**Sequential Control**

**Software Developer Design Guide**

Version 1.8

July 6, 2022

**ILS Automation, Inc.**

Contents

[1. Introduction 5](#_Toc112260304)

[2. Running Sequential Function Charts 6](#_Toc112260305)

[2.1 Running a Sequential Function Chart From a Client 6](#_Toc112260306)

[2.1.1 Control Panel 6](#_Toc112260307)

[2.2 Running a Sequential Function Chart From the Designer 7](#_Toc112260308)

[3. Recipe Data 8](#_Toc112260309)

[3.1 Synchronizing Ignition and the Database 8](#_Toc112260310)

[3.1.1 Designer Synchronization 8](#_Toc112260311)

[3.1.1.1 Change Listeners 8](#_Toc112260312)

[3.1.1.2 Moved Charts 9](#_Toc112260313)

[3.1.1.3 Moved Steps 9](#_Toc112260314)

[3.1.1.4 Renamed Charts 9](#_Toc112260315)

[3.1.2 Gateway Synchronization 10](#_Toc112260316)

[3.2 Step identification – UUID vs Name 10](#_Toc112260317)

[3.3 Unit Support 10](#_Toc112260318)

[3.4 Steps to Create a New Type of Recipe Data 10](#_Toc112260319)

[4. Step Definitions 11](#_Toc112260320)

[4.1.1 Error Reporting 11](#_Toc112260321)

[4.2 Common Step Behavior 11](#_Toc112260322)

[4.2.1 Long-Running Steps 11](#_Toc112260323)

[4.2.2 Timeout Handling 11](#_Toc112260324)

[4.2.3 Support for Chart Scope and Step Scope Variables 12](#_Toc112260325)

[4.2.3.1 Java Changes 12](#_Toc112260326)

[4.2.3.2 Python Changes 12](#_Toc112260327)

[4.2.3.3 Database Changes 12](#_Toc112260328)

[4.2.3.4 Window Changes 12](#_Toc112260329)

[5. Scripting / Action Steps 13](#_Toc112260330)

[5.1 Types of Actions 13](#_Toc112260331)

[5.2 Logging Support 13](#_Toc112260332)

[5.3 Accessing Tags and Tag History in Action Steps 13](#_Toc112260333)

[5.3.1 Reading a Tag’s Value at a Specific Moment in Time 13](#_Toc112260334)

[5.4 Scripting API 13](#_Toc112260335)

[5.4.1 Argument Dictionary 14](#_Toc112260336)

[5.4.2 Tag History Examples 14](#_Toc112260337)

[5.4.3 Triggerring a Final Diagnosis from a Chart 15](#_Toc112260338)

[6. Error Handling 16](#_Toc112260339)

[6.1 Exception Handling in Action Blocks 16](#_Toc112260340)

[6.2 Error Handling 16](#_Toc112260341)

[7. Extensions to Transition Expressions 18](#_Toc112260342)

[7.1 Recipe Data 18](#_Toc112260343)

[7.2 Tag Values 18](#_Toc112260344)

[8. Designer Extensions 19](#_Toc112260345)

[8.1 Find/Replace 19](#_Toc112260346)

[8.1.1 Find/Replace Overview 19](#_Toc112260347)

[8.1.2 Find/Replace Implementation 20](#_Toc112260348)

[8.1.3 Recipe Data Search Implementation 20](#_Toc112260349)

[8.2 Running Charts From Designer 20](#_Toc112260350)

[8.2.1 Custom Chart Parameters 21](#_Toc112260351)

[8.2.2 Mock Call Hierarchy 22](#_Toc112260352)

[8.3 Context Sensitive Help 24](#_Toc112260353)

[8.3.1 User Manual Preparation 24](#_Toc112260354)

[8.3.2 Java Module Implementation 25](#_Toc112260355)

[8.3.3 Build Script 26](#_Toc112260356)

[9. Substitution Expressions in Messages 27](#_Toc112260357)

[10. SFC KPI 28](#_Toc112260358)

[10.1 Data Collection enhancements 28](#_Toc112260359)

[11. Architecture 29](#_Toc112260360)

[11.1 Persistent Windows 29](#_Toc112260361)

[11.2 Step Implementation 30](#_Toc112260362)

[11.2.1 SDK model for Custom Steps 30](#_Toc112260363)

[11.2.2 Java Implementation 30](#_Toc112260364)

[11.2.3 Python Implementation 31](#_Toc112260365)

[11.2.3.1 Short Running Step 31](#_Toc112260366)

[11.2.3.2 Long Running Step 32](#_Toc112260367)

[11.2.3.3 Long-Running Step with User Interface 35](#_Toc112260368)

[12. Export / Import Support 39](#_Toc112260369)

[12.1 Recipe Data Converter Class 39](#_Toc112260370)

[12.1.1 Internalize Recipe Data 39](#_Toc112260371)

[12.1.2 Storing Internalized Recipe Data Into the Database 40](#_Toc112260372)

[12.2 Python Methods 40](#_Toc112260373)

[13. Testing 42](#_Toc112260374)

# Introduction

This document explains certain aspects of the design of the ILS SFC system which consists of a module, database, windows, and external Python. This document is intended for the software developers responsible for the design of the module, Python, database, or Ignition project. It may also be of interest to advanced users that are responsible for debugging and support of an application using the toolkit.

