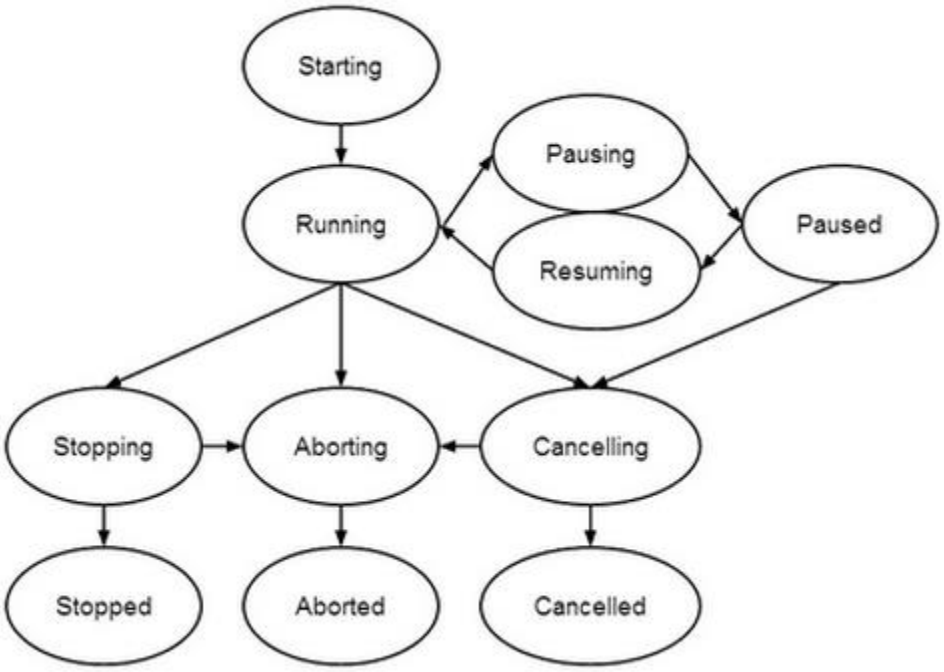
SFCs are used to execute logic in ways that are more convenient to structure than with Python scripts alone. Because of their inherently visual depiction, they help to illuminate logic to users, and facilitate intuitive development and refinement. Charts can be monitored as they run visually, making troubleshooting easier than with scripting alone.

SFCs provide a recipe-based approach to implementing a sequence of discrete control and monitoring steps. The elements of the recipe consist of a hierarchical set of blocks. The recipe elements refer to specific process actions or tasks and include relevant data.

The extensions implemented by ILS are derived from the ANSI/ISA-S88.01-1995 Batch Control Part 1: Models and Terminology and ANSI/ISA-S88.00.02-2001 Batch Control Part 2: Data Structures and Guidelines for Languages, hereafter referred to as the S88 Specification. These two standards include a broad range of topics, many of which are not applicable to this particular definition of sequential control. The standard is very flexible and primarily provides a guideline with terminology and definitions. In particular, this implementation follows the concepts of Control Recipes and Procedural Control Elements as outlined in the standards. This document will only provide brief descriptions as required of S88 and refer the reader to the standards where necessary.

2.1 Chart Lifecycle

Each running chart is effectively a finite-state machine: the chart can be in one of many different states, but it is only ever in a single state at any given moment. The current state of the chart carries significant meaning.



SFC Lifecycle

2.2 Overview of Sequential Control Elements

Sequential Control consists of a hierarchy of control elements, or blocks. A recipe contains a:

Unit Procedures which contains

Operations which contains

Phases which contains

Tasks which can contain

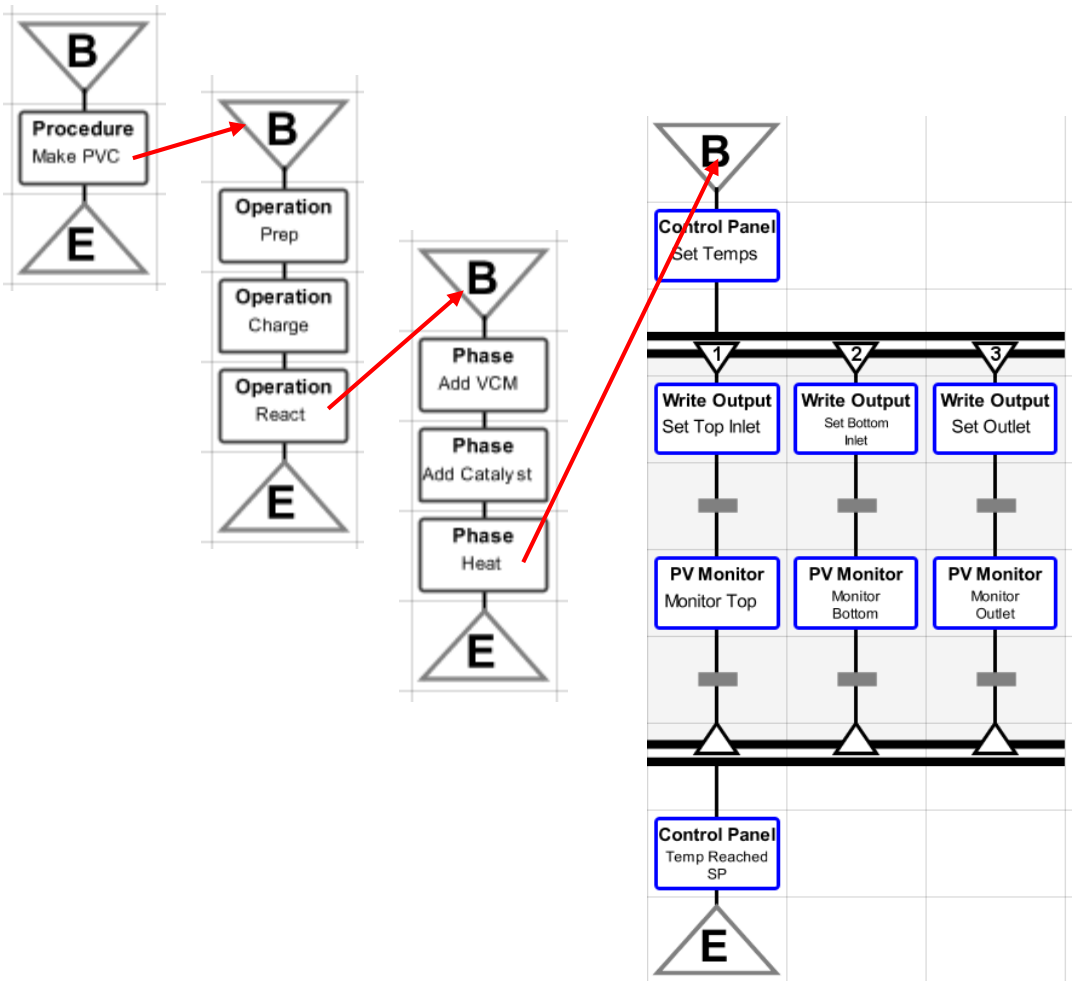
Tasks

Element	Description
Unit Procedure	Consists of an ordered set of operations that causes a contiguous production sequence to take place within a unit. Within a unit procedure, only a single operation can execute at any time. Example: "Polymerize VCM", "Dry PVC".

Element	Description
Operation	An ordered set of phases that defines a major processing sequence that takes the material being processed from one state to another, usually involving a chemical or physical change. It is often desirable to locate operation boundaries at points in the procedure where normal processing can safely be suspended. . Example: "Preparation: Pull a vacuum on the reactor and coat the walls with antifoulant", "React: Add VCM and catalyst, heat, and wait for the reactor pressure to drop".
Phase	Smallest element that can perform a process-orientated task. A phase is subdivided into Tasks. Phases can issue commands, change setpoints, and set or clear alarm limits. Example: "Add VCM", Add Catalyst"
Task	This level is defined within this document. S88 allows implementers to define these elements with only limited guidelines. This level will include changing set points, controller modes, monitoring instrument values versus set points, etc.

2.3 Typical Example

A typical sequential control example that shows the full recipe structure from the high-level abstract unit procedure down to concrete write output and PV Monitoring tasks is shown below. The unit procedure, labeled "Make PVC", encapsulates three operations labeled, "Preparation", "Charge", and "React". The three Operations run sequentially, one after the other. The Operation labeled "React" consists of the three phases labeled "Add VCM", "Add Catalyst", and "Heat". Although numerous Phases can run simultaneously, in this diagram, the phases are constructed to run serially, one after the other. The "Heat" phase encapsulates the tasks that post a message to the control panel and then executes three parallel threads of execution for writing and monitoring set points. The phase is complete when all three threads are complete. The details of Unit Procedures, Operations, and Phases are discussed in section 4.3, task level blocks are discussed in detail in section 4.



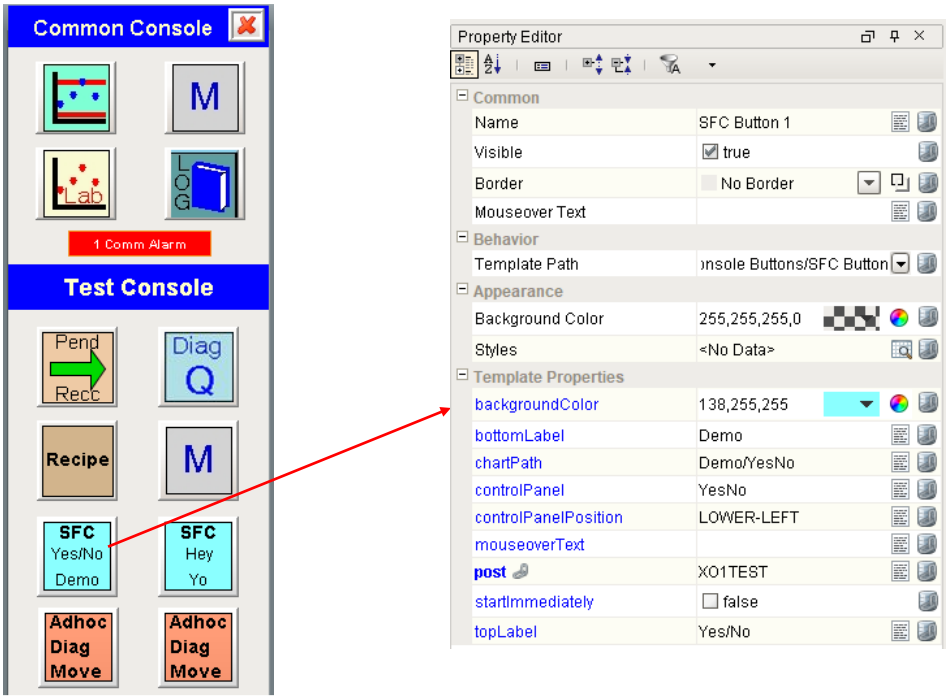
Typical Unit Procedure

2.4 Running Sequential Function Charts

There are several ways to run a sequential function chart.

2.4.1 Running a SFC from a Standard Console Button

The project contains a template for a button that can be placed on a console window and configured without writing any custom code as shown below.



Standard Console Button for Running a SFC

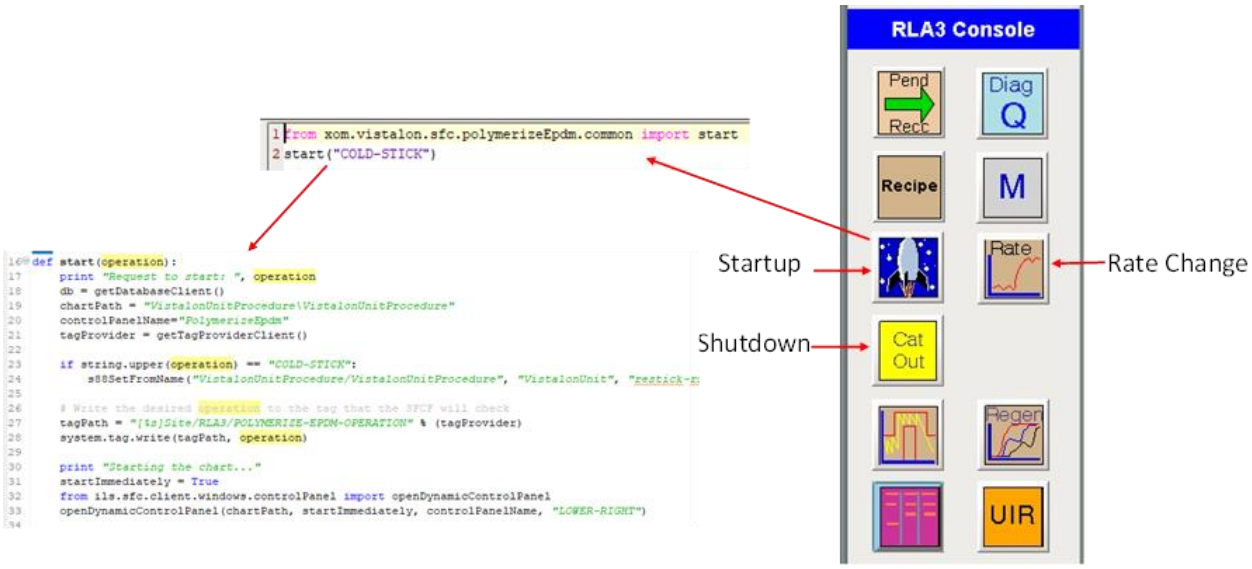
2.4.2 Running a SFC from a Custom Console Button

A custom action button can also be used to run a SFC if a custom icon is desired, but this will require writing a small amount of code. Running a SFC ultimately uses the standard *system.sfc.startChart()* system procedure. However, in order to set up the proper environment to support isolation mode testing, persistent windows, and to open the control panel; charts should be started using the following API:

ils.sfc.client.windows.controlPanel.openDynamicControlPanel(chartPath, startImmediately, controlPanelName, position)

The framework assumes that the chart that is named by the first argument contains a unit procedure step. The second argument, *startImmediately*, specifies if the chart will start automatically or if the operator needs to press the "*Start*" button on the control panel.

Although there is no limit to the number of unit procedures defined at a site, sites typically have a fairly small number of fairly complicated unit procedures. Therefore, there is generally a console for the operator with buttons mapped to the common unit procedures / operations as shown below.

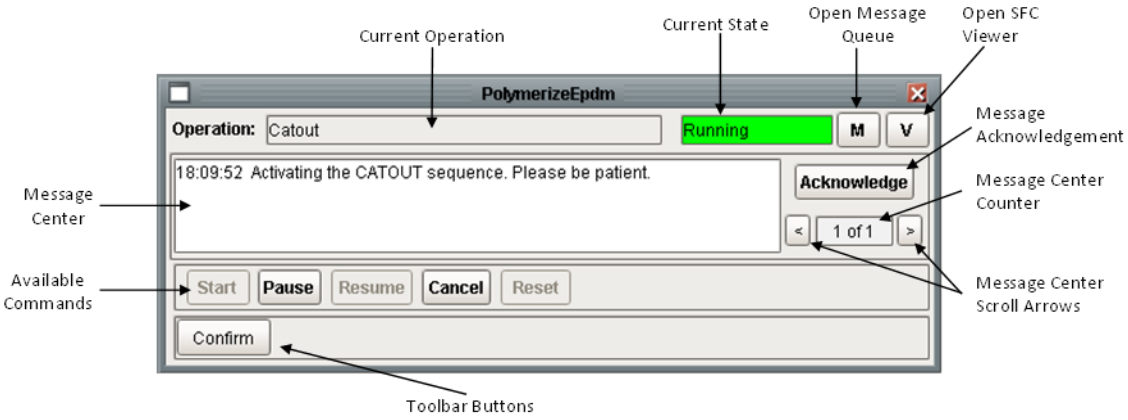


Running a SFC From from a Custom Button on the Control Panel

2.5 Control Panel

The Control Panel is used to interact with a running SFC. The control panel is a vision window that has the following capabilities: indicators for the state and current operation, buttons to open the SFC viewer and message queue, buttons for controlling chart execution; a message center monitoring the progress of the SFC, and toolbar buttons for viewing active windows that were closed but are still active.

The control panel that is running the Catout operation for Vistalon unit is shown below:



Control Panel

The top of the control panel has indicators that show the current operation that is running and the current state of the unit procedure. The button labelled "M" open the message queue window. The button labelled "V" opens the SFC viewer which allows the operator to view the SFCs as they execute.

The middle of the control panel contains a "Message Center". It displays a single message at a time, although a queue of messages is maintained. The most recently posted message will be

displayed and the current message index and the number of messages in the queue are displayed in the message center counter. The message center scroll arrows can be used to navigate through the messages in the queue. It also contains a button to acknowledge the message that is currently displayed. When a message is acknowledged it is removed from the message center and the next message will be displayed. Messages that require acknowledgement will flash between red and blue. The message center is used to notify the operator immediately. It most cases, the control panel will always be visible. It is suggested that other windows should be displayed in such a way that they do not cover the control panel. In general, when a message is viewed by the operator then it should be acknowledged, and removed from the message center. The message center should not be used to lead the operator through a sequence of instructions or for debugging. Also, it does not have any logging or auditing capabilities.

Below the message center is a container of buttons for controlling the execution of the unit procedure. The buttons are context sensitive based on the current state per the state transition diagram shown in **Error! Reference source not found.**. The "Cancel" button is one of several ays that a running SFC can be aborted. A unit procedure consists of many charts, many of which can be running at the same time. Whereas Ignition allows any running chart to be cancelled if you know the instance id of the chart by using *system.sfc.cancelChart();* pressing "Cancel" will cancel the top chart and all of the charts that it has spawned, from the bottom up

Finally, the bottom of the control panel contains an area for toolbar buttons. These buttons are used to display windows that have been posted by various blocks in the recipe and then inadvertently closed. It is also useful if the client window is inadvertently closed or is a new client connects after the window was originally posted. Any number of windows may be registered with the control panel. When the number of toolbar buttons exceeds the available space on the control panel, the toolbar button scroll arrows can be used to access any button.

3. RECIPE DATA

Recipe Data are created, viewed and configured using the Recipe Data Browser or the Recipe Data Viewer in a client under the Admin -> SFCs menu. Creating recipe data at run time is not supported. Recipe data values can be read and set at runtime, but new recipe entities cannot generally be created at runtime. The only exception is by using the SQL blocks describes in section 4.10

3.1 Recipe Data Classes

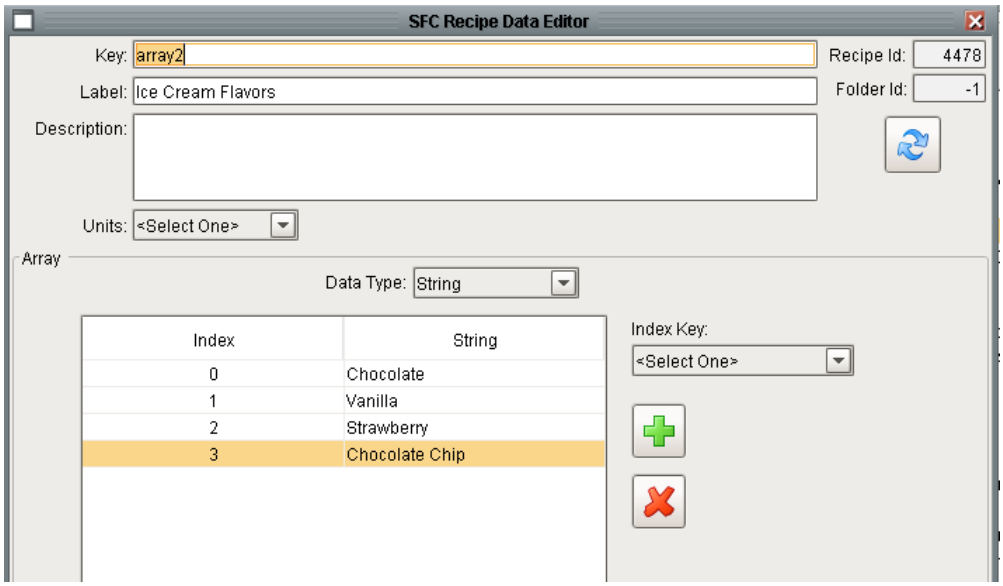
The following table describes the various types of recipe data.

Type	Description
Array	An n dimensional array of any datatype. All elements in the array must be of the same data type.
Input	Defines an input value from a tag.
Matrix	An n X m dimensional matrix of any datatype. All elements in the matrix must be of the same data type.
Output	Defines a output value to a tag. This is used extensively by the Download Monitor, Write Output, and PV Monitoring blocks.
Output Ramp	An extension of the Output type with additional properties for ramping an output to a target value over a time interval.
Recipe	Integrates to the recipe toolkit. Corresponds to a record in the recipe database.
Simple Value	Stores a single datum. Provides full support for units and unit conversion.
SQC	Integrates to the SQC table in the recipe toolkit. Specifies a target value and upper and lower limits.
Timer	Provides built-in support for starting, stopping, pausing, and resuming a timer. This class is specifically designed for the download timer used in Write Output, PV Monitoring, and Download GUI steps.

3.1.1 Array

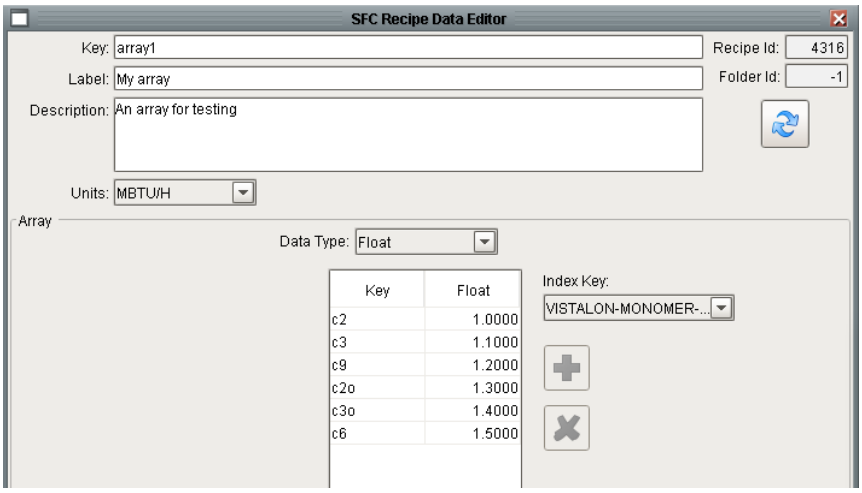
The array class is a holder for an n dimensional array of any datatype. All elements in the array must be of the same data type. The size of the array is defined through the recipe data editor. Changing the size of the array is not supported at run time. The array is initially empty, the green

"+" and the red "x" are used to configure the number of elements in the array. Like all recipe data types, the array also supports unit conversions.



Array Recipe Data Editor

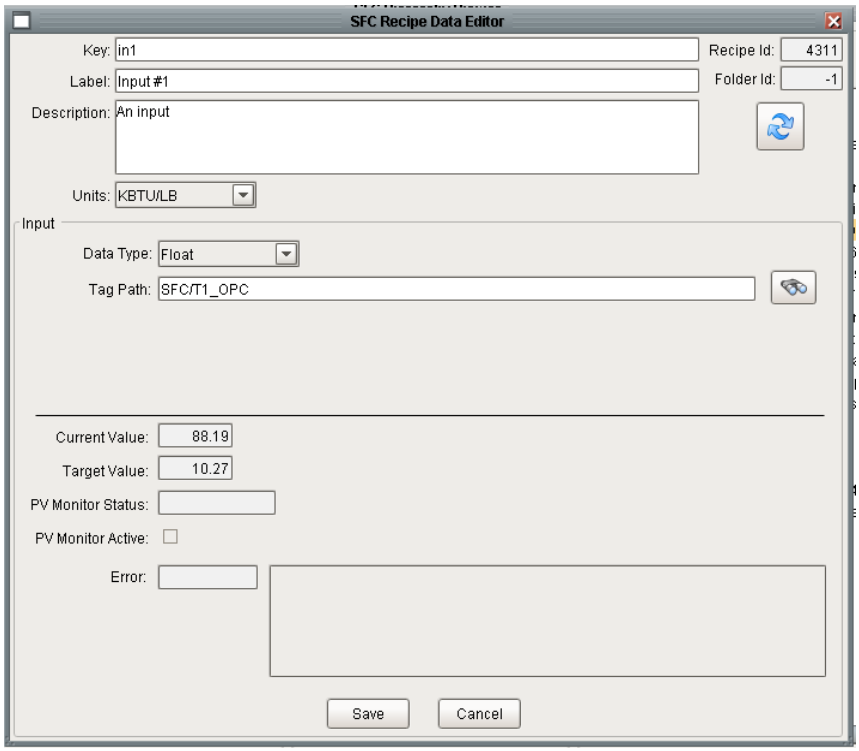
Arrays also support using an index to access elements and determine the size of the array. Defining an index is useful if you will have a number of arrays that would all use the same index. For example, suppose you need arrays for *LeftFeed*, *RightFeed*, *MaxTemp*, *MinTemp* and the index is the same for all of them. By defining an index key you don't have to worry about consistency across all of the arrays. The example below shows a float array which uses monomers as the key. When using a key, the dimension of the array cannot be changed; therefore the add row and delete row buttons are disabled.



Array Recipe Data Editor with a Key

3.1.2 Input

The Input class of recipe data is class of recipe data is used when a value will be read from an external system. It is often used if the input will be subject to PV monitoring.



Input Recipe Data Editor

The fields in the middle of the window need to be configured:

Label	Description
Data Type	The normal data types: Boolean, Float, Integer, String
Tag Path	The full path to the Ignition UDT. If writing a setpoint to a controller. The tagpath should be to the root of the UDT.

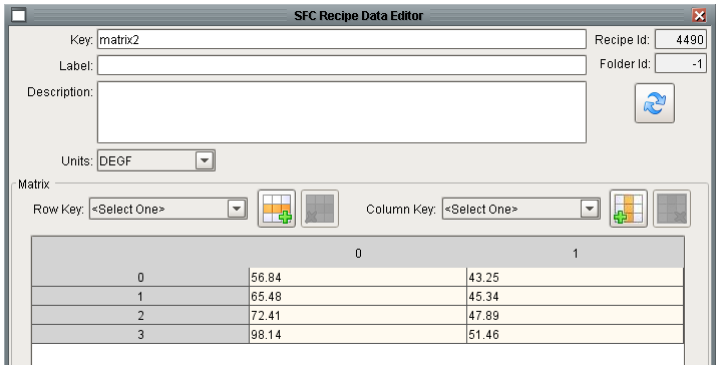
The fields on the lower third of the window, below the horizontal line, are read-only and reflect the status of the most recent PV monitor.

Label	Description
Current Value	The monitored value for a PV monitor.
Target Value	The target value for a PV monitor.
PV Monitor Status	The status of the PV monitor.
PV Monitor Active	A Boolean that reflects if the PV monitor is active.

Error	The error from a write output, if the write did not succeed.

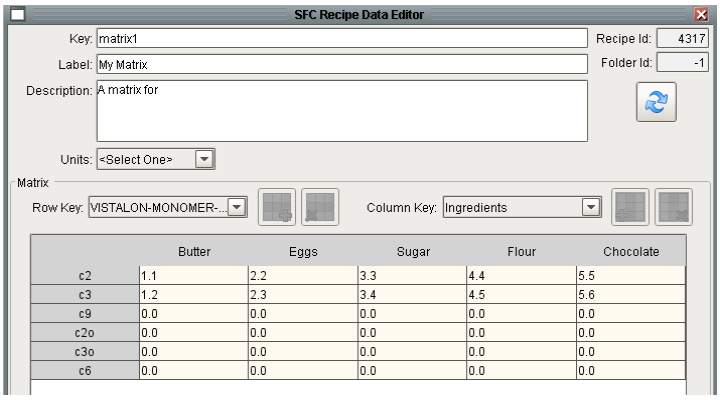
3.1.3 Matrix

The matrix class is a holder for an n X m dimensional matrix of any datatype. All elements in the matrix must be of the same data type. The size of the matrix is defined through the recipe data editor. Changing the dimensions of the matrix is not supported at run time. The matrix is initially empty, the green "+" and the red "x" are used to configure the number of elements in the matrix. Like all recipe data types, the matrix also supports unit conversions.



Matrix Recipe Data Editor

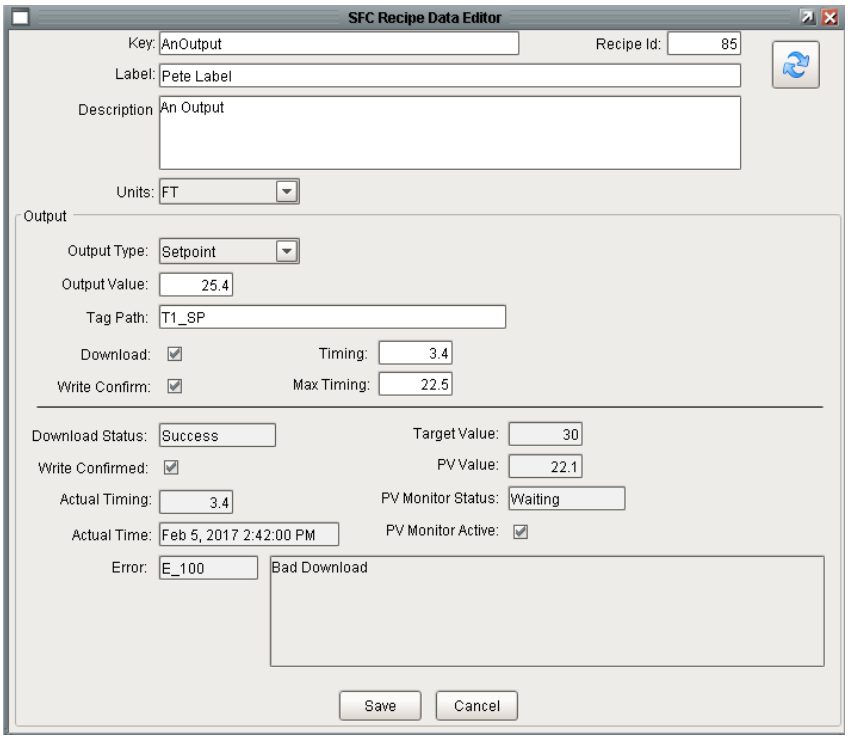
Like the array, the matrix also supports the use of keys for both dimensions:



Matrix with Keys Recipe Data Editor

3.1.4 Output

The Output class of recipe data is used when a value will be written to an external system.



Output Recipe Data Editor

The fields in the middle of the window are specific to "Output" data are explained below:

Label	Description
Output Type	A dropdown with the following values: Mode, Output, Setpoint, Value. When writing to a controller UDT, the first three choices are relevant. When writing to a stand-alone OPC output, regardless of the type of tag in the DCS, then "Value" should be used.
Data Type	The normal data types: Boolean, Float, Integer, String
Output Value	The value that will be written
Tag Path	The full path to the Ignition UDT. If writing a setpoint to a controller. The tagpath should be to the root of the UDT.
Download	A Boolean that determines if the value will be downloaded / monitored. This allows steps to be fully configured and then customized at runtime by setting or clearing this field. This is used by the Write Output, PV Monitoring, and Download GUI steps. Setting the flag on a single recipe data will affect all of these steps that reference that recipe.

Timing	The time that the value will be written, since the beginning of the download, in minutes.
Write Confirm	A Boolean that specifies if the write should be confirmed. The confirmation reads back the value after it has been written in addition to checking the error code returned from the write.
Max Timing	Not sure if this used, maintained for backwards compatibility.

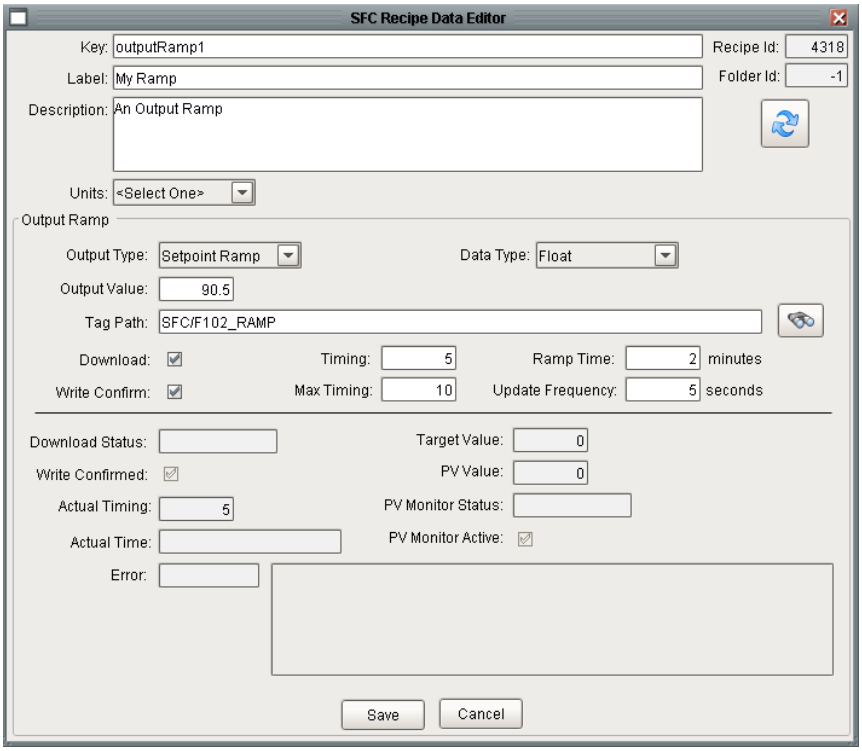
Timing has the following additional features. If the timing has a value of 0.0, it will be written as soon as the block runs, which may be before the block that starts the timer. If the desire is to write the point as soon as the download begins, then a very small positive number should be used. If the timing is > 1000 minutes then the output will be written after all of the timed outputs have been written. This is useful if there are one or more outputs that need to be written at the end after all timed outputs have been written. It is assumed that all reasonable outputs will be written in less than 1000 minutes. Because these parameters are stored in recipe data they can be accessed and set programmatically from action step calculations.

The fields on the lower third of the window, below the horizontal line, are read-only and reflect the status of the most recent write.

Label	Description
Download Status	The status of the most recent Write Output.
Target Value	The target value for a PV monitor.
Write Confirmed	A Boolean that reflects if the most recent write was confirmed.
PV Value	The monitored value for a PV monitor.
Actual Timing	The time that the value was actually written, since the beginning of the download, in minutes.
PV Monitor Status	The status of the PV monitor.
Setpoint Status	The overall status which incorporates the download status and PV monitor status.
Actual Time	The actual time that the value was actually written.
PV Monitor Active	A Boolean that reflects if the PV monitor is active.
Error	The error from a write output, if the write did not succeed.

3.1.5 Output Ramp

The Output Ramp class of recipe data is an extension of the Output class. The editor is shown below. When an output ramp is reference in a Write Output step, a ramp will be written. How the ramp is written is determined by the I/O layer and the type of tag / UDT referenced by the tag path.



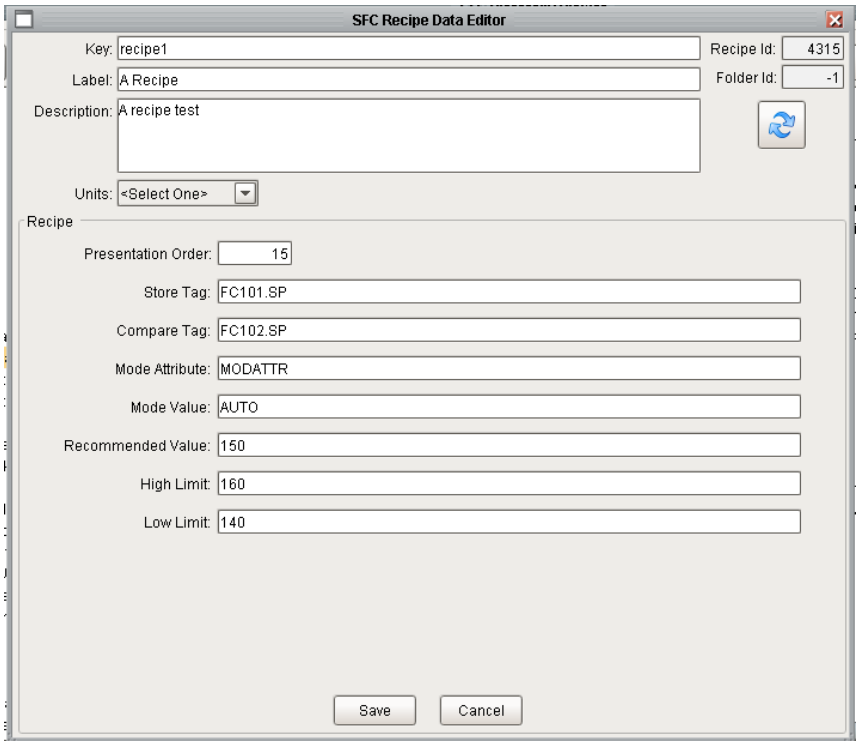
Output Ramp Recipe Data Editor

The properties that differ from an Output are explained below:

Label	Description
Output Type	A dropdown with the following values: Output Ramp or Setpoint Ramp.
Ramp Time	The length of time in minutes for the ramp.
Update Frequency	The time in seconds that defines the rate of updates for the ramp.

3.1.6 Recipe

The Recipe class of recipe data is used to coordinate with the recipe database, aka DB Manager. Typically, data from the recipe database is download to the database via the "Recipe" button on the console window. This class of recipe data is provided if additional data is needed by the SFC directly from the recipe database. The editor for Recipe type of recipe data is shown below. The fields are familiar to users of the Recipe Toolkit.



"Recipe" Recipe Data Editor

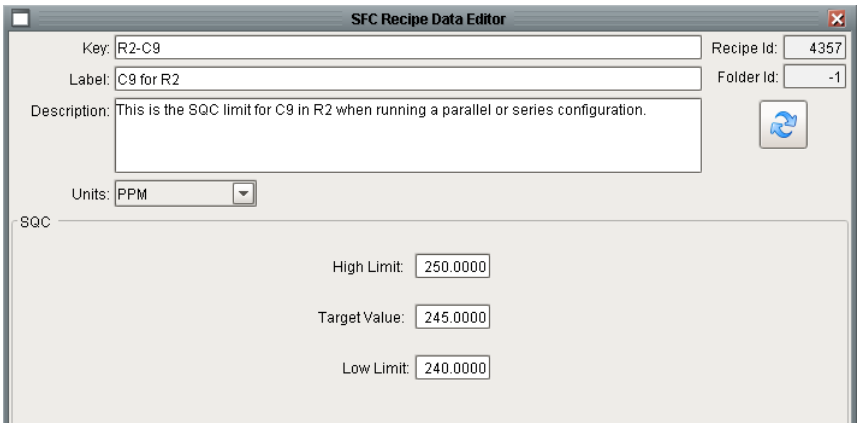
3.1.7 Simple Value

The Simple Value class is the most basic class of recipe data. It is used to store a single datum. It supports all of the data types and automatic unit conversion. It is commonly used for constants, a place for user inputs, and a place for the results of calculations.

Simple Data Recipe Editor

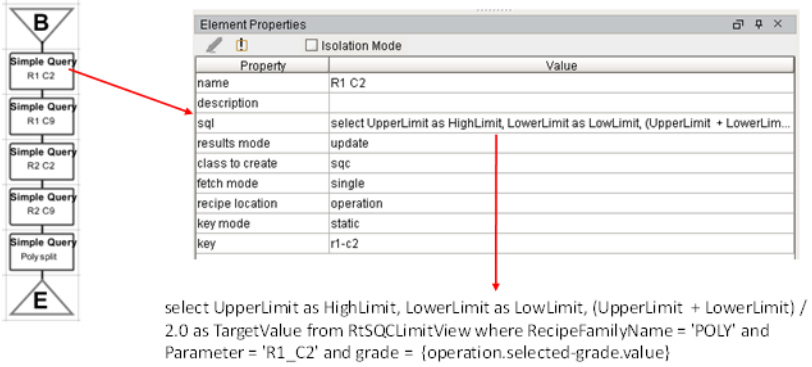
3.1.8 SQC

The SQC class of recipe data is used to store SQC limits. It is designed to be consistent with the SQC recipe data. It has target value, high limit, and low limit properties.



SQC Recipe Data Editor

This recipe data is often used with the Simple Query step to read the SQC limits from the recipe data and to store them into SFC recipe data for easy access from SFCs. This simple Query step is described in more detail in section 4.10.2.



SQC Recipe Data Typical Use

3.1.9 Timer

The Timer class of recipe data is significantly different from all other recipe data classes. All other recipe data classes merely provide a storage location for data. The Timer class implements a state model and behavior to implement a timer. The timer behavior is implemented in the *fetchRecipeData()* and *setRecipeData()* methods in *ils.sfc.recipeData.core*. These are the private methods that back the *s88Get()* and *s88Set()* public methods.

The purpose of the Timer class of recipe data is to provide stopwatch like functionality for the I/O steps; specifically the Download Monitor, Write Output, and PV Monitoring steps. The timer is controlled by setting the "command" attribute of the data using *s88Set()*. The set of commands is: *clear*, *start*, *stop*, *pause*, and *resume*. It is important to note that "command" is a virtual column and does not get stored in the database. The command does get translated and reflected in the state which is stored in the database. The "TimerState" values are: *running*, *cleared*, *paused*, and *stopped*. The elapsed time of the timer is via the virtual column "*elapsedMinutes*" which returns the time in minutes that the timer has been running. The returned value is calculated when the value is requested.

3.1.9.1 Unit Procedure State Awareness

Timers are also aware of the state of the unit procedure. Most importantly, if the unit procedure is paused then the timer stops running. When the unit procedure is resumed the timer resumes running without including the time that the chart was paused.

3.1.9.2 Time Scaling

When run in Isolation, Timers are automatically subject to time scaling. When configuring the isolation execution environment, one of the settings is a time acceleration factor. When setting or getting "TIME" recipe data this acceleration is automatically applied to scale the time. Refer to section 8.3 for a more in-depth discussion about isolation mode.

3.1.9.3 Timer Recipe Data versus Timestamps

The goal of Timer recipe data is to avoid using simple value recipe data to record the current time, either as a float, or a string, and then having subsequent steps use it and calculate the time difference between it and the current time and all the data type conversions. This is all provided by the timer. The Download Monitor, Write Output, and PV Monitor steps have properties to control the clearing and starting of the timer without having to write any custom Python. While it is common for a number of these blocks to cooperate, one is generally configured to start the timer.

3.1.9.4 Custom use of a Timer

While timers were designed with the I/O blocks in mind, a timer can easily be used for custom purposes and be explicitly controlled through *s88set()* API. The following example demonstrates complete control of a timer from action steps.

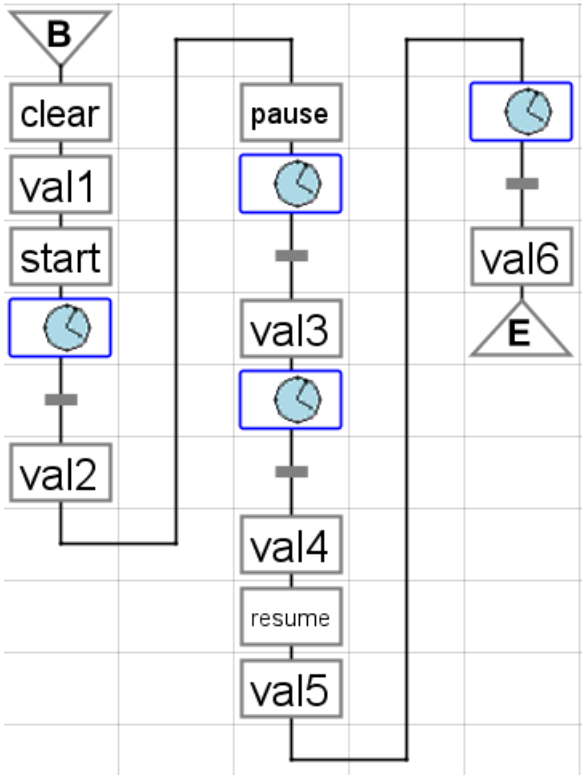
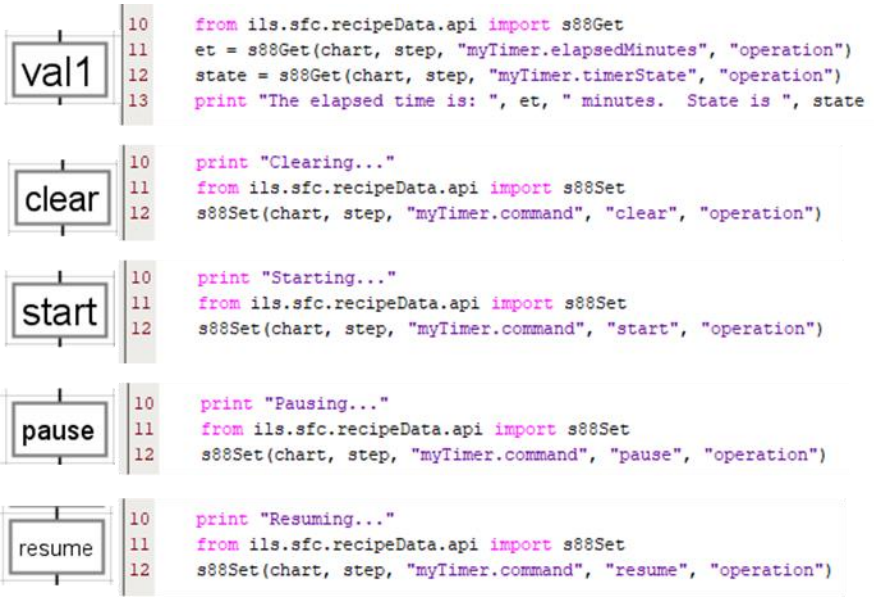


Chart to Test Timer Recipe Data Commands

The "val" steps are all identical, the other action steps do what their label indicates.



Step Details for Testing Timer Recipe Data

The performance is best shown by an excerpt from the wrapper log:

```
14:51:48 | Starting the unit procedure
14:51:48 | In ils.sfc.common.notify.sfcNotify(), Sending <sfcMessage> messa
14:51:58 | Clearing...
14:51:58 | The elapsed time is:  0.0  minutes.  State is  cleared
14:51:58 | Starting...
14:53:28 | I [U.U.Timer Operation          ] [19:53:28]: Step D1 in Use Ca
14:53:28 | The elapsed time is:  1.5  minutes.  State is  running
14:53:28 | Pausing...
14:53:33 | The elapsed time is:  1.5  minutes.  State is  paused
14:54:03 | The elapsed time is:  1.5  minutes.  State is  paused
14:54:03 | Resuming...
14:54:03 | The elapsed time is:  1.5  minutes.  State is  running
14:54:18 | I [U.U.Timer Operation          ] [19:54:18]: Step D4 in Use Ca
14:54:18 | The elapsed time is:  1.75  minutes.  State is  running
14:54:18 | The unit procedure is done
```

Wrapper Log Tracing Timer

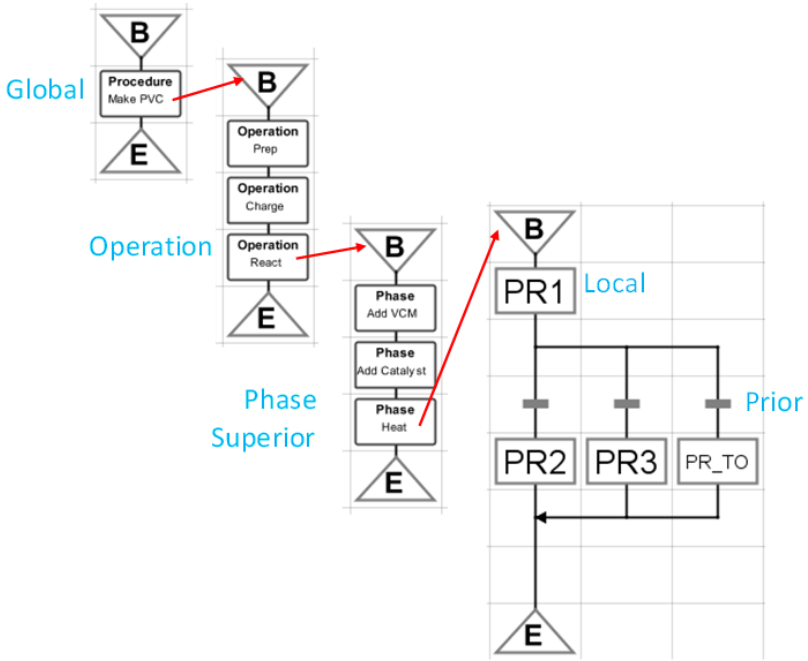
3.2 Recipe Data Scopes

Recipe data is associated with a step. The step effectively defines a name space. Recipe data within a name space / for a step must have a unique key. Of course recipe data can accessed elsewhere in the unit procedure by specifying the scope. The following scopes are supported:

Scope	Description
Global	The unit procedure superior to the step.
Operation	The operation superior to the step.
Phase	The phase superior to the step.

Superior	The enclosing step that called the chart on which the step is.
Prior	The step upstream from the current step. This is only valid for transitions.
Local	The step.
Reference	The key will name a chart variable that has the scope and key of the recipe data to use.
Tag	This is not a recipe data scope, but it is a choice in the location dropdown for many steps. This means that the value will come from/be written to a tag.

From the Action step named PR1's perspective, the scope locators would refer to the various steps as shown below. The transitions following PR1 can refer to PR1's recipe data using the *prior* scope locator.



Recipe Data Scope Locators

3.3 Referencing Recipe Data

The power of recipe data is that is can be conveniently referenced throughout the toolkit.

3.3.1 Referencing Recipe Data from Steps

Many steps are configured by specifying recipe data for one or more properties of the step. For example, the Yes/No task (section 4.7.2) can use recipe data to store the user response. Other

steps, such as the Enter Data (section 4.7.4) or PV Monitor (section 4.6.4) operate on numerous entities and have a configuration property that takes a variable number of recipe data entities.

3.3.2 Referencing Recipe Data from Action Steps

Recipe data can be fully accessed in the Python of an action step. The API is fully described in section Scripting API6.6. The most common interfaces are *s88Get()* for getting recipe data and *s88Set()* for setting recipe data, both in ils.sfc.recipeData.api:

```
val = getGet(chart, step, keyAndAttribute, scope)
getSet(chart, step, keyAndAttribute, val, scope)
```

3.3.3 Referencing Recipe Data from Transitions

Recipe data may be accessed in transitions using the following syntax:

{scope.key.attribute}

Where scope is one of the standard recipe data scopes.

Recipe Data Access from a Transition

As mentioned above, transitions are the only entity that can use the *prior* scope locator. One area of confusion is that there is also a scope locator, *previous*, which is used to reference a step property of the upstream step.

3.3.4 Dynamic Text Strings

There are a number of steps that post messages, warnings, and instructions to the operator. The steps have one or more text properties that define the text that will be displayed. An expression in curly braces in a text string will be substituted with the value of run-time chart data. The following steps support this: Queue Insert, Control Panel, and Simple Query. The form of the expression is scope, key, and attribute separated by periods, e.g. "{global.MyString.value}". Allowable scopes are tag, chart, step, and any recipe data scope (from ils.sfc.common.constants.py).

An example that substitutes the value attribute from the recipe data with attribute *value* and key *MyString* from *global* scope for a Control Panel step is:

Element Properties

Property	Value
name	CP7
description	Test a message with dynamic text
message	This is a message with some dynamic text <{global.MyString.value}>. How did that look?
ack required	☐
priority	Info
post to queue	☐
timeout	-1.0
timeout unit	SEC

Dynamic Text Substitution in an Control Panel Step

3.4 Engineering Unit Support

Many of the recipe data classes support Engineering Units. If Engineering Units are specified in the recipe data and a client requests the value in different units then the value is automatically converted. A client could be a step like Review Data or it could be Python called in an action step. The interfaces for accessing recipe data with unit conversions are *s88GetWithUnits()* for getting recipe data and *s88SetWithUnits()* for setting recipe data, both in *ils.sfc.recipeData.api*:

```
val = getGetWithUnits(chart, step, keyAndAttribute, scope, units)
getSetWithUnits(chart, step, keyAndAttribute, val, scope, units)
```

3.5 Creating a new Type of Recipe Data

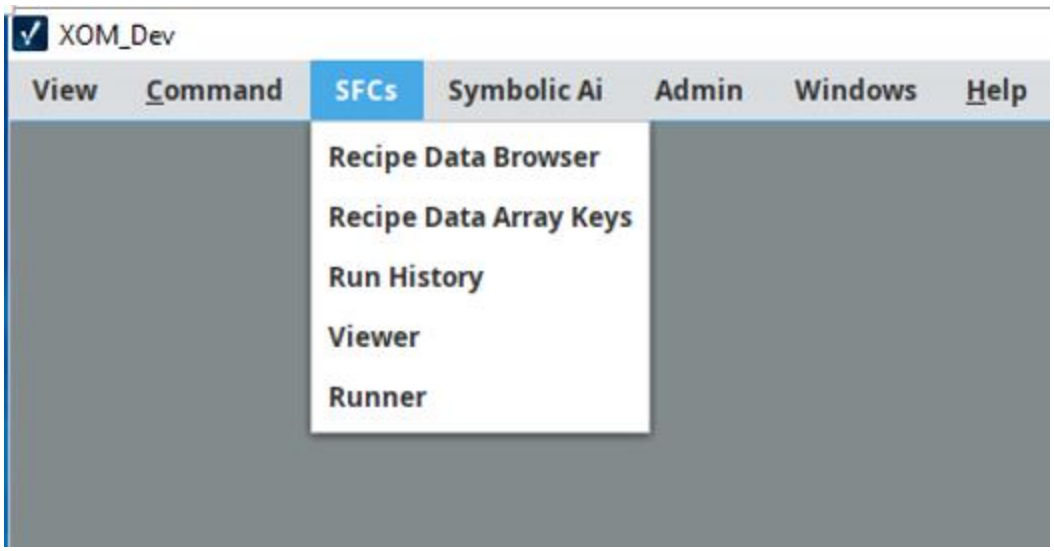
Creating a new type of recipe data is NOT a trivial task. This section provides an overview of the process of creating a new class of recipe data.

1. Define the new type by adding a record to SfcRecipeDataType
2. Design the new tables(s) and then create them in in SQL*Server.
3. Create a View patterned after SfcRecipeDataSimpleValueView.
4. Update the RecipeDataEditor Vision Window
    a. Add a dataset property for the a record from the view
    b. Add a new container to the window for the class specific properties.
5. Update Python supporting the Vision Window (ils.sfc.recipeData.visionRecipeDataEditor.py)
6. Add support for s88Set() and s88Get() to ils.sfc.recipeData.core.py
    a. fetchRecipeData()
    b. setRecipeData()

3.6 User Interface

The user interface for recipe data is implemented entirely using Vision windows and Python and therefore is only available in Ignition clients. While working with SFCs it will be necessary to use the Designer to construct the chart and the client to edit recipe data.

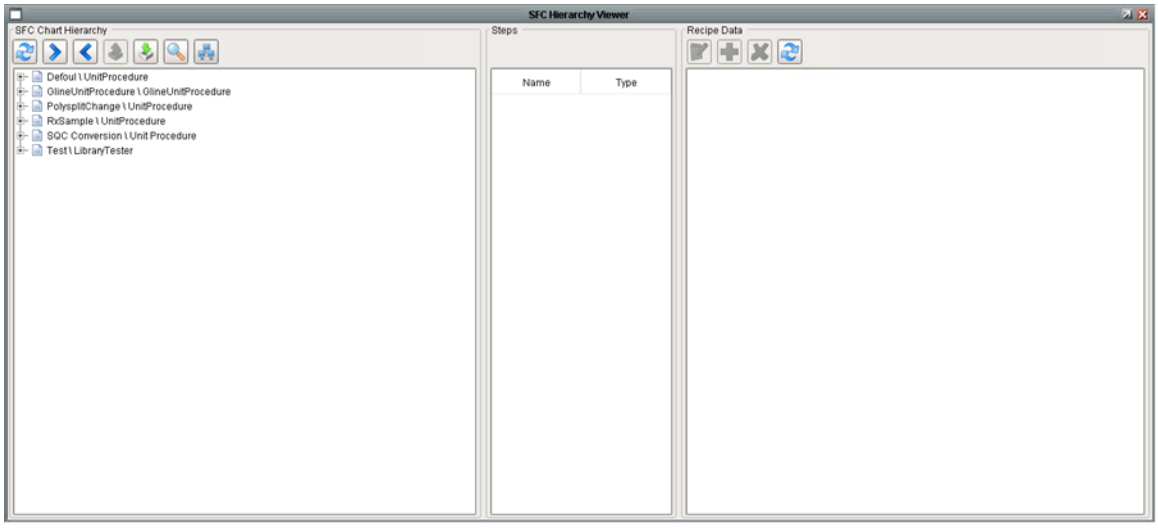
The user interface provides several screens which are accessed as follows:



Menu for SFCs

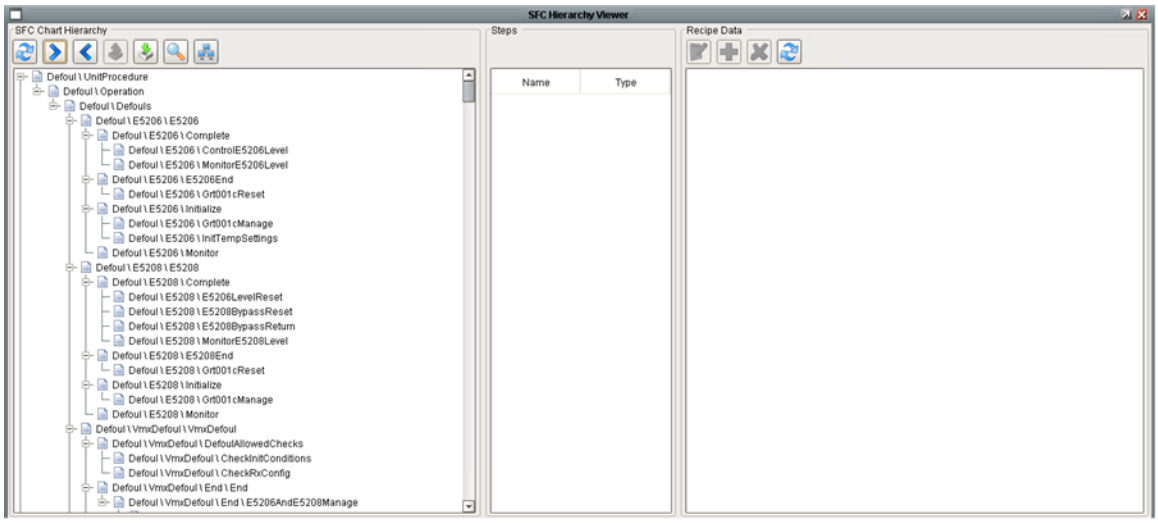
3.6.1 Recipe Data Browser

The Recipe Data Browser window serves two very important purposes. First, it shows the logical call hierarchy of charts. This is different from the physical organization of the charts in the project resource tree. The call hierarchy is shown in the left pane of the window. When the window opened, only the root charts are visible.



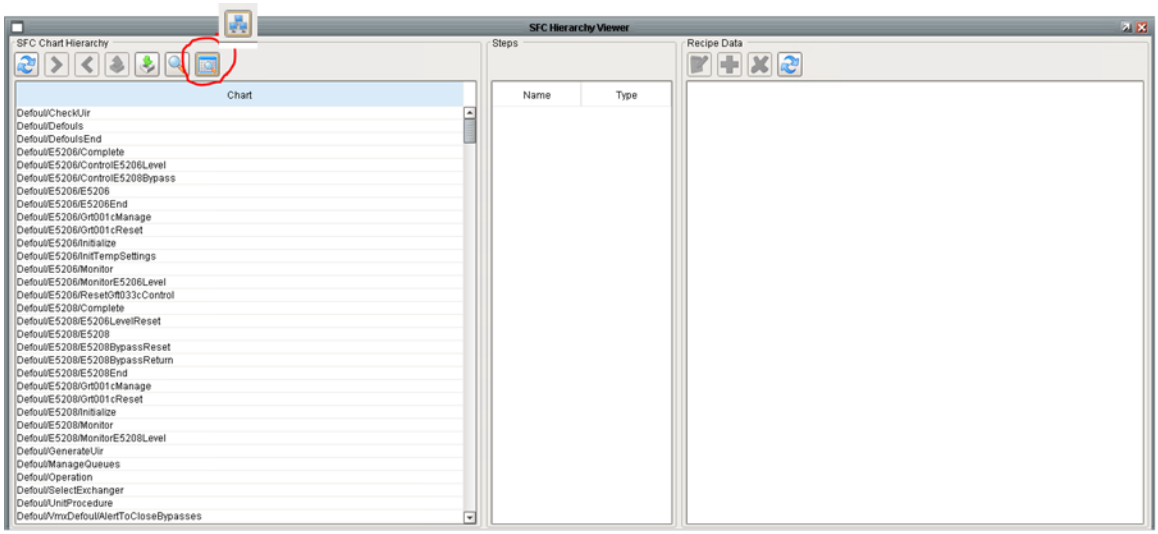
Recipe Data Browser in Tree Mode

Pressing the "+" to the left of a chart name will expand to show the charts called by that chart. Pressing the ">" button just above the view will expand the view of all charts as shown below.



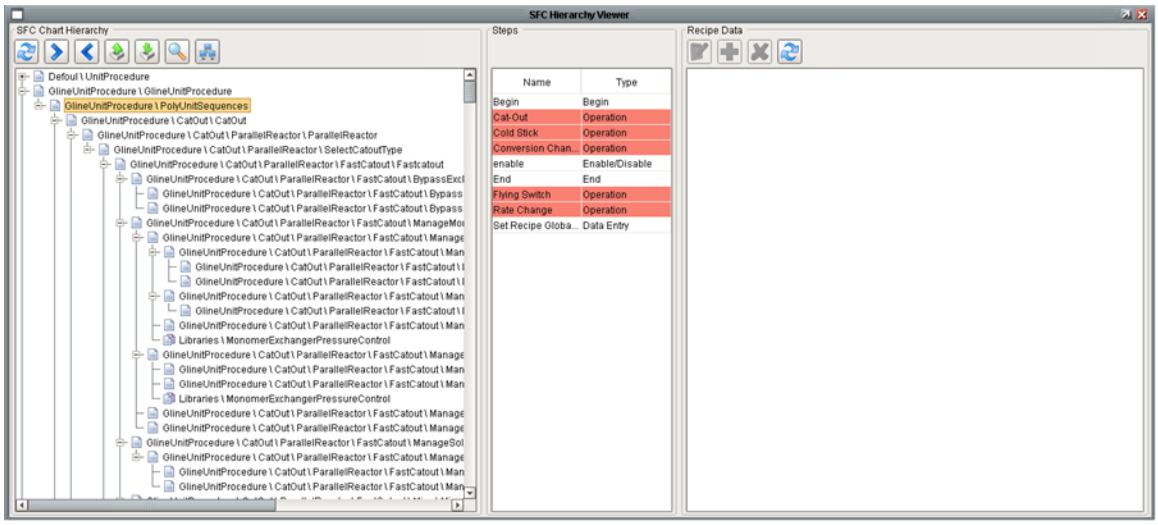
Recipe Data Browser in Expanded Tree Mode

There are two different modes for the left pane. The mode described above is the tree view. An alphabetical list view is available by pressing the "toggle view" button.



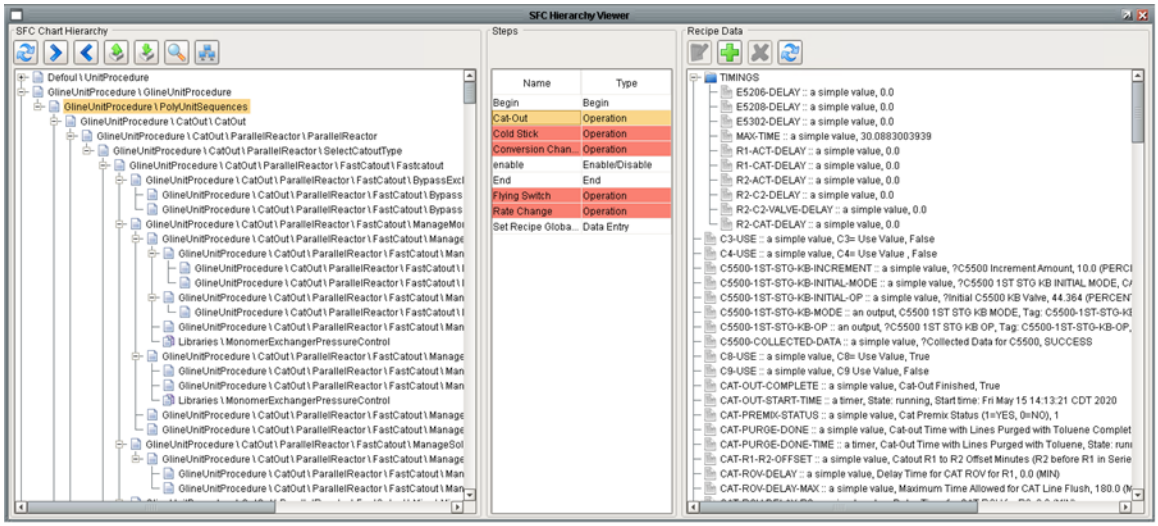
Recipe Data Browser in List Mode

The middle pane shows the steps of the chart selected in the left pane. The step name is highlighted in salmon if the step has recipe data.



Recipe Data Browser Showing Steps with Recipe Data

The right pane of the window shows recipe data for the step selected in the middle pane.

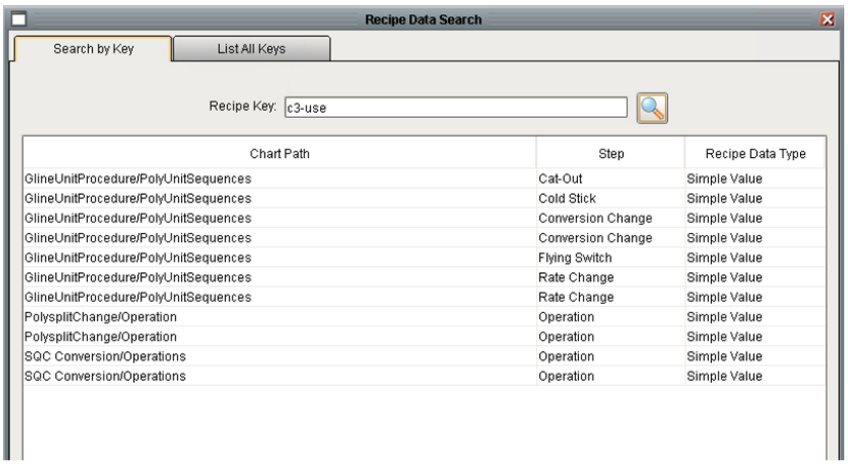


Recipe Data Browser Showing Recipe Data for Selected Step

Selecting any recipe data entity and pressing the notepad button opens the recipe data editor. The refresh button can be used to refresh the view in the event that the recipe data is being updated by a running chart or another user.

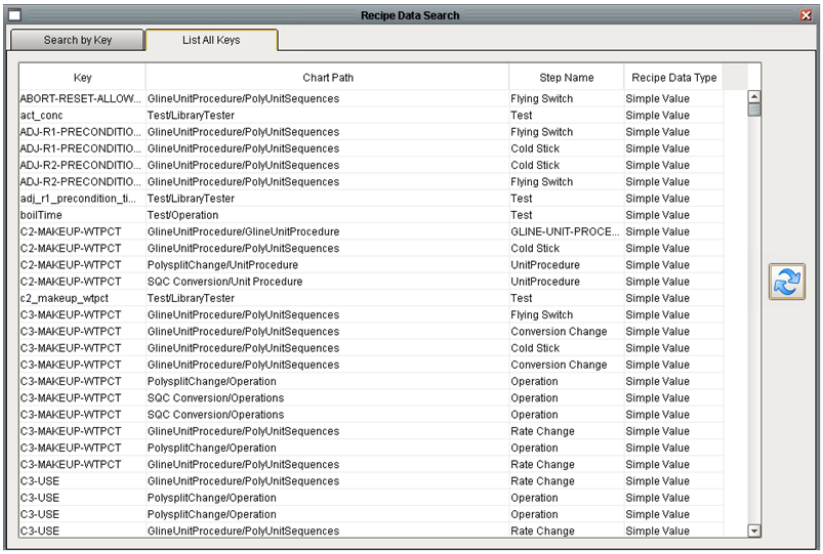
3.6.2 Recipe Data Search capabilities

Pressing the magnifying glass above the left pane of the Recipe Data Browser opens the Recipe Data Search window. The purpose of this window is to find all of the locations for a specific recipe data key. A common mistake is to define the same key on multiple steps and then to reference it inconsistently.



Recipe Data Search by Key

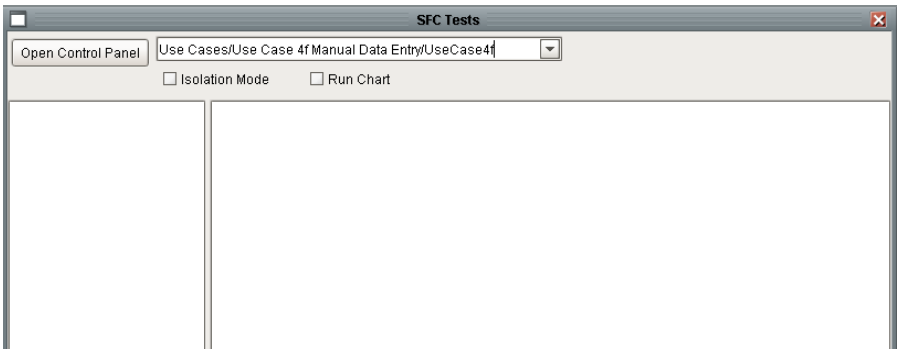
Alternatively, recipe data keys can be browsed by using the "List All Keys" tab. The table is initially sorted by key, but the table can be sorted by any column.



Recipe Data Bulk Search

3.6.3 SFC Runner

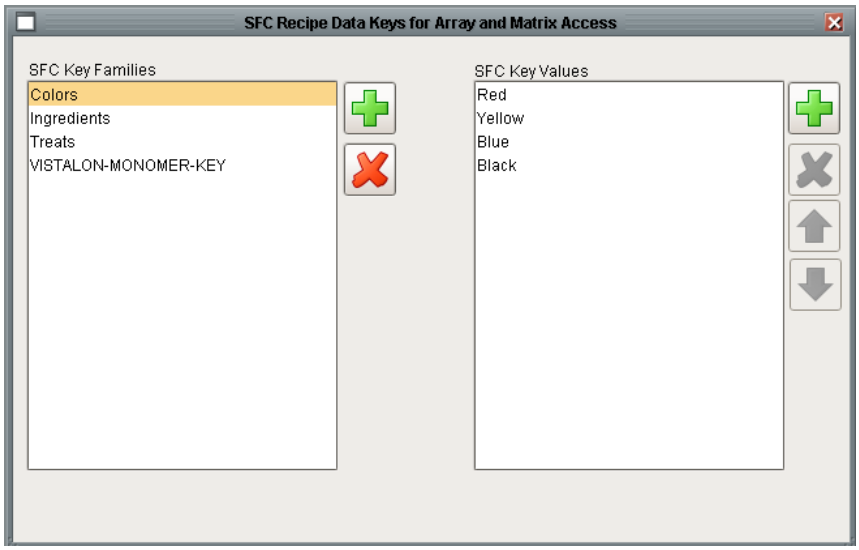
This screen is used for development to easily run a SFC from a client. It requires an entry in the *SfcNames* table. Running SFCs in production from this user interface is discouraged.



SFC Runner

3.6.4 SFC Recipe Data Array Keys

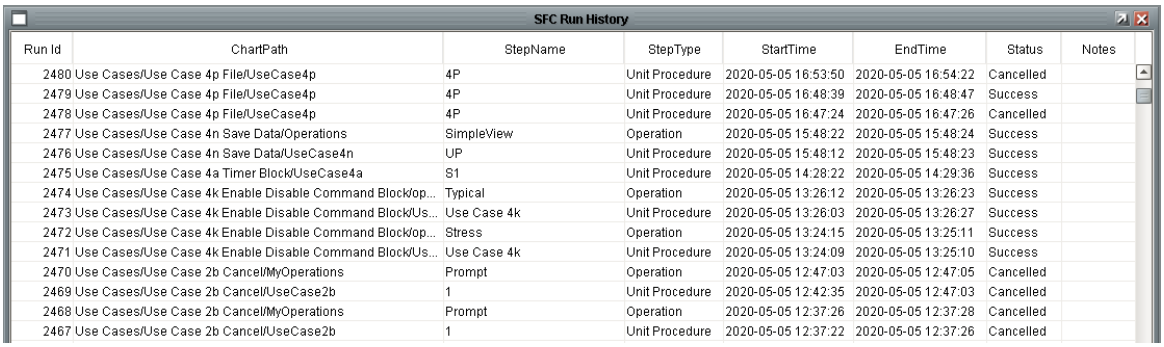
Recipe data keys are used to provide a consistent means to access arrays and matrices. Refer to section 3.1.1 and 3.1.3 for details. Keys are configured from the Admin -> SFCs -> SFC Recipe Data Array Keys menu which opens the following window. The left panel defines the key families and the right panel defines the key values in the family. The order of the values in the list defines the indices that will be used to access the arrays. The keys are used to translate from a symbolic index, *Blue*, to a physical index, *2*. Reordering the key's values once the data has been stored DOES NOT rearrange the data. When referencing an array using symbolic indices (*Red, Yellow, Blue, Black*) or numeric indices (*0, 1, 2, 3*) is equivalent.



Recipe Data Array Key Definition

3.6.5 SFC Run Log

This window provides a convenient way of viewing the history of unit procedures and operations. It shows the start and stop time and the completion status.



Run Id	ChartPath	StepName	StepType	StartTime	EndTime	Status	Notes
2480	Use Cases/Use Case 4p File/UseCase4p	4P	Unit Procedure	2020-05-05 16:53:50	2020-05-05 16:54:22	Cancelled	
2479	Use Cases/Use Case 4p File/UseCase4p	4P	Unit Procedure	2020-05-05 16:48:39	2020-05-05 16:48:47	Success	
2478	Use Cases/Use Case 4p File/UseCase4p	4P	Unit Procedure	2020-05-05 16:47:24	2020-05-05 16:47:26	Cancelled	
2477	Use Cases/Use Case 4n Save Data/Operations	SimpleView	Operation	2020-05-05 15:48:22	2020-05-05 15:48:24	Success	
2476	Use Cases/Use Case 4n Save Data/UseCase4n	UP	Unit Procedure	2020-05-05 15:48:12	2020-05-05 15:48:23	Success	
2475	Use Cases/Use Case 4a Timer Block/UseCase4a	S1	Unit Procedure	2020-05-05 14:28:22	2020-05-05 14:29:36	Success	
2474	Use Cases/Use Case 4k Enable Disable Command Block/op...	Typical	Operation	2020-05-05 13:26:12	2020-05-05 13:26:23	Success	
2473	Use Cases/Use Case 4k Enable Disable Command Block/Us...	Use Case 4k	Unit Procedure	2020-05-05 13:26:03	2020-05-05 13:26:27	Success	
2472	Use Cases/Use Case 4k Enable Disable Command Block/op...	Stress	Operation	2020-05-05 13:24:15	2020-05-05 13:25:11	Success	
2471	Use Cases/Use Case 4k Enable Disable Command Block/Us...	Use Case 4k	Unit Procedure	2020-05-05 13:24:09	2020-05-05 13:25:10	Success	
2470	Use Cases/Use Case 2b Cancel/MyOperations	Prompt	Operation	2020-05-05 12:47:03	2020-05-05 12:47:05	Cancelled	
2469	Use Cases/Use Case 2b Cancel/UseCase2b	1	Unit Procedure	2020-05-05 12:42:35	2020-05-05 12:47:03	Cancelled	
2468	Use Cases/Use Case 2b Cancel/MyOperations	Prompt	Operation	2020-05-05 12:37:26	2020-05-05 12:37:28	Cancelled	
2467	Use Cases/Use Case 2b Cancel/UseCase2b	1	Unit Procedure	2020-05-05 12:37:22	2020-05-05 12:37:26	Cancelled	

SFC Run Log

4. STEP DEFINITIONS

This section describes the performance of each step. It also defines behavior common to all steps.

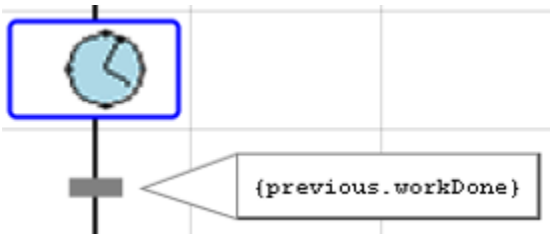
4.1 Common Step Behavior, Best Practices, and Gotchas

The section describes behavior that is common to all steps or a subset of steps.

4.1.1 Long-Running Steps and workDone

Some steps run for a long time, or may wait a long time for user input. In order to work correctly all such steps must be followed by a transition with the expression {`previous.workDone`}, as shown below.

It is very import to pay attention to the case of the step variable: *workDone*. The uppercase "D" is required!

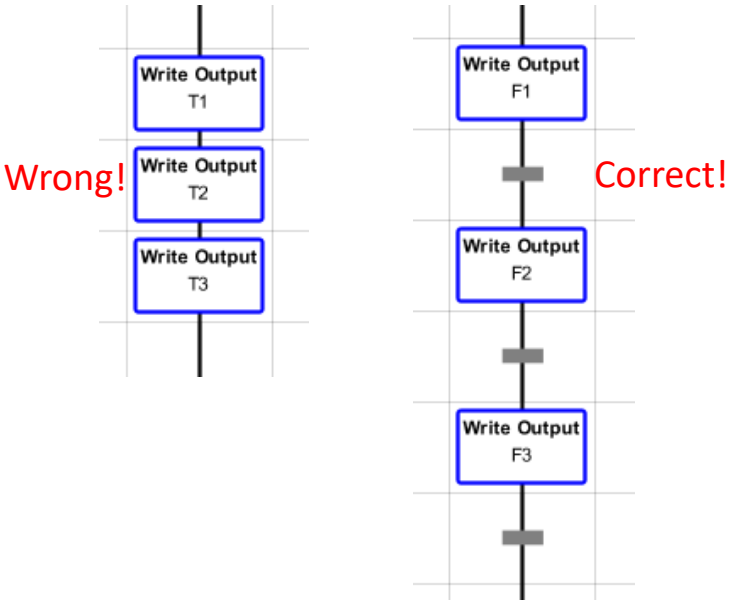


Long Running Step workDone Flag

Long running steps types include: Control Panel Message, Data Entry, Get Input, Notify Dialog, Review Data, Review, Review Flows, Time Delay, PV Monitor, Write Output, and Yes/No.

In order to provide a visual cue, all long running steps have a blue border.

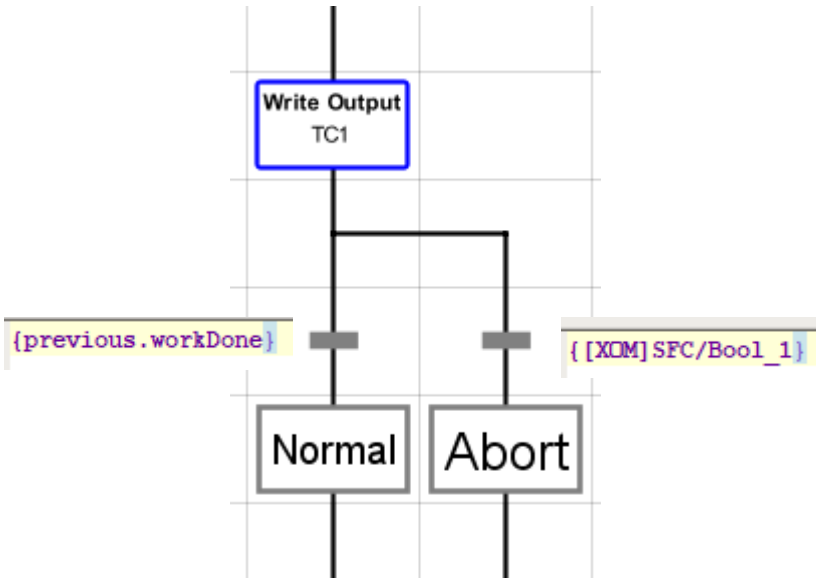
Another example shows a series of write output steps. Assume that the goal is to write 3 different sets of outputs. Each set of writes must complete before proceeding to the next set. It is imperative that a transition follow the Write Output as shown on the right.



Right and Wrong Examples of Long Running Steps

The transition that typically follows a long-running step is configured with `{previous.workDone}` which monitors a step variable that is set by the step. The assumption is that the step knows how much work it has to do and will set the `workDone` flag when all of the work is complete.

Another use of the transition that follows a long-running step is to allow the step to be terminated. Assume that the Write Output step below is configured to write several timed outputs and will run for a long time. The path on the left is for the normal successful completion of the writes. The path on the right monitors a Boolean tag (set by some external logic) and when it becomes True the Write Output step will be aborted immediately.

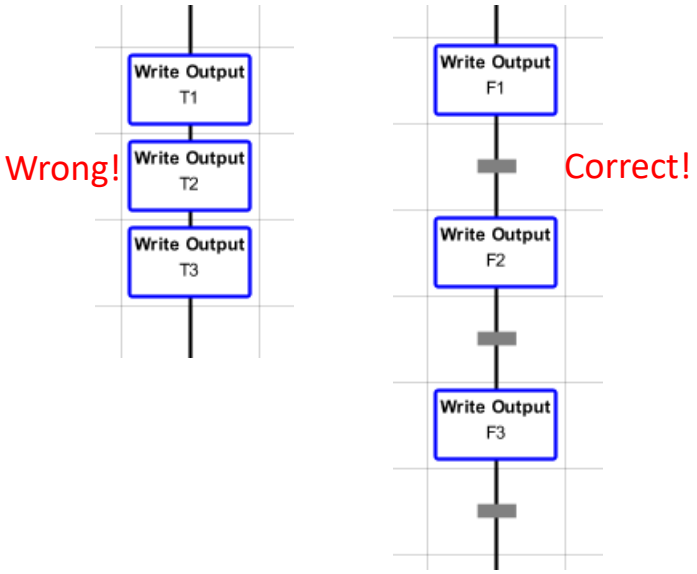


Aborting a Long Running Step

4.1.2 Timeout Handling

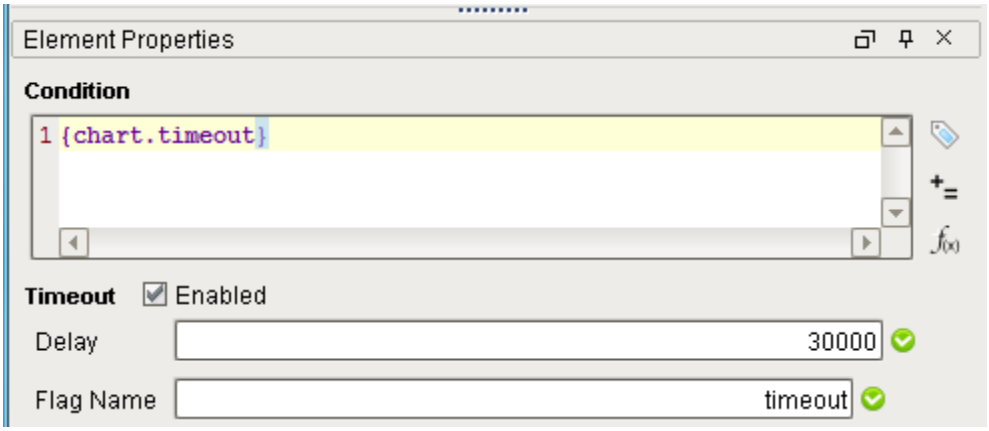
Timeouts may be appropriate for any step that waits for user input, an event, or a tag value. In general, any long running step can coordinate with a timeout handler. Timeout handling uses standard Ignition facilities. A timeout is configured on a transition that follows the step. In this way the block does not have or need any knowledge about the timeout. All the step knows it that it has been deactivated by the SFC engine.

The following example, taken from Use Case 4b demonstrates how the Yes/No step works with the standard timeout pattern:



Standard Timeout Handling Pattern

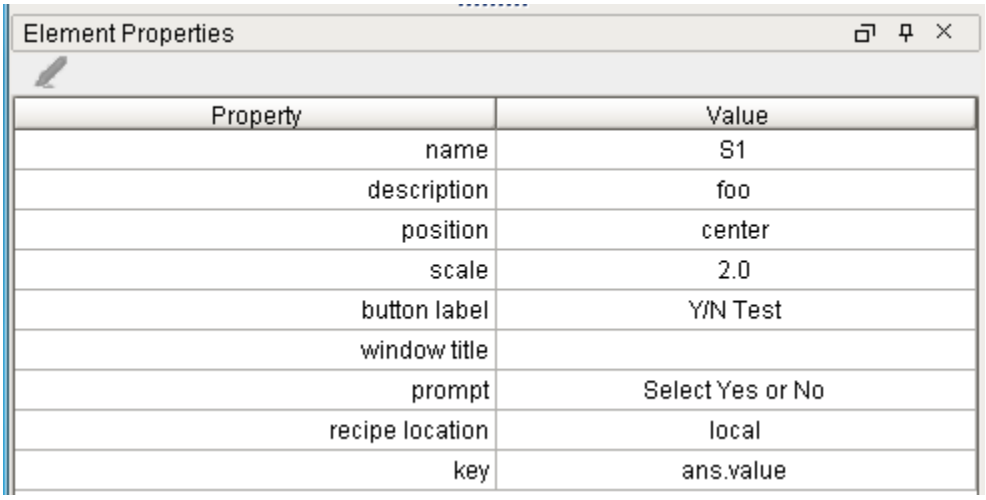
A 30 second timeout is implement in the 3rd transition. The bottom portion of the transition defines the time. The time units for the delay are in milliseconds, so this transition defines a 30 second timeout. When the transition has run for 30 seconds it sets the chart scope variable: *timeout*, to True. The condition that the transition checks the same variable.



Transition Configured to Set and Check for Timeout

Taking a closer look at what happens reveals that as soon as the Yes/No step starts running all of the transitions following the step also start running. Each transition maintains a timer of how long it has been running. The first transition that evaluates to True wins and the other transitions stop running. If none of the transitions have become True after 30 seconds, then the flag becomes True, the timeout transition becomes True, the other transitions stop executing, a deactivate command is sent to the Yes / No block, giving it time to do cleanup, and the step connected to the output of the transition starts executing.

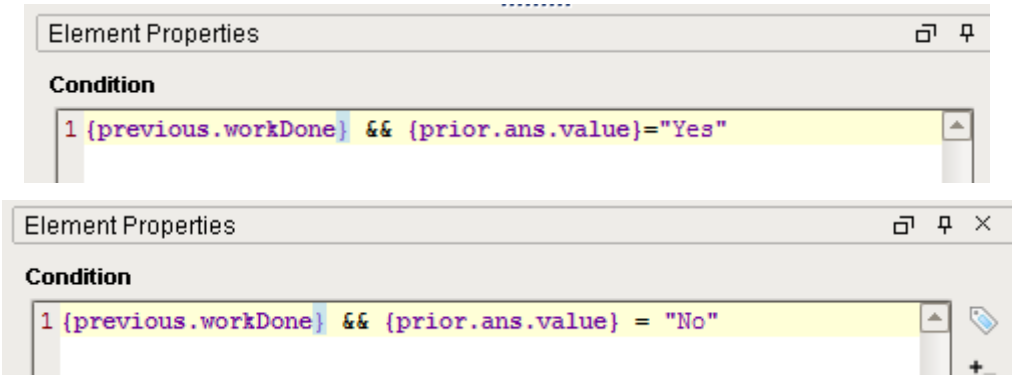
The Yes/No step does not have any properties associated with the timeout:



Property	Value
name	S1
description	foo
position	center
scale	2.0
button label	Y/N Test
window title	
prompt	Select Yes or No
recipe location	local
key	ans.value

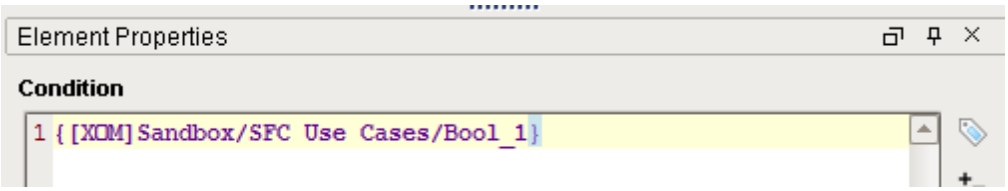
Yes / No Step Properties

The "Yes" and "No" transitions are configured as follows:



Transition Configuration

The "Cancel" transition is used to represent monitoring of some external condition that would override waiting for the user's response. In this case it is simply monitoring a Boolean tag. When this transition becomes True, it will cancel the Yes/No step which will in turn clean up any windows that may be shown on various clients.



Transition Configured to Cancel the Yes / No

Whether the transition times out or the chart is cancelled by an external event, the step shuts down in the same manner. In other words, the step does not know why it was deactivated.

4.1.3 Engineering Unit Support

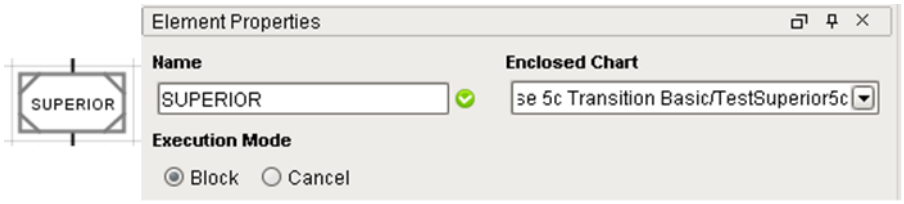
The Standard palette contains the steps provided by Inductive Automation. Refer to the Ignition documentation for details.

4.1.4 Previous vs Prior Scope Locator

The previous and prior scope locators but do two entirely different things. The "previous" locator refers to step variable defined in the upstream task and "prior" defines recipe data defined on the previous task. Both are valid only in transitions.

4.1.5 Enclosing Step Configuration

The enclosing step, a standard Ignition step, has an execution mode that is critical to the performance of the block.



Enclosing Step and its Properties

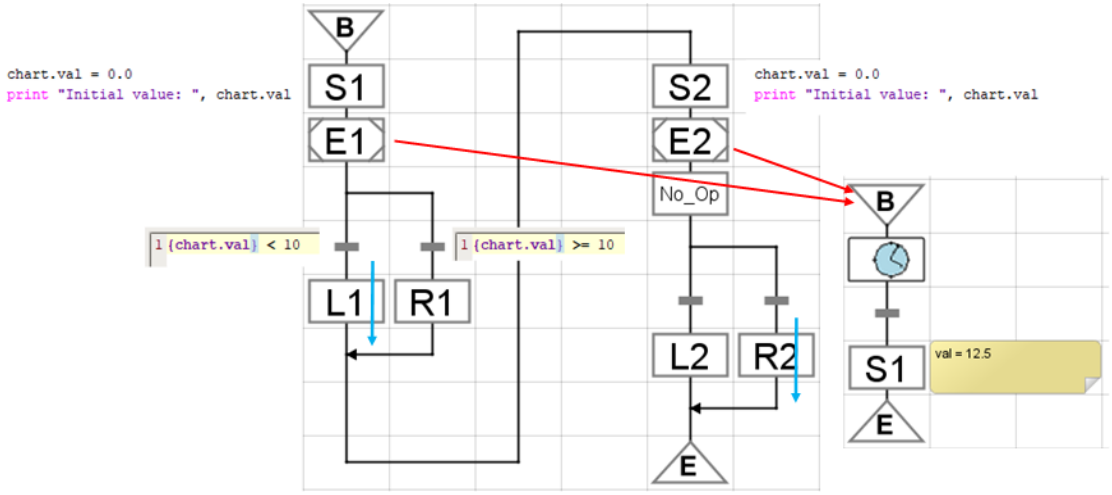
The common practice by experienced users is to always set this property to "Block". The Ignition documentation describes these properties as follows:

> When flow reaches an enclosing step, it starts its enclosed chart. Using the enclosing step's Execution Mode property, the step can be configured to work in one of two very different ways:
>
> **Execution Mode = Block**
> Let the enclosed chart run to completion. This means that the enclosed chart should have an **End Step** in it, and that flow will not be able to move beyond the enclosing step until the enclosed chart stops by reaching its end step.
>
> **Execution Mode = Cancel**
> Cancel the subchart when the enclosing step is ready to stop. This means that the subchart is canceled when flow is ready to move beyond the enclosing step. Any running steps in the enclosed chart are told to stop, and flow ceases in the enclosed chart.

Both modes are valid and but it is important understand the difference. It is also important to note that the default value is "Cancel", which is not the mode normally used in our implementations.

4.1.6 Transitions Following an Enclosing Step

A fundamental understanding of transitions is important when designing charts. Transitions begin running when the upstream step begins running. The transitions stop running when any one of them becomes *True*. This may lead to unexpected results when a transition, or multiple transitions, follow an enclosing step that is configured in block mode. Assuming that the chart called by the enclosing step runs for a long time, the presumption is that the transitions will be evaluated when the enclosed chart completes using the tag values and/or recipe data values at the time that the chart completes. But this is NOT the way it works! In order to force the transitions to evaluate after the enclosing step completes you need to insert a dummy step as shown below. It calls the same enclosing chart followed by the same transitions. The only difference is that the second enclosing step is followed by an action step labelled "No_Op", which does nothing. In the first case the left path, containing the action step "L1" is taken. In the second case, the right path, containing the action step "R2" is taken. The transitions check the value of the chart scope variable: val. The steps labelled S1 and S2 initialize the variable to 0.0. Step S1 on the enclosing chart set the variable to 12.5 after a 10 second delay.



No Op between Enclosing Step and Transitions

The performance is documented in the wrapper log:

```
17:17:50 | I [U.U.UseCase5c
17:17:50 | Starting operations
17:17:50 | Initial value:  0.0
17:18:00 | Local value:  12.5
17:18:00 | Left - val =  12.5
17:18:00 | Initial value:  0.0
17:18:10 | Local value:  12.5
17:18:10 | Right - val =  12.5
17:18:10 | operations finished
17:18:10 | Ran to completion!
```

4.1.7 Persistent Windows

A basic requirement of the application is to be able to recover the windows that were present if the client is closed either deliberately or as a result of a client failure. Also, if a control panel for a currently running chart is opened on a second client, all windows visible to the operator will be visible on this second client as well.

This functionality is supported by storing all information for currently open windows in the database. Each open window has a record in the *SfcWindow* table, as well as a record in one or more window-specific tables. For instance, the Yes/No step will have a record in the *SfcInput* table as well.

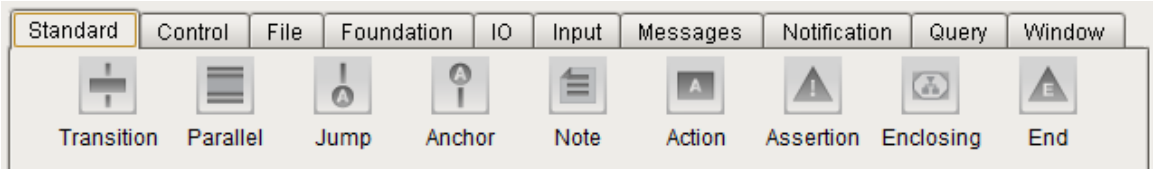
When a client reconnects, or if a new client connects, the windows will not automatically reopen but they will all be registered with the SFC control panel. If the operator opens the control panel they can then press the toolbar buttons which will reopen the window using data in the database.

4.1.8 Multiple Client Support

If multiple clients are "interested" in an SFC, any window that is posted will be displayed on multiple clients. If the purpose of the window is to get operator input then the first window to respond wins. As soon as the gateway receives the first response it will close the window on all other clients.

4.2 Standard Palette

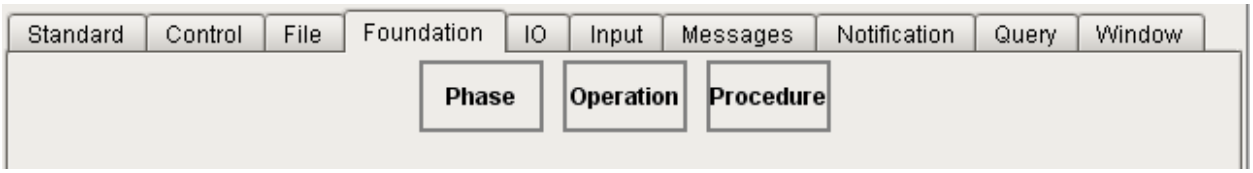
The Standard palette contains the steps provided by Inductive Automation. Refer to the Ignition documentation for details.



Standard Step Palette

4.3 Foundation Palette

The foundation palette contains the Unit Procedure, Operation, and Phase steps. Refer to section 2.2 for more details about these steps.

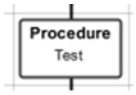


Foundation Step Palette

These blocks are used to organize charts, tasks, and recipe data into a top down structure. Each of these steps are special types of a standard Enclosing step. Once placed on a chart each of these must be configured with the name of a chart that will be called. Unlike a pure Ignition Enclosing step, these steps do not support argument passing. All of these steps operate in "block" mode whereas a task encapsulation can be configured to operate in "block" or "cancel" mode. In addition to organizing charts, these steps play an important role in organizing recipe data which is discussed in detail in section 3.

4.3.1 Unit Procedure Step

The Unit procedure is the highest organizational level step. A Unit procedure generally calls a chart with one or more operations. A unit procedure and its properties are shown below. A unit procedure references a message queue which is used by the Message Queue steps that are used in the unit procedure. The chart-path is a reference to the chart that will be called.



Property	Value
name	Test
description	
chart-path	Use Cases/Use Case 4n Save Data/Operations
msg queue	SFC-Message-Queue

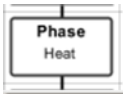
4.3.2 Operation Step

The Operation step is the next highest organizational level. An operation generally calls a chart with one or more phases, or if the phase level is not used, then tasks and task enclosures. An operation and its properties are shown below. When an operation runs, the queue specified for the operation will be written to the unit procedure. This allows a unit procedure with multiple operations to use a separate queue for each operation. Remember that by definition, only one operation can run at a time.

Property	Value
name	File Only
description	
chart-path	Use Cases/Use Case 4n Save Data/File Only Test
msg queue	SFC-Message-Queue

4.3.3 Phase Step

A phase is the next level of organization. The only property for a phase is the chart-path of the chart it will call. Unlike operations, phases are encouraged to run in parallel.



Property	Value
name	Heat
description	
chart-path	Use Cases/Use Case 1 Recipe Data Access/My Phase

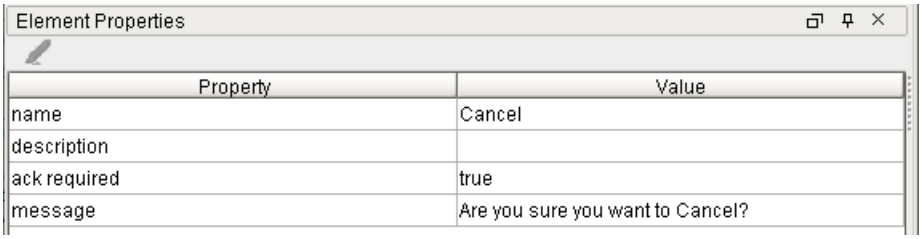
4.4 Control Palette

The control palette has four steps which control execution of the SFC.

Control Step Palette

4.4.1 Cancel Step

The Cancel step immediately cancels the unit procedure from the top down. This will invoke "on Cancel" handlers from the bottom up. The step has the following configuration:



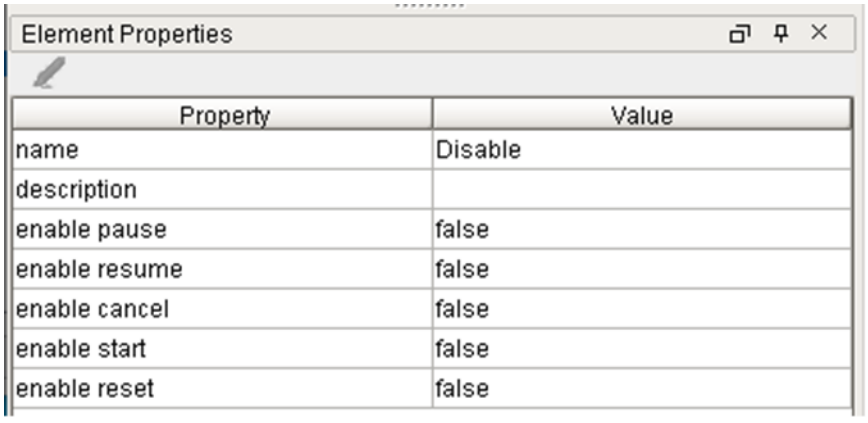
The "*ack required*" property can be used to require confirmation from the operator before cancelling using the "*message*" property as a prompt.

The "ack required" property is NOT currently implemented.

4.4.2 Enable/Disable Step

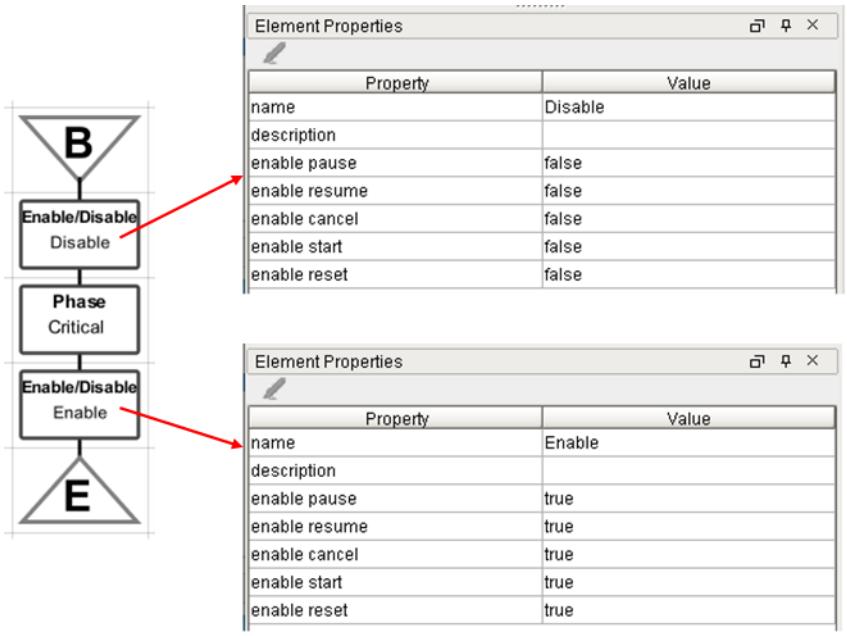
The Enable/Disable step is used to enable or disable the command buttons on the control panel. It does not affect the ability to issue commands via the API or via the command steps described above. The step does not alter the run state of a chart, rather, it configures the ability of the operator to alter the run state of a chart.

The step has the following step properties. The properties correspond to the buttons on the control panel.



A typical use of this block is shown below where a critical section of a chart is "protected" from interruptions from the operator while it is executing by disabling commands before the critical code and enabling them after the critical code. Extreme care should be exercised when using this

technique because by disabling the commands on the control panel there is nothing the operator can do to correct a bad situation. It is recommended that the critical code use error handlers and be designed to "expect the unexpected."

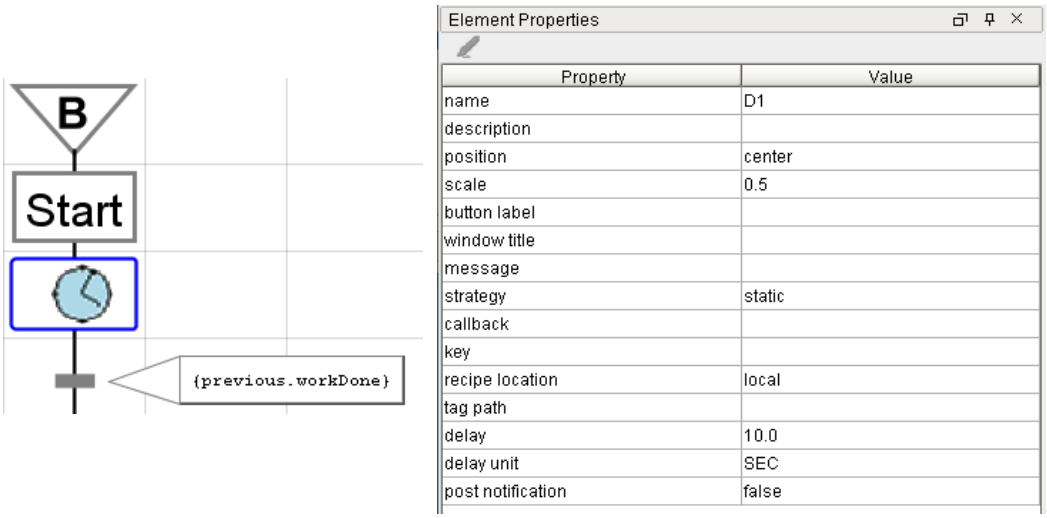


4.4.3 Pause Step

The Pause step immediately pauses the unit procedure from the top down. This step does not have any step specific properties. Once a chart is paused, the normal way to resume execution is from the control panel by pressing the resume button. Execution can also be resumed by using the chart execution viewer although this is not recommended because it is easy to resume execution from a chart other than the top. In addition to pausing the execution of charts, timers, which are commonly used with I/O, will also be paused.

4.4.4 Time Delay Step

The time delay is a commonly used step to perform a delay. The step must always be followed by a transition checking for completion. The delay step properties are shown below.



Delay Step and Properties

The properties are:

Property	Description
Position	Specifies the position of the notification window when "post notification" is true. Choices are center, topLeft, topCenter, topRight, bottomLeft, bottomCenter, bottomRight
Scale	Scale factor that the window will be scaled by when "post notification" is true. 1.0 is full scale.
Button Label	Label for the button that will be added to the control panel for the window when "post notification" is true. The button is useful if the window is accidentally closed, if the client is accidentally closed, or if a new client connects after the step has started but before it has completed.
Window Title	The title for the window when "post notification" is true.
Message	The text that will be displayed in the notification window when "post notification" is true.
Strategy	Choices are: static, recipe, tag, callback
Callback	Name of a Python script, with no arguments, that returns a float value that specifies the delay time subject to the units configured in the step.
Key	If the strategy is recipe, then this specifies the key and attribute that contains the value of the delay.
Recipe Location	If the strategy is recipe, then this specifies the scope of the recipe data. Choices are: global, operation, phase, superior, local, and reference.

Property	Description
Tag Path	If the strategy is tag, then this specifies the tagpath that contains the value of the delay. The tag should be a float.
Delay	Static value that specifies the delay time subject to the delay units.
Delay Unit	Specifies the units of the delay, regardless of how the delay is specified. Choices are SEC, MIN, HR
Post Notification	A Boolean that determines if a notification window is posted to the client.

Dynamic Values: Regardless of the strategy, except for static, the delay is checked every second from the source. This allows the delay to be dynamic assuming that there is a mechanism that manipulates the tag, recipe data, or the callback performs a calculation.

Hard Coded Value: The configuration of the Time Delay block that specifies a static value of 5 seconds is shown below.

Property	Value
name	S2
description	
g2 xml	
position	center
scale	0.5
button label	
window title	
message	
strategy	static
callback	
key	
recipe location	local
tag path	
delay	5.0
delay unit	SEC
post notification	☐

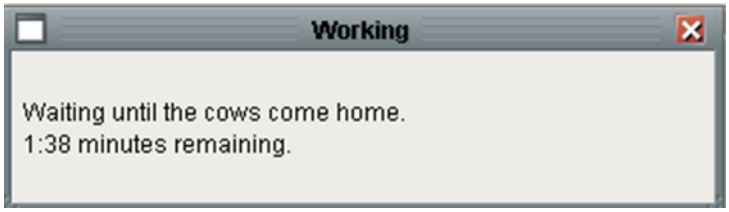
Time Delay Configuration with a Static Delay

Delay with Notification: Several properties are only relevant when using notification. There properties are: position, scale, button label, window title, message, and post notification. These are all used to specify the look and feel of the notification window.

Property	Value
name	S2
description	
position	topRight
scale	1.0
button label	Test
window title	Working
message	Waiting until the cows come home
strategy	static
callback	
key	
recipe location	local
tag path	
delay	20.0
delay unit	SEC
post notification	☑

Time Delay with Notification

When the step runs it will post the following window on the client. The window is updated to display how much time is remaining.



Time Delay Notification Window

Callback: The procedure that is called must return a single float value that specifies a time delay, in seconds, from when the step started to run. The callback, which does not take any arguments, is called every second until the elapsed time is greater than the float value returned from the callback. Because the callback is called every second, the callback can be used to specify a very dynamic delay that can be altered as the delay is underway.

Property	Value
name	S4
description	
g2 xml	
position	center
scale	0.5
button label	
window title	
message	
strategy	callback
callback	ils.sfc.tests.useCase4a.delayCallback
key	
recipe location	local
tag path	
delay	0.0
delay unit	SEC
post notification	☐

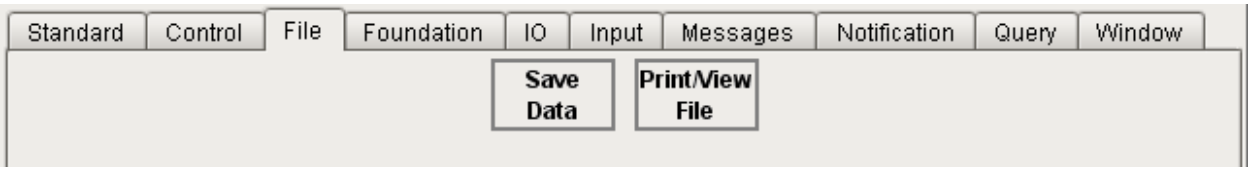
useCase4a

```
'''
Created on Sep 8, 2015

@author: Pete
'''
def delayCallback():
    print "In ils.sfc.tests.useCase4a.delayCallback()..."
    return 12.3
```

Time Delay Callback

4.5 File Palette

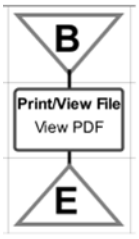
The File palette contains two steps. One for viewing and printing a file and another for saving recipe data to a file.



File Palette

4.5.1 Print/View File Step

This step is used to view and/or print a pre-existing file. This block does not create, write, or update the file in any way. It can be used to print a file that is created dynamically from a custom callback or any other mechanism, as long as the file exists before the block runs.



Property	Value
name	View PDF
description	
position	center
scale	1.0
button label	E/R
window title	E/R Diagram
file location	client
filename	C:\temp\TEST.pdf
print file	false
show print dialog	false
view file	true

Print / View File Step and Properties

The behavior of this step is largely drive by the location of the file. The behavior is summarized in the following table:

File Location	Print File	View File	Description
Gateway	True	True	File is printed by the gateway. File is opened in the gateway and stored in SfcSaveData table, client reads database and displays in window in window.
Gateway	True	False	File is printed by the gateway.
Gateway	False	True	File is opened in the gateway and stored in SfcSaveData table, client reads database and displays in window in window.
Gateway	False	False	Valid but useless configuration
Client	True	True	File name is stored in SfcSaveData table, client opens file in window. Printing is at the discretion of the client.
Client	True	False	Invalid configuration, since client printing is only done at the discretion of the client to avoid one print for each client.
Client	False	True	File name is stored in SfcSaveData table, client opens file in window.
Client	False	False	Valid but useless configuration

When viewing a file, a PDF file will be displayed in the PDF viewer widget and everything else will be displayed in a Document Viewer widget. Graphics files are not supported.

4.5.2 Save Data Step

This block is used to save the recipe data to a file. This task is useful for archiving the state of recipe data before or after a critical point in a recipe, at the start of execution or upon completion

of an operation or unit procedure. When the step runs a file will be created. The file may optionally be printed and/or viewed. The step and its step properties are shown below.

Element Properties

Property	Value
name	View
description	
position	center
scale	1.0
button label	Recipe
window title	Pete Recipe Data
recipe location	global
directory	c:\temp
filename	view
extension	pdf
timestamp	false
print file	false
show print dialog	false
view file	true

Save Data Step and Properties

The properties are:

Property	Description
Position	If *View File* is True, specifies the position of the window. Choices are center, topLeft, topCenter, topRight, bottomLeft, bottomCenter, bottomRight
Scale	If *View File* is True, specifies the factor by which the window will be scaled. 1.0 is full scale.
Button Label	If *View File* is True, specifies the label for the button that will be added to the control panel for the window. The button is useful if the window is accidentally closed, if the client is accidentally closed, or if a new client connects after the step has started but before it has completed.
Window Title	If *View File* is True, specifies the title for the window.
Recipe Location	Specifies the scope of the recipe data that will be included in the report.
Directory	Path to the directory where the file will be written.
Extension	This is used to determine the format of the file. There are three choices: csv, rtf, and pdf. The extension txt will be mapped to csv
Timestamp	Species if a timestamp will be appended to the filename. This is convenient for archiving recipe data every time a unit procedure runs.

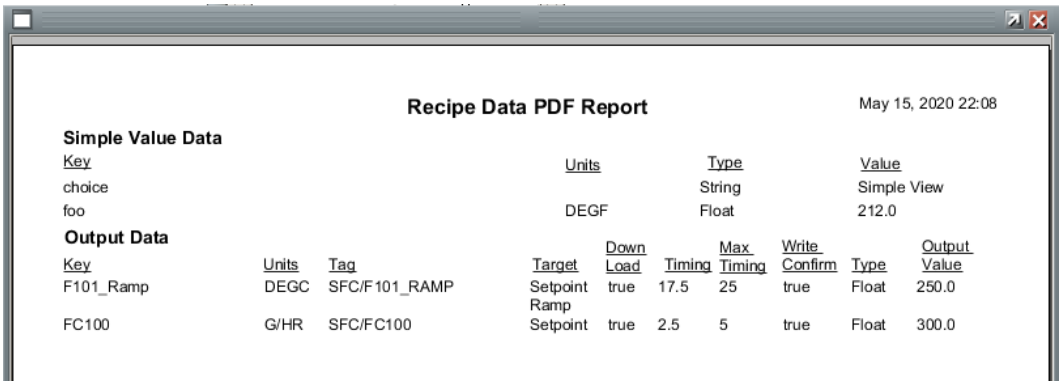
Property	Description
Print File	If True, the file will be printed (to the default printer I think)
Show Print Dialog	If True, then the Operating System Print window will be posted. This requires that a client be connected and should not be used for SFCs that are run automatically without operator interaction.
View File	If True, then the report will be displayed on the client's window. The report will always be displayed as a PDF, the Extension property does not influence how the file will be viewed.

This example shows a step that is configured to save the global recipe data in CSV format to a file named c:\temp\recipe-*.csv where the * will be replaced with a timestamp at run time. The file will also be displayed to the client in pdf format.

Property	Value
name	ViewPDF
description	
position	center
scale	1.0
button label	Recipe
window title	Recipe Data PDF Report
recipe location	global
directory	c:\temp
filename	recipe
extension	csv
timestamp	true
print file	false
show print dialog	false
view file	true

Save Data Step and Properties

When the step runs, the following window opens on every interested client:



Recipe Data PDF Report May 15, 2020 22:08

Simple Value Data

Key	Units	Type	Value
choice		String	Simple View
foo	DEGF	Float	212.0

Output Data

Key	Units	Tag	Target	Down Load	Timing	Max Timing	Write Confirm	Type	Output Value
F101_Ramp	DEGC	SFC/F101_RAMP	Setpoint Ramp	true	17.5	25	true	Float	250.0
FC100	G/HR	SFC/FC100	Setpoint	true	2.5	5	true	Float	300.0

Save Data Report

In addition the following file is written:

recipe-202005152208.csv

	A	B	C	D	E	F	G	H	I	J	K	L
1	Key	Simple Va	Units	Type	Value							
2	choice		String	Simple View								
3	foo	DEGF	Float	212								
4	Key	Output Da	Units	Tag	Target	Down	Timing	Max Timir	Write Con	Type	Output Value	
5	F101_Ram	DEGC	SFC/F101_	Setpoint F	TRUE	17.5	25	TRUE	Float	250		
6	FC100	G/HR	SFC/FC10(	Setpoint	TRUE	2.5	5	TRUE	Float	300		

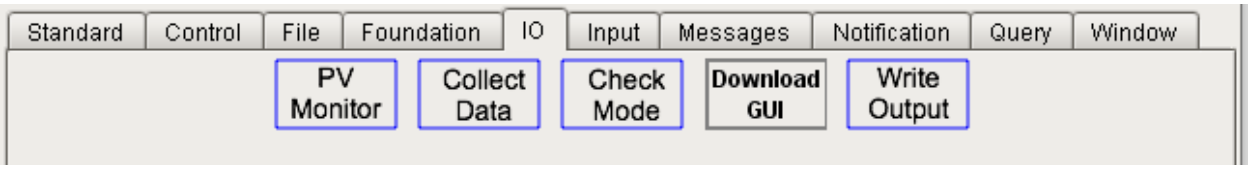
Save Data File

Caution:

There are two points of caution. First, the RTF format does not generally produce a readable report. Second, printing when a default printer has not been configured for the Ignition user has been known to crash the gateway. The file is printed from a gateway process, not the client that may have launched the SFC or is presented with a view of the file. The "user" is whoever owns the Ignition gateway process.

4.6 I/O Palette

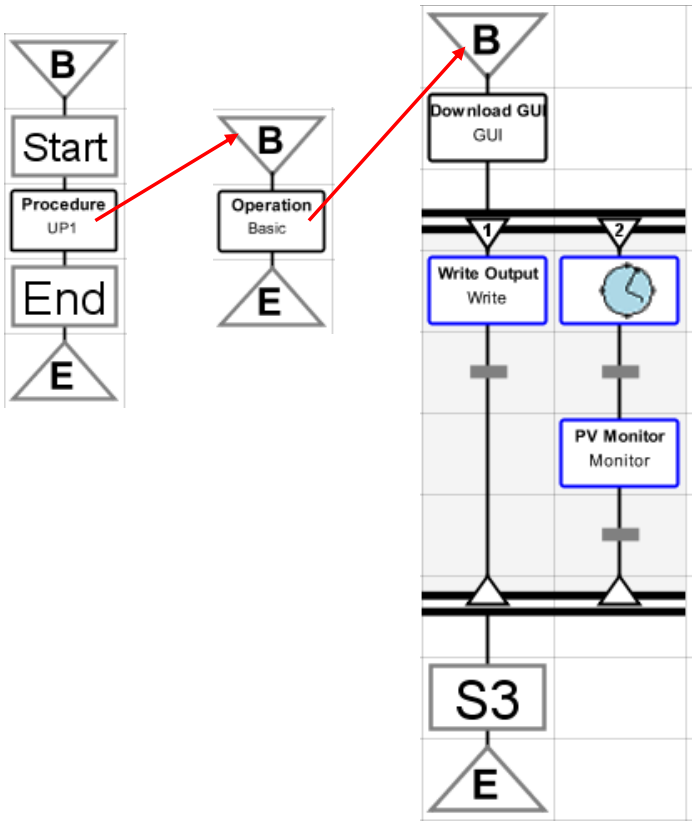
The I/O palette contains steps that make tag I/O easier. The Download Monitor, Write Output, and PV Monitoring steps are three important steps that may work independently but often cooperate.



I/O Palette

4.6.1 Common Behavior

Several of the steps, the Write Output, Monitor Download, and PV Monitoring, reference a timer. Although these blocks may be used individually, they are generally used in a coordinated fashion. When they are working together, they all need to reference a common download timer, which is stored in recipe data. The application designer needs to determine which block is the master; the master should be the only block that "Sets" the timer, all of the other blocks merely reference the timer. This diagram is used throughout section 4.6.


B
Start
Procedure
UP1
End
E
B
Operation
Basic
E
B
Download GUI
GUI
1
2
Write Output
Write
PV Monitor
Monitor
S3
E

Typical Use

4.6.2 Download Monitor Step

The purpose of the Download Monitor, aka Download GUI, step is to post a window for monitoring the state of a group of Write Output and PV Monitoring steps.

Step Configuration

A Download Monitor step and its properties are shown below:

Property	Value
name	GUI
description	
position	center
scale	1.0
button label	Down
window title	My Download GUI
window	SFC/MonitorDownloads
timer location	global
timer key	timer
timer clear	☑
timer set	☐
recipe location	operation
secondary sort key	alphabetical
config	<Use Editor>

Key	Label Attribute	Units
OUTPUT1	name	
OUTPUT2	name	
OUTPUT3	name	

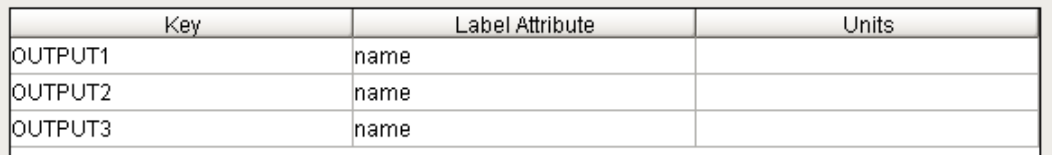
Download Monitor and Properties

The properties are:

Property	Description
Position	Specifies the position of the window. Choices are center, topLeft, topCenter, topRight, bottomLeft, bottomCenter, bottomRight
Scale	Specifies the factor by which the window will be scaled. 1.0 is full scale.
Button Label	Specifies the label for the button that will be added to the control panel for the window. The button is useful if the window is accidentally closed, if the client is accidentally closed, or if a new client connects after the step has started but before it has completed.
Window Title	Specifies the title for the window.
Window	Allows the window that is posted to be a custom window. *This is not supported at this time.*

Property	Description
Recipe Location	The recipe data scope locator where the keys in the config table are located.
Timer Key	The key to the timer
Timer Clear	Boolean. If True then the timer will be cleared when this step starts running.
Timer Set	A Boolean. If True then the timer will be started when this step starts running.
Timer Location	The recipe data scope locator where the timer recipe data is located.
Config	A list of dictionaries that specify the I/O to monitor. See below for details.

The config table lists the keys of the recipe data that will be monitored in the GUI that will be posted. It also allows the display to be configuring the engineering units for each line. It also specifies which property of the recipe data will be used in the display. The order of outputs in the table does not determine the order of outputs in the GUI. The outputs are ordered by the download time in the recipe data. There are downloads where the download time cannot be predetermined because it is event based. In this type of a download, all of the recipe data will have a download time of 0.0, but there are blocks upstream from the Write Output block that prevent the output from being written until the appropriate conditions have been met. In the case where the download time is the same for all outputs, then the outputs in the table are ordered in alphabetical order.



Key	Label Attribute	Units
OUTPUT1	name	
OUTPUT2	name	
OUTPUT3	name	

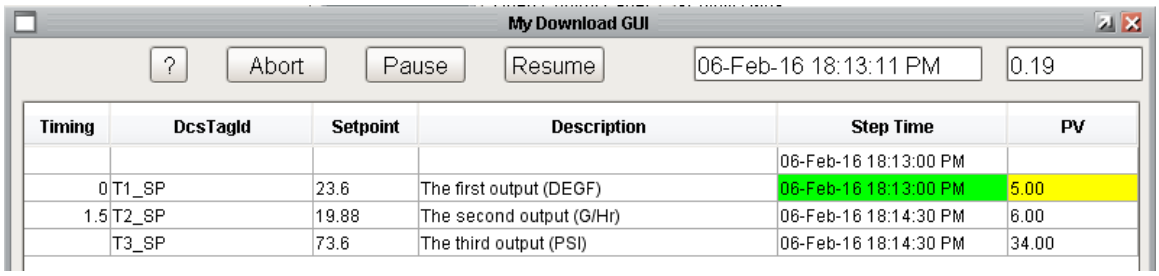
Download Monitor Configuration

Note: The download monitor step is not a long running step even though the window that it posts remains open for a long time. Because it is not a long running task it does not need to be followed by a transition checking for if work is done.

Note: There is an enhancement request to allow the second sort key for ordering outputs to be the order in the table.

Run-Time User Interface

The run-time user interface is shown below. It features a spreadsheet with one row per output. The user interface has a number of buttons across the top that can be used to control the download. The actions of the buttons are consistent with their labels and are a subset of the buttons contained on the control panel. There are two time displays at the top right of the window. One is a real time display of the current time and just to the right of that is a display of the time since the initial download commenced. The time is from the timer specified for the Download Monitor step.

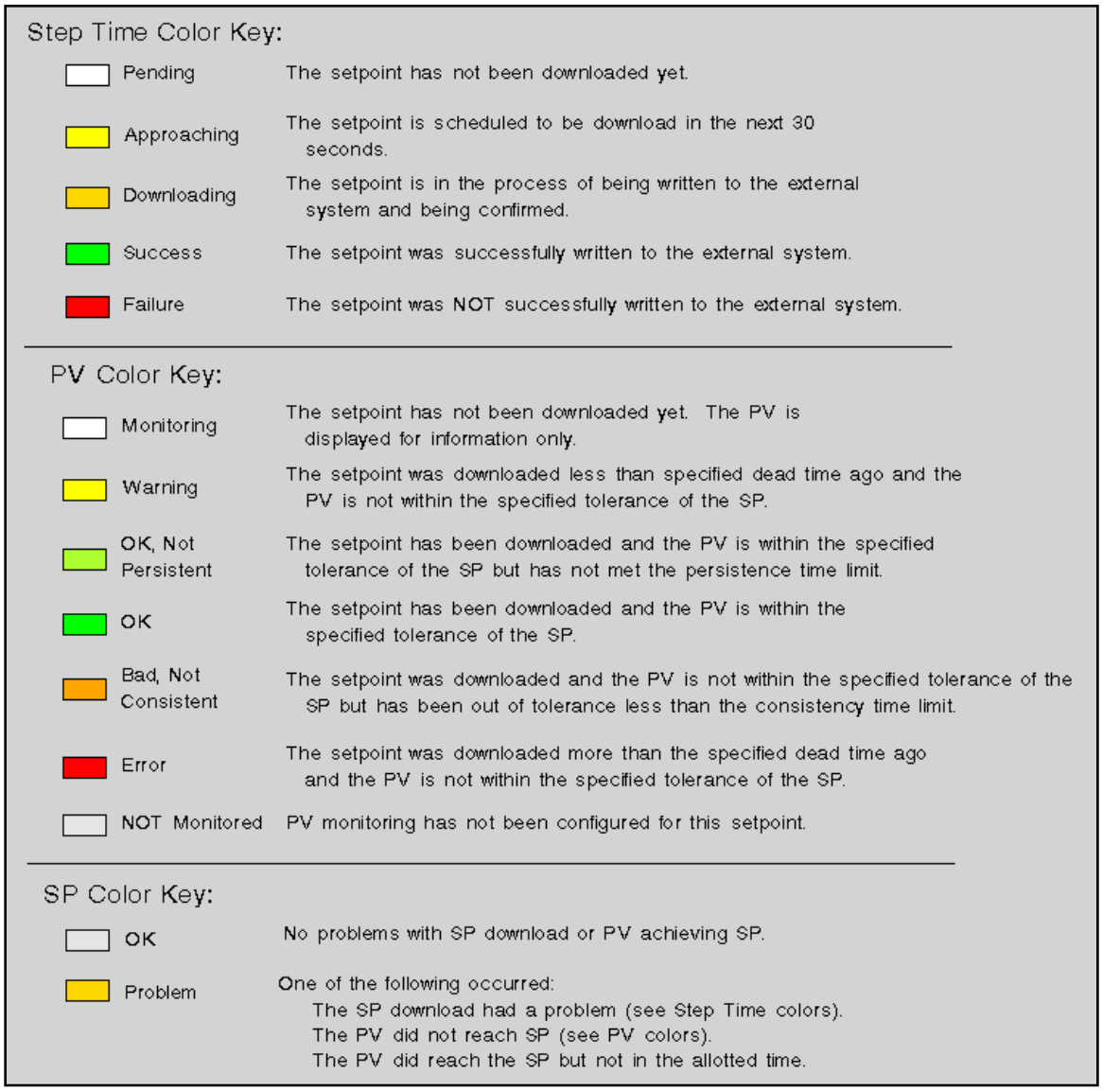


Download Monitor Window

The columns are:

Column	Description
Timing	The time in minutes from when the download commences that this point is scheduled to be written. This value comes from the recipe data. It may be static or dynamically calculated by a custom callback. If the value is >= 1000 then the write is an event-based write and a value will not be displayed.
DCS Tag Id	The label or name of the output.
Setpoint	The value that will be written. This column is animated as defined below.
Description	A description of the output from the recipe data.
Step-Time	The scheduled time of the download. If blank, then the output is an event-based write rather than a scheduled write. The column will be filled in once the output is written. This column is animated as defined below.
PV	The actual value of the I/O point. This column is animated as described below.

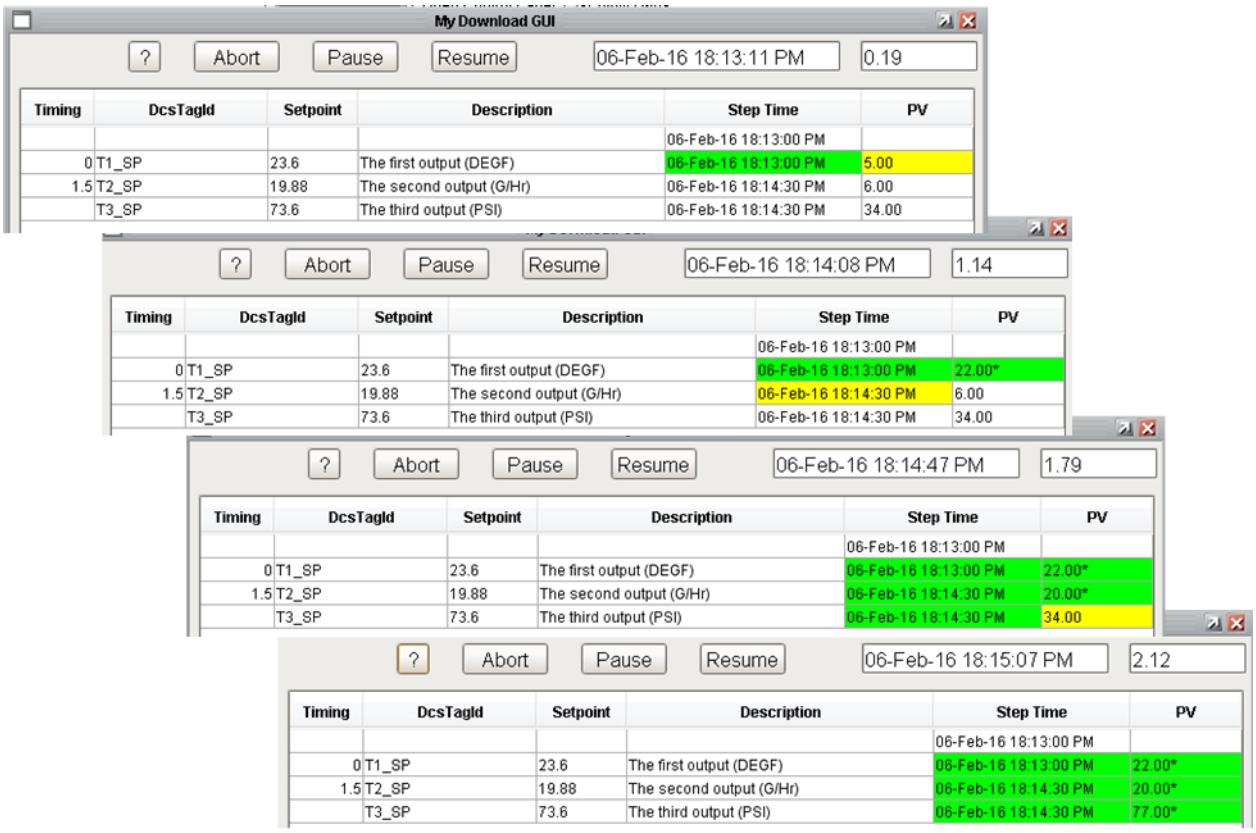
The key for the colors used to animate the table is described in the help screen which is shown below:



Download Monitor Color Key

Examples

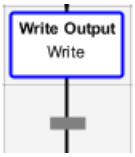
This example traces the execution of the chart described in section 4.6.1. When the recipe executes, it will write the three outputs and then monitor the three inputs. The download monitor is updated in real-time to display the state of the download in the column labelled "Step Time", the present value is displayed in the column titled "PV". The state of the monitored PV is indicated by the color. Finally, if any problem was encountered during the download of a set point then the "Setpoint" column will be animated.



Download Monitor Window Over Time

4.6.3 Write Output

The Write Output step writes values to tags, usually OPC tags that reference an external system, at specific times, and confirms that the values were successfully written. The Write Output and its properties are shown below.



Property	Value
name	Write
description	
recipe location	operation
timer location	global
timer key	timer
timer set	☑
config	<Use Editor>
error count scope	stepScope
error count key	errorCount
error count mode	absolute

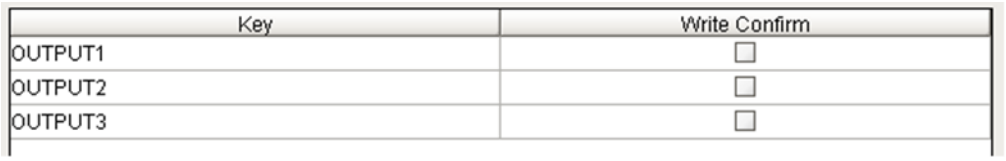
Key	Write Confirm
OUTPUT1	☐
OUTPUT2	☐
OUTPUT3	☐

Write Output Step and Properties

The properties are:

Property	Description
Recipe Location	The recipe data scope locator where the keys in the config table are located.
Timer Location	The recipe data scope locator where the timer recipe data is located.
Timer Key	The key to the timer
Timer Set	A Boolean. If True then the timer will be started when this step starts running.
Config	A list of dictionaries that specify the outputs to write. See below for details.
Error Count Scope	The recipe data scope locator where the simple value recipe data object is located or "chart" or "step" for chart and step variables.
Error Count Key	The key of the recipe data or chart/step variable.
Error Count Mode	Incremental or Absolute. Absolute will report the errors for just this step for this run. Incremental can be used to get a total count of errors from a number of Write Output steps.

The config table lists the recipe data keys that will be written and if the download will be confirmed. The details of what will be written, when, and where are contained in the recipe data. The important configuration is in the recipe data object. The key must reference either a Recipe Output or a Recipe Output Ramp type of recipe data. The recipe data has a *download* field must be **True** in order for the Write Output block to write the output. Typically the download field is somehow managed programatically based on reactor configuration, grade, or some other criteria.



Write Output Config Property

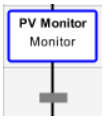
4.6.4 PV Monitor

The PV Monitor task is used to confirm that the present value (PV), or instrument value, of a controller has reached and/or maintained a desired set point (SP) within a certain amount of time. More generally, it is used to monitor or watch an input or an output. Monitor may be used after a setpoint has changed to ensure that the PV drives towards the new setpoint within the expected amount of time. It may also be used when a setpoint has been constant

for a reasonably long period of time and the PV is expected to be within the specified tolerance of the setpoint at all times. A watch is used merely to collect the PV, generally in order to update a user interface that the operator is viewing.

Configuration

The PV Monitoring step and its property table is shown below.



Property	Value
name	Monitor
description	
recipe location	operation
timer location	global
timer key	timer
timer set	☐
time limit strategy	recipe
time limit recipe key	time-limit.value
time limit static value (min)	0.0
time limit recipe location	local
error count scope	stepScope
error count key	errorCount
error count mode	absolute
activation callback	
config	<Use Editor>

Enabled	PV Key	Target Type	Target Nam...	Strategy	Limits	Download	Persistence	Consistency	Deadtime	Tolerance	Type	Status
☑	PV1	setpoint	OUTPUT1	monitor	High/Low	Wait	0.0	0.0	1.0	5.0	Abs	
☑	PV2	value	OUTPUT2	monitor	High	Wait	0.5	0.0	1.0	5.0	Abs	
☑	PV3	tag	OUTPUT3	watch	Low	Immediate	0.0	0.0	1.0	5.0	Pct	

PV Monitor Step and Properties

The properties are:

Property	Description
Recipe Location	The recipe data scope locator where the keys in the config table are located.
Timer Location	The recipe data scope locator where the timer recipe data is located.
Timer Key	The key to the timer
Timer Set	A Boolean. If True then the timer will be started when this step starts running.
Time Limit Strategy	No Limit, Static, Recipe. This determines how long the block will run. _ No Limit: The block will run until all of the monitored items have completed _ Static: Used in conjunction with Time Limit Static Value _ Recipe: Used in conjunction with Time Limit Recipe Key and Time Limit Recipe Location properties to specify the time limit in recipe data.

Property	Description
Time Limit Recipe Key	If the Time Limit Strategy is Recipe then this property, in combination with Time Limit Recipe Location, specifies the recipe data, generally a simple value, this specifies how long the block will run in minutes.
Time Limit Static Value (min)	If the Time Limit Strategy is Static then this property specifies how long the block will run in minutes.
Time Limit Recipe Location	If the Time Limit Strategy is Recipe then this property, in combination with Time Limit Recipe Key, specifies the recipe data, generally a simple value, this specifies how long the block will run in minutes.
Error Count Scope	The recipe data scope locator where the simple value recipe data object is located or "chart" or "step" for chart and step variables.
Error Count Key	The key of the recipe data or chart/step variable.
Error Count Mode	Incremental or Absolute. Absolute will report the errors for just this step for this run. Incremental can be used to get a total count of errors from a number of Write Output steps.
Activation Callback	
Config	A list of dictionaries that specify the outputs to write. See below for details.

Note: It is often desireable that the PV monitoring block should run forever and some other mechanism will stop it (such as a cancel condition on a parallerl encapsulation). There are two ways to configure the step to run forever. The prteferred way is to specify the "time limit strategy" as *No Limit*. A less obvious way is to specify the "time limit strategy" as *Static* and then specify the "time limit static value (min)" as *-1*. A Time limit of 0.0 will cause the step to complete immediately.

The config table consists of a table that describes the I/O that will be monitored or watched.

Enabled	PV Key	Target Type	Target Nam...	Strategy	Limits	Download	Persistence	Consistency	Deadtime	Tolerance	Type	Status
☑	PV1	setpoint	OUTPUT1	monitor	High/Low	Wait	0.0	0.0	1.0	5.0	Abs	
☑	PV2	value	OUTPUT2	monitor	High	Wait	0.5	0.0	1.0	5.0	Abs	
☑	PV3	tag	OUTPUT3	watch	Low	Immediate	0.0	0.0	1.0	5.0	Pct	

PV Monitor Configuration Table

The properties of each row are:

Attribute	Description
Enabled	A Boolean, if true then monitoring is enabled for this item. This allows an item to not be monitored by flipping a bit rather than deleting it entirely from the configuration.

Attribute	Description
PV Key	The key of the recipe data that identifies the object to be monitored. See section 0.
Target Type	Specifies the type of the target; the choices are Setpoint, Value, Tag, or Recipe. This column and the 'Target Name, Id, or Value' columns are related.
Target Name	The contents of this cell vary depending on the contents of the 'Target Type' as follows: _ Setpoint: the key to the recipe data that references the setpoint. The target value is stored in the ".val" attribute of the recipe data. Because the setpoint data structure is system-defined, the ".val" attribute does not need to be specified. _ Value: The target value, essentially a "hard-coded" target. _ Tag: The name of the tag. _ Recipe: The key to the recipe data that has the target value. Because the value is held in recipe data, it is generally set via API in some calculation procedure. The referenced recipe data may be any class although it is typically a Simple Value. The key and attribute must be specified.
Strategy	Defines the strategy to be used for this particular monitored item. The strategies are: Monitor or Watch, which are described below. If the strategy is "Watch" then all of the remaining fields are disabled.
Limits	This column has three possible values: "High/Low", "High", or "Low".
Download	This column has two possible values: "Immediate" or "Wait". "Immediate" means that the PV must reach the target within dead-time minutes from when the block began execution. This setting is generally used when monitoring is done without an associated download. "Wait" means that the PV must have reached the target within dead-time minutes from when the setpoint was downloaded. This strategy is generally used when PV monitoring is done with an associated download

Attribute	Description
Persistence	The amount of time that the PV must be "in range" for the monitoring to be a success. This guarantees that the PV is stable and prevents monitoring to prematurely complete due to a momentary spike in the PV value. Note: persistence may cause the monitored time to extend the dead-time. For example: Consider a configuration with a dead-time of 10 minutes and persistence of 2 minutes. If the PV becomes within the tolerance at 9 minutes, then after 10 minutes, the dead-time, even though the PV is within tolerance, because the persistence has not been met, the monitoring cannot be deemed a success. Consequently, the monitored time is extended beyond the dead-time until the persistence is met or the PV goes out of tolerance.
Consistency	The minimum amount of time that the PV is not within its limits before the PV is identified as "out of range".
Deadtime	The maximum amount of time, in minutes, that the PV will be monitored. If the PV has not reached the target at the end of this time, then the monitoring is considered a failure and the error counter is incremented. The download setting, i.e., immediate or wait, determines if the dead-time is measured from when the setpoint is written or when the block begins execution.
Tolerance	Defines the acceptable deviation for the PV from the target value, either as an absolute value or as a percentage of the target.
Type	This column has two values: "Abs" or "Pct". This refers to the type of the tolerance, either absolute or a percentage of the target. Implementing the tolerance as a percent is problematic if the target is 0.0. (A future version may address this limitation by implementing an absolute value scheme if the target value is 0. For example, 5% would be interpreted as +/- 0.05).
Status	This column is read-only and reflects the latest real-time status of the item being monitored.

Specifying the PV

The PV must be specified by an input, output, or output ramp recipe data. The recipe data must exist at the location specified by *Recipe Location* block property. It is not possible to just specify

a tag name because the results of the monitor update properties of the recipe data. A PV Monitoring block often coordinates with a Download Monitor and/or Write Output block; the three blocks communicate with each other through the recipe data.

If the PV Key references an input, then the PV value comes directly from the tag referenced by the input. Typically, an input recipe data is used if we are monitoring a stand alone OPC tag.

If the PV Key references an output or output ramp the value that is used for the PV is determined by output-type of the recipe data:

Output Type	Tag
Mode	Mode/Value
Setpoint	value
Output and Output Ramp	op/Value
Value	value

If the recipe data is an Output or Output Ramp then the download flag must be set to True, othewise PV monitoring will essentially be disabled. The rationale is that if the output isn't downloaded then there isn't anything to monitor. This is a reasonable approach when the recipe download flag is set dynamically based on grade or reactor configuration and setting the download flag in recipe will be picked up by Write OPutput and PV Monitoring blocks throughout the recipe.

Specifying the Target

Specifying the target is a little more flexible than specifying the PV There are two columns that define the target: *Target Type* and *Target Name / Id / Value*. For all of the target types, the value is determined when the step starts running. In the case of a ramp, the target value is the final value. It is not possible to monitor that the PV is within a certain tolerance at each step of the ramp.

Target Type	Target Name / Id / Value
Value	A fixed value
Tag	A tag path whose value will be read.
Recipe	Recipe data specified by *key.attribute* at the scope defined by the block property: *Recipe Location*.
Setpoint	This is used when the PV Key is Output or Output Ramp recipe data. It is confusing because this is used when the recipe data *Output Type* is *Output* or *Output Ramp*. In all cases the target value is the final value that is written.

Strategy

For each item in the configuration structure, there are two strategies: "Monitor" or "Watch".

MONITOR STRATEGY

A "monitor" strategy is used for monitoring I/O specified in input or output recipe data. Generally an output is used if there is a setpoint download to ensure that the PV drives towards the new setpoint within the expected amount of time. An input is typically used after a setpoint has been constant for a reasonably long period of time and the PV is expected to be within the specified tolerance of the setpoint at all times. It can also be used when there isn't a setpoint associated with the PV being monitored.

WATCH

A "watch" strategy is used when there is a Monitor Download task that is configured to display the PV for an I/O that does not have a monitoring requirement. A watch will merely collect the PV value and update the recipe data so that a Download GUI can display the latest value. The watch strategy cannot timeout; it will execute until the block completes.

Monitoring Results

The PV monitoring step does not have a run-time user interface. However, the step is often used in conjunction with a Download Monitoring step which displays data that is collected by this step. The results of the monitoring update the recipe data associated with the item. There are three properties that are updated:

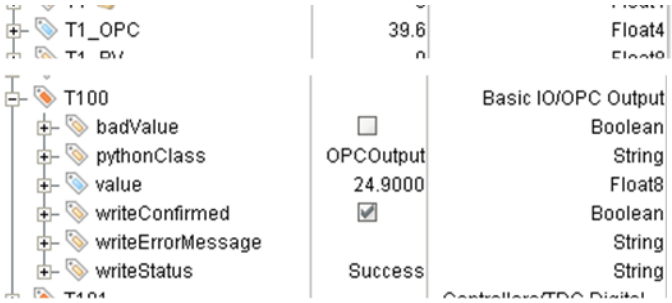
- pvValue: the actual present value for the item being monitored or watched.
- *pvMonitorActive:* specifies if the item is currently being monitored. It may be False if the monitoring has not yet begun, or if the monitoring has completed due to successfully meeting the target or if it has timed out.
- *pvMonitorStatus:* specifies the status of the monitoring. The possible values are:
    - Monitoring: The initial status of the PV once monitoring has started and prior to a value being analyzed.
    - Warning: The PV does not meet the target value but the time is less than the specified dead-time.
    - Error: The PV did not meet the target value within the specified dead-time and the PV is still being monitored. (If the PV meets the target AFTER the deadtime, the status is still ERROR).
    - Timeout: The PV did not meet the target value within the specified dead-time and the PV is no being monitored. The status turns from ERROR to TIMEOUT when the monitoring stops.
    - OK: The PV has met the target value within the specified dead-time.

Examples

The behavior of the PV monitoring is best explained through examples.

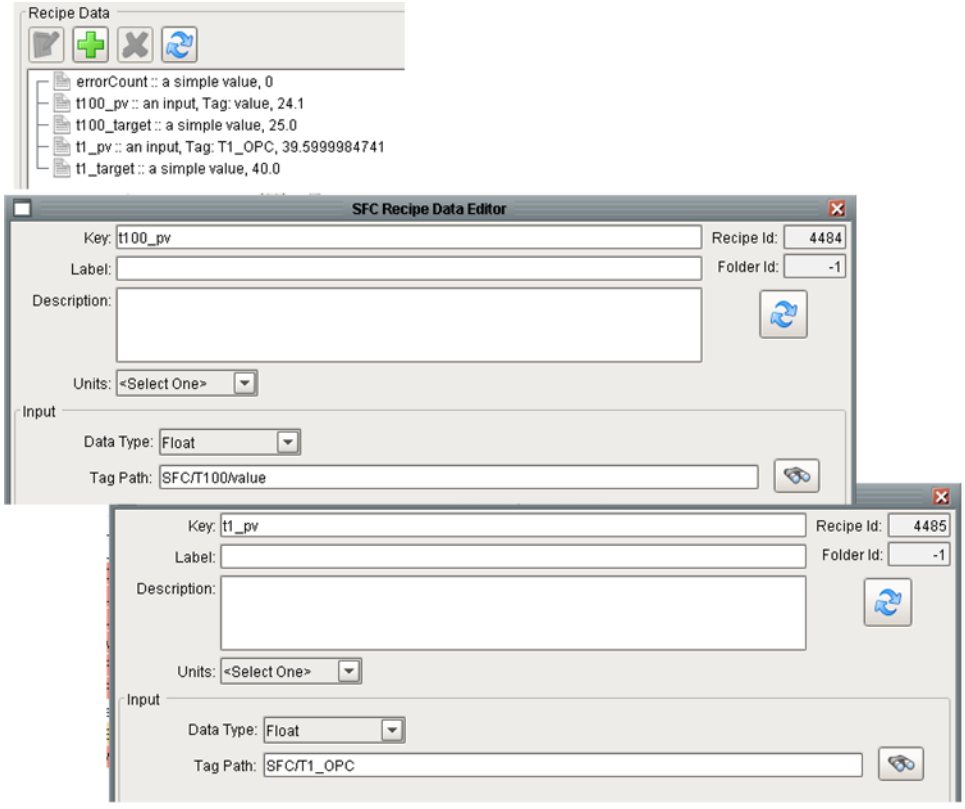
MONITORING AN OPC TAG

The first example monitors two OPC tags, a simple stand-alone OPC tag and an OPC Output UDT, without any download. The two tags are both in the SFC folder:



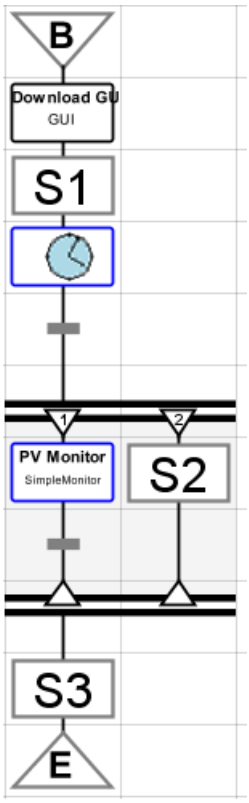
OPC Tags to be Monitored

The next example uses INPUT recipe data is used for the PV because a download is not involved. The target is specified in SIMPLE VALUE recipe data.



Recipe Data for a Simple Monitor of an Input

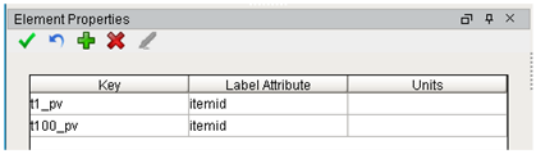
The following diagram is used.



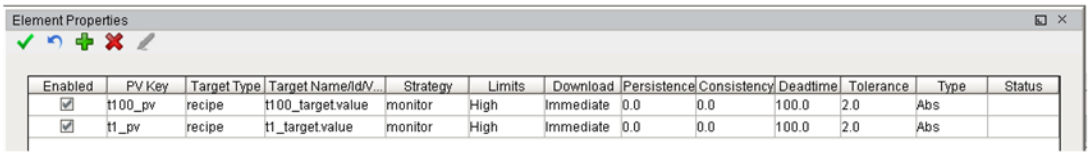
Typical Diagram to Monitor an Input

The three action steps write values to the tags for simulation purposes. The configuration of the Download GUI and PV Monitoring are shown below. The performance criteria for T100_PV, from t100_target recipe, is 40 +/- 2. The performance criteria for T1_PV, from t1_target recipe, is 25 +/- 2

Download GUI:

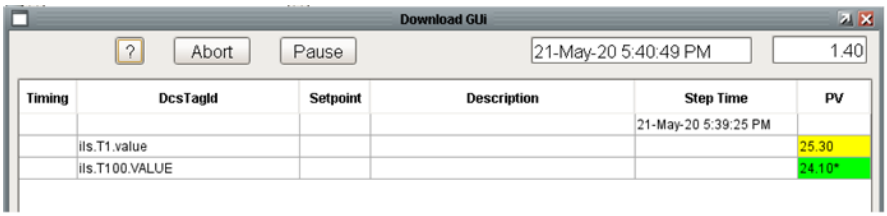


PV Monitor:



Download Monitor and PV Monitor Step Configurations

The runtime GUI is shown below after 1.4 minutes. Because there is not a download the timing fields are blank, but the PV field is animated to reflect the PV performance criteria.



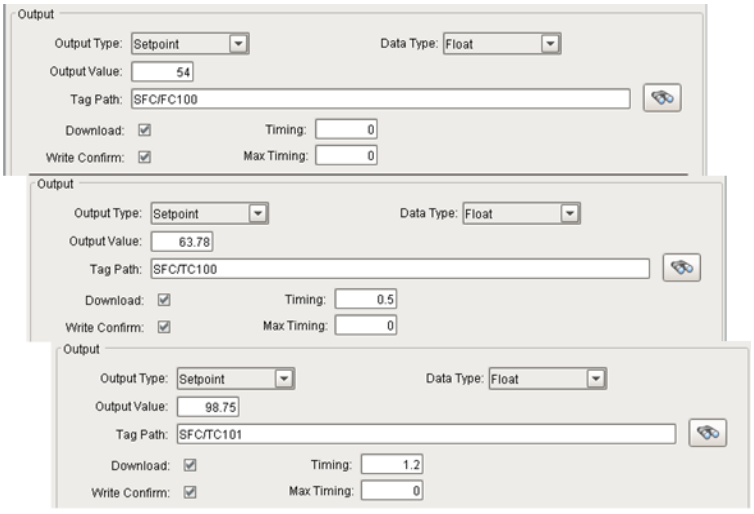
Download Monitor Window for a PV Monitor of an Input

MONITORING A CONTROLLER PV AFTER A SP WRITE

The second example involves monitoring the PV of a controller after writing to the SP. This is probably the most common configuration. The example writes to the following recipe data:

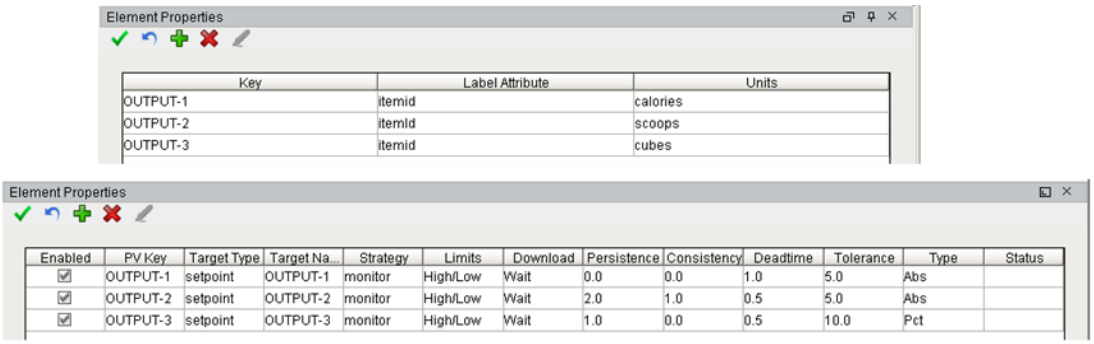
PV Monitoring Recipe Data

This example involves writing values to the setpoint of the controllers so OUTPUT recipe data is used. The values are written over 1.2 minutes from the start of the download.



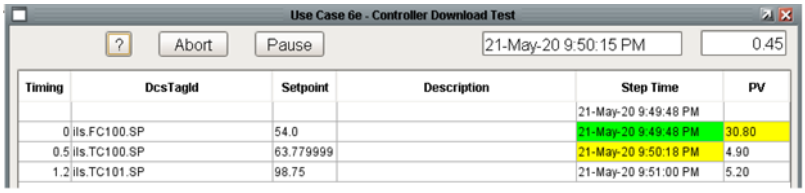
Output Recipe Data

The Download GUI and PV Monitor configuration are shown below. The PV Key and the Target both point to the same OUTPUT recipe data. The combination of OUTPUT recipe data referencing a controller and a target type of setpoint configures the PV Monitoring step to use the setpoint of the controller as the target.



Download Monitor and PV Monitoring Configurations

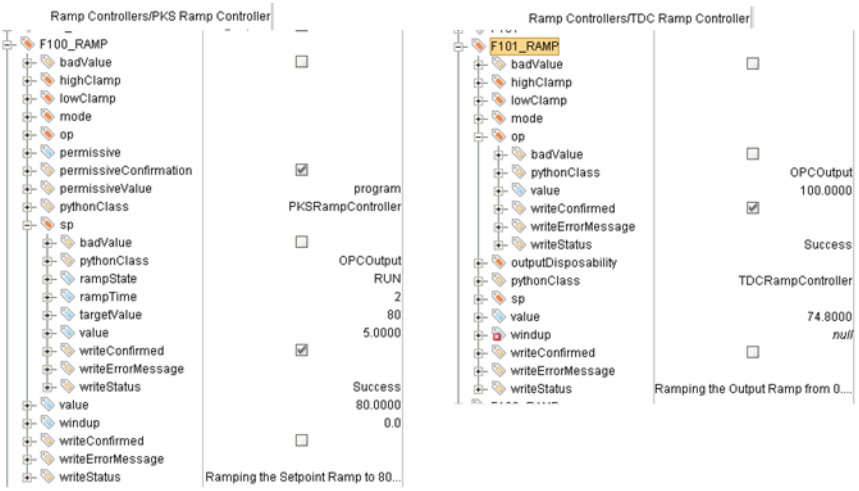
The run time GUI after 0.45 minutes after the start is shown below.



Download Monitor Window

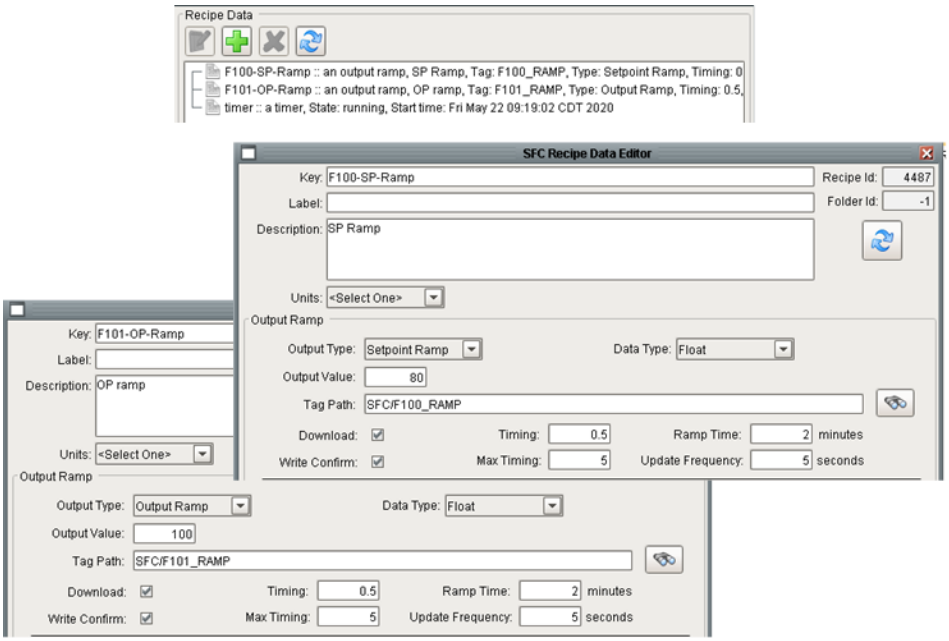
MONITORING RAMP CONTROLLERS

A special case to consider is ramp controllers. This example examines ramping the SP and OP to ramp controllers. It is important to understand that an SP ramp control implements the ramp in the DCS controller but there is no equivalent for OP ramps. Even if we are writing to a ramp controller, when ramping the OP, the ramp is implemented in Ignition by making many incremental writes to implement a linear ramp from the current OP to the target OP. The example writes to the following two controllers:



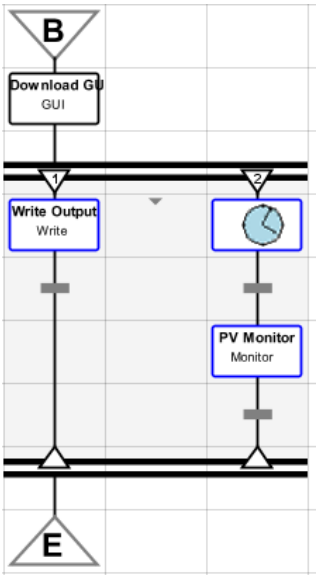
Ramp Controller UDTs

This test uses the following recipe data. Both recipe data are instances of OUTPUT-RAMP recipe data.



Output Ramp Recipe Data

The chart uses the standard block configuration where the Download GUI step is followed by the Write Output and PV Monitoring steps in parallel.



PV Monitoring Typical Chart

The configuration of the I/O blocks is shown below. It is worth pointing out that the PV Monitoring configuration for the Target Type for F101-OP-Ramp is not intuitive. The Write Output is ramping the OP. The combination of an OUTPUT-RAMP recipe data configured as an "Output Ramp" and the Target Type of "setpoint" specifies that the PV monitor step will

monitor the actual OP vs the target OP. The rationale behind this is that it may not be known what the PV should be when the OP reaches its target value. Therefore the only thing that can be monitored is the OP. On the other hand, the WRITE OUTPUT already monitors that the individual writes succeed.

The runtime GUI at various times after chart starts is shown below. The GUI shows the difference between a ramp implemented in Ignition (the OP ramp) and a ramp implemented in the DCS (the SP ramp). The Setpoint column shows the target value from recipe data. For Step Time column displays the actual time when the write begins. The Step Time cell is orange while the write is being processed. As specified in the recipe data, the ramp time is 2 minutes for both the SP and the OP ramp. Writing the SP ramp, implemented in the DCS, completes when the ramp parameters are written to the DCS; so the write is complete before the SP value starts ramping. Writing the OP ramp, implemented in Ignition, is not complete until the final OP value is written. This explains the different animation in the Step Time column even though the ramps begin at the same time (0.5 minutes) and are the same length (2.0 minutes). The OP ramp column will be orange for 2.0 minutes while the OP is being ramped. The PV column for the setpoint ramp will read the PV of the controller. The PV column for the OP ramp will display the OP of the controller. The "Download" column of the PV monitoring configuration is set to "wait" for both items being monitored. The PV monitoring step will collect the PV value and the Download GUI will display it even before the download is complete even though the performance criteria will not be evaluated until the download completes. Therefore, the performance of the SP ramp will be begin being evaluated two minutes before the performance of the OP ramp. It may be important to add the ramp time to the deadtime to achieve the desired goal.

Timing	DcsTagId	Setpoint	Description	Step Time	PV
				22-May-20 10:13:35 AM	
0.5	GFT033C.OP	100.0	OP ramp	22-May-20 10:14:05 AM	0.10
0.5	ils.F100.sp	80.0	SP Ramp	22-May-20 10:14:05 AM	11.80

Timing	DcsTagId	Setpoint	Description	Step Time	PV
				22-May-20 10:13:35 AM	
0.5	GFT033C.OP	100.0	OP ramp	22-May-20 10:14:05 AM	20.91
0.5	ils.F100.sp	80.0	SP Ramp	22-May-20 10:14:05 AM	55.00

Timing	DcsTagId	Setpoint	Description	Step Time	PV
				22-May-20 10:13:35 AM	
0.5	GFT033C.OP	100.0	OP ramp	22-May-20 10:14:05 AM	37.56
0.5	ils.F100.sp	80.0	SP Ramp	22-May-20 10:14:05 AM	81.00

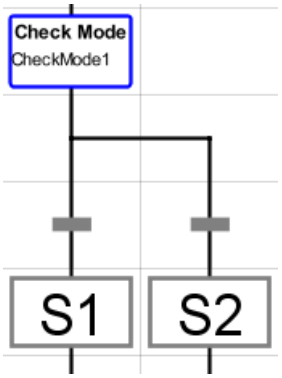
Timing	DcsTagId	Setpoint	Description	Step Time	PV
				22-May-20 10:13:35 AM	
0.5	GFT033C.OP	100.0	OP ramp	22-May-20 10:14:05 AM	100.00*
0.5	ils.F100.sp	80.0	SP Ramp	22-May-20 10:14:05 AM	79.00*

PV Monitoring of Output Ramps

4.6.5 Check Mode

The Check Mode step, also known as the Confirm Controller Mode step, is used to confirm the mode of a controller to ensure that it is set up to allow a setpoint or output write. This is typically used upstream of a Write Output step.

The Check Mode step and properties are shown below:



Property	Value
name	CheckMode1
description	
config	<Use Editor>
error count scope	stepScope
error count key	errorCount
error count mode	incremental

Check Mode Step and Properties

The properties are:

Property	Description
Config	A list of dictionaries that specify the outputs to write. See below for details.
Error Count Scope	The recipe data scope locator where the simple value recipe data object is located or "chart" or "step" for chart and step variables.
Error Count Key	The key of the recipe data or chart/step variable.
Error Count Mode	Incremental or Absolute. Absolute will report the errors for just this step for this run. Incremental can be used to get a total count of errors from a number of Write Output steps.

The step is configured to check the mode of two controllers. The controllers are indirectly referenced using recipe data which must be either output or output ramp recipe data types.

The step performs the following checks for each controller specified in the configuration:

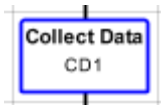
- The quality of the value of the controller is good
- The quality of the value of the mode of the controller is good
- The quality of the windup of the controller is good
- If the output type of the recipe data is OP or OUTPUT then verify that the value of the mode is 'MAN'
- If the output type if the recipe data is SP or SETPOINT then verify that the value of the mode is 'AUTO'

In addition to these checks, which are performed for every controller, there are two additional checks that may be specified, these only apply if the output type of the recipe data is SP or SETPOINT :

- If the "Check Path To Valve" check box is checked then verify that the windup of the controller is not 'HiLo'
- If the "Check SP for 0" check box is checked then verify that the value of the controller is near zero. This is used when we expect the valve to be closed. If it isn't closed then the state of the plant is not what we expect it to be and the SFC should not proceed. Because a valve that is closed may not read exactly 0.00000, we use a near zero check which checks if the current value is less than 0.03 * the output value of the recipe data (the value that will be written).

4.6.6 Collect Data Step

The collect data block is a convenient way of reading tag values and storing the value in recipe data. It can read the current value or the average, minimum, maximum, and standard deviation. The Collect Data Step and its properties are shown below.



Property	Value
name	CD1
description	
timeout behavior	abort
config	<Use Editor>

Recipe Key	Location	Tag Path	Value Type	Past Window (min)	Default Value
MIN.value	operation	SFC/6a/T1	minimum	10.0	0.0
MAX.value	operation	SFC/6a/T1	maximum	10.0	0.0
AVG.value	operation	SFC/6a/T1	average	10.0	0.0
PV.value	operation	SFC/6a/T1	current	10.0	0.0
SIGMA.value	operation	SFC/6a/T1	stdDeviation	10.0	0.0

Collect Data Step and Properties

The step has two properties:

Property	Description
Timeout Behavior	Three choices on what to do if any item in the list times out or returns a bad value. _ Abort – Immediately abort the chart _ Timeout – Set the step scope variable timeout to True so a transition can test for a timeout and implement custom logic. _ Default Value - Use the default value as if nothing happened.
Config	A list of dictionaries that specify data to collect. See below for details.

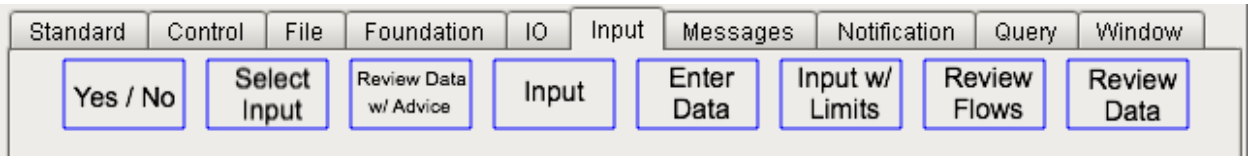
The config table specifies the recipe data that will be updated, the tab value that will be read, the value type, the window over which to perform the statistics, and the default value to return if the timeout strategy is Default.

Recipe Key	Location	Tag Path	Value Type	Past Window (min)	Default Value
MIN.value	operation	SFC/6a/T1	minimum	10.0	0.0
MAX.value	operation	SFC/6a/T1	maximum	10.0	0.0
AVG.value	operation	SFC/6a/T1	average	10.0	0.0
PV.value	operation	SFC/6a/T1	current	10.0	0.0
SIGMA.value	operation	SFC/6a/T1	stdDeviation	10.0	0.0

Collect Data Configuration

4.7 Input Palette

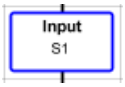
There are numerous steps that post a user interface to the client soliciting some sort of response. These steps are described below.



Input Step Palette

4.7.1 Get Input and Get Input with Limits Steps

The Get Input and Get Input steps are used to prompt the user for input and store it in recipe data. Both of these steps get a single value. The step requires that an entry is made. The step does not force or check the data type but the recipe data update of the response will fail if the data types are incompatible.



Property	Value
name	S1
description	Get some user input
position	bottomLeft
scale	1.2
button label	Color
window title	User Preference
prompt	What is your favorite colour?
recipe location	local
key	ans.value
default value	

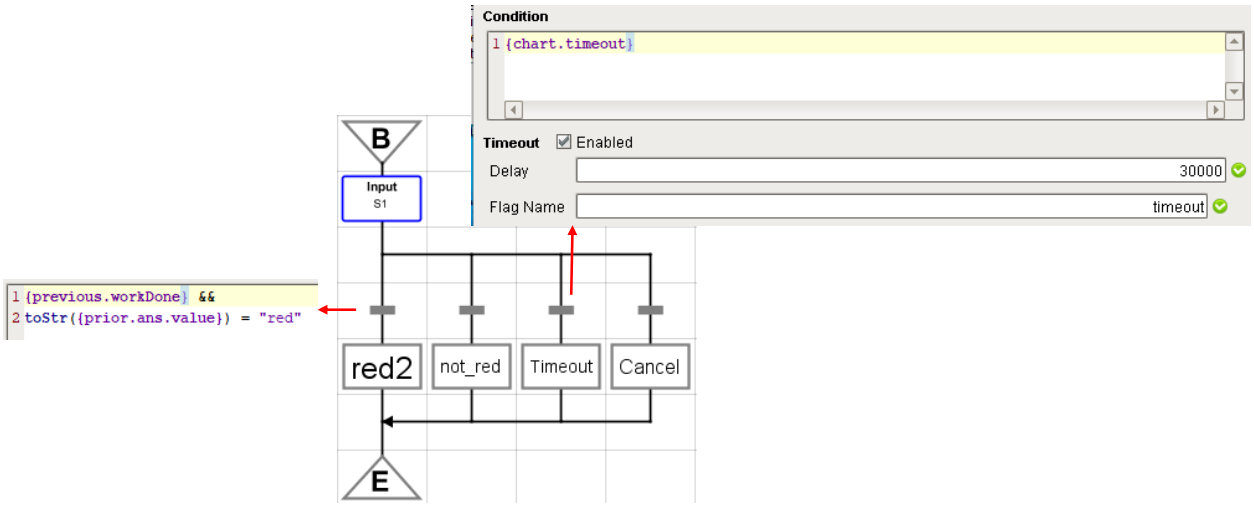
Get Input Step and Properties

The step properties are:

Property	Description
Position	Specifies the position of the window when in semiautomatic mode. Choices are center, topLeft, topCenter, topRight, bottomLeft, bottomCenter, bottomRight
Scale	Factor that the window will be scaled by when in semiautomatic mode. 1.0 is full scale.
Button Label	Label for the button that will be added to the control panel for the window when in semiautomatic mode. The button is useful if the enter data window is accidentally closed, if the client is accidentally closed, or if a new client connects after the step has started but before it has completed.
Window Title	The title for the window when in semiautomatic mode.
Prompt	Text string that will be displayed above the input field.
Recipe Location	The recipe data scope locator where the input will be stored.
Key	The recipe data key and attribute where the input will be stored. Typically, the response will be saved in a Simple Value recipe data.
Default Value	Optional – value that will be populated into the input field.

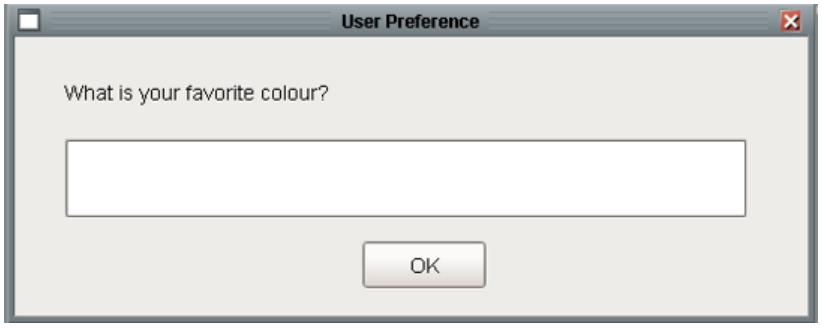
One common pattern involving this step is to select a path to take. In this case the step is followed by any number of transitions. This step is a long running step so it is importanty that the transitions check the *previous.workDone* flag. The step does not have timeout specified in the step but a timeout can be implemented using a common Ignition pattern shown below. The third transition defines the timeout interval and chart variable that will be sets and the condition checks the flag.

This is a standard, compact way of implementing a timeout using regular Ignition capabilities. When the transition sets the flag and then cheks the flag, the step will be cancelled. It is important to NOT use the *previous.workdone* in this condition.



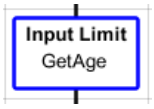
Get Input Typical Usage

When the chart runs, the following window is displayed on every interested client.



Get Input Window

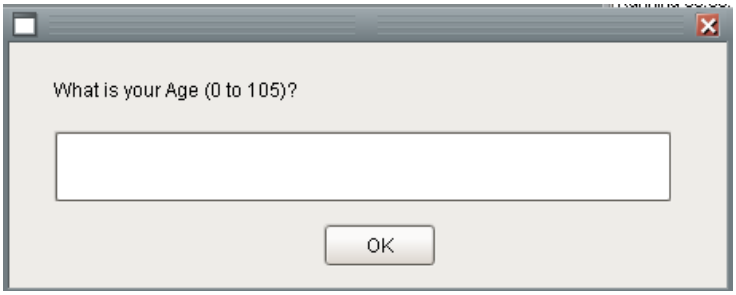
The Get Input with Limits is an extension of the Get Input step. It works with numeric data and validates that the entry is between the upper and lower limits. The property table is shown below:



Property	Value
name	GetAge
description	Ask them their age
position	center
scale	1.0
button label	Age
window title	
prompt	What is your Age?
recipe location	local
key	age.value
maximum value	105.0
default value	
minimum value	0.0

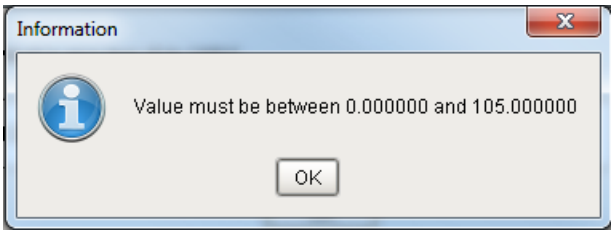
Get Input with Limits Step and Properties

The user interface posted to the client is:



Get Input with Limits Window

If the entry is outside of the limits, which are not automatically added to the prompt, then the following message is displayed and the user can enter another value.



Get Input Limit Violation Warning

4.7.2 Select Input Step

The Select Input Step is another way of getting user input. The difference between this step and the Get Input is that this step selects a value from a combo box rather than typing a value into an edit box. Like the get input step, this is a long running step and should be followed by transitions checking *previous.workDone*. Select Input handles timeouts the same way as the Get Input.

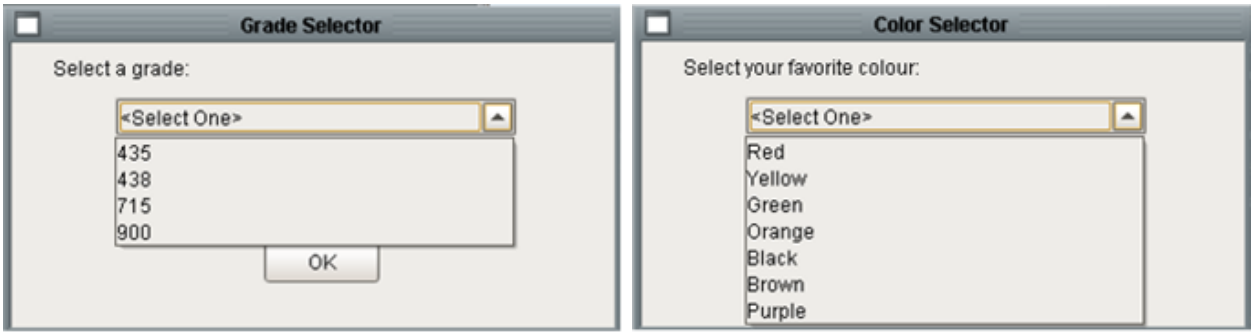
Property	Value
name	Select Grade
description	
position	center
scale	1.0
button label	Grade
window title	Grade Selector
prompt	Select a grade:
choices recipe location	operation
choices key	grades
recipe location	operation
response key and attribute	selectedGrade.value

Select input Step and Properties

The step properties are:

Property	Description
Position	Specifies the position of the window when in semiautomatic mode. Choices are center, topLeft, topCenter, topRight, bottomLeft, bottomCenter, bottomRight
Scale	Factor that the window will be scaled by when in semiautomatic mode. 1.0 is full scale.
Button Label	Label for the button that will be added to the control panel for the window when in semiautomatic mode. The button is useful if the enter data window is accidentally closed, if the client is accidentally closed, or if a new client connects after the step has started but before it has completed.
Window Title	The title for the window when in semiautomatic mode.
Prompt	Text string that will be displayed above the input field.
Choices Recipe Location	The recipe data scope locator where the choices are stored.
Choices Key	The recipe data key of the Array recipe data that contains the choices. The choices must be defined in and Array recipe data!
Recipe Location	The recipe data scope locator where the selected value will be stored.
Response Key and Attribute	The recipe data key and attribute where the selected value will be stored. Typically, the response will be saved in a Simple Value recipe data.

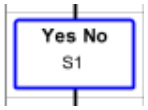
The window that is displayed on interested clients is shown below:



Select Input Windows

4.7.3 Yes/No Step

The Yes/No step is used to get a Yes or No answer from the operator. The Yes/No step and its property editor is shown below.



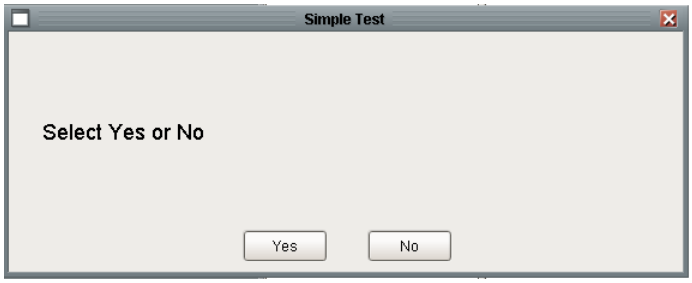
Property	Value
name	S1
description	Most basic of tests
position	center
scale	1.0
button label	Y/N Test
window title	Simple Test
prompt	Select Yes or No
recipe location	local
key	ans.value

Yes /No Step and Properties

The step properties are:

Property	Description
Position	Specifies the position of the window when in semiautomatic mode. Choices are center, topLeft, topCenter, topRight, bottomLeft, bottomCenter, bottomRight
Scale	Factor that the window will be scaled by when in semiautomatic mode. 1.0 is full scale.
Button Label	Label for the button that will be added to the control panel for the window when in semiautomatic mode. The button is useful if the enter data window is accidentally closed, if the client is accidentally closed, or if a new client connects after the step has started but before it has completed.
Window Title	The title for the window when in semiautomatic mode.
Prompt	Text string that will be displayed above the input field.
Recipe Location	The recipe data scope locator where the selected value will be stored.
Key	The recipe data key and attribute where the selected value will be stored. Typically, the response will be saved in a Simple Value recipe data.

The window that is displayed on interested clients for the step defined above is shown below:



Yes / No Window

4.7.4 Enter Data Step

The Enter Data step is used to set recipe data or tag values. The step can be configured to initialize the values automatically to constants defined in the step or it can be configured to post a window for the user to enter values. If configured for user entry, the step will post a window with a table for entry of multiple pieces of data.

If this step is configured with a timeout, then it is important to understand that any data partially entered into the GUI will not be saved if the step times out.

The step and its properties are shown below:

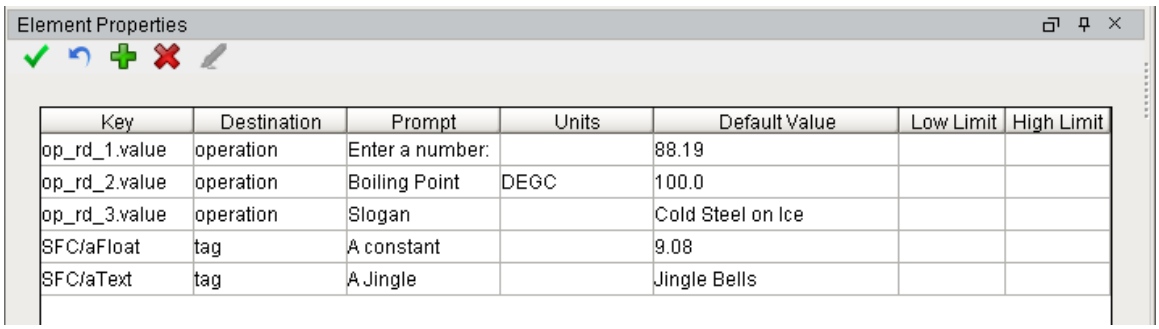
Property	Value
name	Entry_2
description	
position	topCenter
scale	1.0
button label	Enter
window title	Test NEW Default notation
window header	Test the new notation for specifying default values
window	SFC/ManualDataEntry
auto mode	semiAutomatic
require all inputs	☑
config	<Use Editor>

Enter Data Task and Properties

Property	Description
Position	Specifies the position of the window when in semiautomatic mode. Choices are center, topLeft, topCenter, topRight, bottomLeft, bottomCenter, bottomRight

Property	Description
Scale	Factor that the window will be scaled by when in semiautomatic mode. 1.0 is full scale.
Button Label	Label for the button that will be added to the control panel for the window when in semiautomatic mode. The button is useful if the enter data window is accidentally closed, if the client is accidentally closed, or if a new client connects after the step has started but before it has completed.
Window Title	The title for the window when in semiautomatic mode.
Window Header	The window has space for optional instructions above the table when in semiautomatic mode.
Window	The name of the Vision window to use if in semiautomatic mode.
Auto Mode	Automatic or semiautomatic. If automatic the target will be set from the default values. If semiautomatic then a window will be posted to the client.
Require All Inputs	If True then an entry is required for every row.
Config	A dictionary of values that define the layout of the table.

Configuration Parameters: The config parameters are presented in a table where each row specifies a recipe data entity that will be initialized. A typical table is shown below:



The columns are:

Property	Description
Key	An n dimensional array of any datatype. A
Destination	
Prompt	An n X m dimensional matrix of any datatype. All elements in

Property	Description
Units	Engineering Units that define the units that the recipe data will be displayed in. Conversion between the recipe data units and the display units is automatic.
Default Value	Integrates to the recipe toolkit. Corresponds to a record in the recipe database.
Low Limit	Optional – only applies if the mode of the block is semi-automatic
Upper Limit	Optional – only applies if the mode of the block is semi-automatic

Automatic Mode

When the "Auto Mode" property is set to "automatic" the Enter Data step can be used to initialize recipe data values. In automatic mode, the default values are written to the destination, if there is not a default value then the destination is not affected. This is useful because recipe data is a permanent data structure. Any changes made during the course of a run will be permanent and be used the next time the unit procedure runs. Therefore, in order to guarantee a known starting point, recipe data that is not calculated or read from tags should be initialized.

SemiAutomatic Mode

This should be referred to as "manual" mode. When the "Auto Mode" property is set to "semiAutomatic" the Enter Data step posts a window for the user to enter values into a table. The values will be stored in recipe data when the user presses OK.

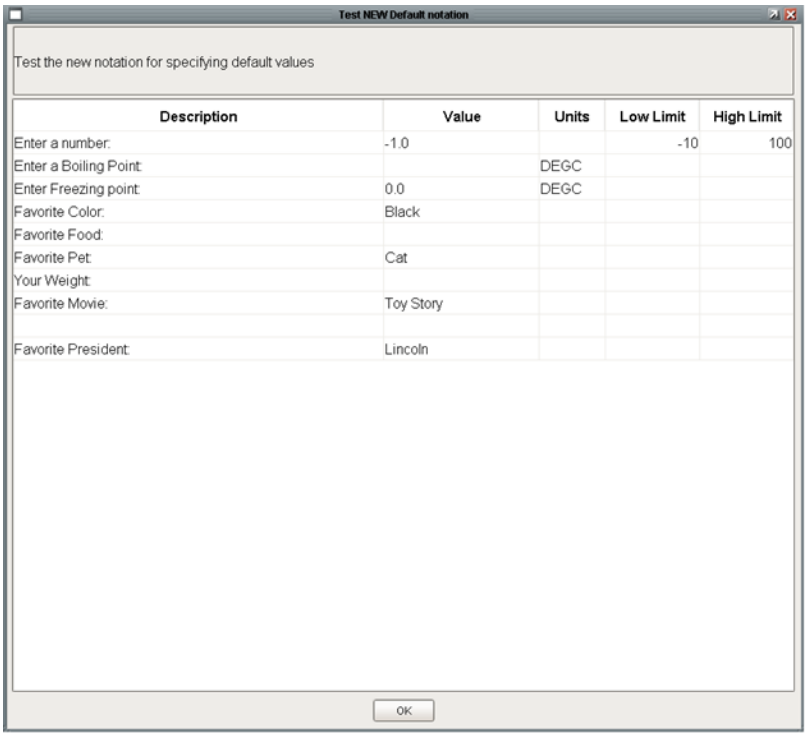
DEFAULT VALUES

The table that is displayed may provide default values in one of two ways. First, static values can be entered into the Default Value column. The other way is to use the magic word: *recipe* as the default value. This will use the current value from the recipe data destination.

Key	Destination	Prompt	Units	Default Value	Low Limit	High Limit
op_rd_1.value	operation	Enter a number:		recipe	-10.0	100.0
op_rd_2.value	operation	Enter a Boiling Point:	DEGC			
op_rd_3.value	operation	Enter Freezing point:	DEGC	0.0		
op_rd_4.value	operation	Favorite Color:		recipe		
op_rd_5.value	operation	Favorite Food:				
op_rd_6.value	operation	Favorite Pet:		Cat		
SFC/aFloat	tag	Your Weight:				
SFC/aString	tag	Favorite Movie:		recipe		
op_rd_7.value	operation	Favorite President:		Recipe		

Data Entry Configuration with Default Values

The window that is posted is:



Data Entry Window

4.7.5 Review, Review with Advice, and Review Flows Steps

These steps provide a user interface for operators to review and approve data. These data may originate from model calculations, process measurements, diagnosis diagrams, etc., and may be in recipe data or named items. The data is presented to the user in a read-only table on a window. The window will generally contain a button to accept the values and to proceed with recipe execution, another will generally return the user to an input screen where parameters can be changed that will affect the values shown in the review data screen. All three steps support a primary and secondary table of data. The secondary table is optional, if there is nothing on the secondary page then the tab strip used to select the table will not be shown.

There are three different review data tasks, each of which is discussed below.

Review Data Step

The Review Data Step posts a window that contains a table with three columns. The first column contains a label or description. The second column displays a value and the third column displays the units.

The block configuration is shown below.

Property	Value
name	RD1
description	
position	center
scale	1.0
button label	Review 1
window title	Use Case 4L - Review Data
button key	selectedButton
button key location	local
primary data	<Use Editor>
primary tab label	Primary
secondary data	<Use Editor>
secondary tab label	Secondary
activation callback	
custom window path	

Review Data Step

The step properties are:

Property	Description
Position	Specifies the position of the window when in semiautomatic mode. Choices are center, topLeft, topCenter, topRight, bottomLeft, bottomCenter, bottomRight
Scale	Factor that the window will be scaled by when in semiautomatic mode. 1.0 is full scale.
Button Label	Label for the button that will be added to the control panel for the window when in semiautomatic mode. The button is useful if the enter data window is accidentally closed, if the client is accidentally closed, or if a new client connects after the step has started but before it has completed.
Window Title	The title for the window when in semiautomatic mode.
Button Key	The recipe data key where the button that the operator pressed will be stored. The recipe data type is generally a Simple Value. DO NOT include the ".value" attribute.
Button Key Location	Scope where the button that the operator pressed will be recorded.
Primary Data	See Below
Primary Tab Label	Label for the second tab if secondary data is present.
Secondary Data	See Below

Property	Description
Secondary Tab Label	Label for the second tab if secondary data is present.
Activation Callback	Custom Python script that will be executed in the gateway after the review data is inserted into the database and before the message is sent to clients to open the window. This gives the system a chance to perform some really site specific customization of the data in the database.
Custom Window Path	Path to a custom window if the standard window is not appropriate. The window will need to meet certain standards so that the standard support code works.

The configuration for the primary tab is shown below. Note that the first three rows specify that the data will come from recipe data and that the fourth and sixth rows specify hard-coded data using the "value" destination. Notice that the units for row 1 and 3 are M and DEGF even though the referenced recipe data (not shown) is specified in FT and DEGF. The values will be automatically converted.

Config Key	Value Key	Destination	Prompt	Units
row1	val1.value	operation	Value 1:	M
row2	val2.value	operation	Value 2:	
row3	val3.value	operation	Value 3:	DEGF
row4	54.78	value	A Value:	PSI
row5				
row6	76.99	value	Almost 77	

Primary Tab Specification

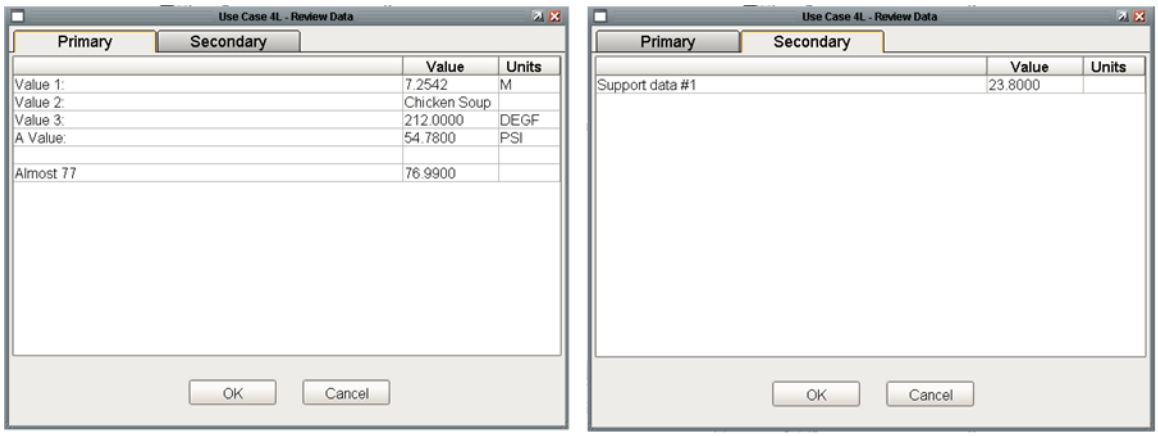
Config Key	Value Key	Destination	Prompt	Units
row1	val1.value	operation	Support data A	

Secondary Tab Specification

The columns are:

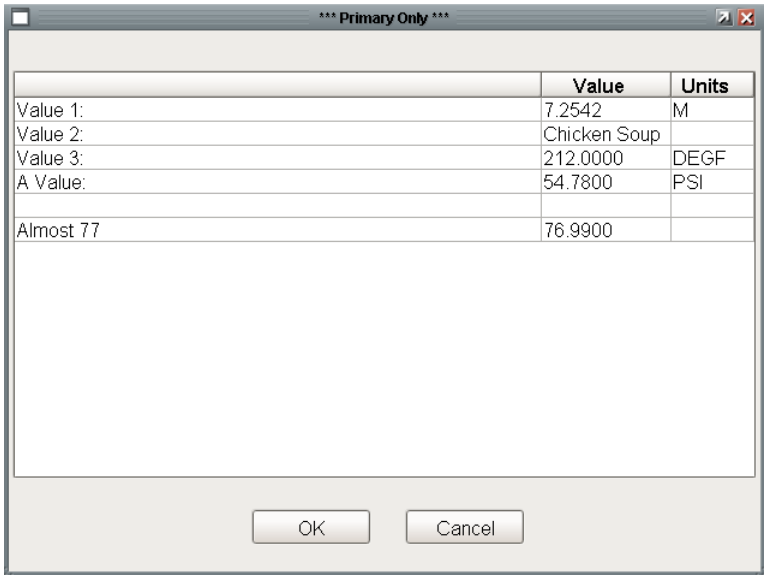
Property	Description
Config Key	An n dimensional array of any datatype. A
Value Key	The key and attribute of the recipe data that will be displayed.
Destination	Recipe data scope where the recipe data is located.
Prompt	String that will be displayed in the first column
Units	Engineering Units that define the units that the recipe data will be displayed in. Conversion between the recipe data units and the display units is automatic.

The user interface that is displayed when the step runs is shown below.



Review Data Step Run-Time Window

The same step without the secondary tag data would look like:



Review Data Task with Primary Data Only

Review Data with Advice Step

The Review Data with Advice task posts a window that is very similar to the Review Data task discussed in the previous section. The only difference is that this screen contains an additional column for displaying advisory text. The advisory text can be whatever is appropriate.

The task and its property editor, which is identical to the Review Data step, are shown below:

Property	Value
name	Review
description	Test Review Data with advice step
position	center
scale	1.0
button label	Power
window title	Use Case 4L - Review With Advice
button key	selectedButton
button key location	local
primary data	<Use Editor>
primary tab label	Boiler Room
secondary data	<Use Editor>
secondary tab label	Steam Room
activation callback	
custom window path	

Review Data with Advice Step

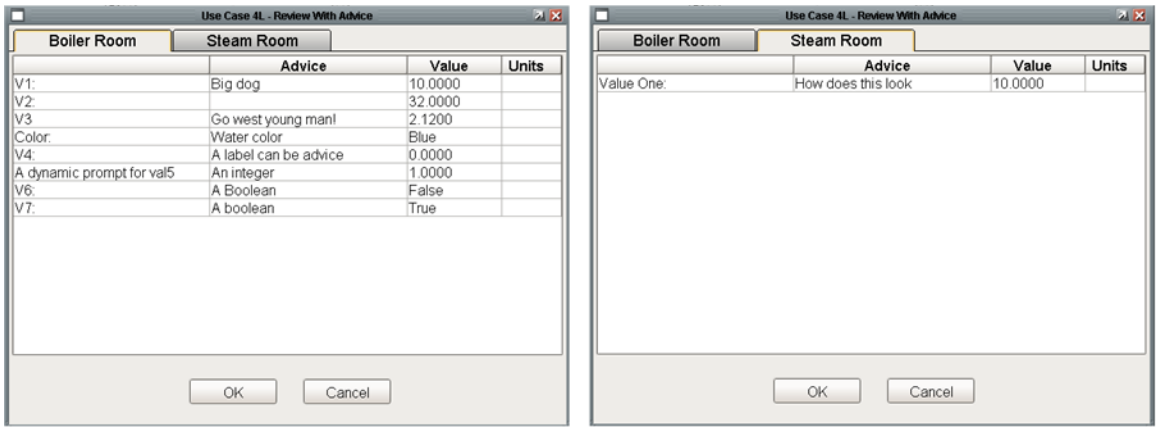
The configuration for the primary and secondary tabs are shown below. The only difference between this step and the Review Data is the "Advice" Column.

Config Key	Value Key	Destination	Prompt	Units	Advice
r1	val1.value	operation	V1:		Big dog
r2	val2.value	operation	V2:		
r3	val3.value	operation	V3		{operation.val3.description}
r4	Blue	value	Color:		Water color
r5	val4.value	operation	V4:		{operation.val4.label}
r6	val5.value	operation	{operation.val5.label}		An integer
r7	val6.value	operation	V6:		A Boolean
r8	val7.value	operation	V7:		A boolean

Config Key	Value Key	Destination	Prompt	Units	Advice
r1	val1.value	operation	Value One:		How does this look

Primary and Secondary Tab Specification

The following user interface is displayed when the block executes.

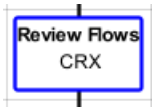


Review Data with Advice Run-Time Window

Both the prompt and advice columns support dynamic text using the *{scope.key.attribute}* notation. This is demonstrated in the *Advice* column for r3 and r5 in **Error! Reference source not found.** nd for the Prompt for r6 in the same figure.

Review Flows Step

The **Review Flows** step is similar to the **Review Data** step described above. The difference is that this block is designed to present three columns of data with the additional capability that the third column may be the sum of the first two. The block is named review flows because it was designed to address the situation where each raw material is fed through two inlets and the individual and total feed rates are of interest. The individual feeds could be left and right, top and bottom, R1 and R2, etc. So while the block is fairly specialized; it can still be adapted for a number of scenarios. The step and its properties are shown below:



Property	Value
name	CRX
description	
position	center
scale	1.0
button label	Flows
window title	C-Reactor Flow Review
button key	button
button key location	local
heading1	Top
heading2	Bottom
heading3	Sum
review flows	<Use Editor>
primary tab label	Pete's First Page
secondary data	<Use Editor>
secondary tab label	Pete's Second Page
activation callback	
custom window path	

Review Flows Step

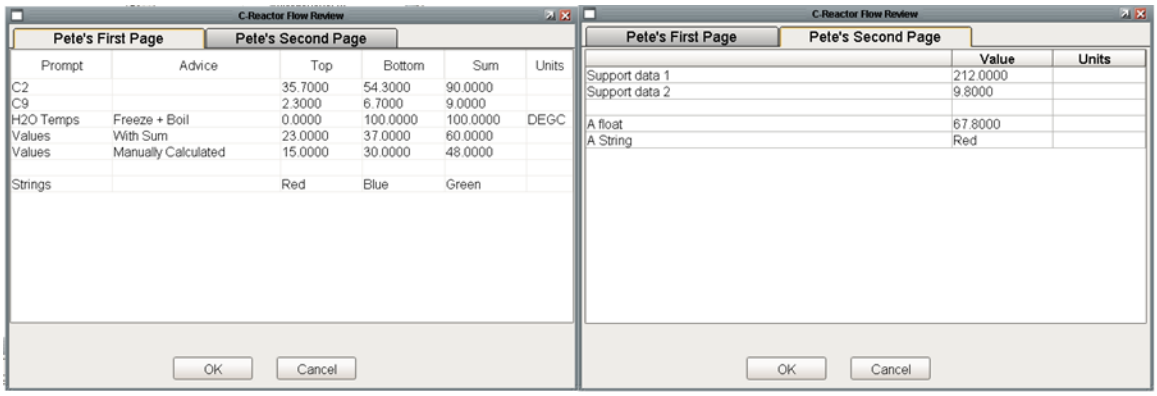
The configuration for the primary and secondary tabs are shown below. The only difference between this step and the Review Data with Advice step is that instead of a single value column, there are three: *Flow 1, Flow 2*, and *Total Flow*. Total Flow can contain a recipe reference, a hard-coded value, or the magic word "sum" in which case the values in the first two columns will be summed.

Config Key	Flow 1	Flow 2	Total Flow	Destination	Prompt	Units	Advice
c2	c2Top.value	c2Bottom.value	sum	operation	C2		
c9	c9Top.value	c9Bottom.value	sum	operation	C9		
Temps	freeze.value	boil.value	sum	operation	H2O Temps	DEGC	Freeze + Boil
Vals1	23	37	sum	value	Values		With Sum
Vals2	15	30	48	value	Values		Manually Calcu
Vals3	Red	Blue	Green	value	Strings		

Config Key	Value Key	Destination	Prompt	Units
row1	val1.value	operation	Support data 1	
row2	val2.value	operation	Support data 2	
row3				
row4	67.8	value	A float	
row5	Red	value	A String	

Primary and Secondary Tab Specification

The run time window that is displayed to interested clients is:



Review Flows Run Time Window

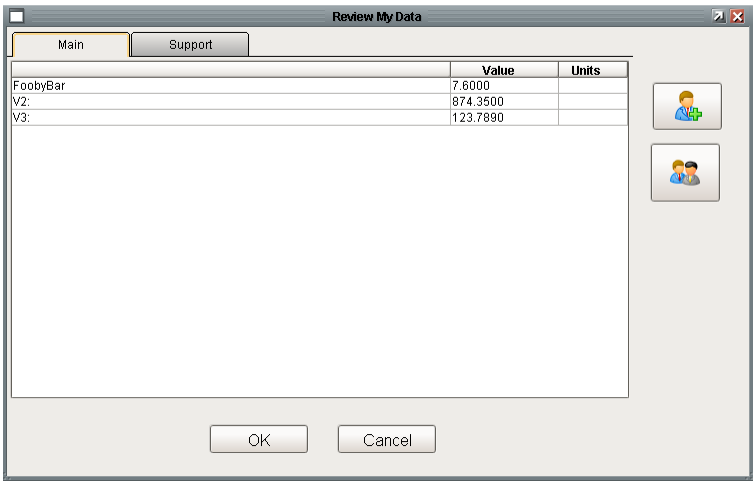
Advanced Capabilities

There are two advanced capabilities that apply to all three steps. The Custom Window Path and Activation Callback both provide a high degree of customization. The following step demonstrates both features:

Property	Value
name	Review
description	
position	center
scale	1.0
button label	review
window title	Review My Data
button key	selectedButton
button key location	global
primary data	<Use Editor>
primary tab label	Main
secondary data	<Use Editor>
secondary tab label	Support
activation callback	ils.sfc.tests.useCase4m.activationCallback
custom window path	SFC/Test/ReviewData

Review Data Step with Advanced Capabilities

A custom window is generally used when additional buttons or information is desired on the window. The custom window needs to have the same tables, tab strip, and root container properties to still operate within the Review Data Framework. If the OK and Cancel buttons are replaced, the replacements should call the same callbacks. An example window is shown below:



Custom Review Data Window

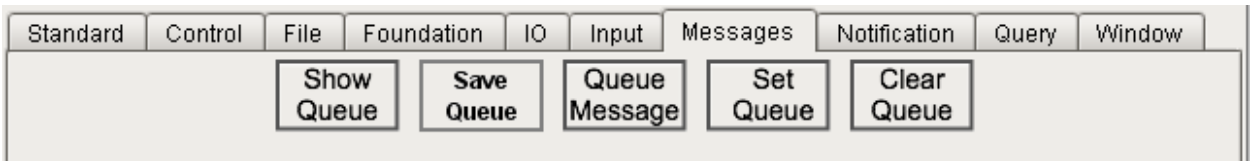
An Activation Callback is a Python script that will be executed in the gateway after the review data is inserted into the database and before the message is sent to clients to open the window. This provides an opportunity to perform some site specific customization of the data in the database. An activation callback is shown below which updates the advice for one of the rows.

```
def activationCallback(scopeContext, stepProperties):
    print "In %s.activationCallback() " % (__name__)

    chartScope = scopeContext.getChartScope()
    stepScope = scopeContext.getStepScope()

    # Test that the arguments that passed are usable
    title = getStepProperty(stepProperties, WINDOW_TITLE)
    database = getDatabaseName(chartScope)
    windowId = stepScope[WINDOW_ID]

    prompt = "FoobyBar"

    SQL = "update SfcReviewDataTable set prompt = '%s' where windowId = %s and configKey = 'row1' and isPrimary = 1" % (prompt, str(windowId) )
    print SQL

    rows = system.db.runUpdateQuery(SQL, database)
    if rows != 1:
        print "Warning: %d rows were updated when exactly 1 was expected.  SQL = %s" % (rows, SQL)

    print "-------------------------"
    print "Title:    ", title
    print "Database: ", database
    print "Window Id: ", windowId
    print "-------------------------"
```

Review Data Activation Callback

4.8 Message Palette

The Messages palette contains five steps for interacting with message queues. Message queues are discussed in detail in the *Common Facilities User's Guide*. Message queues are especially useful when working with SFCs to provide high level logging of steps taken by a SFC. The blocks provide complete control over the Message Queue facility. All of the queue steps use the message queue that is specified by the Unit Procedure.



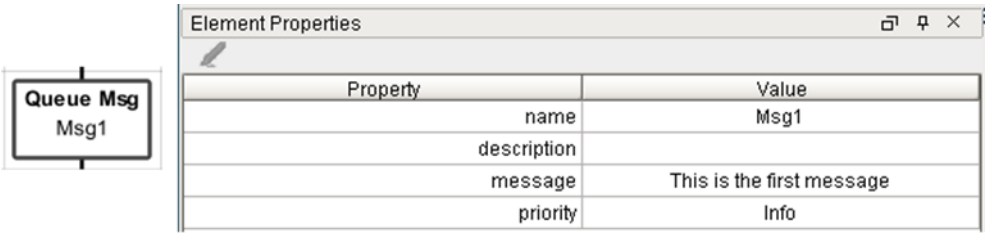
Message Step Palette

4.8.1 Clear Queue Step

This step does not have any step specific properties. When it runs it will clear all of the messages in the queue named by the unit procedure using the standard message queue interface. Good Practice is to clear the queue at the beginning of an operation. It is also good practice to NOT clear the queue at the completion of an operation so that the steps taken during the operation can be reviewed.

4.8.2 Post Message Step

This step will insert a message into the message queue named by the unit procedure. A Post Message step and its properties are shown below. The step supports the three priorities provided by the message queue facility: info, warning, and error. The animation of the messages is left to the message queue facility, but typically info messages have a white background, warnings a yellow background, and errors are red. This step posts a very simple message into the message queue at info level.

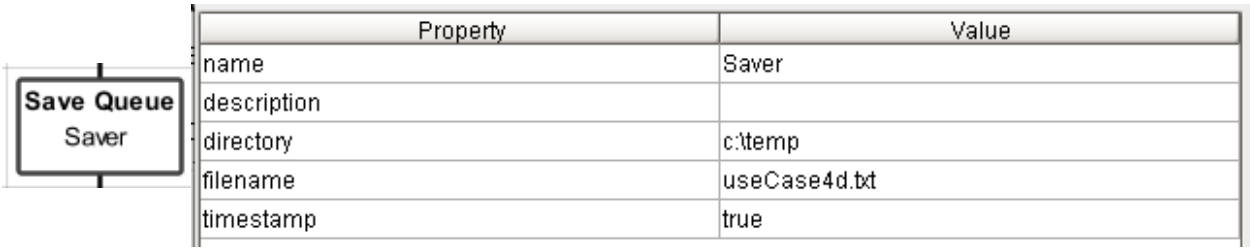


Post Message Step and Properties

The step supports dynamic text substitution using tag or recipe data values as shown below:

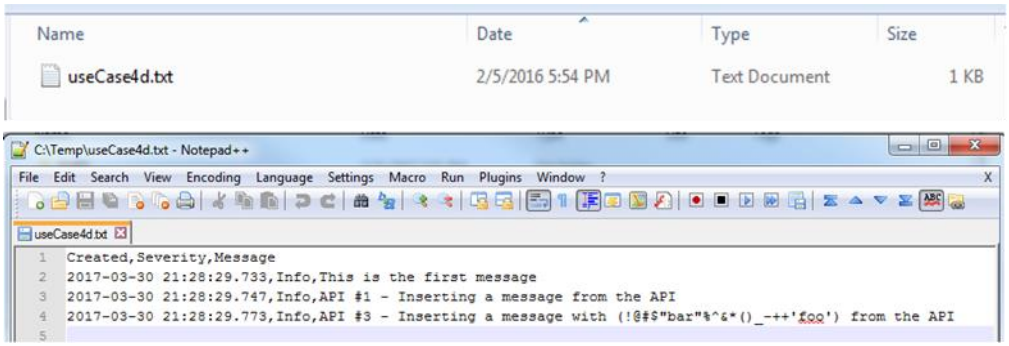
4.8.3 Save Queue Step

This step is used save the current contents of a queue to a file for permanent archiving. Good practice is to save the queue to a file at the end of an operation. A save queue task and its properties are shown below:



Save Queue Step and Properties

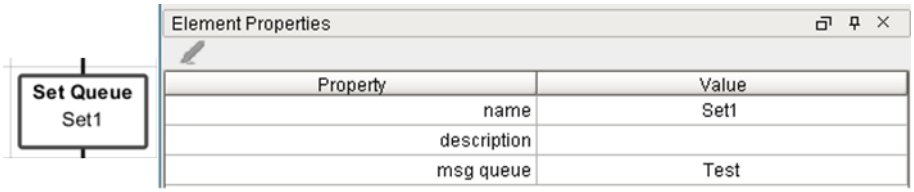
When the step runs it will save the contents of the queue to a text file as shown below.



Save Queue File and Format

4.8.4 Set Queue Step

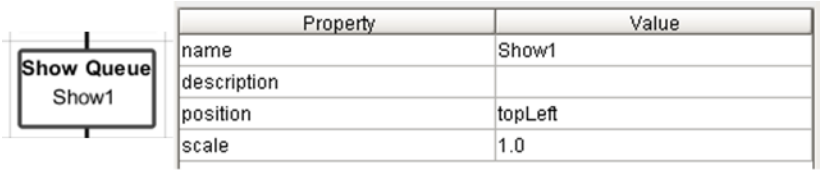
This step is used to set the queue for the unit procedure.



Set Queue Step and Properties

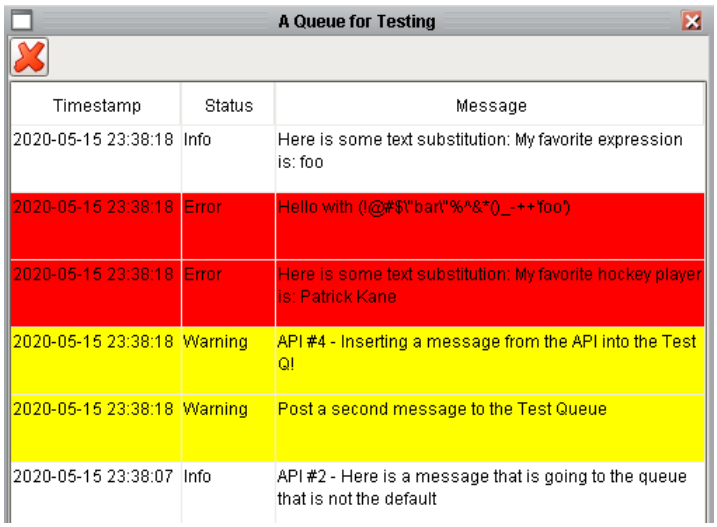
4.8.5 Show Queue Step

This step will post the standard Queue View window on all interested clients. The step and its properties re shown below. The step allows you to specify the position and scale of the window. This will show the contents of the queue referenced by the unit procedure.



Show Queue Step and Properties

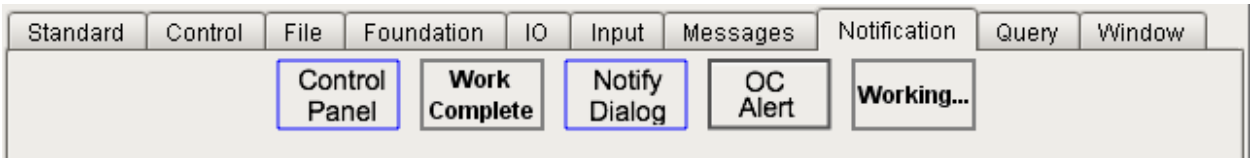
The window that is displayed is provided by the Message Queue facility and is shown below:



Message Queue Window

4.9 Notification Palette

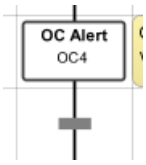
The Notification palette contains three steps for notifying the operator and one step for removing the notification.



Notification Palette

4.9.1 OC Alert Step

The OC Alert step interfaces to the OC Alert package to post the "loud" workspace to the appropriate clients.



Property	Value
name	OC4
description	
post	XO1TEST
top message	The rain in Spain...
bottom message	...falls mainly in the plain.
main message	My Fair Lady
button label	Acknowledge
button callback	ils.common.ocAlert.testCallback
oc alert window	Common/OC Alert
ack required	☑

OC Alert Step and Properties

The GUI that is posted is shown below:

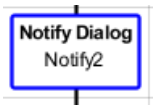
The Standard OC Alert Window

The step properties are:

Property	Description
Post	The post that will determine which clients will display the window.
Top Message	The text for the message at the top of the OC alert window.
Bottom Message	The text for the message at the bottom of the OC alert window.
Main Message	The text for the message in the middle of the OC alert window.
Button Label	The text for the button in the middle of the OC alert window.
Button Callback	A user written callback that will be called when the button is pressed. The callback will execute in the client and must take two arguments: event, and payload. Payload is a dictionary with three keys: chartId, stepId, and the button callback.
OC Alert Window	The name of the Vision window that will be posted. The default is the built-in window. A custom window may be substituted if it has the same properties as a minimum.
Ack Required	If True, then a client must acknowledge the window before chart execution will proceed.

4.9.2 Notify Dialog / Post Dialog Step

The Notify Dialog Step is designed to post a generic window with a custom message and a button to dismiss the window. The step contains a property, ack required, that if True requires that the window is acknowledged before execution proceeds.



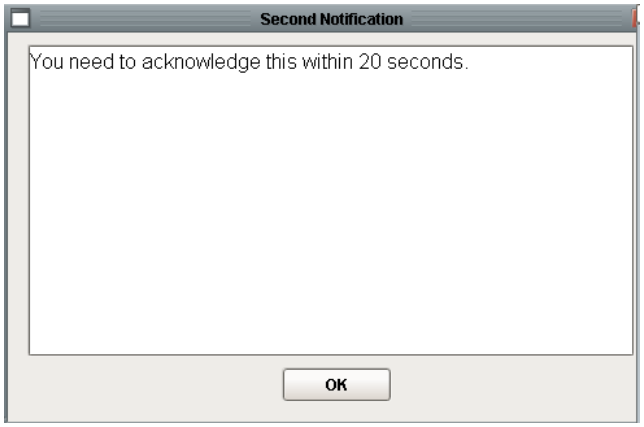
Property	Value
name	Notify2
description	
position	topRight
scale	1.0
button label	Notice 2
window title	Second Notification
message	You need to acknowledge this within 20 seconds
strategy	static
recipe location	local
key	
window	
ack required	☑

Notify Dialog Step and Properties

The step properties are:

Property	Description
Position	Specifies the position of the window when in semiautomatic mode. Choices are center, topLeft, topCenter, topRight, bottomLeft, bottomCenter, bottomRight
Scale	Factor that the window will be scaled by when in semiautomatic mode. 1.0 is full scale.
Button Label	Label for the button that will be added to the control panel for the window when in semiautomatic mode. The button is useful if the enter data window is accidentally closed, if the client is accidentally closed, or if a new client connects after the step has started but before it has completed.
Window Title	The title for the window when in semiautomatic mode.
Message	The message that will be use in the center of the window.
Strategy	Not used.
Recipe Location	Not used.
Key	Not used.
Window	Not used.
Ack Required	If True, then a client must acknowledge the window before chart execution will proceed.

The step posts the following window on interested clients:



Notify Dialog Window

4.9.3 Post Delay Notice Step

The Post Notice step displays a window that can only be removed by the Delete Notice step. The step is used to display a window while the chart proceeds to do work. The step has the normal window properties (*position*, *scale*, *button label*, and *window title*). The step specific property: *message*, contains the step specific message. As shown below, the message can contain a dynamic expression in curly braces.

Property	Value
name	TextSubstitution
description	
position	center
scale	1.0
button label	DText
window title	Dynamic Text
message	Here is some text substitution: My favorite expression is: {tag.SFC/Text_1}

Post Notice Step and Properties

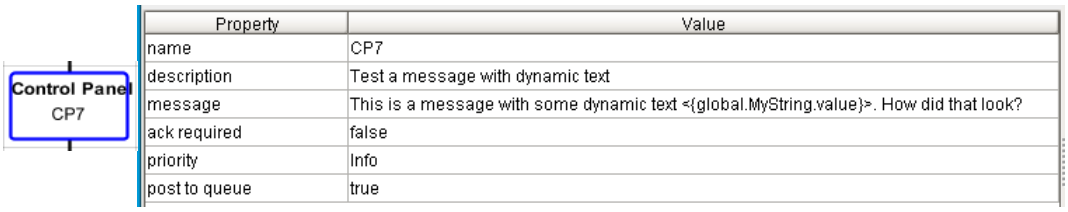
4.9.4 Delete Delay Notice Step

The Delete Notice step will delete all of the notice windows that have been posted. The step does not have any configurable properties.

Work Compete Step

4.9.5 Control Panel / Post Message Step

The Control Panel message is used to post a message to the control panel message center. A Control Panel step and its properties is shown below.



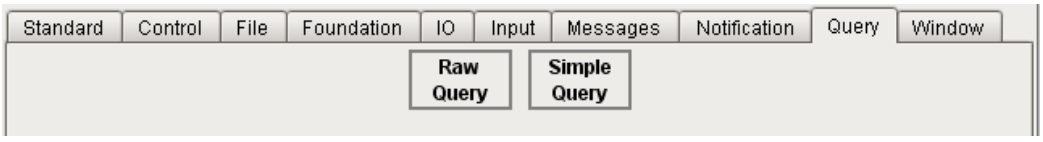
Control Panel Step and Properties

The step properties are:

Property	Description
Message	The message that will be posted to the message center. The message supports dynamic text substitution using curly braces.
Ack Required	If True, then a client must acknowledge the window before chart execution will proceed and the message will flash between white and blue to catch the operator's attention.
Priority	Info, Warning, or Error. The priority is used to animate the message using: white, yellow, and red respectively.
Post To Queue	If True then the message will also be posted to the message queue. Generally, if the message is important enough to bring to the operator's attention it should also be logged to the message queue so that it will be permanently archived.

4.10 Query Palette

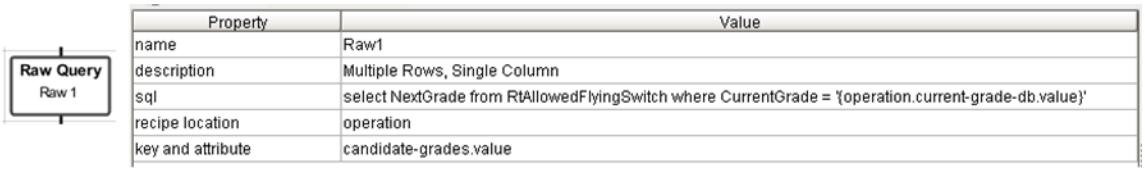
The Query palette has two steps that make it easy to integrate data from a SQL database into a running SFC.



Query Palette

4.10.1 Raw Query Step

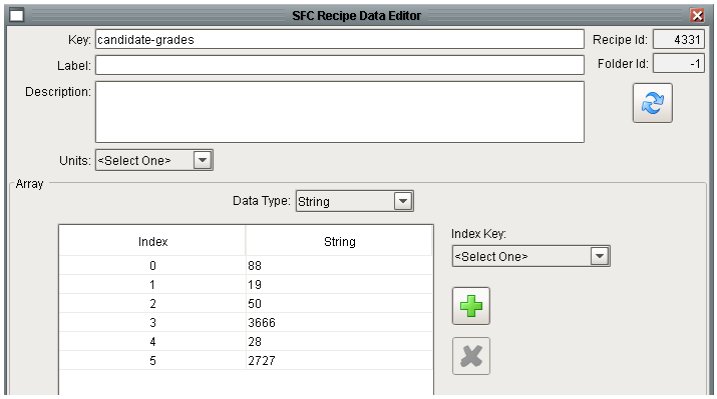
The Raw Query Step is the easiest way to integrate data from a SQL database. It is designed to handle a single column from one or more rows and to store the results into a preexisting recipe data array. A Raw Query step and its properties are shown below. The SQL statement can contain a dynamic reference enclosed by curly braces that will be evaluated before sending the SQL to the database.



Property	Value
name	Raw1
description	Multiple Rows, Single Column
sql	select NextGrade from RtAllowedFlyingSwitch where CurrentGrade = '{operation.current-grade-db.value}'
recipe location	operation
key and attribute	candidate-grades.value

Raw Query Step and Properties

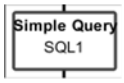
The recipe data after running the step is shown below:



Raw Query Results in Recipe Data

4.10.2 Simple Query Step

The Simple Query Step is designed to query any SQL database table and store the results into recipe data without writing any custom Python. A Simple Query step and its property table is shown below. The SQL executes against the database connection determined by the production / isolation mode settings.



Property	Value
name	SQL1
description	Test a simple query in update mode with a static key
sql	select max(FamilyPriority) value from DtFamily
results mode	update
class to create	sqc
fetch mode	single
recipe location	operation
key mode	static
key	val1

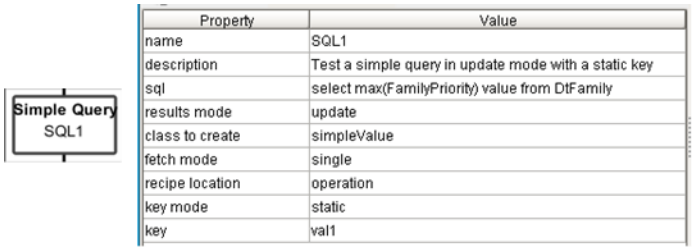
Simple Query Step and Properties

The properties are:

Property	Description
sql	The SQL statement that will execute.
results mode	update or updateOrCreate

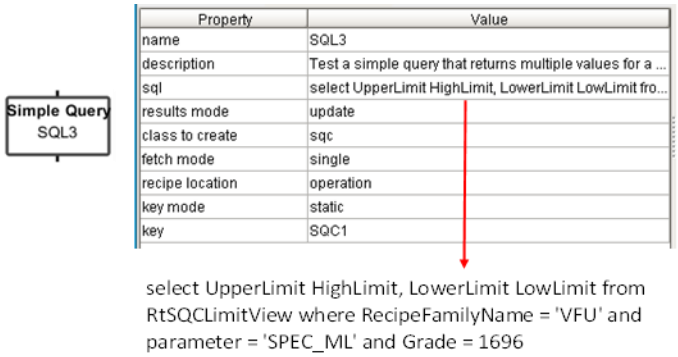
Property	Description
class to create	The class of recipe data to create. Only relevant if mode is updateOrCreate and new recipe data needs to be created.
fetch mode	single or multiple. Limits the amount of data that will be processed.
recipe location	The recipe data scope for the results.
key mode	static or dynamic. If static, then the key for updating the recipe data is specified in the *key* property.
key	Optional – only applies if the key mode is static and only valid if fetch mode is single.

The performance of this step is best demonstrated through a series of examples. The first example fetches a single record from the database and updates a single recipe data item at: *operation.val1.value*. The attribute, *value,* is determined by the column alias in the SQL statement. Because the results mode is update, if the key that is specified does not exist then the operator will be notified and the unit procedure cancelled.



Simple Query Update of a Single Record

The next example is very similar to the first except that the query updates multiple values in a single recipe data item. This example updates the HighLimit and LowLimit attributes of the SQC recipe data at operation scope with key SQC1. Even though multiple values are returned, a single record is returned and a single recipe data item are updated so it meets the single and update properties of the step.

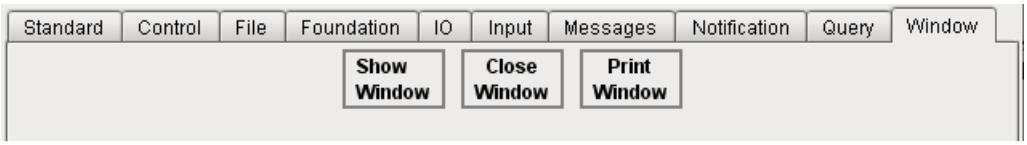


Simple Query that Updates Multiple Values in a Single Query

Dynamic SQL doesn't quite work correctly

4.11 Window Palette

There are three steps that provide a convenient way to manage windows. There are steps to open, close, and print a window.



Window Palette

4.11.1 Show Window Step

The Show Window step is used to display a Vision window on clients. A Show Window step and its properties is shown below.

Property	Value
name	Show Window 1
description	
position	topLeft
scale	1.0
button label	Screen 1
window title	
window	SFC/Test/Use Case 4h Screen 1
security	private
is sfc window	false

Show Window Step and Properties

Property	Description
Position	Specifies the position of the window when in semiautomatic mode. Choices are center, topLeft, topCenter, topRight, bottomLeft, bottomCenter, bottomRight
Scale	Factor that the window will be scaled by when in semiautomatic mode. 1.0 is full scale.
Button Label	Label for the button that will be added to the control panel for the window when in semiautomatic mode. The button is useful if the enter data window is accidentally closed, if the client is accidentally closed, or if a new client connects after the step has started but before it has completed.
Window Title	The title for the window.
Window	The full path to the window.
Security	public or private. If public then every client will open the window.

Property	Description
Is sfc window	A Boolean, if True then the window must have a windowId property. A SFC window has the capability of being tiled if multiple instances of the window are open.

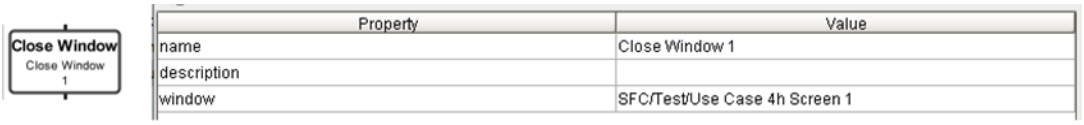
SFCs run in the gateway and it is not possible for the gateway to open a window on a client. The mechanism is for the gateway to send a message to the client and then the client opens the window. When the step runs, the properties are packaged into a message payload and sent in a message to every client running the same project that initially launched the SFC.

Because the message is sent to every client, there are some rules to determine if the client should show the window:

- The window will not be shown if the production / isolation mode does not match. A message sent from an isolation SFC will not show a window on a production client
- The window will be shown if security is *public*.
- The window will be shown client has the same control panel open that was used to launch the original SFC.
- The client has the same username as the client that launched the SFC.
- Otherwise, the window will not be shown.

4.11.2 Close Window Step

The Close Window step is used to close a Vision window on clients. A Close Window step and its properties is shown below.

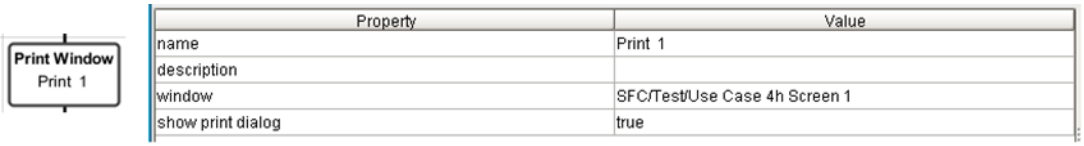


Close Window Step and Properties

This step has a single property, window, which is the path of the window. Because the SFC runs in the gateway, a message is sent to every client connected to the gateway to close the named window.

4.11.3 Print Window Step

The Print Window step is used to print a window that is already open. A Print Window step and its properties are shown below.



Print Window Step and Properties

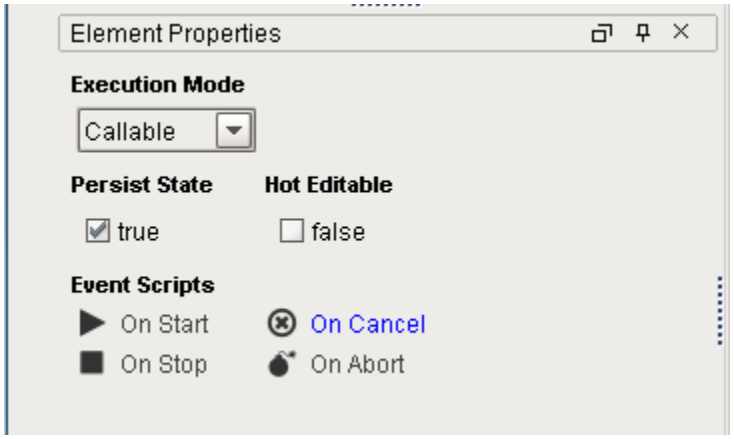
SFCs run in the gateway and a window can only be printed from a client. When this step runs it sends a message to every client. If the window is open in a client then the window will be immediately printed unless "Show Print Dialog" is True in which case the native print window will be posted from which the user can choose a printer, cancel the print, etc.

5. ERROR HANDLING

Error handling is critical for any robust control system. While an error fre system is the goal, the system needs to be designed to handle unexpected errors in a controlled safe way. This section describes facilities provided by the IA SFC framework and best practices for using the framework to design reliable applications.

5.1 Ignition Facilities

The Ignition SFC framework allows for every chart to define four different chart level event scripts, On Start, On End, On Cancel, and On Abort:



Standard Ignition Chart Handlers

Ignition	Description
On Start	This gets called when a chart begins executing before any step executes.
On Stop	This is called when the chart completes normally.
On Cancel	This is called in response to a user requested Cancel command. The cancel command can be issued to any executing chart. The running chart will transition to the "cancelling" state while the cancel command is sent to each chart called by an enclosing step on the chart. Once all of the sub charts have been cancelled, then the parent chart will be "cancelled".
On Abort	This is automatically called by the Ignition SFC framework when an unexpected error is detected on the chart. Unlike the cancel command, there is no way to "send" an abort command, nor is there an API to "abort" a chart.

5.1.1 Cancel Handlers

This section focuses on the cancel handler. As described above, a chart may be cancelled by a user in one of several ways:

- From the SFC control panel

- By inserting a Cancel step into a chart
- Selecting "Cancel" on a running chart from the SFC viewer
- By calling an *system.sfc.cancelChart()* with the chart id

The following example demonstrates how the cancel command and onCancel handlers work hand in hand. It demonstrates that by sending the cancel command to the top chart Ignition will cancel the charts from the bottom up.

The example has the normal unit procedure, operation, phase, task structure. The operation runs three phases in parallel which call three different charts which will run forever.

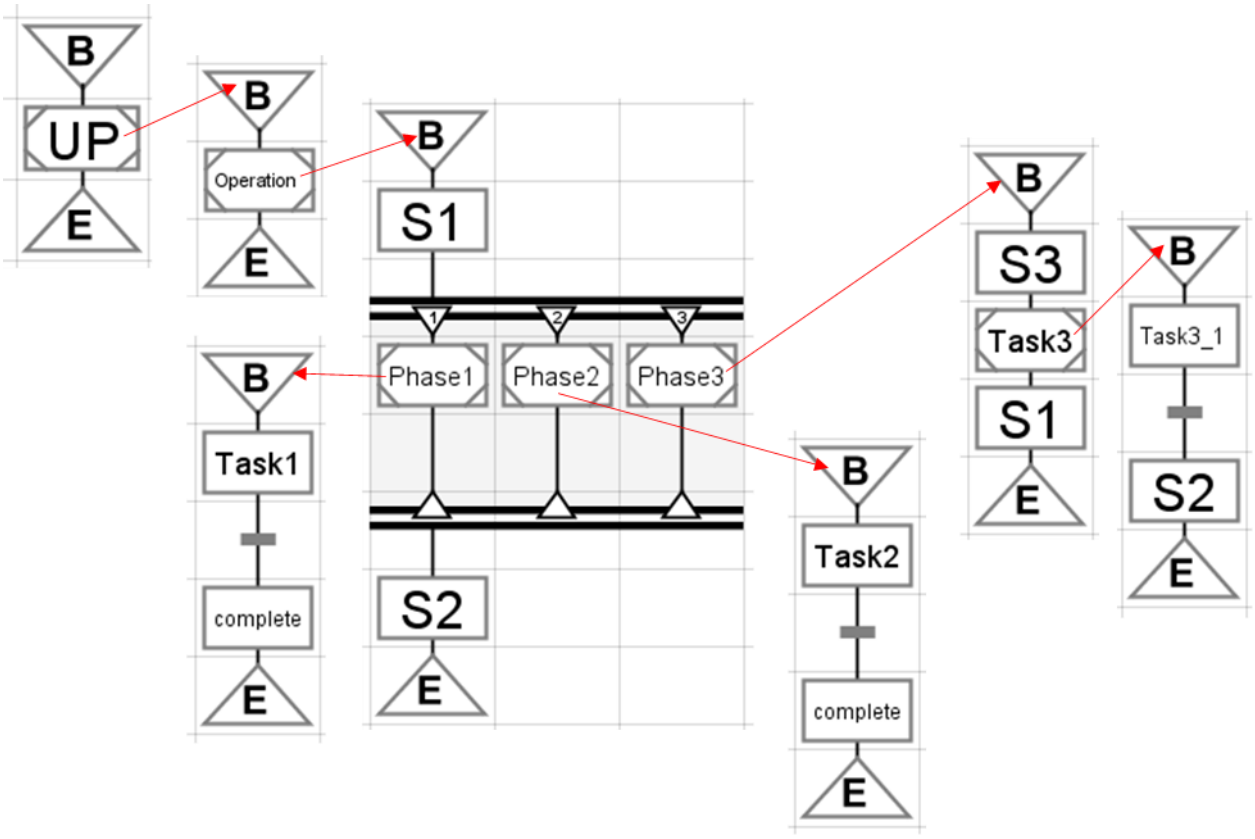


Chart Demonstrating Cancel Handlers

All of the transitions are hard-coded to be permanently false. This has the effect that the chart will run forever with the tasks named: Task1, Task2, and Task3_1 repeating infinitely. Each of these will print a message to the wrapper log every second. Every chart has an "On Cancel" handler that is simply a print statement similar to the one shown below (for the Operations chart).

```
def onCancel(chart):
    """
    This will run once if the chart is cancelled.

    Arguments:
        chart: A reference to the chart's scope.
    """
    print "Starting onCancel for Operations..."
    import time
    time.sleep(2)
    print "...finished onCancel for Operations!"
```

onCancel Handler

The unit procedure was started and execution was allowed to reach the transitions. Then, from the Control panel, the Cancel button was pressed. The wrapper log shows that the onCancel handlers for the three phases all start simultaneously and when all three are complete then Ignition walks up the call stack executing the cancel handlers as it goes, waiting for each to complete before proceeding to the next one.

```
| 2016/10/25 08:09:38 | ...task3_1 is working...
| 2016/10/25 08:09:39 | ...phase 2 work...
| 2016/10/25 08:09:39 | ...phase 1 work...
| 2016/10/25 08:09:40 | Starting onCancel for Phase1...
| 2016/10/25 08:09:40 | Starting onCancel for Phase2...
| 2016/10/25 08:09:40 | Starting onCancel for Task3_1...
| 2016/10/25 08:09:42 | ...finished onCancel for Phase1!
| 2016/10/25 08:09:42 | ...finished onCancel for Phase2!
| 2016/10/25 08:09:42 | ...finished onCancel for Task3_1, !
| 2016/10/25 08:09:42 | Starting onCancel for Phase3...
| 2016/10/25 08:09:44 | ...finished onCancel for Phase3!
| 2016/10/25 08:09:44 | Starting onCancel for Phases...
| 2016/10/25 08:09:46 | ...finished onCancel for Phases!
| 2016/10/25 08:09:46 | Starting onCancel for Operations...
| 2016/10/25 08:09:48 | ...finished onCancel for Operations!
| 2016/10/25 08:09:48 | Starting onCancel for UseCase8d...
| 2016/10/25 08:09:50 | ...finished onCancel for UseCase8d!
```

Wrapper Log for Cancelled Chart

The time delays in the handlers accentuate the fact that the handlers are called from the bottom up and they wait until the lower level handlers complete before walking up the hierarchy.

5.1.2 The Abort Handler and the "Silent Error"

When performing critical operations in a production environment there really isn't such a thing as an acceptable error, but the worst thing that can happen is that an error occurs, the operator is not notified, and the operation continues as if nothing happened even though some critical steps may have been skipped. We refer to this as a "Silent Error". Unfortunately, unless we pay attention to adding abort handlers this is exactly what will happen.

When a step on a chart throws an error, then that chart aborts, execution is transferred to the abort handler on that chart, if one exists, and then execution is returned to the calling step and continues as if nothing happened. The presence of an abort handler does not change the flow of execution which returns to the calling chart and continues from there just as if no error occurred.

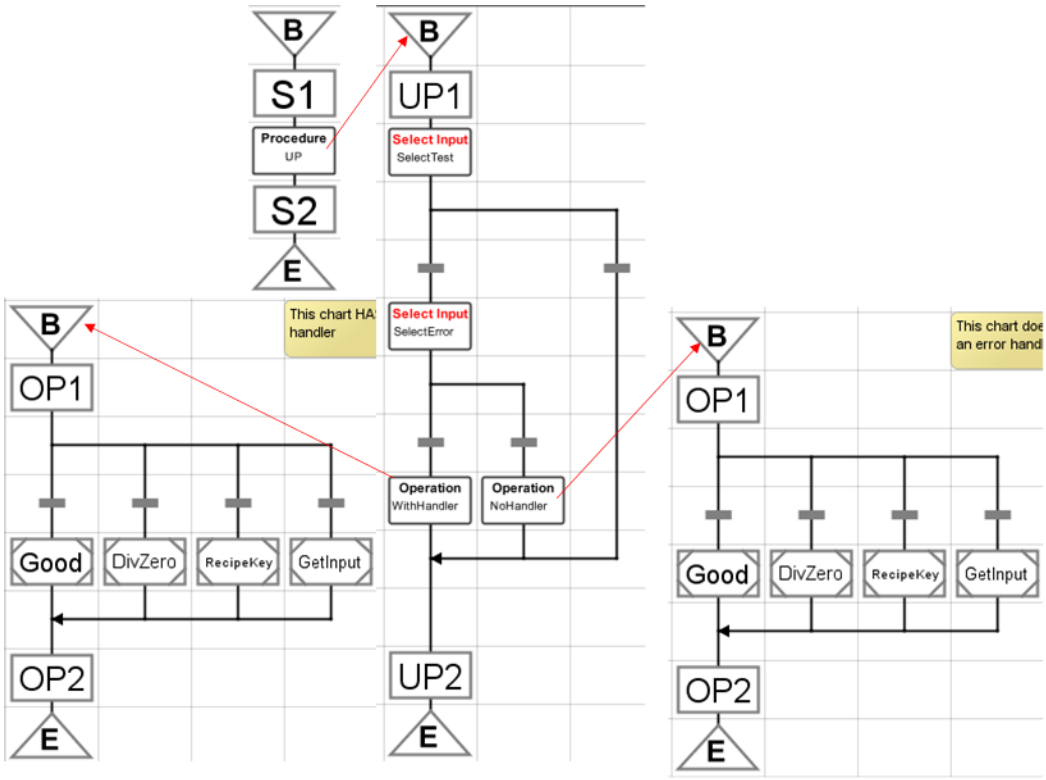
This is demonstrated in the following unit procedure which tests three types of errors:

- A divide by zero error in Python in an action step
- A recipe data access error in Python in an action step
- An error in a custom ILS step due to missing recipe data

The first two cases are identical from an error handling perspective. All ILS custom steps handle errors differently and will be discussed in section 5.2.1 so only the divide by zero and custom step error handling will be discussed. Each of these were tested with and without an abort handler. Abort handlers were defined on each calling charts. Cancel handlers were defined on every chart. The hierarchy of charts are:

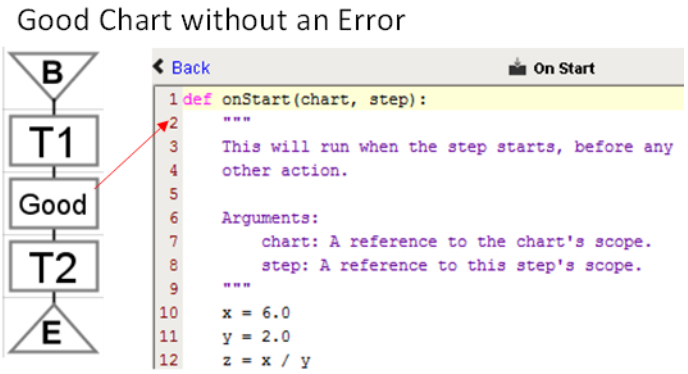
Use Case Charts for Testing Silent Error

The main charts are shown below:



Main Charts for Testing Silent Error

First, consider the case with an action step that does not have an error.



Error-Free Task

The SFC viewer widget shows that each chart successfully completes.

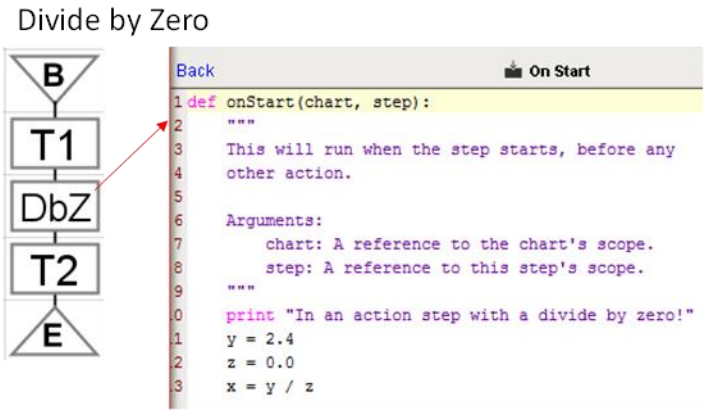
Chart Viewer Status for Error-Free Chart

The action steps print messages to the wrapper log which show the flow of execution.

```
In step S1 - chart Use Cases/Use Case 8k Error Handling/UseCase8k is starting...
T [U.U.UseCase8k                ] [17:53:35]: In ils.sfc.gateway.steps.procedure.activate() sfc-chart-id=c
T [U.U.UseCase8k                ] [17:53:35]: chart Scope: {u'isolationMode': False, u'parent': None, u'cc
T [U.U.UseCase8k                ] [17:53:35]: Step Properties: {return-parameters=[], name=UP, chart-path=
I [U.U.UseCase8k                ] [17:53:35]: The chart run id is: c11f197d-2c6c-4946-9b10-d37095fe3f09 st
T [U.U.UseCase8k                ] [17:53:35]: Database: XOM sfc-chart-id=c11f197d-2c6c-4946-9b10-d37095fe3
In step UP1 - chart Use Cases/Use Case 8k Error Handling/Operations is starting...
In ils.sfc.common.notify.sfcNotify(), Sending <sfcMessage> message to project: <XOM>, post: <XO1TEST>, payl
I [U.U.Operations               ] [17:53:45]: Step SelectTest in Use Cases/Use Case 8k Error Handling/Oper
In ils.sfc.common.notify.sfcNotify(), Sending <sfcMessage> message to project: <XOM>, post: <XO1TEST>, payl
I [c.i.s.s.OperationStep        ] [17:53:51]: In OperationStep.calling Python... sfc-chart-id=8329aa66-97
I [c.i.s.s.OperationStep        ] [17:53:51]: In OperationStep.activating Step... sfc-chart-id=8329aa66-97
I [c.i.s.s.OperationStep        ] [17:53:51]: In OperationStep.doneActivating()
In step OP1 - chart Use Cases/Use Case 8k Error Handling/PhaseWithoutHandler is starting...
In step T1 - chart Use Cases/Use Case 8k Error Handling/ActionStepGood is starting...
In step T2 - chart Use Cases/Use Case 8k Error Handling/ActionStepGood completed!
In step OP2 - chart Use Cases/Use Case 8k Error Handling/PhaseWithoutHandler completed!
In step UP2 - chart Use Cases/Use Case 8k Error Handling/Operations completed!
In step S2 - chart Use Cases/Use Case 8k Error Handling/UseCase8k completed!
```

Wrapper Log for an Error Free Run

Next, consider a very similar chart whose action step has a divide by zero error.



Task with a Divide By Zero Error

The SFC viewer widget shows that the chart that had the divide by zero step aborted and that all of the other charts ran to completion.

Chart Viewer Status for Chart with an Error

The wrapper log below confirms that the steps following the offending step do not run but that the charts that called the offending chart run to completion as evidenced by the last three log entries.

```
| In step S1 - chart Use Cases/Use Case 8k Error Handling/UseCase8k is starting...
| T [U.U.UseCase8k                ] [16:48:02]: In ils.sfc.gateway.steps.procedure.activate() sfc-chart-id=0e63dca7-c4
| T [U.U.UseCase8k                ] [16:48:02]: chart Scope: {u'isolationMode': False, u'parent': None, u'controlPanel
| T [U.U.UseCase8k                ] [16:48:02]: Step Properties: {return-parameters=[], name=UP, chart-path=Use Cases/
| I [U.U.UseCase8k                ] [16:48:02]: The chart run id is: 0e63dca7-c4df-421c-819a-956da802507f sfc-chart-id
| T [U.U.UseCase8k                ] [16:48:02]: Database: XOM sfc-chart-id=0e63dca7-c4df-421c-819a-956da802507f, sfc-c
| In step UP1 - chart Use Cases/Use Case 8k Error Handling/Operations is starting...
| In ils.sfc.common.notify.sfcNotify(), Sending <sfcMessage> message to project: <XOM>, post: <XO1TEST>, payload: <{'ha
| I [U.U.Operations               ] [16:48:06]: Step SelectTest in Use Cases/Use Case 8k Error Handling/Operations dea
| In ils.sfc.common.notify.sfcNotify(), Sending <sfcMessage> message to project: <XOM>, post: <XO1TEST>, payload: <{'ha
| I [c.i.s.s.OperationStep        ] [16:48:11]: In OperationStep.calling Python... sfc-chart-id=c03594b2-419a-4a57-bac
| I [c.i.s.s.OperationStep        ] [16:48:11]: In OperationStep.activating Step... sfc-chart-id=c03594b2-419a-4a57-ba
| I [c.i.s.s.OperationStep        ] [16:48:11]: In OperationStep.doneActivating()
| In step OP1 - chart Use Cases/Use Case 8k Error Handling/PhaseWithoutHandler is starting...
| In step T1 - chart Use Cases/Use Case 8k Error Handling/ActionStepDivByZero is starting...
| In an action step with a divide by zero!
| W [c.i.s.ChartInstance          ] [16:48:11]: Chart 'Use Cases/Use Case 8k Error Handling/ActionStepDivByZero' abort
| com.inductiveautomation.ignition.common.script.JythonExecException: Traceback (most recent call last):
|   File "<Script name="">", line 13, in onStart
| ZeroDivisionError: float division
|
|       at org.python.core.Py.ZeroDivisionError(Py.java:255)
|       at org.python.core.PyFloat.float___div__(PyFloat.java:415)
|       at org.python.core.PyFloat.__div__(PyFloat.java:401)
|       at org.python.core.PyObject._basic_div(PyObject.java:2341)
|       at org.python.core.PyObject._div(PyObject.java:2327)
|       at org.python.pycode._pyx9372.onStart$1(<Script name="">:13)
|       at org.python.pycode._pyx9372.call_function(<Script name="">)
|       at org.python.core.PyTableCode.call(PyTableCode.java:165)
|       at org.python.core.PyBaseCode.call(PyBaseCode.java:301)
|       at org.python.core.PyFunction.function___call__(PyFunction.java:376)
|       at org.python.core.PyFunction.__call__(PyFunction.java:371)
|       at org.python.core.PyFunction.__call__(PyFunction.java:361)
|       at org.python.core.PyFunction.__call__(PyFunction.java:356)
|       at com.inductiveautomation.ignition.common.script.ScriptManager.runFunction(ScriptManager.java:647)
|       at com.inductiveautomation.sfc.elements.steps.StepScriptRunner.invoke(StepScriptRunner.java:50)
|       at com.inductiveautomation.sfc.elements.steps.StepScriptRunner.invoke(StepScriptRunner.java:38)
|       at com.inductiveautomation.sfc.elements.steps.action.ActionStep.activateStep(ActionStep.java:49)
|       at com.inductiveautomation.sfc.elements.StepContainer.onActivateRequested(StepContainer.java:142)
|       at com.inductiveautomation.sfc.fsm.element.ElementInactive.action(ElementInactive.java:35)
|       at com.inductiveautomation.sfc.fsm.element.ElementInactive.action(ElementInactive.java:8)
|       at com.inductiveautomation.sfc.fsm.StateContext.lambda$handleEvent$0(StateContext.java:49)
|       at com.inductiveautomation.sfc.api.ExecutionQueue$ThrowableCatchingRunnable.run(ExecutionQueue.java:146)
|       at com.inductiveautomation.sfc.api.ExecutionQueue$PollAndExecute.run(ExecutionQueue.java:120)
|       at java.util.concurrent.Executors$RunnableAdapter.call(Unknown Source)
|       at java.util.concurrent.FutureTask.run(Unknown Source)
|       at java.util.concurrent.ThreadPoolExecutor.runWorker(Unknown Source)
|       at java.util.concurrent.ThreadPoolExecutor$Worker.run(Unknown Source)
|       at java.lang.Thread.run(Unknown Source)
| Caused by: org.python.core.PyException: Traceback (most recent call last):
|   File "<Script name="">", line 13, in onStart
| ZeroDivisionError: float division
|
|       ... 28 common frames omitted
| In step OP2 - chart Use Cases/Use Case 8k Error Handling/PhaseWithoutHandler completed!
| In step UP2 - chart Use Cases/Use Case 8k Error Handling/Operations completed!
| In step S2 - chart Use Cases/Use Case 8k Error Handling/UseCase8k completed!
```

Wrapper Log for Dive By Zero Error

The "Silent Error" is discussed in more detail in use case 8k.

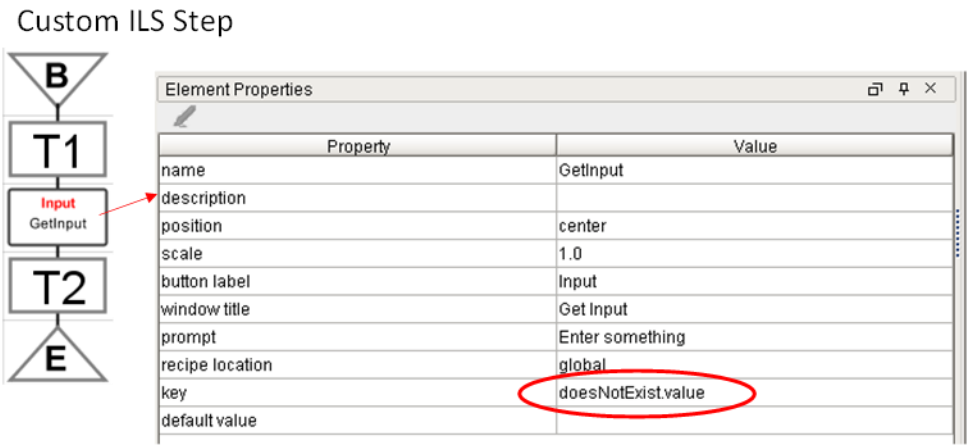
5.2 ILS Facilities

This section discusses enhancements that ILS has implemented in the ILS-SFC module to compensate for deficiencies in the standard Ignition module.

5.2.1 Error Handling in Custom Steps

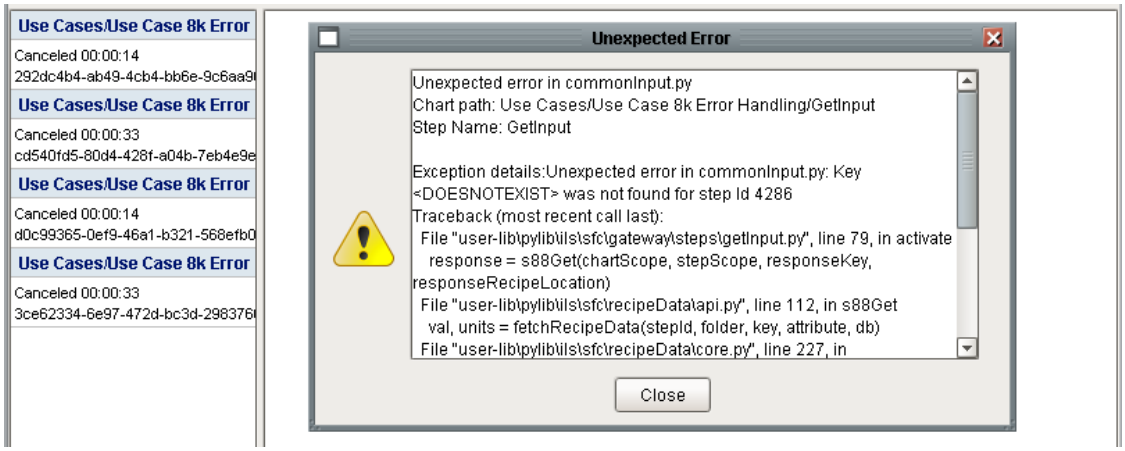
In the ILS toolkit, the practice is to send the cancel command to the top chart, the one with the unit procedure step. Ignition then propagates the Cancel command to all running sub charts which then cancel from the bottom up. The "Cancel" button on the control panel and the "Cancel" step both send the cancel command to the top chart.

Custom ILS steps handle errors differently than errors in action steps. They are implemented to avoid the "silent error" problem demonstrated above. Using the same use case example described above, the third type of step is the ILS custom "Select Input" step, shown below. The step is intentionally misconfigured with a nonexistent recipe data key. This will cause an error at run time.



Misconfigured Step Error

As the screen capture below and the wrapper log shows, the error is trapped by the step, the operator is notified of the error, and the top chart is cancelled which automatically cancels charts from the bottom up.



Runtime Error for a Misconfigured Step

The wrapper log traces the flow of execution after the error. It confirms that the offending step traps the error and then cancels the unit procedure. It also shows that the cancel handlers were called from the top down. It is important to note that no abort handlers were called.

```
In step S1 - chart Use Cases/Use Case 8k Error Handling/UseCase8k is starting...
T [U.U.UseCase8k                ] [18:44:45]: In ils.sfc.gateway.steps.procedure.activate() sfc-chart-id=dbe7b464-f03e-44fb-a382-c34d7aebbb
T [U.U.UseCase8k                ] [18:44:45]: chart Scope: {u'isolationMode': False, u'parent': None, u'controlPanelId': 4, u'project': u'X
T [U.U.UseCase8k                ] [18:44:45]: Step Properties: {return-parameters=[], name=UP, chart-path=Use Cases/Use Case 8k Error Handl
I [U.U.UseCase8k                ] [18:44:45]: The chart run id is: dbe7b464-f03e-44fb-a382-c34d7aebbba0 sfc-chart-id=dbe7b464-f03e-44fb-a38
T [U.U.UseCase8k                ] [18:44:45]: Database: XOM sfc-chart-id=dbe7b464-f03e-44fb-a382-c34d7aebbba0, sfc-chart-name=Use Cases/Use
In step UP1 - chart Use Cases/Use Case 8k Error Handling/Operations is starting...
In ils.sfc.common.notify.sfcNotify(), Sending <sfcMessage> message to project: <XOM>, post: <XO1TEST>, payload: <{'handler': 'sfcOpenWindow',
I [U.U.Operations               ] [18:44:51]: Step SelectTest in Use Cases/Use Case 8k Error Handling/Operations deactivated before complet
In ils.sfc.common.notify.sfcNotify(), Sending <sfcMessage> message to project: <XOM>, post: <XO1TEST>, payload: <{'handler': 'sfcOpenWindow',
I [c.i.s.s.OperationStep        ] [18:44:55]: In OperationStep.calling Python... sfc-chart-id=48e329ab-5458-4bf7-9efb-71d0063dc2e3, sfc-cha
I [c.i.s.s.OperationStep        ] [18:44:55]: In OperationStep.activating Step... sfc-chart-id=48e329ab-5458-4bf7-9efb-71d0063dc2e3, sfc-cha
I [c.i.s.s.OperationStep        ] [18:44:55]: In OperationStep.doneActivating()
In step OP1 - chart Use Cases/Use Case 8k Error Handling/PhaseWithoutHandler is starting...
In step T1 - chart Use Cases/Use Case 8k Error Handling/GetInput is starting...
E [c.i.s.r.core                 ] [18:44:55]: Error the key <DOESNOTEXIST> was not found sfc-chart-id=a2ab63d7-d0ec-4f0e-9544-ebdd70c60587,
E [U.U.GetInput                 ] [18:44:55]: Unexpected error in commonInput.py: Key <DOESNOTEXIST> was not found for step Id 4381 sfc-cha
E [U.U.GetInput                 ] [18:44:55]: Traceback (most recent call last):
  File "user-lib\pylib\ils\sfc\gateway\steps\getInput.py", line 60, in activate
    s88Set(chartScope, stepScope, responseKey, "NULL", responseRecipeLocation)
  File "user-lib\pylib\ils\sfc\recipeData\api.py", line 310, in s88Set
    setRecipeData(stepId, folder, key, attribute, value, db)
  File "user-lib\pylib\ils\sfc\recipeData\core.py", line 635, in setRecipeData
    recipeDataId, recipeDataType, targetUnits = getRecipeDataId(stepId, folder + "." + key, db)
  File "user-lib\pylib\ils\sfc\recipeData\core.py", line 205, in getRecipeDataId
    raise ValueError, "Key <%s> was not found for step Id %d" % (key, stepId)
ValueError: Key <DOESNOTEXIST> was not found for step Id 4381
 sfc-chart-id=a2ab63d7-d0ec-4f0e-9544-ebdd70c60587, sfc-chart-name=Use Cases/Use Case 8k Error Handling/GetInput
In ils.sfc.common.notify.sfcNotify(), Sending <sfcMessage> message to project: <XOM>, post: <XO1TEST>, payload: <{'handler': 'sfcUnexpectedE
E [U.U.GetInput                 ] [18:44:55]: Canceling the chart due to an error. sfc-chart-id=a2ab63d7-d0ec-4f0e-9544-ebdd70c60587, sfc-c
I [c.i.s.g.api                  ] [18:44:55]: Cancelling chart with id: dbe7b464-f03e-44fb-a382-c34d7aebbba0 sfc-chart-id=a2ab63d7-d0ec-4f0
E [c.i.s.r.core                 ] [18:44:55]: Error the key <DOESNOTEXIST> was not found
E [U.U.GetInput                 ] [18:44:55]: Unexpected error in commonInput.py: Key <DOESNOTEXIST> was not found for step Id 4381

In ils.sfc.common.notify.sfcNotify(), Sending <sfcMessage> message to project: <XOM>, post: <XO1TEST>, payload: <{'handler': 'sfcUnexpectedE
E [U.U.GetInput                 ] [18:44:55]: Canceling the chart due to an error.
I [c.i.s.g.api                  ] [18:44:55]: Cancelling chart with id: dbe7b464-f03e-44fb-a382-c34d7aebbba0
I [U.U.GetInput                 ] [18:44:55]: Step GetInput in Use Cases/Use Case 8k Error Handling/GetInput deactivated before completing
-------------------------------
In the Get Input CANCEL handler
-------------------------------
-----------------------------------------
In the PhaseWithoutHandler CANCEL handler
-----------------------------------------
-----------------------------------------
In the Operations CANCEL handler
-----------------------------------------
-----------------------------------------
In the UseCase8k CANCEL handler
-----------------------------------------
```

5.2.2 Scripting Interfaces

Now that we have an understanding of cancel and abort handlers and how errors propagate in the Ignition SFC framework it is appropriate to define best practices that should be used to guarantee a robust SFC recipe.

5.3 Best Practices

Now that we have an understanding of cancel and abort handlers and how errors propagate in the Ignition SFC framework it is appropriate to define best practices that should be used to guarantee a robust SFC recipe.

The best practices outlined here will ensure:

- Operators are notified of all unexpected errors
- Whenever an unexpected error occurs, the abort handler will be called on the chart and then the unit procedure will be cancelled from the bottom up running cancel handlers at each level.

5.3.1 Abort Handler

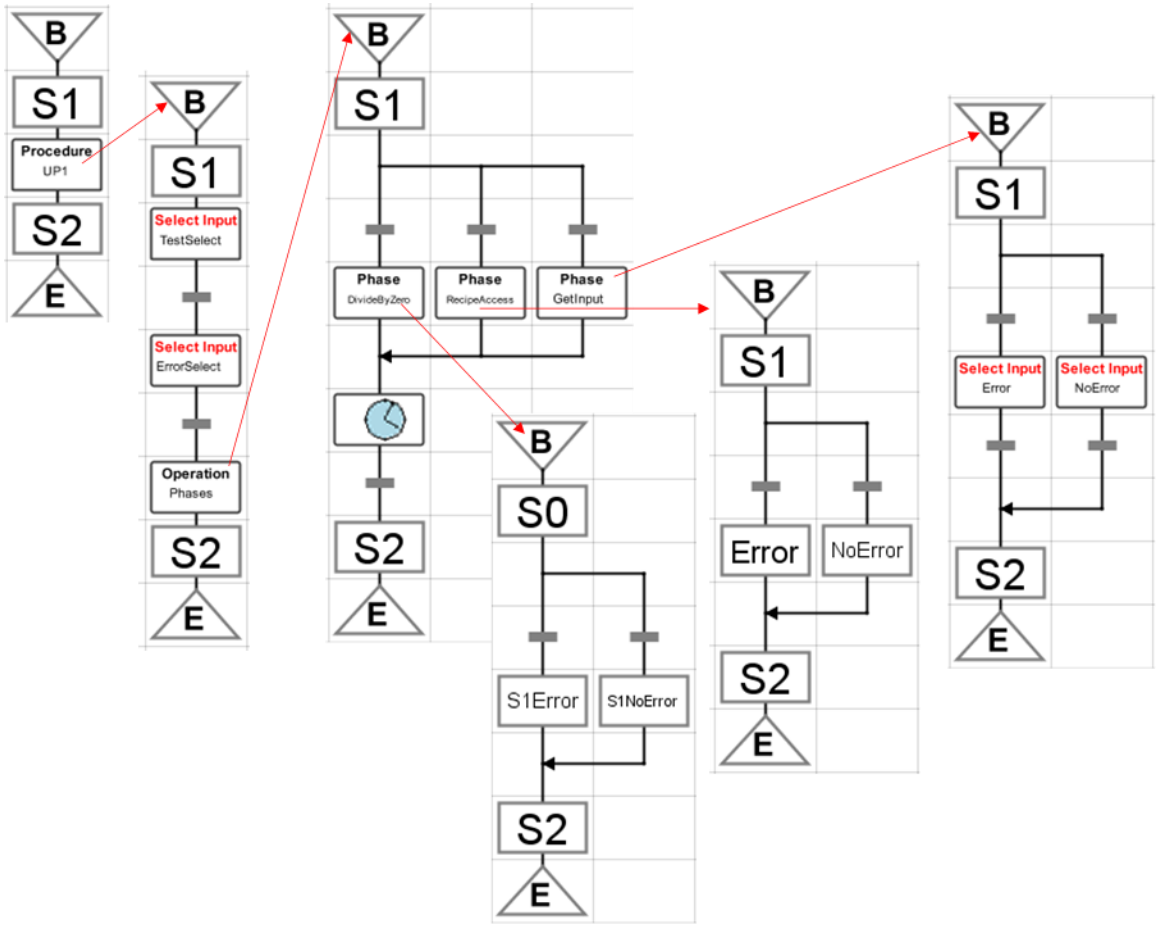
Every chart should provide an abort handler. When designing a chart it is easy to think that this chart is really simple, no way can this ever have an error. This is a dangerous mindset! Over time the chart may be edited and more complex logic added, a tag or recipe data that is needed may

exist today but may be accidentally deleted sometime later. Therefore it is best to design defensively and add an abort handler which calls the ILS *abortHandler* API.

```
def onAbort(chart):
    """
    This will run once if the chart is aborted.
    The exception that caused the abort is
    available via chart.abortCause

    Arguments:
        chart: A reference to the chart's scope.
    """

    print "****************************************"
    print "In the %s ABORT handler - cancelling superior charts" % (chart.chartPath)
    print "****************************************"
    from ils.sfc.gateway.api import abortHandler
    abortHandler(chart, "Caught an error")
```

In conjunction with this, appropriate onCancel handlers should be placed on every chart to anticipate actions to leave a chart in a safe state should a sub chart abort. An onCancel handler is not needed on every chart, only charts where specific actions can be identified.

The following chart (use case 8L) demonstrates the technique.

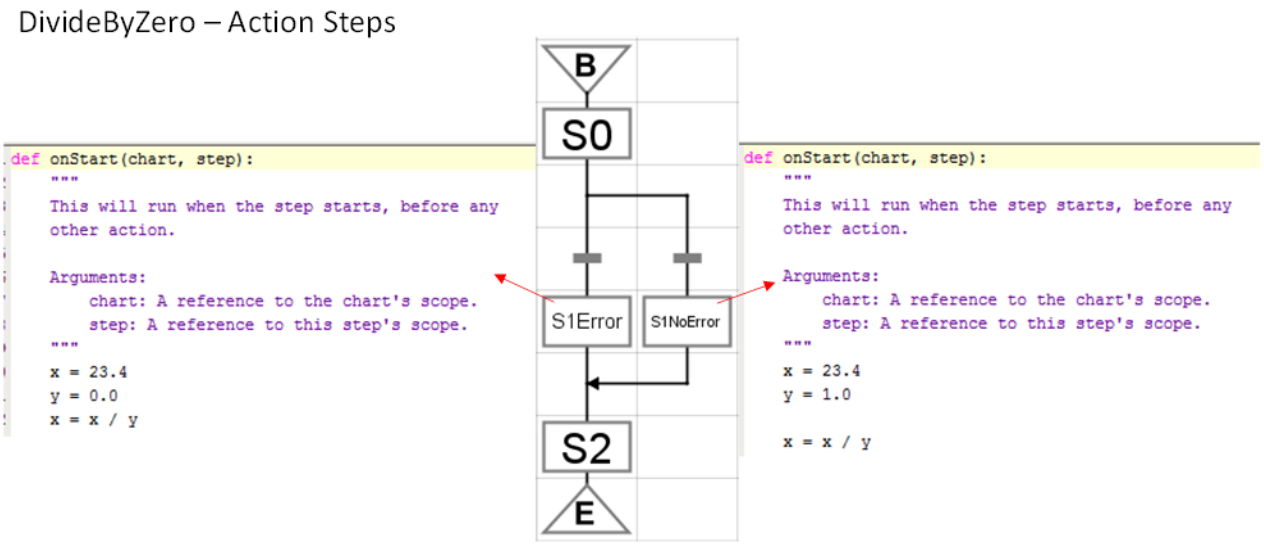


This use case tests three types of errors:

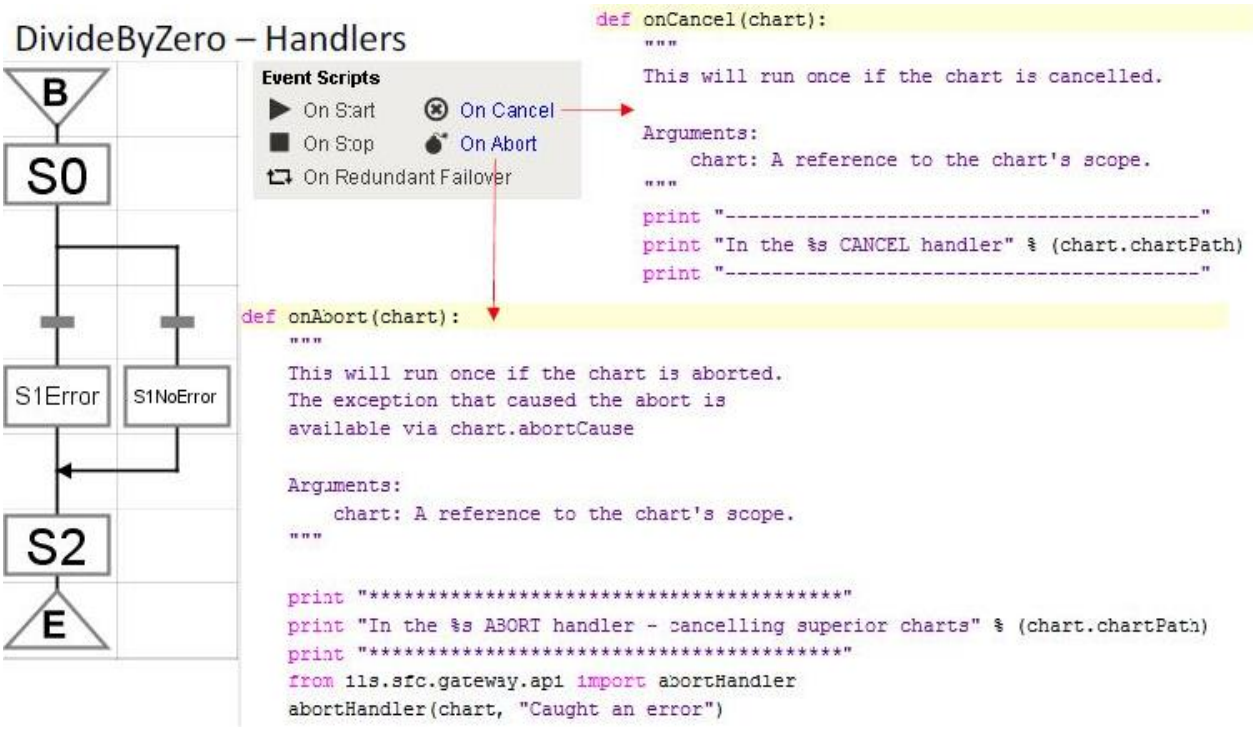
1. A Python script in an action step with a divide by zero error
2. A Python script in an action step with an invalid key used to access recipe data using the recipe data access API
3. A custom step, the Select Input step, which is misconfigured.

The behavior of both action steps is identical and the custom ILS steps already exhibit the desired performance, so the rest of the discussion will use the divide by zero example.

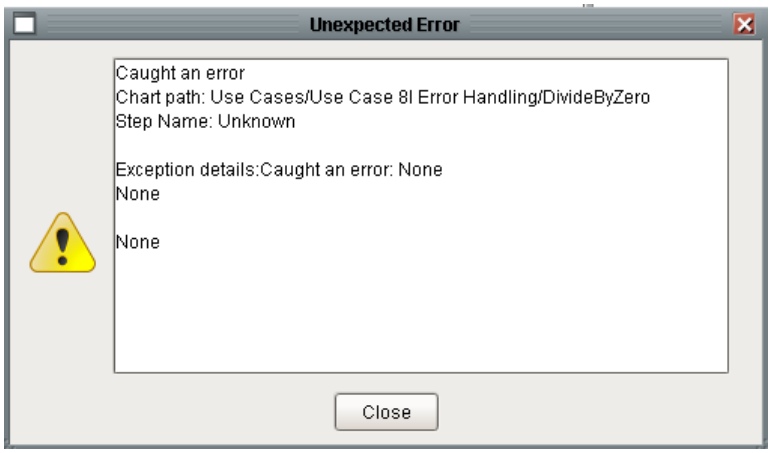
The chart is:



The chart has the following handlers:



When the chart runs, the operator is notified of the error. Unfortunately, the error call stack is not available, nor is the step name. Additional context sensitive error information should be coded in the abort handler to provide a meaningful error message.



The SFC viewer widget shows that chart that had the offending step is aborted and all other charts are cancelled

The wrapper log below shows that the abort handler on the offending chart ran which then called the *abortHandler()* api which cancels the unit procedure and then Ignition cancels charts from the bottom up.

```
12:22:42 | Starting the Unit procedure...
12:22:42 | I [U.U.UseCase81                ] [17:22:42]: The chart run id is: bab0421b-1d75-4d3e-a12d-fc2f09b45264 sfc-chart-id=b
12:22:42 | Starting the operation...
12:22:42 | In ils.sfc.common.notify.sfcNotify(), Sending <sfcMessage> message to project: <XOM>, post: <XO1TEST>, payload: <{'handl
12:22:46 | In ils.sfc.common.notify.sfcNotify(), Sending <sfcMessage> message to project: <XOM>, post: <XO1TEST>, payload: <{'handl
12:22:53 | Starting phases..
12:22:54 | Starting the chart that has an error...
12:22:54 | W [c.i.s.ChartInstance          ] [17:22:53]: Chart 'Use Cases/Use Case 81 Error Handling/DivideByZero' aborted in 'Sc
12:22:54 | com.inductiveautomation.ignition.common.script.JythonExecException: Traceback (most recent call last):
12:22:54 |   File "<Script name="">", line 12, in onStart
12:22:54 | ZeroDivisionError: float division
12:22:54 |
12:22:54 |       at org.python.core.Py.ZeroDivisionError(Py.java:255)
12:22:54 |       at org.python.core.PyFloat.float___div__(PyFloat.java:415)
12:22:54 |       at org.python.core.PyFloat.__div__(PyFloat.java:401)
12:22:54 |       at org.python.core.PyObject._basic_div(PyObject.java:2341)
12:22:54 |       at org.python.core.PyObject._div(PyObject.java:2327)
```

```
12:22:54 |       at java.util.concurrent.ThreadPoolExecutor.runWorker(Unknown Source)
12:22:54 |       at java.util.concurrent.ThreadPoolExecutor$Worker.run(Unknown Source)
12:22:54 |       at java.lang.Thread.run(Unknown Source)
12:22:54 | Caused by: org.python.core.PyException: Traceback (most recent call last):
12:22:54 |   File "<Script name="">", line 12, in onStart
12:22:54 | ZeroDivisionError: float division
12:22:54 |
12:22:54 |       ... 28 common frames omitted
12:22:54 | ****************************************
12:22:54 | In the Use Cases/Use Case 81 Error Handling/DivideByZero ABORT handler - cancelling supe
12:22:54 | ****************************************
12:22:54 | E [c.i.s.g.api                  ] [17:22:54]: Caught an error: None sfc-chart-id=811099
12:22:54 | E [c.i.s.g.api                  ] [17:22:54]: None
12:22:54 |  sfc-chart-id=811099c5-bf34-499d-87f1-0c0efc26419a, sfc-chart-name=Use Cases/Use Case 81
12:22:54 | In ils.sfc.common.notify.sfcNotify(), Sending <sfcMessage> message to project: <XOM>, pa
12:22:54 | E [c.i.s.g.api                  ] [17:22:54]: Canceling the chart due to an error. sfc-
12:22:54 | I [c.i.s.g.api                  ] [17:22:54]: Cancelling chart with id: bab0421b-1d75-4
12:22:54 | In cancelWork(), an asynchronous thread, sleeping...
12:22:55 | ...cancelling...
12:22:55 | ...the asynchronous thread is complete!
12:22:55 | ----------------------------------------
12:22:55 | In the Use Cases/Use Case 81 Error Handling/Phases CANCEL handler
12:22:55 | ----------------------------------------
12:22:55 | ----------------------------------------
12:22:55 | In the Use Cases/Use Case 81 Error Handling/Operations CANCEL handler
12:22:55 | ----------------------------------------
12:22:55 | ----------------------------------------
12:22:55 | In the Use Cases/Use Case 81 Error Handling/UseCase81 CANCEL handler
12:22:55 | ----------------------------------------
```

5.3.2 Implementing an Error Handling Chart

Sequential Function Charts is a programming language. It is sometimes desirable to write an error handler as an SFC. This section describes a technique for doing just that while maintaining all of the S88 facilities such as recipe data and the scope locator context.

The following example is taken from use case 8f.

Even though the event handlers are specified in Python, the Python can call a SFC. This use case demonstrates how to call a SFC from an "On Cancel" event. The same technique can be used for "On Stop" and "On Abort" handlers. An "On Cancel" event handler is shown below:

```
def onCancel(chart):
    """
    This will run once if the chart is cancelled.

    Arguments:
        chart: A reference to the chart's scope.
    """
    print "In the operation's cancel handler, starting a chart..."
    chartPath = "Use Cases/Use Case 8f Enhanced Abort Handling/Safe State"

    from ils.sfc.gateway.api import endHandlerRunner
    endHandlerRunner(chartPath, chart)

    print "...back in the cancel handler, the chart is done!"
```

onCancel Handler

The chart that is called by the abort handler is started by calling the API named *endHandlerRunner()* in *ils.sfc.gateway.api*. The API takes two arguments, the full path of the chart to be called and the chart variable. The API starts the chart using the Ignition system function *system.sfc.startChart()* API. The *endHandlerRunner()* will monitor the chart status and wait until it completes. Control will then return to Ignition which walks up the chart call hierarchy, calling cancel handlers at each level.

The chart that implements the cancel logic must define a chart variable named *callerChartScope* as shown below:

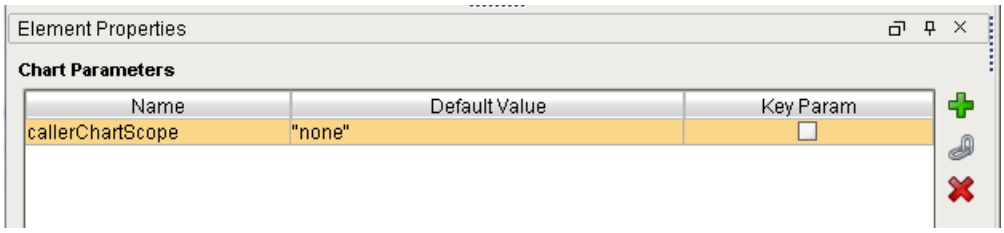
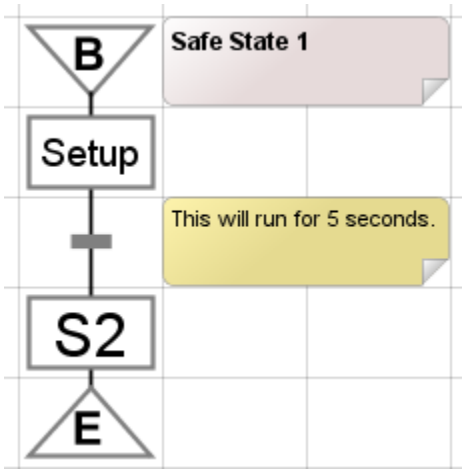


Chart Scope Variable Requireed for Cancel Handler

Because the chart is called by *system.sfc.startChart()* it does not automatically inherit the chart scope of the calling chart. The calling scope is restored from the chart scope variable, *callerChartScope*, which is passed by *endHandlerRunner()*, by calling the API named *endHandlerSetup()*, also in *ils.sfc.gateway.api*. This must be the first thing that is done in the first action step in the chart as shown below. The transition and step S2 are not required, in an actual chart there would likely be some combination of Enter Data, Write Output, PV Monitoring, etc. The execution control steps (Cancel & Pause) should not be used in error handling charts.



Template for a Chart-based Abort Handler

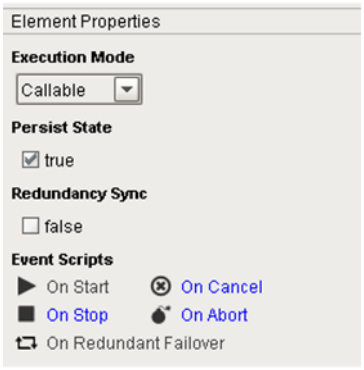
The Python for the "Setup" action step is shown below. Lines 11 and 12 are required, the rest of the lines demonstrate that they worked.

```python
def onStart(chart, step):
    """
    This will run when the step starts, before any
    other action.

    Arguments:
        chart: A reference to the chart's scope.
        step: A reference to this step's scope.
    """

    from ils.sfc.gateway.api import endHandlerSetup
    endHandlerSetup(chart)

    from ils.sfc.recipeData.api import s88Set, s88Get
    from ils.sfc.common.constants import GLOBAL_SCOPE, OPERATION_SCOPE, PHASE_S

    key="foo.value"
    val=s88Get(chart, step, key, GLOBAL_SCOPE)
    print key, " => ", val
```

Python for the Setup Step

5.3.3 Multi-Mode End Handler Charts

The previous example showed how to use a SFC to implement an abort handler. This section describes a more general design to allow a chart to function as a generic end handler that would accommodate End, Cancel, and Abort commands. This is achieved by passing the handler name to the *endHandlerRunner()* API as shown below.

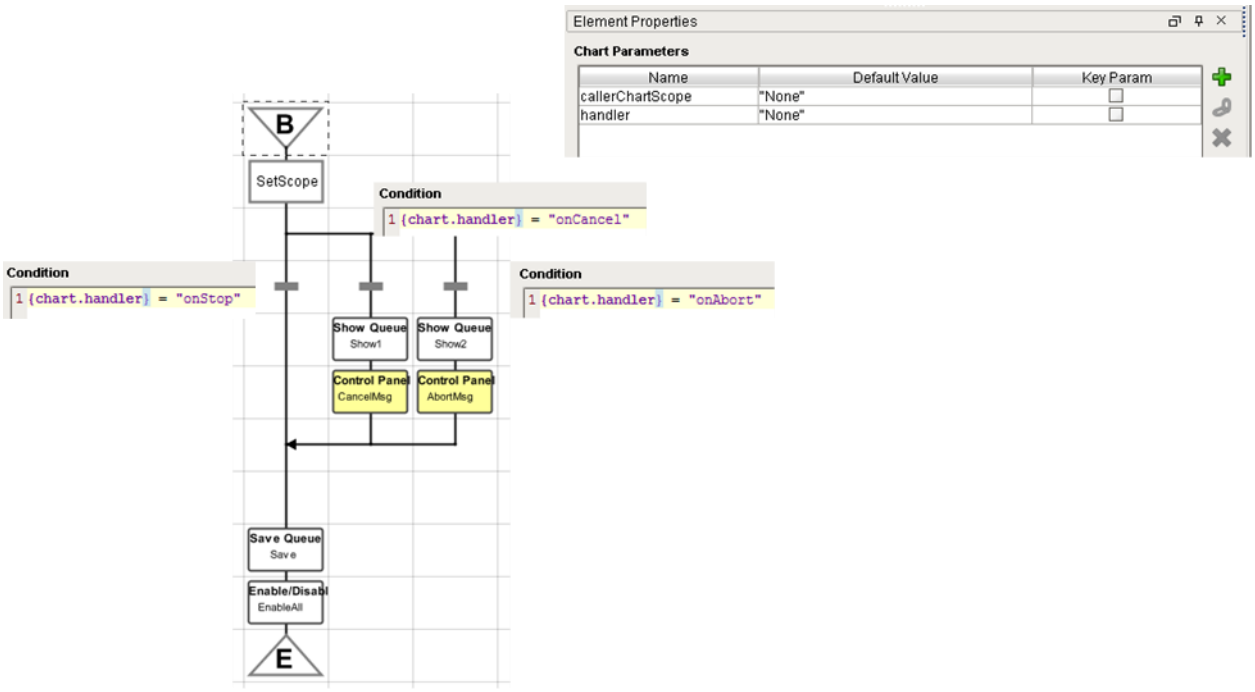



```
def onStop(chart):
    """
    This will run once when the chart is stopped
    normally.

    Arguments:
        chart: A reference to the chart's scope.
    """
    chartPath = "TfaRundown/End"

    from ils.sfc.gateway.api import endHandlerRunner
    endHandlerRunner(chartPath, chart, "onStop")
```

```
def onCancel(chart):
    """
    This will run once if the chart is cancelled.

    Arguments:
        chart: A reference to the chart's scope.
    """
    chartPath = "TfaRundown/End"

    from ils.sfc.gateway.api import endHandlerRunner
    endHandlerRunner(chartPath, chart, "onCancel")
```

```
def onAbort(chart):
    """
    This will run once if the chart is aborted.
    The exception that caused the abort is
    available via chart.abortCause

    Arguments:
        chart: A reference to the chart's scope.
    """
    chartPath = "TfaRundown/End"

    from ils.sfc.gateway.api import endHandlerRunner
    endHandlerRunner(chartPath, chart, "onAbort")
```

The end handler chart then needs to define a new chart parameter: *handler*. The diagram then specifies transitions that check which path to take.


Element Properties
Chart Parameters
Name
Default Value
Key Param
callerChartScope
"None"
handler
"None"
B
SetScope
Condition
1 {chart.handler} = "onCancel"
Condition
1 {chart.handler} = "onStop"
Condition
1 {chart.handler} = "onAbort"
Show Queue
Show1
Show Queue
Show2
Control Panel
CancelMsg
Control Panel
AbortMsg
Save Queue
Save
Enable/Disabl
EnableAll
E

6. SCRIPTING / ACTION STEPS

Action steps are a standard feature of IA's SFC module. Action steps perform actions specified by a Python script.

6.1 Types of Actions

Action steps allow Python to run in four different ways as described in this excerpt from the IA user manual:

> ### On Start
>
> This action runs when the step is started. These actions always run to completion before flow can move beyond the step, and before any other scripts on the action step will run.
>
> ### On Stop
>
> When flow is ready to move beyond the step, it is told to stop. The step will then wait for any currently executing action, for example, **On Start** or **Timer** actions, to finish. Then it will execute its **On Stop** action, if configured. Only after these actions complete will the chart be allowed to move on.
>
> ### Timer
>
> Timer actions run every so often while the action step is running. The timer actions only start running after the **On Start** action finishes (if there is one). In addition to the timer action's script, it must have a rate, specified in milliseconds. This is the amount of time to wait between running the timer action's script. The clock starts after the **On Start** action finishes.
>
> It is important to realize that, unlike **On Start** and **On Stop** scripts, a timer action can not run at all for an action step. If the step is ready to stop before the timer is ready, it will never run.
>
> ### Error Handler
>
> This action will run only if any of the other actions throw an unexpected error. This provides a chance for chart designers to do their own error handling, and gracefully recover from an error. If this script throws an error, the chart will abort, see Chart Concepts for more information.

6.2 Python Location

There are three locations where the Python associated with an Action Step can be placed. The first, and easiest, is to embed the Python directly in the step. The second, is to use Global scope. The third, and the method preferred by ILS, is to use external Python in the user-lib/pylib file system. When using global or external Python there still needs to be a line or two embedded in the step to call the global or external Python. Because SFCs are global, an action step cannot reference project scope Python.

Location	Pros	Cons
Embedded	Easy to find the code	Ignition Editor, no source code control, difficult to reuse.
Global	Promotes reuse.	Ignition editor, no Source code control
External	Promotes reuse, Eclipse editor, Source Code Control, easy to move files around.	Not integrated with Find

6.3 Logging Support

It is important to consider logging, discussed further in section 8.2, when implementing an action step. By placing log messages in your code at the appropriate level will allow the system to be debugged, by putting the logger into trace mode, and to run quietly when in info mode. It is a best practice to always use log messages rather than print statements.

The logger level is set in the gateway at run time from the gateway web page. An example of getting and writing to the logger is shown below.

```
5 import system
6 from ils.sfc.common.constants import GLOBAL_SCOPE, SUPERIOR_SCOPE, MSG_STATUS_INFO
7 from ils.sfc.gateway.api import getChartLogger, s88Get, s88Set, postToQueue, getProviderName, getDatabaseName
8
9 def onStart(chart,step):
10     provider = getProviderName(chart)
11     database = getDatabaseName(chart)
12     log = getChartLogger(chart)
13     log.tracef("In %s...", step.get("name",""))
```

Logger Snippet Example

6.4 Accessing Tags and Tag History in Action Steps

The first thing to remember when accessing tags and tag history in actions steps is that SFCs run in the gateway and are not associated with a project. Therefore, they do not have a default tag provider which means that the tag provider must always be specified.

6.4.1 Reading a Tag's Current Value

The following example demonstrates how to read the current value of a tag. It is a best practice to always check the quality of the tag. It is also a best practice to never hard-code the tag provider,

rather you should always get the provider from the chart scope dictionary in order to properly support isolation mode.

```
qv = system.tag.read("[%s]LabData/RLA3/C2-LAB-DATA-SQC/target" % (tagProvider))
if qv.quality.isGood():
    postToQueue(chart, MSG_STATUS_ERROR, "copyCRxData() - unable to acquire the ethylene SQC
    target because the data quality for <LabData/RLA3/C2-LAB-DATA-SQC/target> is not good")
    return
target = qv.value
```

Reading a Tag Value Example

6.4.2 Reading a Tag's Average Value over a Period of Time

The following example demonstrates how to read the average value of a one or more tags for any given time period. The *system.tag.queryTagHistory()* API supports a number of aggregation modes and also allows specific start and end times. Unlike *system.tag.read()* which returns a qualified value, this API returns a dataset of qualified values. If one of the aggregation functions is used then the dataset will have a single row where the first column is a timestamp and each tag in the tagPaths list will be in a subsequent column. There are two special values that can be returned, NaN and None. In order to be robust, the quality of each returned value should be checked. The API will not throw an error.

The following example shows a typical usage of *queryTagHistory()*. The example will query the average value of a list of tags over the past *n* minutes. It demonstrates using *getQualityAt()* to verify that the quality of the returned value is good. When a list of tags is provided, it is generally useful to abort whatever calculation is being performed if the quality of any tag in the return dataset is bad. This routine will return a badValue = True if any value is bad. This convenient wrapper function is provided in *ils.common.util*.

```
def readAverageValues(tagPaths, tagProvider, timeIntervalMinutes, log):
    fullTagPaths = []
    for tagPath in tagPaths:
        fullTagPaths.append("[%s]%s" % (tagProvider, tagPath))

    ds = system.tag.queryTagHistory(
        paths=fullTagPaths,
        endDate=system.date.now(),
        rangeMinutes=-1*timeIntervalMinutes,
        aggregationMode="Average",
        returnSize=1,
        ignoreBadQuality=True
        )

    badValue = False
    for i in range(0,len(tagPaths)):
        isGood = ds.getQualityAt(0, i + 1).isGood()
        print "Average value for %s isGood %s" % (tagPaths[i], str(isGood))
        if not(isGood):
            badValue = True
            log.warnf("Unable to collect average value for %s", tagPaths[i])

    return badValue, ds
```

ILS Utility to read Tag Averages

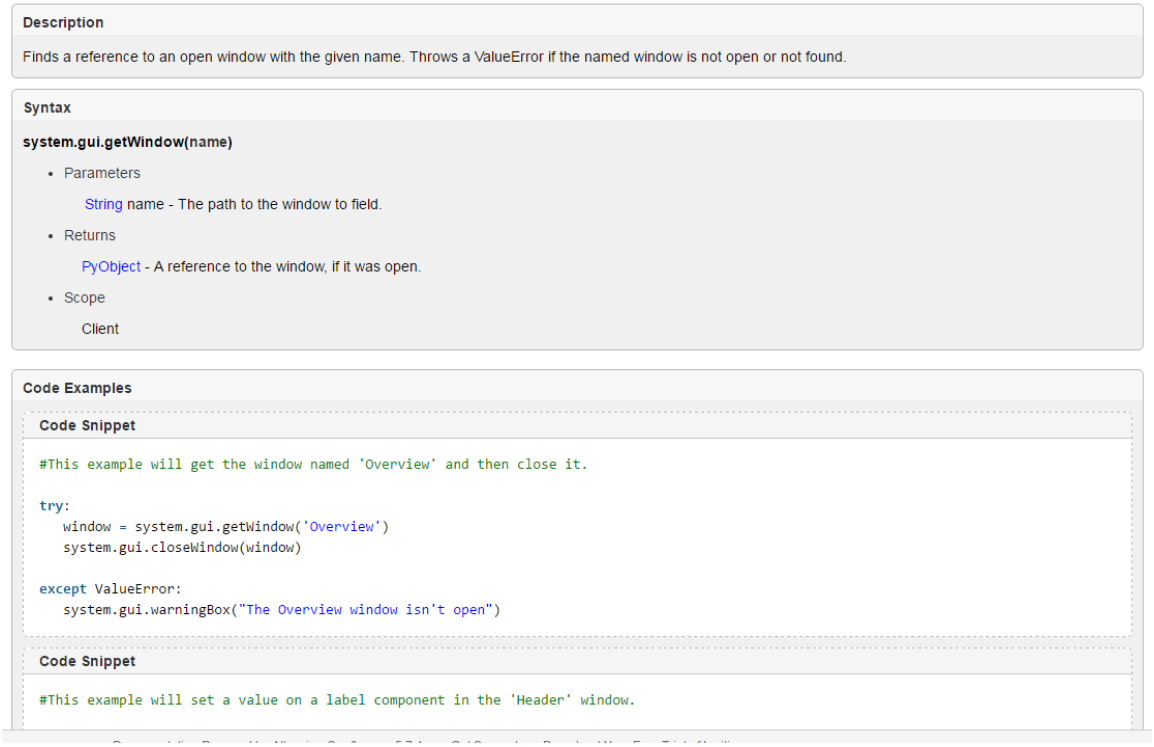
Note: The tag provider that is used is the real-time tag provider, not the history tag provider.

6.4.3 Reading a Tag's Value at a Specific Moment in Time

Similar to above except that the startDate, endDate, or intervalMinutes should be specified as need to determine the moment in time of interest.

6.5 Exception Handling in Action Blocks

It is important to realize that if an error occurs in the Python of an Action Block, the default Ignition action is to abort the chart **AT THAT LEVEL ONLY** (i.e. an enclosing chart will continue to run)! The top-down cancellation and abort behavior in the ILS steps depends on calling particular scripts. This means that if the desired top-down behavior is to be maintained in Action blocks, a call to the proper error handling method must be called explicitly by either a) placing a try/except block around all the Python in the onStart and/or onStop methods or b) placing the call to the error handler in the Error Handler method of the Action step.



Description

Finds a reference to an open window with the given name. Throws a ValueError if the named window is not open or not found.

Syntax

system.gui.getWindow(name)

- Parameters

  String name - The path to the window to field.

- Returns

  PyObject - A reference to the window, if it was open.

- Scope

  Client

Code Examples

Code Snippet

```
#This example will get the window named 'Overview' and then close it.

try:
   window = system.gui.getWindow('Overview')
   system.gui.closeWindow(window)

except ValueError:
   system.gui.warningBox("The Overview window isn't open")
```

Code Snippet

```
#This example will set a value on a label component in the 'Header' window.
```

6.6 Scripting API

The external Python module *ils.sfc.gateway.api* contains methods intended for use from ActionSteps. The Pydoc documentation for these is shown below. In most cases, the chart scope that is passed to the onStart method (e.g.) is also passed in to the api method. The naming is a bit inconsistent; chartScope and chartProperties are synonymous.

It is a recommended practice to use definitions from `ils.sfc.common.constants` wherever string constants are required (e.g. recipe data scope). Unless otherwise noted, all of the interfaces described below are provided in ils.sfc.api.

6.6.1 Argument Dictionary

This section defines that arguments that are used.

Argument	Description
chartProperties	
Message	A string that will be posted to a queue, the control panel, etc
ackRequired	A Boolean, True if the message requires acknowledgement from the operator
Scope	Used for recipe data step location. Acceptable scopes are: LOCAL_SCOPE, PRIOR_SCOPE, SUPERIOR_SCOPE, PHASE_SCOPE, OPERATION_SCOPE, GLOBAL_SCOPE which are defined in ils.sfc.common.constants.py

6.6.2 addControlPanelMessage

Display a message on the control panel.

```
addControlPanelMessage(chartProperties, message, ackRequired)
```

6.6.3 cancelChart

Cancel the entire chart hierarchy.

```
cancelChart(chartProperties)
```

6.6.4 convertUnits

This should be simplified, no need for chartProperties if we have the value and the to and from units.

Convert a value from one unit to another.

```
convertUnits(chartProperties, value, fromUnitName, toUnitName)
```

6.6.5 getChartLogger

Get the logger associated with this chart.

getChartLogger(chartScope)

6.6.6 getChartPath

Get the full path to the current running chart.

getChartPath(chartScope)

6.6.7 getControlPanelId

Get the id of the control panel for the currently running chart.

getControlPanelId(chartScope)

6.6.8 getControlPanelName

Get the name of the control panel for the currently running chart.

getControlPanelName(chartScope)

6.6.9 getCurrentMessageQueue

Get the currently used message queue.

getCurrentMessageQueue(chartProperties)

6.6.10 getDatabaseName

Get the name of the database this chart is using, taking isolation mode into account.

getDatabaseName(chartProperties)

6.6.11 getIsolationMode

Returns true if the chart is running in isolation mode.

getIsolationMode(chartProperties)

6.6.12 getOriginator

Get the username that started this SFC.

getOriginator(chartProperties)

6.6.13 getPostForControlPanelName

Get the post associated with the named control panel.

```
getPostForControlPanelName(chartProperties)
```

6.6.14 getProject

Get the project associated with the client side of this SFC (not the global project!

```
getProject(chartProperties)
```

6.6.15 getProvider

Like getProviderName(), but puts brackets around the provider name.

```
getProvider(chartProperties)
```

6.6.16 getProviderName

Get the name of the tag provider for this chart, taking isolation mode into account.

```
getProviderName(chartProperties)
```

6.6.17 getTimeFactor

Get the factor by which all times should be multiplied (typically used to speed up tests).

```
getTimeFactor(chartProperties)
```

6.6.18 getTopChartRunId

Get the run id of the top chart.

```
getTopChartRunId(chartProperties)
```

6.6.19 getTopChartStartTime

Get timestamp for chart start.

```
getTopChartStartTime(chartProperties)
```

6.6.20 getTopLevelProperties

Get the properties of the top level chart.

```
getTopLevelProperties(chartProperties)
```

6.6.21 handleUnexpectedGatewayError

Called from an error handler to consistently handle a SFC gateway error including cancelling the chart and notifying clients of the error with a suitable stack trace.

```
handleUnexpectedGatewayError(chartProperties, stepProperties, msg,
   logger)
```

6.6.22 notifyGatewayError

Called from an error handler to consistently handle a SFC gateway error and notify clients of the error with a suitable stack trace.

```
notifyGatewayError(chartProperties, stepProperties, msg, logger)
```

6.6.23 parseValue

parse a value of the given type from a string.

```
parseValue(strValue, tagType)
```

6.6.24 pauseChart

pause the entire chart hierarchy.

```
pauseChart(chartProperties)
```

6.6.25 postToQueue

Post a message to a queue from an SFC. If the queueKey is left blank then the current default queue for the unit procedure is used. Expected status are Info, Warning, or Error, best practice is to use constants QUEUE_ERROR, QUEUE_WARNING, QUEUE_INFO from ils.constants.constants.py

```
postToQueue(chartScope, status, message, queueKey=None)
```

6.6.26 readTag

Read an ordinary tag (ie not recipe data), substituting provider according to isolation mode setting

```
readTag(chartScope, tagPath)
```

6.6.27 resumeChart

resume the entire chart hierarchy

```
resumeChart(chartProperties)
```

6.6.28 s88DataExists

Returns true if the specified recipe data exists

```
from ils.sfc.recipeData.api import s88DataExists
s88DataExists(chartScope, stepScope, keyAndAttribute, scope)
```

6.6.29 s88Get

Get the given recipe data's value

```
from ils.sfc.recipeData.api import s88Get
s88Get(chartScope, stepScope, keyAndAttribute, scope)
```

6.6.30 s88GetFullTagPath

Get the full path to the recipe data tag, taking isolation mode into account

```
from ils.sfc.recipeData.api import s88GetFullTagPath
s88GetFullTagPath(chartScope, stepScope, keyAndAttribute, scope)
```

6.6.31 s88GetType

Get the underlying recipe data type; return one of STRING, INT, FLOAT, or BOOLEAN

```
from ils.sfc.recipeData.api import s88GetType
```

```
s88GetType(chartScope, stepScope, keyAndAttribute, scope)
```

6.6.32 s88GetUnits

Get the units associated with a recipe data value; None if not found.

```
from ils.sfc.recipeData.api import s88GetUnits
s88GetUnits(chartScope, stepScope, keyAndAttribute, scope)
```

6.6.33 s88GetWithUnits

Like s88Get, but adds a conversion to the given units

```
from ils.sfc.recipeData.api import s88GetWithUnits
s88GetWithUnits(chartScope, stepScope, keyAndAttribute, scope,
   returnUnits)
```

6.6.34 s88Set

Set the given recipe data's value

```
from ils.sfc.recipeData.api import s88Set
s88Set(chartScope, stepScope, keyAndAttribute, value, scope)
```

6.6.35 s88SetWithUnits

Like s88Set, but adds a conversion from the given units

```
from ils.sfc.recipeData.api import s88SetWithUnits
s88SetWithUnits(chartScope, stepScope, keyAndAttribute, value, scope,
   units)
```

6.6.36 scaleTimeForIsolationMode

If the supplied unit is a time unit and we are in isolation mode, scale the value appropriately--otherwise, just return the value

```
scaleTimeForIsolationMode(chartProperties, value, unit)
```

6.6.37 sendMessageToClient

Send a message to the client(s) of this chart

```
sendMessageToClient(project, handler, payload, clientSessionId=None)
```

6.6.38 sendOCAlert

Send an OC alert

```
sendOCAlert(chartProperties, stepProperties, post, topMessage,
   bottomMessage, buttonLabel, callback=None,
   callbackPayloadDictionary=None, timeoutEnabled=False,
   timeoutSeconds=0)
```

6.6.39 setCurrentMessageQueue

Set the currently used message queue

```
setCurrentMessageQueue(chartProperties, queue)
```

6.6.40 writeLoggerMessage

Write a message to the system log file from an SFC.

writeLoggerMessage(chartScope, block, unit, message)

6.7 Scripting Examples

Typical examples of setting and getting recipe data are shown below

6.7.1 Recipe Data Access Examples

Typical examples of setting and getting recipe data are shown below

```
def onStart(chart, step):
    """
    This will run when the step starts, before any
    other action.

    Arguments:
        chart: A reference to the chart's scope.
        step: A reference to this step's scope.
    """
#   from system.ils.sfc import s88Get, s88Set
    from ils.sfc.gateway.api import s88Set, s88Get
    from ils.sfc.common.constants import LOCAL_SCOPE, SUPERIOR_SCOPE, PHASE_SCOPE, OPERATION_SCOPE, GLOBAL_SCOPE, PRIOR_SCOPE

    print "Setting #1... "

    s88Set(chart, step, "foo.value", 2, GLOBAL_SCOPE)
    s88Set(chart, step, "foo.value", 25.4, OPERATION_SCOPE)
    s88Set(chart, step, "foo.value", 7.8, PHASE_SCOPE)
    s88Set(chart, step, "foo.value", "Hello World", SUPERIOR_SCOPE)
    s88Set(chart, step, "foo.value", 6.02, LOCAL_SCOPE)
    s88Set(chart, step, "foo.value", 78, PRIOR_SCOPE)
```

```
def onStart(chart, step):
    """
    This will run when the step starts, before any
    other action.

    Arguments:
        chart: A reference to the chart's scope.
        step: A reference to this step's scope.
    """
    from ils.sfc.gateway.api import s88Get, s88Set
    from ils.sfc.common.constants import GLOBAL_SCOPE, OPERATION_SCOPE, PHASE_SCOPE, SUPERIOR_SCOPE, PRIOR_SCOPE

    print "Getting #2... "

    key="foo.value"
    for scope in [GLOBAL_SCOPE, OPERATION_SCOPE, PHASE_SCOPE, SUPERIOR_SCOPE, PRIOR_SCOPE]:
        val=s88Get(chart, step, key, scope)
        print scope, key, " => ", val
```

6.7.2 Tag History Examples

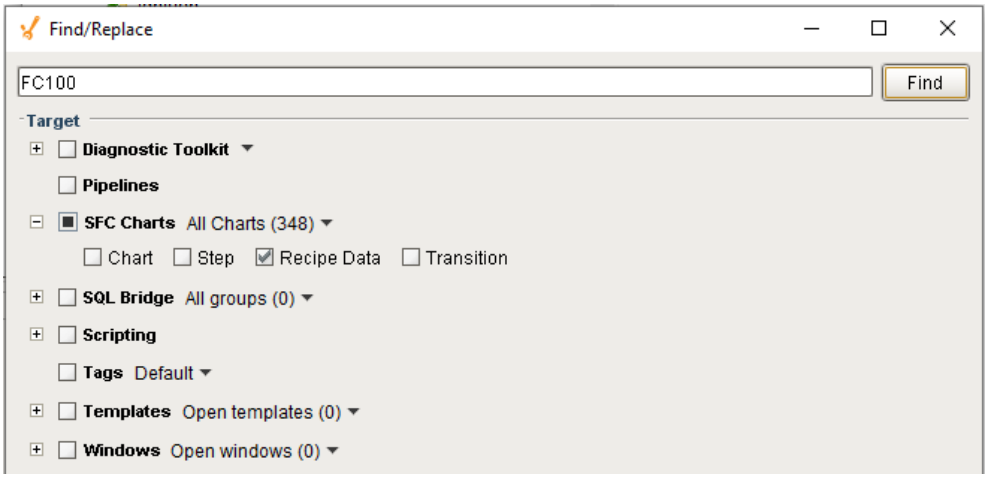
The following example shows how to query tag history for two tags using built-in scripting functions to return the average of each tag over the last 30 minutes. queryTagHistory() returns a dataset where the first column will be the timestamp and each column after that represents a tag.

```
r2C6RateTagPath = "[%s]SFC IO/Fast Scan Data/CS-VRF206/value" % (provider)

# These tags need to have history collected
tagPaths = []
tagPaths.append(mainC6RateTagPath)
tagPaths.append(r2C6RateTagPath)

# Fetch the average value over the last 30 minutes
ds = system.tag.queryTagHistory(paths=tagPaths, aggregationMode='Average', returnSize=1, endDate=system.date.now(), rangeMinutes=-30.0)
if ds.rowCount == 1:
    mainC6Rate = ds.getValueAt(0, 1)
    r2C6Rate = ds.getValueAt(0, 2)
    log.tracef("Querying the average feed rate over the last 30 minutes was successful, mainC6Rate: %s, r2C6Rate: %s",str(mainC6Rate), str
    if mainC6Rate == 0.0 or r2C6Rate == 0.0:
        txt = "Setting the R2 C2 delay for a C-Rx grade to (0.0) because either the main C6 rate (%s) or the r2 C6 rate (%s) is 0." % (str
        log.tracef(txt)
        postToQueue(chart, MSG_STATUS_INFO, txt)
        r2C2Delay = 0.0
    else:
        readOK = True
        log.tracef("The R2 C2 delay will be calculated using the history of the main C6 rate (%s) and the r2 C2 rate (%s)", str(mainC6Rate
else:
    log.warnf("Tag history query of %s did not return any data, reading the tags current values" % (tagPaths))
```

7. ADDITIONAL CAPABILITIES

7.1 Find/Replace

The standard Ignition find/replace capabilities have been extended to include support for Sequential Function Charts.



The SFC search will examine these object types:

- Chart – Chart name, chart scope variables, and chart Python handlers
- Step – Including action step Python, ILS step configuration properties, and step names.
- Recipe Data – Recipe data stored in the database.
- Transitions – Transition Expressions.

The SFC search will search:

- All charts
- Selected Charts – searches charts selected in the project tree
- Open Charts – searches charts that are open in the Designer

The user can control the objects that are searched and the charts that are searched from the Find/Replace window.

There are three elements of this feature: finding, locating, and replacing.

- Finding - the ability to search the specified charts and the specified object types for the desired token.
- Locating – double-clicking on the row in the results grid will open the referenced chart. It would be nice to highlight the referenced step or transition but that is not possible. Note: the SFC context must be already open in the Designer for this to work.
- Replacing – Replacing is not supported for SFCs.

7.2 Sequential Control Integration to Diagnostic Toolkit

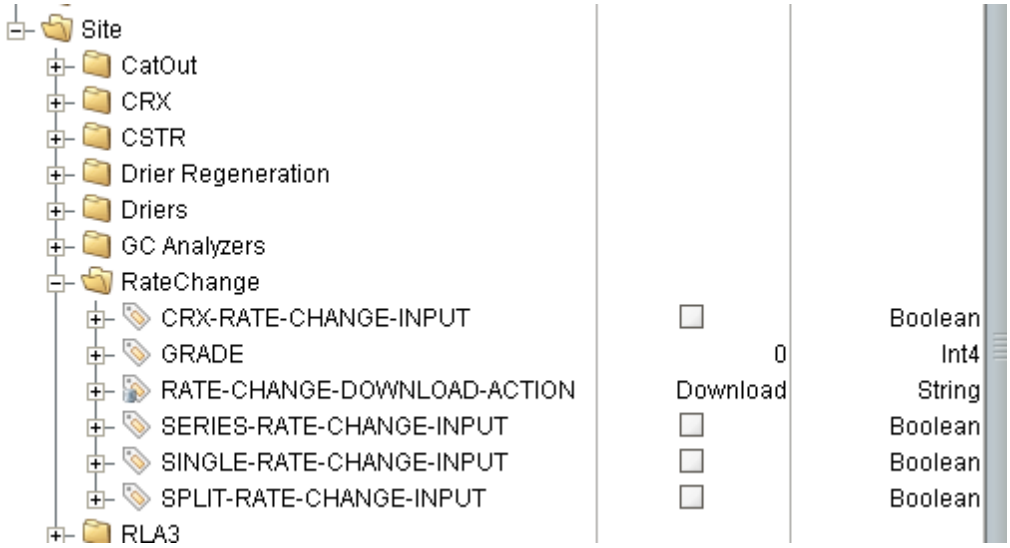
As we have seen throughout this manual, the sequential control toolkit is capable of performing all sorts of automation operations. It is particularly well suited for reactor startup (Coldstick) and reactor shutdown (Catout) operations. There is another category of operations that interface with the diagnostic toolkit. Operations that fall into this category are rate change, conversion rate change and the flying switch. There is a subtle difference between these and the aforementioned operations. These operations are performed on a steady state running unit and move the process to another steady state. In order to do this takes advantage of the recommendation and I/O framework of the diagnostic toolkit along with all of its other features including the setpoint spreadsheet user interface.

The purpose of this section is to describe how to interface the sequential control toolkit with the diagnostic toolkit. The Rate Change application at G-line will be used for demonstration purposes.

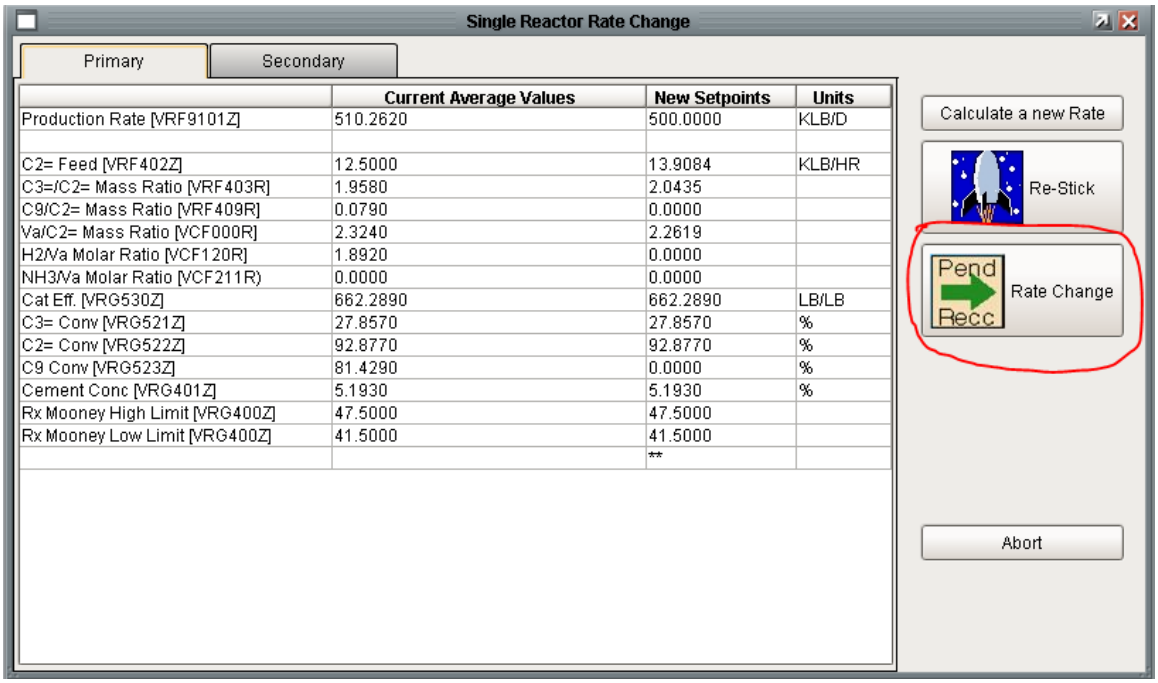
The operator starts the rate change operation from the control panel. The operation utilizes all of the capabilities of the sequential control framework utilizing data entry, review screens, and action steps to calculate the desired setpoints for the new rate.

There are two opportunities where the Rate Change recipe communicates with the diagnostic toolkit.

The first is when the recipe "signals" the diagnostic toolkit that there are rate change setpoints ready to download. It does this by writing a True to the appropriate RATE-CHANGE-INPUT memory tag shown below.



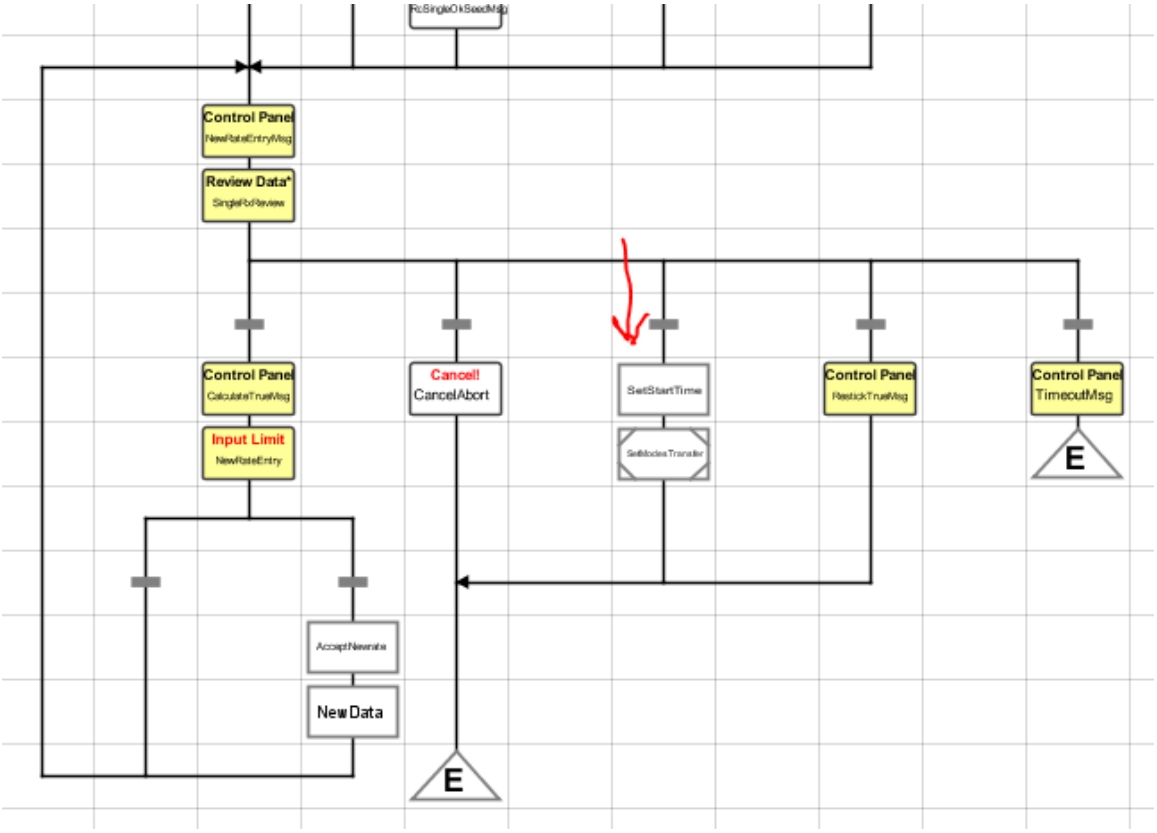
This is implemented in the "transfer" step on the respective "setModesTransfer" chart for each reactor configuration. The user presses the "Rate Change" button:



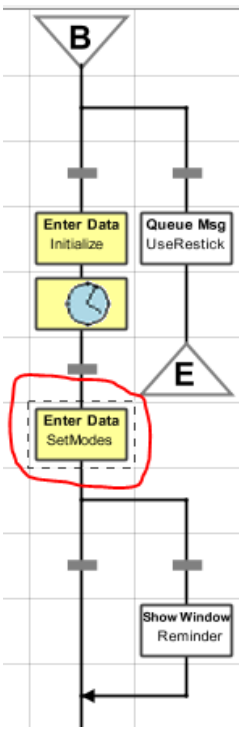
Review Data Window Launching SFCs

It calls *xom.vistalon.sfc.polymerizeEpdm.rateChange.reviewData.transferDataCallback()* which updates DtApplication to set the clientID of the client to be notified with the setpoint spreadsheet and sets the response of the "selected-button" step variable.

The "selected-button" value satisfies the transition shown below.

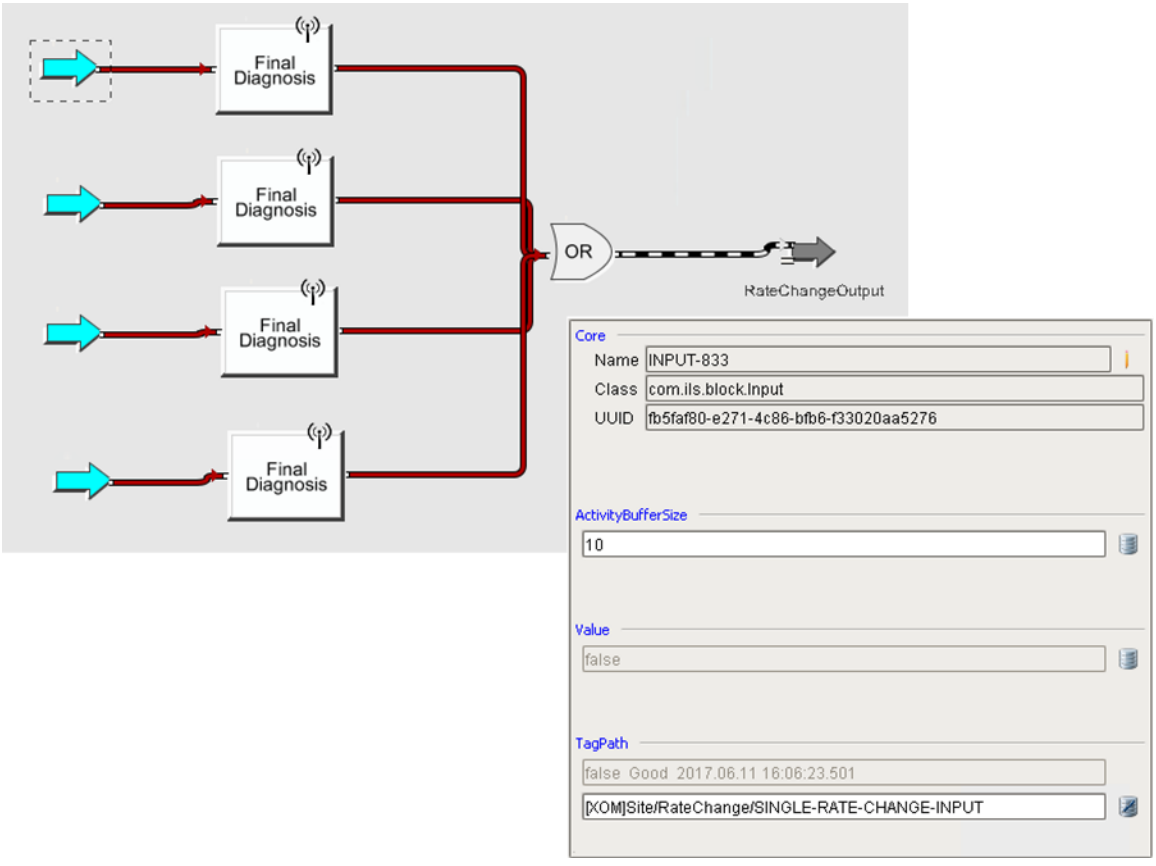


The SetModesTransfer chart, which is reactor configuration specific, signals the diagnostic toolkit by writing to the appropriate RateChange tag in the Enter Data step named "SetModes". In order to guarantee that the diagnostic toolkit sees a transition, the tag is set to False, then waits, and then is set to True. While this is robust, it leads to a processing delay the OPC tag latency time period.

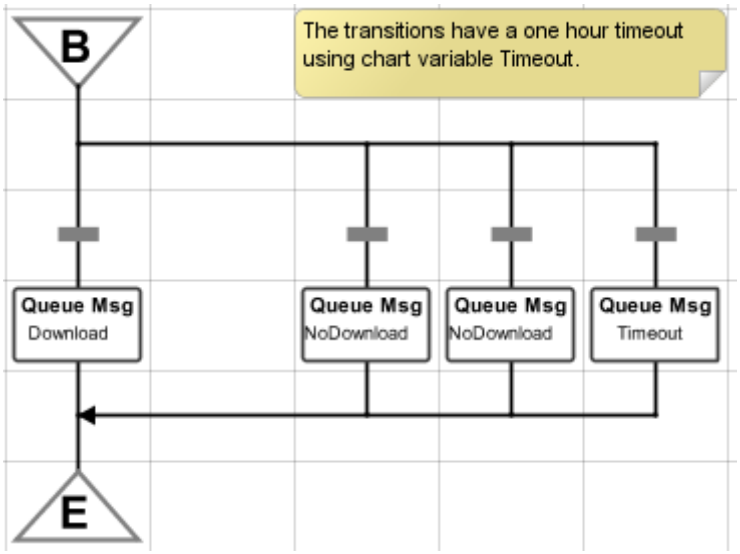


SetModesTransfer Chart

These memory tags are inputs to the Rate Change Diagnostic application shown below.



The second area of communication is when the chart enters a wait state waiting for the operator to press the "Download" or "No Download" buttons. This is implemented in the *CheckForSPWorkspace* chart which is shown below.



The two left transitions check for the button action, the right transition is a timeout.

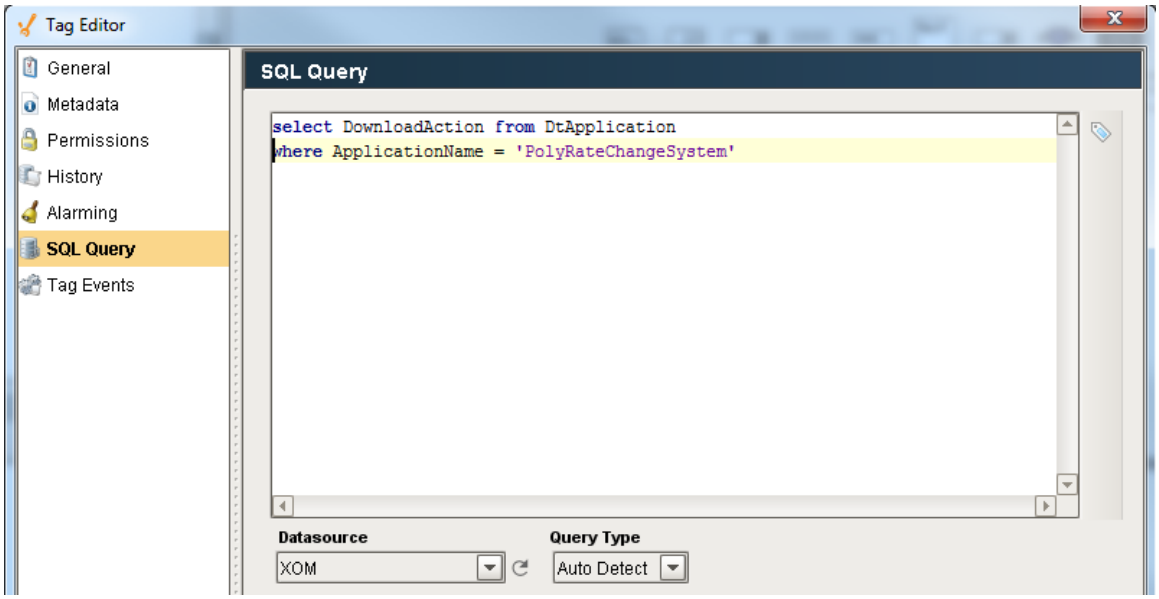
Condition

```
1 upper({[XOM]Site/RateChange/RATE-CHANGE-DOWNLOAD-ACTION}) = "DOWNLOAD"
```

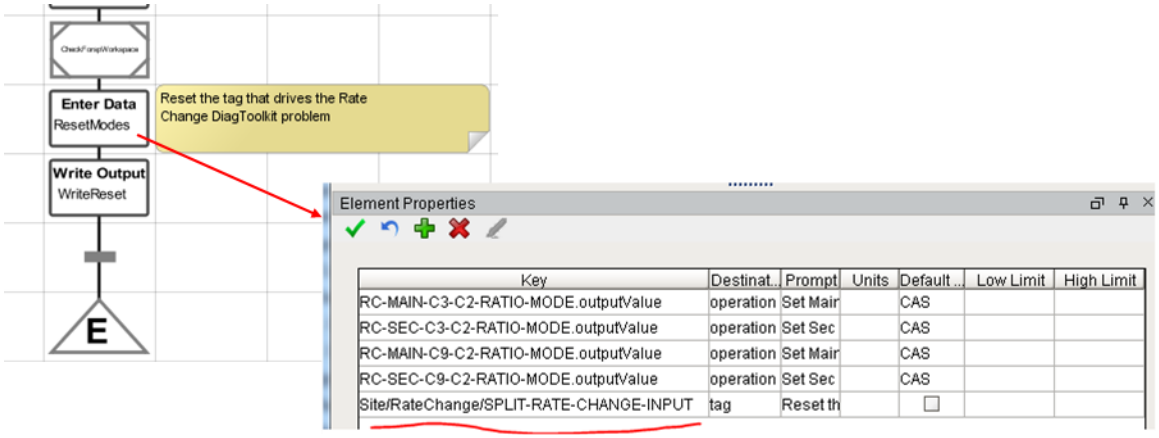
Condition

```
1 upper({[XOM]Site/RateChange/RATE-CHANGE-DOWNLOAD-ACTION}) = "NO DOWNLOAD"
```

The tag referenced in the conditions is a query tag which queries that database for the download action taken on the Rate Change diagnostic application.



The inputs to the diagnostic problems are reset in the final Enter Data step named "ResetModes" on the SetModesTransfer chart. This will cause the Final Diagnosis to become False which should clear the recommendations regardless of whether the operator pressed Download or No Download.



7.2.1 Diagnostic Toolkit Post Processing

After the setpoint spreadsheet has been presented to the operator, and the operator has pressed "Download", all of the diagnostic diagrams for RLA3 need to be reset just as if a grade change has happened. Statistical quality control assumes statistical independence of changes. A rate-change is a move that affects the entire RLA3. Think of rate-change as implementing a change for every possible problem (or family) for RLA3. This is handled by the post processing callback which has been defined on all 4 rate change final diagnosis named *resetRelatedFinalDiagnosis()* in *xom.vistalon.diagToolkit.rateChange.common.py*. This callback executes the "No Download" logic on each of the final diagnosis. All of this runs in the client. Specifically, this calls resetApplication() once for each application defined for RLA3 with a list of the final diagnosis and outputs for the application. This presented one challenge because part of the logic was to call the post processing callback for each final diagnosis. This created an infinite loop because each of the

four rate change final diagnosis called *resetRelatedFinalDiagnosis()*. I got around this by adding an optional argument to call the post processing callback.

7.2.2 Triggering a Final Diagnosis from a Chart

It is common to trigger a final diagnosis from a chart step. Common examples of this are the rate change and flying switch operations. The main reason for triggering a final diagnosis is to take advantage of the framework for writing I/O complete with a common user interface and safety checks on change amounts. Refer to the Diagnostic Toolkit User Manual for an example.

7.3 Library Charts

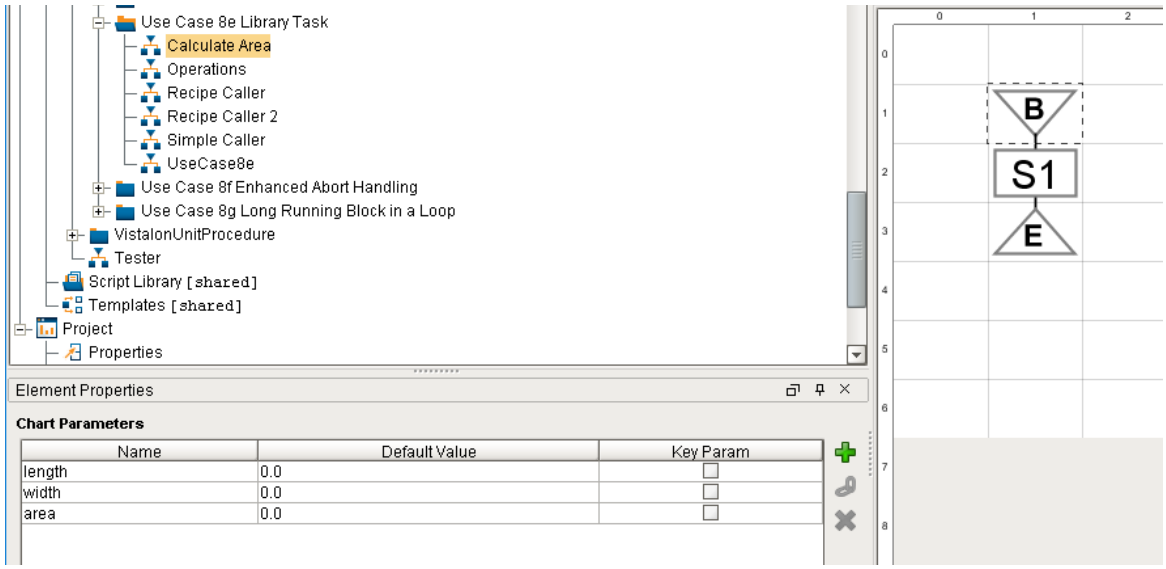
A "Library" chart is the term that we use to refer to a chart that is generic in nature and can be called by numerous steps on numerous charts. The genesis of the term comes from a previous implementation of the SFC toolkit where charts were organized using an encapsulation scheme whereby charts could only be called by their enclosing parent step. It is important to realize that from Inductive Automation's standpoint, every chart can be called from any other charts and the encapsulation constrints of previous systems no longer exist. Therefore the designation of "Library" is purely semantic.

Moving forward, a better use of the term is to refer to a chart that has been designed to be reusable. In order for a chart to be reusable, it must be designed to be reusable. This section describes several ways to design a chart to be reusable.

7.3.1 Library Chart using Pass-by-Value

The first "Library" chart to consider is one that strictly uses standard Ignition steps. This is a textbook example of creating a reusable chart. The key to a reusable chart, just like a reusable function in Python, is to pass parameters to the chart. This demonstrates library arguments using pass by value.

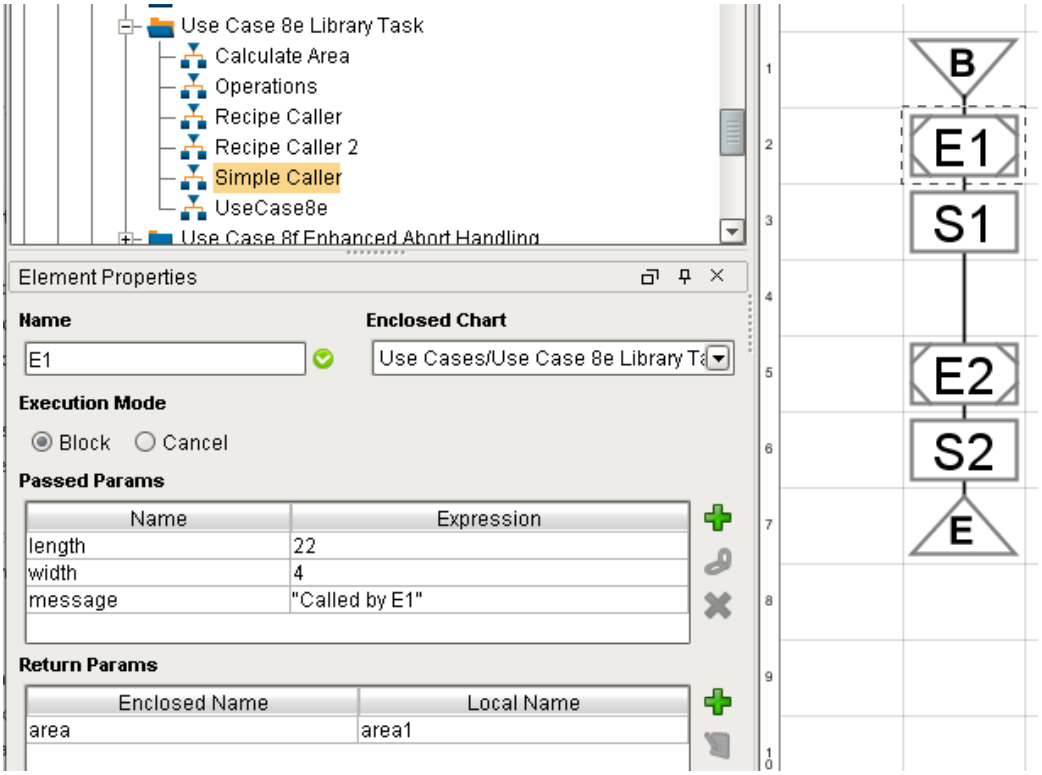
The following trivial "Library" chart calculates the area of a rectangle using the length and width chart scope variables:



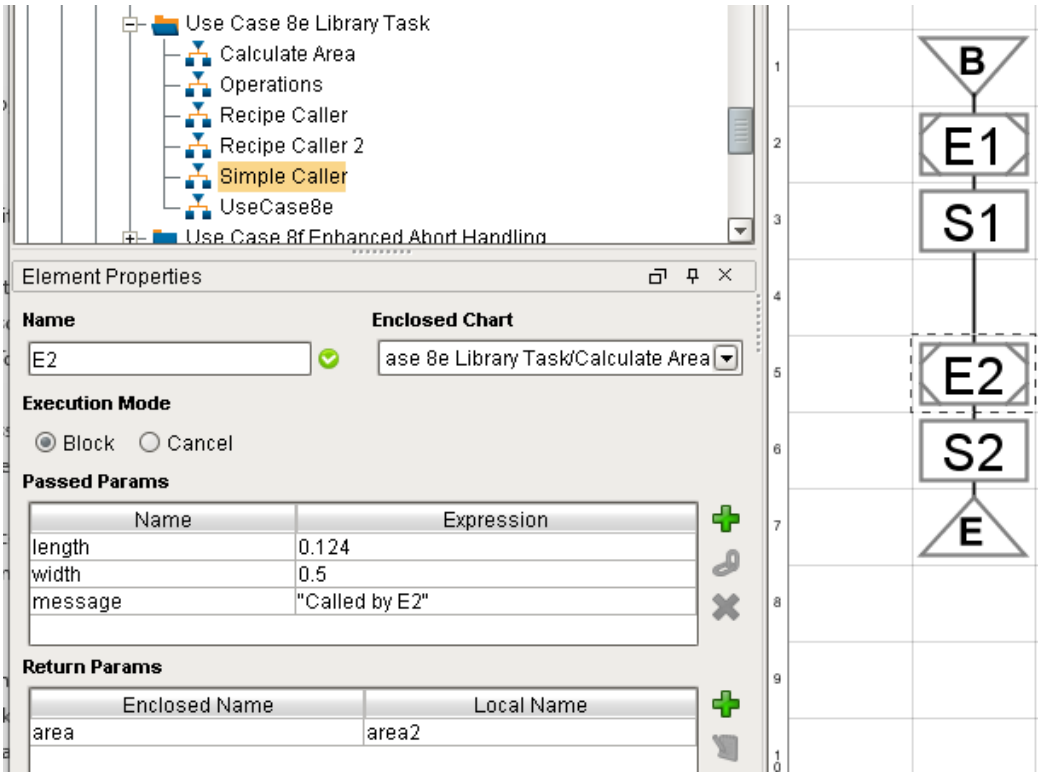
The Python for the action step is:

```
def onStart(chart, step):
    """
    This will run when the step starts, before any
    other action.

    Arguments:
        chart: A reference to the chart's scope.
        step: A reference to this step's scope.
    """

    chart.area = chart.length * chart.width

    print "Calculated the area (%s) from the length (%s) and width (%s)" % (str(chart
```

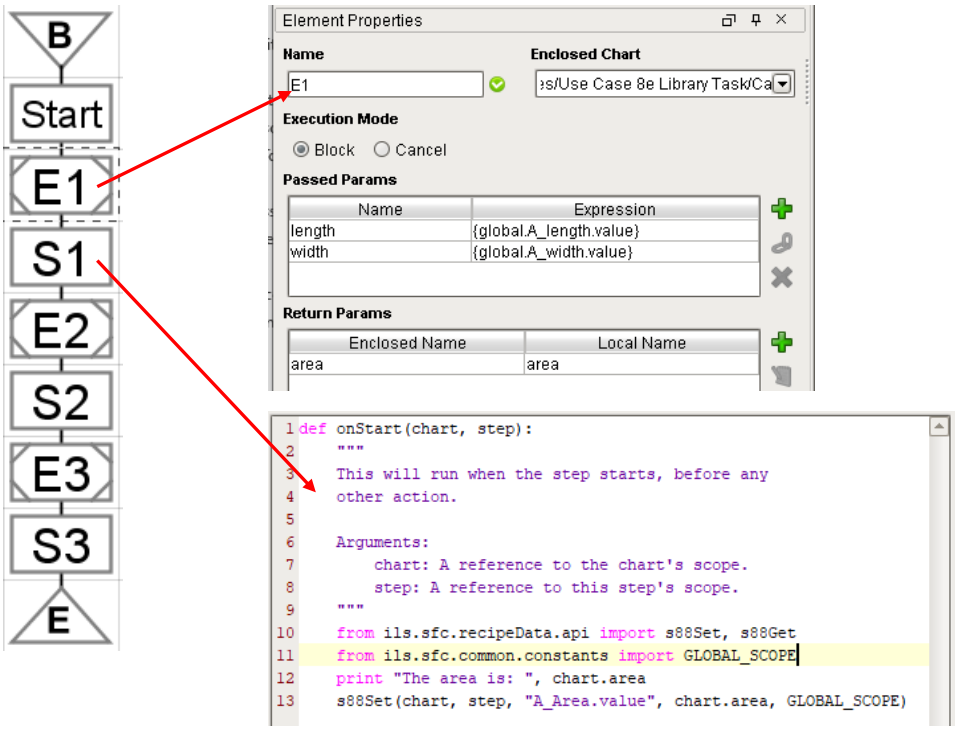
There are several ways to call this chart. The first example demonstrates passing the length and width as hard-coded values in step E1 and returning the area to a chart scope variable.



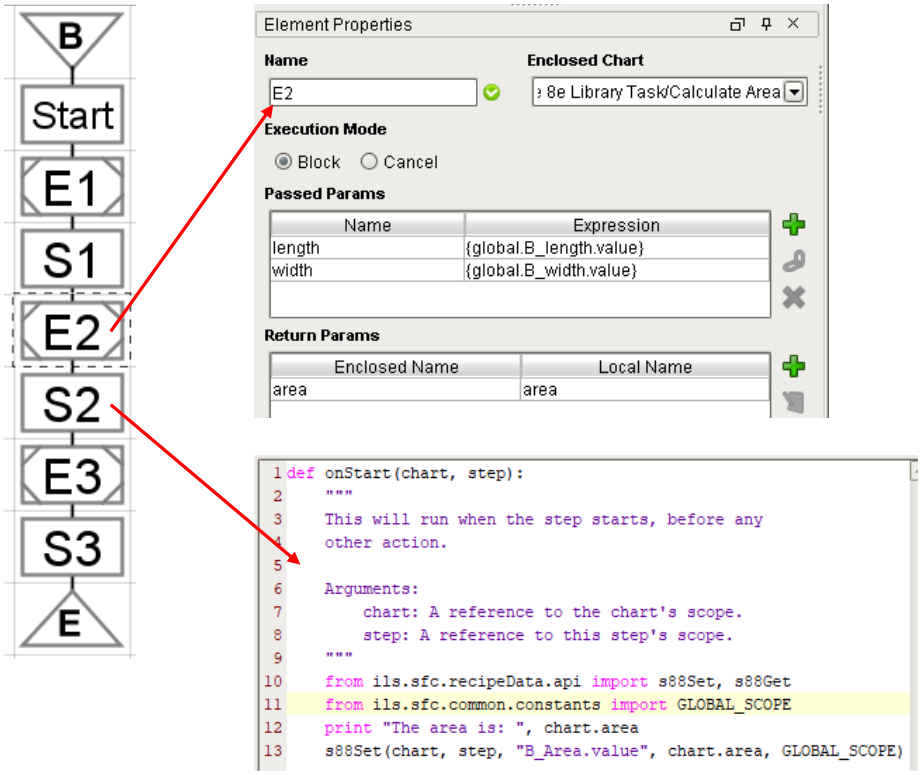
The second caller, step E2, passes different hard-coded values and returns results to a different chart scope variable:



The next example demonstrates using recipe data to pass inputs by valuen values. The syntax used is the same as is used in transitions. Using the recipe data expressions for the inputs works exactly as you would expect. Ignition evaluates these expressions at run time and passes the value to the called chart. However, using recipe data expressions for the return parameters does not work. The "Local Name" must name a chart scope variable and does not support the recipe data notation, therefore a local chart variable must be used and then a subsequent action step must explicitly move it into recipe data.



The second caller is very similar to the first but references different recipe data:



This example worked quite well because the library task contained an action step whose Python script was designed to use chart scope variables. Most charts in the Smart Sequence toolkit will use steps which are designed to work with recipe data. These steps generally do not use chart scope variables.

7.3.2 Library Chart using Recipe Data

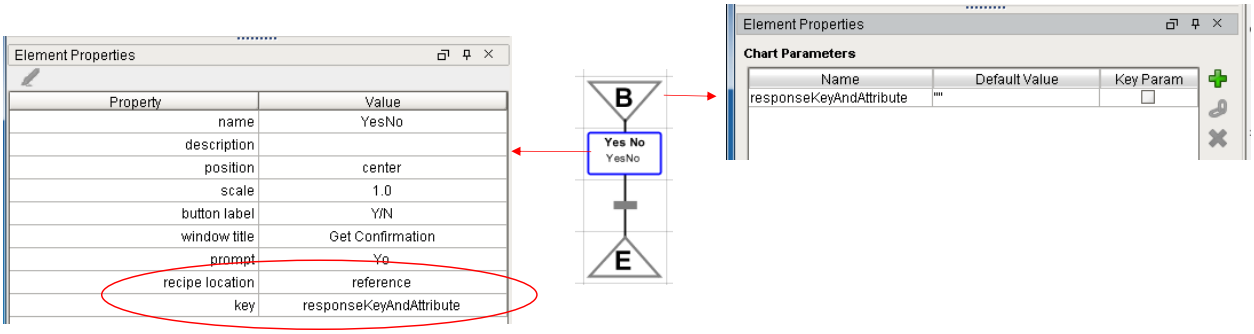
The first "Library" chart to consider is one that strictly uses standard Ignition steps. This is a textbook example of creating a reusable chart. The key to a reusable chart, just like a reusable function in Python, is to pass parameters to the chart. This demonstrates library arguments using pass by value.

Need to design this

7.3.3 Library Chart using Pass-by-Reference to Access Recipe Data

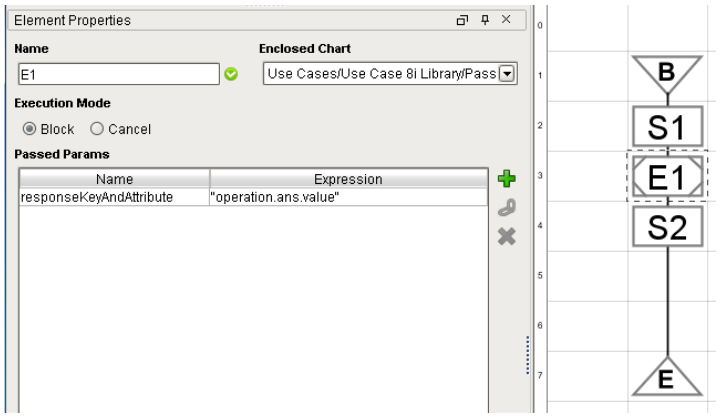
The purpose of this use case is to demonstrate how to pass references to recipe data to a reusable chart that allows for different recipe data to be passed by each caller. This technique is generally known as pass by reference. This example overcomes the limitations of pass-by-value that were pointed out in section 7.3.1. Pass-by-Reference allows the full slate of custom blocks to be used.

"Pass by Reference" is implemented using a new scope locator: *reference* in the "recipe location" dropdown. When the recipe location is reference, then the key must reference a chart parameter, in this case: *responseKeyAndAttribute*.



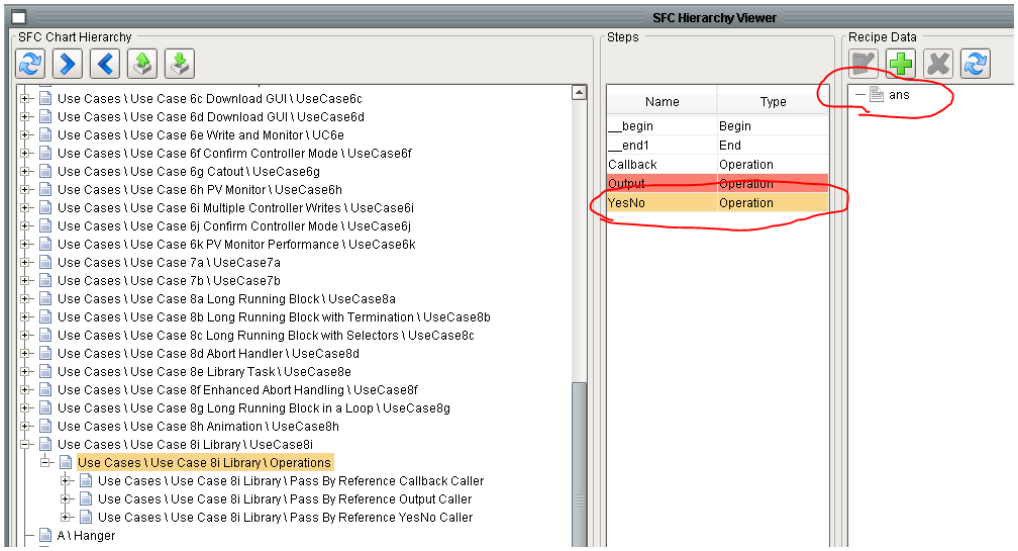
Library Chart using Pass-by-Reference

The caller of this chart must pass a recipe data reference in the form of *scope.key.attribute* as shown below:

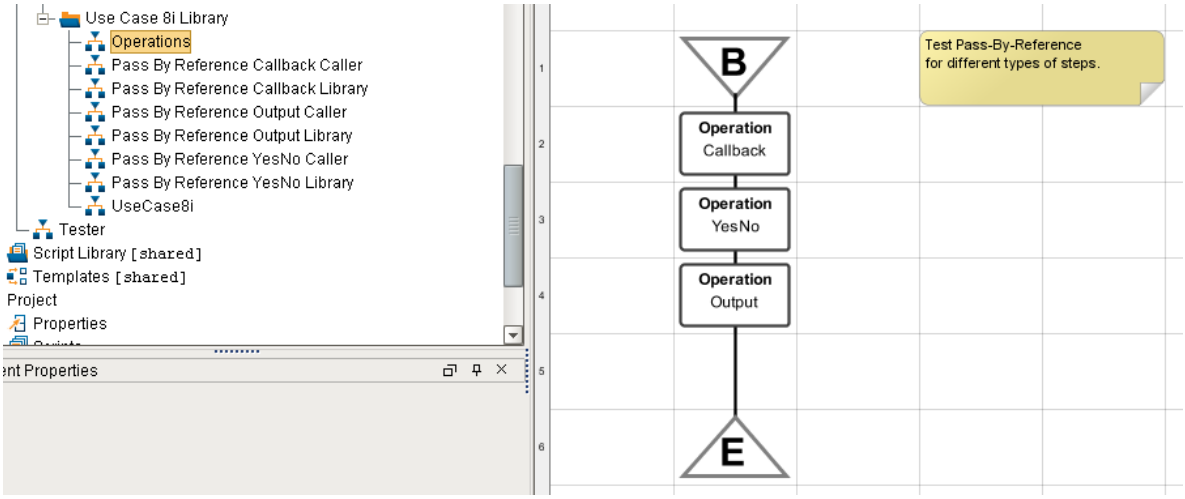


Pass by Reference Caller

This would pass the recipe data shown below:



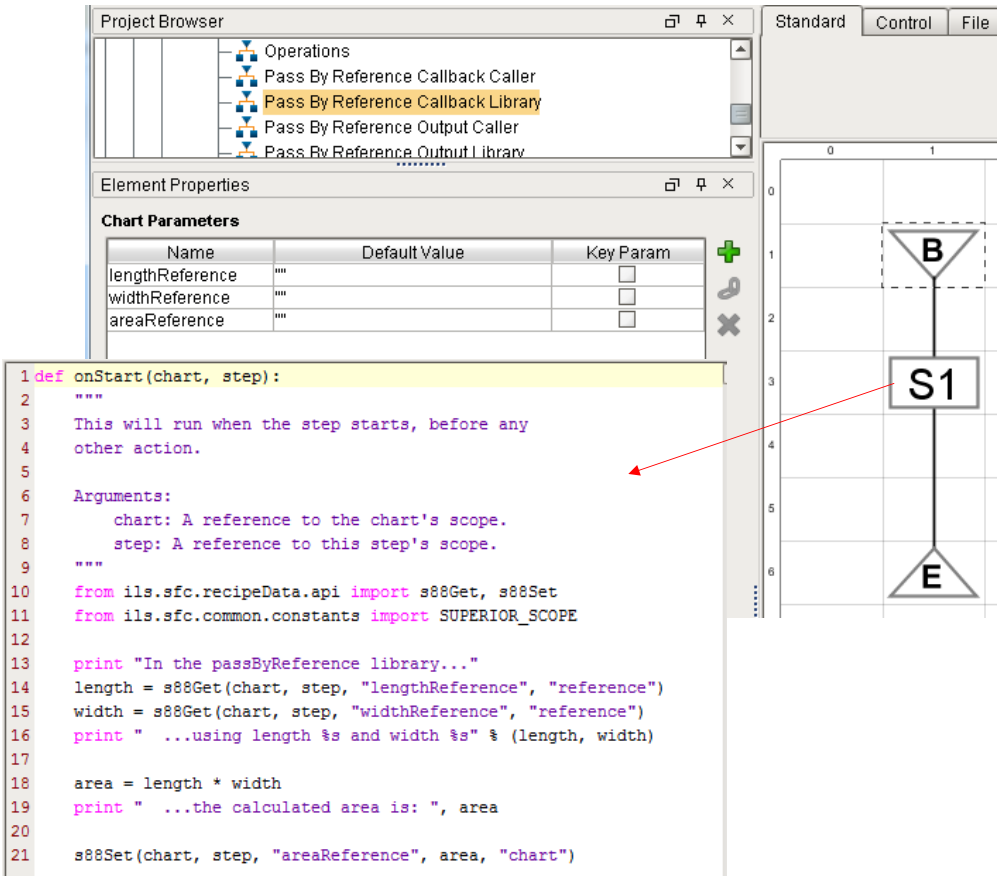
This example consists of three operations which demonstrates pass by reference with three different step types: an action step, a yes/no step, and a write output step. The three operations are shown below:



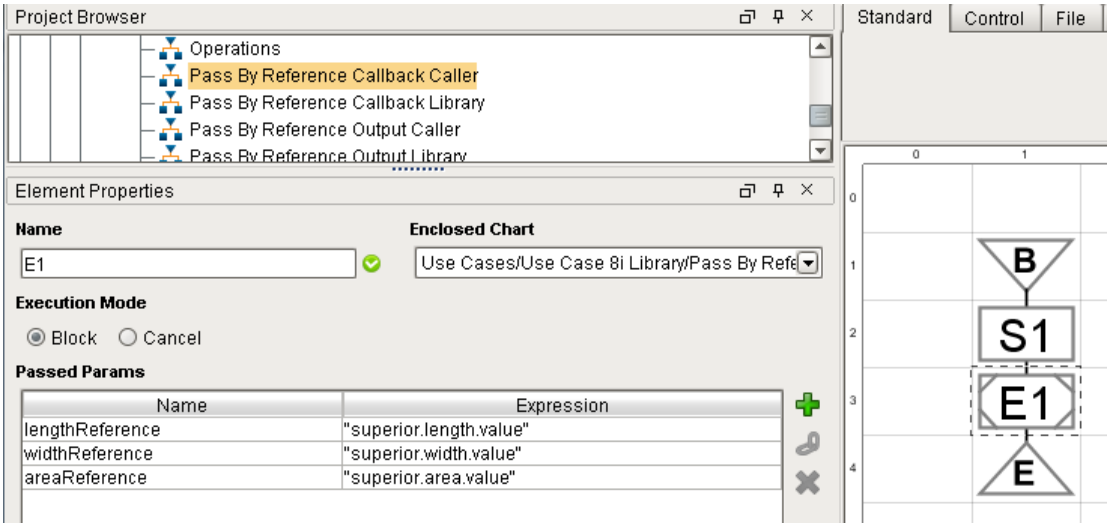
Operations Demonstrating Pass-by-Reference

The first operation shows how pass-by-reference is used in the Python of an action step to construct a reusable chart. The definition of the "library chart" is shown below. The chart calculates the area of a rectangle from recipe data and stores the result in recipe data. It defines three chart scope string variables: lengthReference, widthReferene, and areaReference. The Python for the action step uses these chart scope variables and the scope locator "reference" in the s88Get and s88Set APIs. Note that the Python does not *evaluate* the chart scope variable, rather it passes the chart

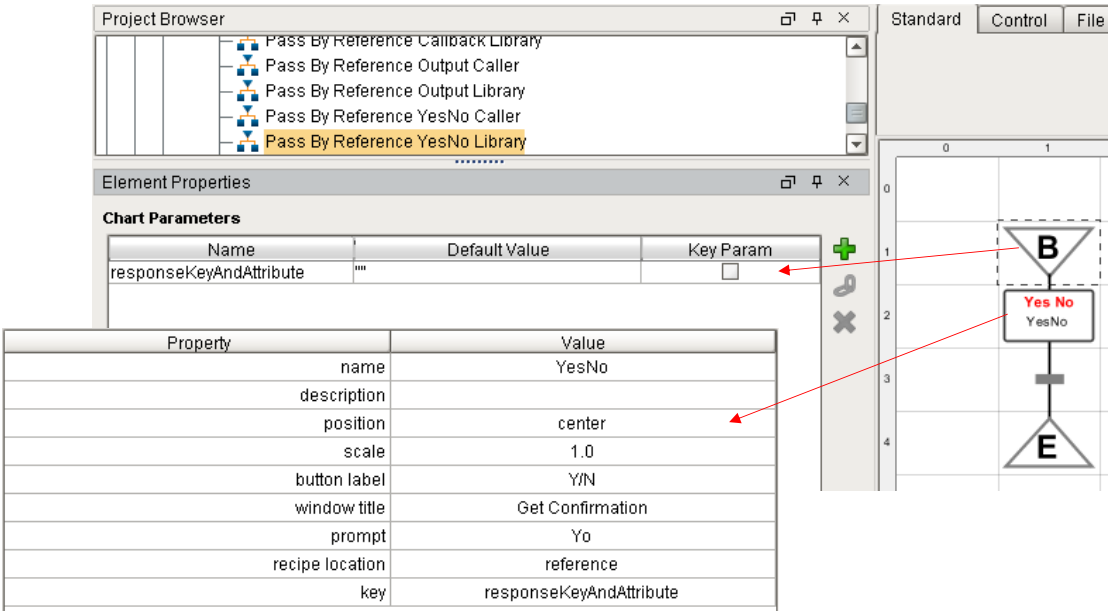
scope variable name and the chart scope to the *s88Get()* API and it evaluates the chart scope variable to get the recipe data reference.



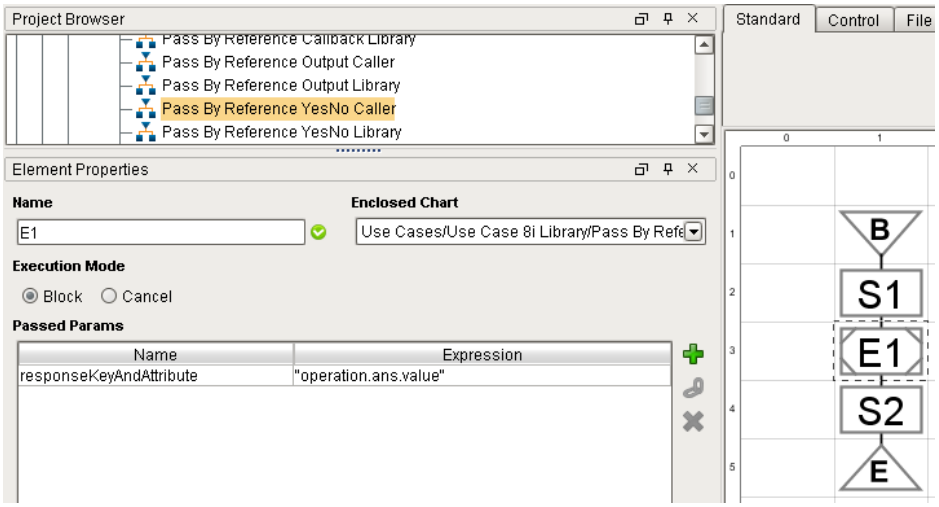
The calling chart is shown below. It specifies the full recipe data address in the form *scope.key.attribute* for the three chart variables. It is important to note that there must be agreement on the type of recipe data between the caller and the library chart. In this case the recipe data type of the three references must be simple value.



This next opeation demonstrates how pass-by-reference is used in a library chart that is constructed from built-in steps. This example uses a Yes/No step as an example of a typical built-in step. The chart defines a single chart variable: responseKeyAndAttribute. The Yes/No step expects a fully specified recipe data key of the form: *scope.key.attribute*.

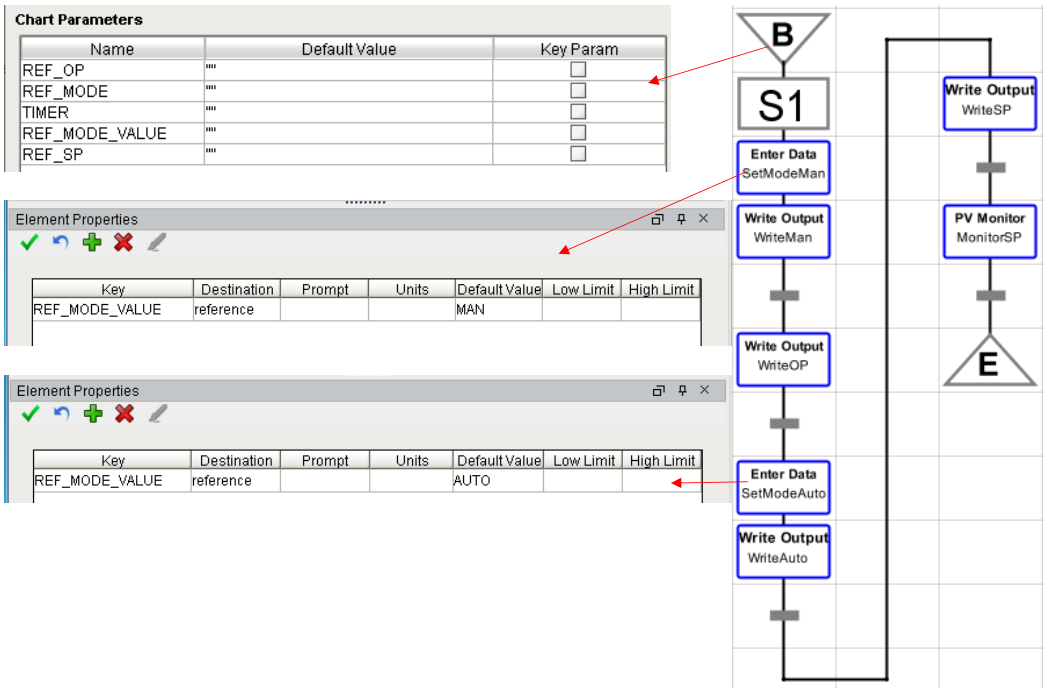


The calling chart is shown below:

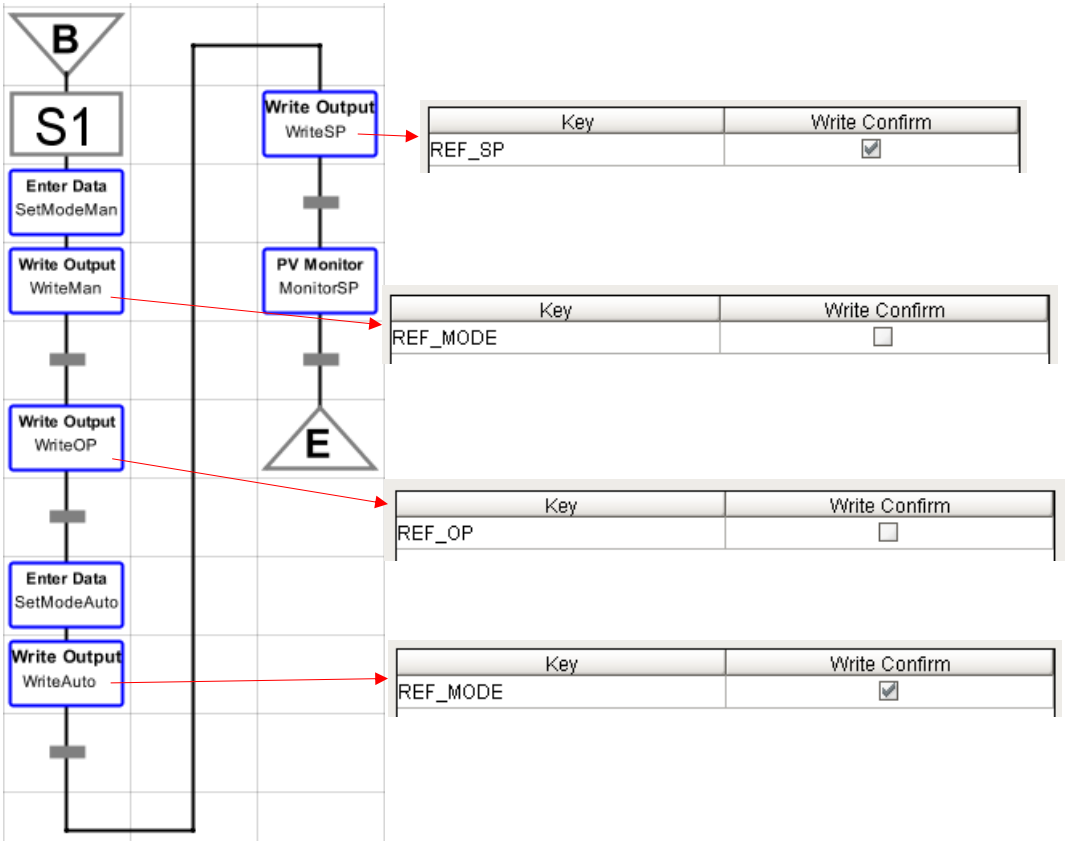


The final operation demonstrates how pass-by-reference is used in a Write Output step. This is a little more complicated but demonstrates a common real-life pattern. The library writes directly

to the OP of a controller but first places the controller into MAN mode and then after the write places the controller back in AUTO mode. The library chart is shown below.



The configuration of the four Write Output steps is shown below. The top level properties are the same for all three steps. The config for each step is different.



It requires four recipe data references which are used multiple times.

Chart Variable Name	Description
REF_MODE	References the OUTPUT class of recipe data that represents the mode tag of the same controller. Used by the first and third Write Output steps.
REF_MODE_VALUE	References the value of the same recipe data as the REF_MODE. Used by the two Enter Data steps and indirectly by the first and third Write Output steps.
REF_OP	References the OUTPUT class of recipe data that represents the OP tag of a controller. Used by the second Write Output step
REF_SP	References the OUTPUT class of recipe data that represents the SP tag of a controller. Used by the last Write Output step
TIMER	References a TIMER class of recipe data. Used by all the Write Output steps.

The calling chart is shown below. It is important to observe that the REF_MODE and the REF_MODE_VALUE variables refer to the same recipe data object but REF_MODE is a reference to the object and REF_MODE_VALUE is a reference to the outputValue of the object.

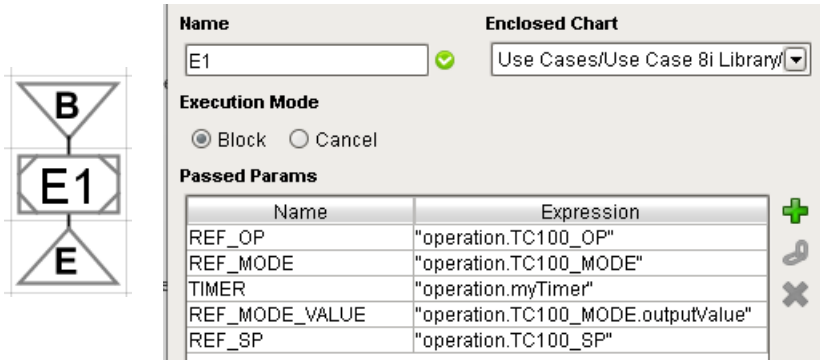


Chart "Pass By Reference Output Caller"

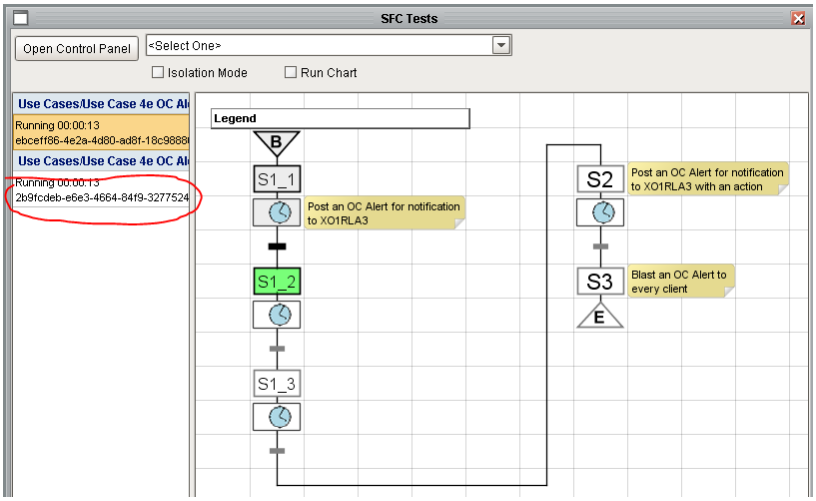
"Library" charts can only be called from the built-in ebcapsulation step; they cannnot be called from a unit procedure, operation, or a phase because they do not provide a way to pass references or values.

8. TESTING

There are numerous facilities for thoroughly testing SFCs some of which are provided as part of Ignition, others are provided as part of the ILS SFC extensions, and some are just best practices.

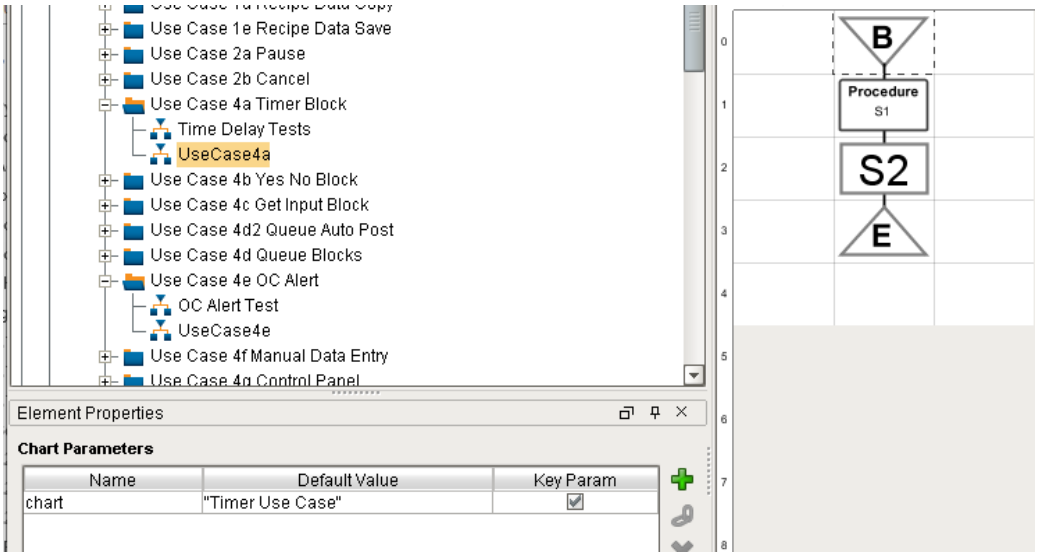
8.1 SFC Viewer

The SFC Viewer is a standard Ignition component that allows running charts, which run in the gateway, to be viewed from a client. The viewer can be easily be accessed by the "V" button on the SFC control panel.

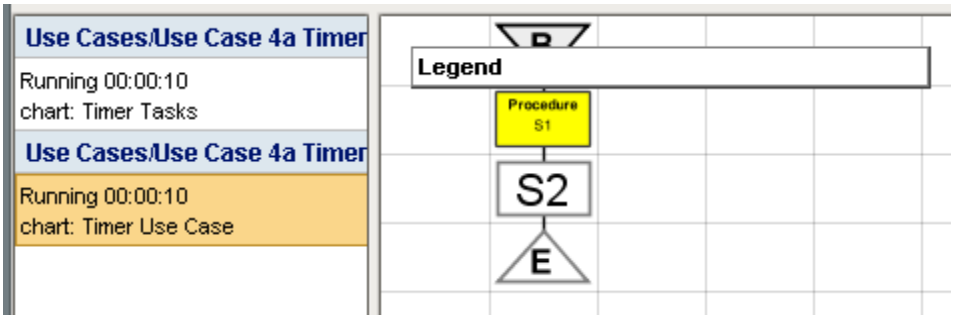


The viewer shows the UUID of the running chart in the browser along with the full path name of the chart. The UUID is particularly useless to the user and chart path is generally too long to be readable unless the window is huge. A best practice has been adopted to make the viewer more useful by defining a "Key" parameter for each chart that labels the chart with a meaningful name.

The parameter is defined by selecting the Begin step, creating a new chart parameter, generally named "chart", and pressing the "Key Param" check box.



The viewer now shows:



A standard behavior of the viewer is that the state of steps on a chart are animated to indicate their state. Some steps offer interactive information about the step can be obtained by hovering over step. A standard behavior of transitions is to display the expression in the transition. Note that only the raw expression is displayed. If tags or recipe data are referenced in the expression that the values are not visible. In addition to transitions, a selected set of steps also display a dynamic tooltip. The steps that support dynamic tooltips are:

- Time Delay – the time left.
- Write Output – the I/O still needs to be written.
- PV Monitoring – the I/O that is still being monitored.

8.2 Trace Logging

The Ignition platform provides a powerful logging capability based on the Java *log4j* package. Loggers operate at five different levels: *error*, *warn*, *info*, *debug*, or *trace*. The logger level is set in the gateway at run time from the gateway web page. A logger can be put in *trace* mode for debugging purposes and in info mode for normal operation. Because SFCs run in the gateway, the log messages are written to the wrapper log.

The ILS SFC extension module extends the logging capabilities of Ignition by automatically creating a logger for each chart on startup. The logger is named with the complete chart path. This creates a tree like structure of loggers that can be accessed from the gateway web page on the Status page. On startup the mode of each logger is *info*. In this more only minimal information is written to the wrapper log.

Built-in steps utilize the logger system extensively. When debugging a chart that consists entirely of built-in steps the user can obtain detailed information about the execution of each step by putting the chart logger into trace mode. Once debugged, the chart logger should be put back into info mode for performance reasons.

Designers can take advantage of this same framework in custom steps as shown below:

```
 6 import system
 7 from ils.sfc.common.constants import  OPERATION_SCOPE
 8 from ils.sfc.gateway.api import getChartLogger, cancelChart, getProviderName
 9 from ils.sfc.recipeData.api import s88Set, s88Get
10 from ils.common.util import readInstantaneousValues
11
12 def getData(chart, step):
13     '''
14     Former Name: rx-sample-get-data
15     Collect the current reactor configuration and grade from the DCS
16     '''
17     log = getChartLogger(chart)
18     chartPath = chart.get("chartPath","")
19     stepName = step.get("name","")
20     provider = getProviderName(chart)
21     log.tracef("In %s.getData() with %s - %s...", __name__, chartPath, stepName)
```

There are a couple things to point out:

- The API that conveniently gets the logger for chart needs to be imported on line 8
- A logger is created on line 17
- A trace log statement is defined on line 21. The message formatted in this statement will only appear in the wrapper log if the logger is in trace mode.
- The *__name__* is a standard Python internal variable which gives the path of the module. This is useful when using external Python and is unnecessay when writing Python directly in the step.
- If you want the message to always appear in the wrapper log use log.infof().

Finally, log statements should be used rather than print statements since they can be easily be turned on and off without editing the code.

8.3 Isolation Mode

There are two modes to run an SFC in, *Production* and *Isolation*. This is specified by a Boolean client tag, *IsolationMode*. The SFC will run in whatever mode the client is in at the time the SFC is started. By default, a client's mode is Production. An engineer can change their mode at any time by selecting Admin -> Test -> Client Isolation Mode Settings from the main menu.

The difference between the two modes is that they use different database and tag providers as specified in the View/External Interface Configuration window. This allows use of an isolated test mode that does not affect actual production equipment.

In order to support isolation mode in a way that is consistent with Ignition's data source binding functionality, there are two Client tags: *Database* and *Provider*, that are set automatically with a change script on the IsolationMode tag. The default data source or provider should NEVER be used in client scripting; bind to those tags instead. There are two scripting functions that should be used to access the client tags rather than reading them directly. They are: *getProvider()* and *getDatabaseName()*. Refer to section 6.6 for details.

For transition expressions to honor isolation mode it is necessary to use the "tag" scope locator in the expression rather than hard coding the tag provider name in the expression. For example: {tag.<tag path>}, e.g. {tag.mytag.value}. Do NOT include the tag provider in this expression; the

provider will be determined by the isolation mode setting. This mode is currently limited to the "value" attribute of the tag, but could be extended to other attributes (e.g. AlarmActiveAckCount).

8.4 Chart Recordings

A standard capabilities of SFCs is to be able to record chart execution. You can think of this like the black box recorder in an airplane. The recording allows you to trace the execution of a chart in very granular detail.

Chart recordings are configured from the gateway web page. The capability can be turned on/off and the number of recording per chart and the retention time can be specified. The chart recording is stored in a file on the filesystem under the Ignition install directory.

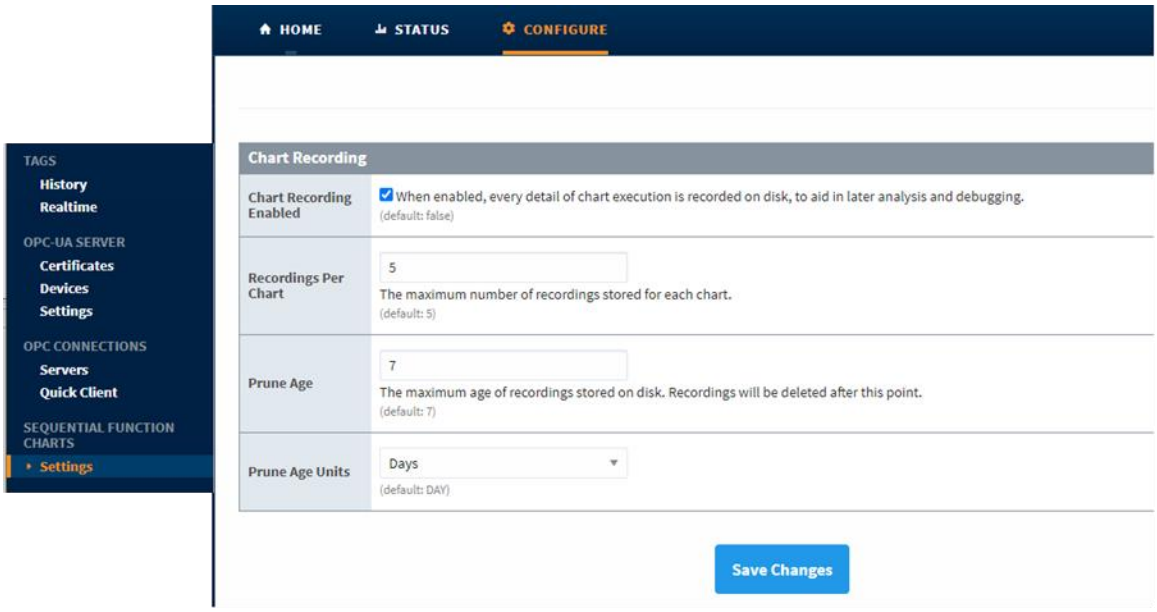
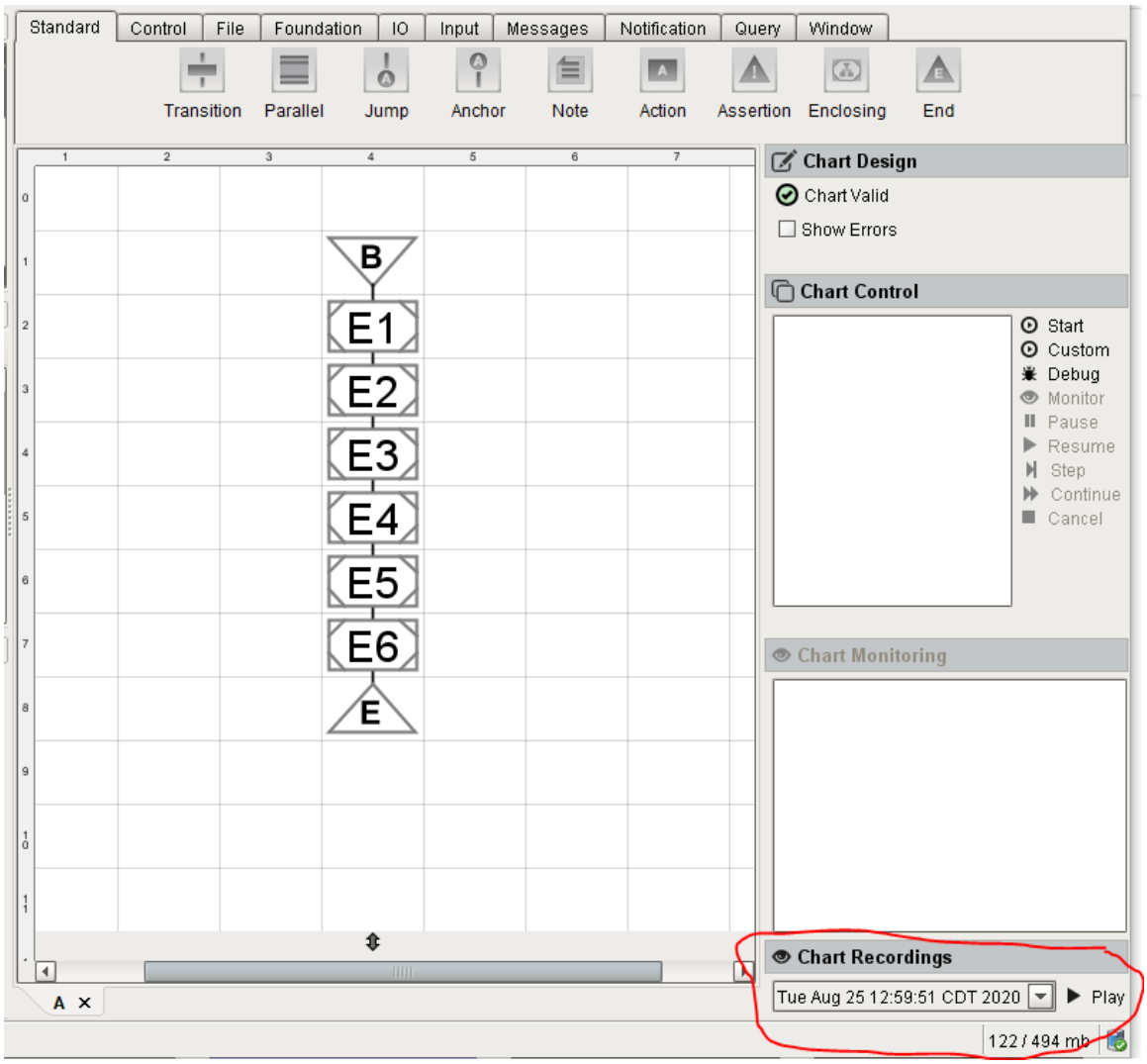
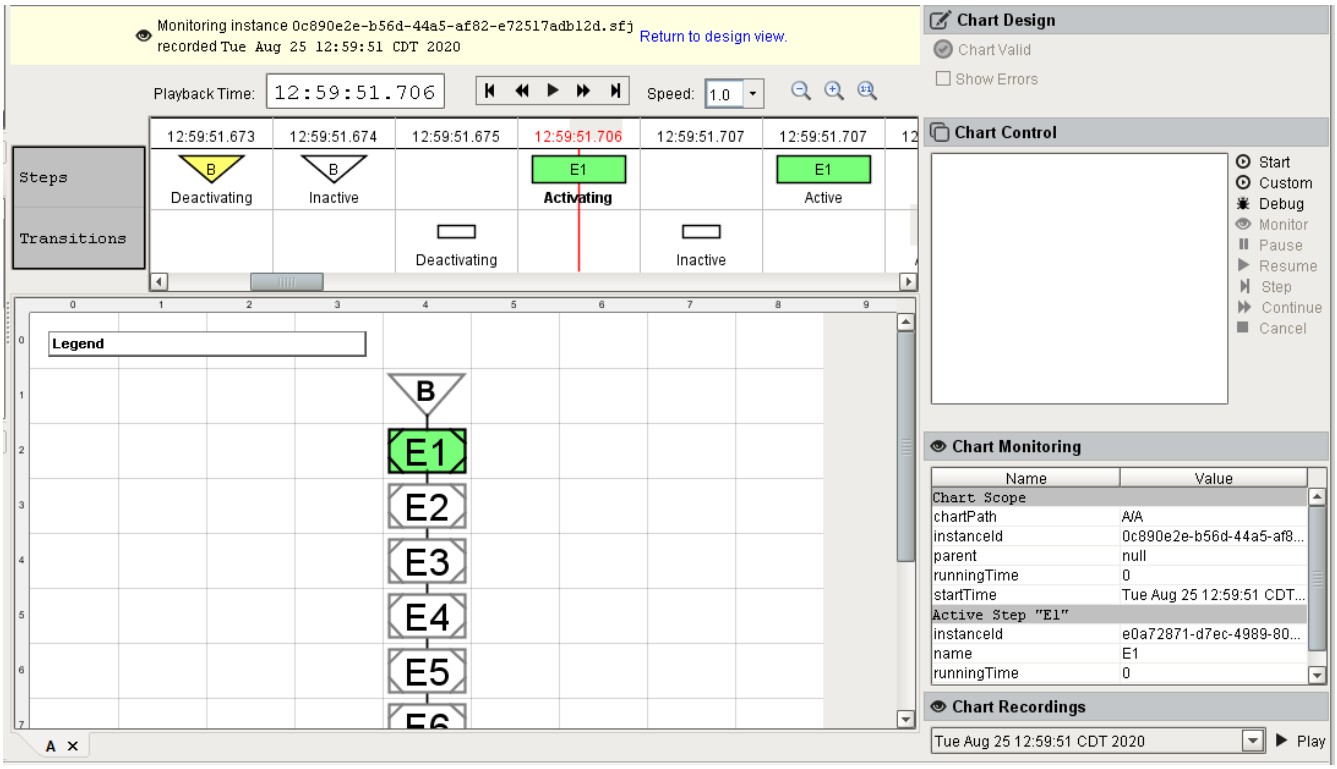


Chart recordings are viewed from Designer. A dropdown list of available recordings is shown in the bottom right. The list shows recordings that are available for the selected chart.



Pressing play will show a chart execution timeline above the chart. The replay controls allow for stepping through the chart one step at a time or replaying the chart from beginning to end. It animates the chart as you step through the recording. It also displays step properties in the lower right panel.



The recording highlights the many states that an individual step goes through as execution proceeds. It does not provide any insight into exactly what actions a step took. For example, if the Python for an action step contains a number of if statements, the replay tool does not give any insight into what the Python did. Likewise, it doesn't provide any insight into the success or failure of tag write operations, either in an action step or in the custom Write Output step.

A chart recording can be a valuable tool when debugging an SFC problem. It can be used to trace the flow of execution through a chart with multiple transitions. It can be used in conjunction with the wrapper log, message queues, and operator logbooks.