# Running Sequential Function Charts

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 control panel interactions, charts should be started using the *ils.sfc.common.util.startChart()* API function. The framework assumes that the chart that is passed to startChart contains a unit prodedure step. The

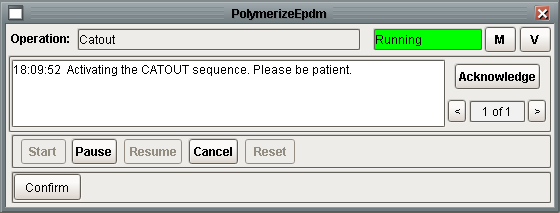
## Running a Sequential Function Chart From a Client

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 control panel interactions, charts should be started using the *ils.sfc.common.util.startChart()* API function. The framework assumes that the chart that is passed to startChart contains a unit prodedure step. The

### 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: buttons for cancelling, pausing, and resuming a chart; 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 is shown below:



The “Cancel” button is one of several ways that a running SFC can be aborted. A unit procedure consist 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.

## Running a Sequential Function Chart From the Designer

Refer to section 7.2 for a discussion about running SFCs from Designer.

# Recipe Data

Recipe Data are created, deleted, and edited in the client using the SFC Hierarchy Viewer window.

## Synchronizing Ignition and the Database

In order to implement recipe data it is necessary to have a model of the SFC charts in the database. If recipe data is not used, then it is not important if Ignition and the database are 100% in sync. However, it is assumed that if the SFC extension module is used, then recipe data is also being used. Therefore, a careful analysis and significant logic has been implemented to maintain a consistent model

### Designer Synchronization

The goal is to keep the database model synchonized with Ignition’s internal database by updating the model as changes are made through the designer.

#### Change Listeners

The *projectResourceModified()* listener rere are three types of event notifications:

* Deleted: called immediately when a chart is deleted. The name is always <n/a>, the chart is identified by the internal resource id. This is also called immediately when a chart is moved.
* Added: called immediately when a chart is added. The name is always <New Chart> for a new chart. This is also called immediately when a chart is moved, the name will be the new chart path.
* Updated : called immediately when a chart is closed, even if nothing was actually changed; also called when the user presses Save for every open chart even if nothing was actually changed.

As indicated above, when a chart is deleted from the project tree, the module is notified immediately, however the chart is not deleted from Ignition’s internal database until the user presses Save. A user may delete a chart, quit the designer without saving, and the chart will NOT be deleted from the internal database. In order to mimic this same behavior with our database, the immediate notifications are recorded in an internal data structure and processed only when the project is saved.

Likewise, when a chart is created, the *projectResourceModified()* callback is called immediately when a chart is created but the chart is not stored in Ignition’s internal database until the project is saved. A user may create a chart, quit the designer without saving and the chart will NOT be saved to the internal database.

The logic to update our database is implemented in Python in *compileCharts()* in the module *ils.sfc.designer.saver.py*. It is called by *projectUpdated()* in the ilsSfcDesignerHook.

#### Moved Charts

A moved chart merits special consideration. When a chart is moved we receive a delete and an added notification with a new resource id. In addition, if the chart was open when it was moved, we also receive an update notification. Moving chart Child from FT to FT/A will generate the following notifications:



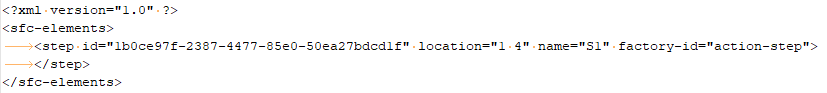
It is important to notice two important things: 1) the moved chart has a new resource id, 2) the delete notification record does not include the old chart path.

#### Moved Steps

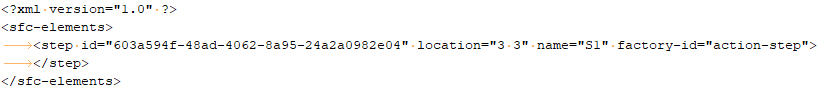
A step can easily be moved from one chart to another by selecting the step, pressing <CTRL>-x, opening the target chart, selecting an empty cell, and pressing <CTRL>-v. If there is recipe data defined on the original step then the recipe data should be copied along with the step.

In order to do this, a link between the new step and the old step is needed. Unfortunately, there is nothing in the chart XML that is received in the update notification. When the step is pasted, Ignition gives the step a new UUID. The only identifying characteristic in the step XML is the step name, which is not unique enough to determine a step is newly created or moved from another chart.

The step XML on the original chart is:



The step XML on the target chart is:



Comparing the two snippets shows that the UUID has changed.

#### Renamed Charts

Whan a chart is renamed, from the Designer’s project tree, even if the chart is not open, the module will receive an updated event. Renaming chart Child in FT to Kid will immediately raise the following notification:



It is important to notice that the event does not specifically indicate that the chart was updated, it also does not contain the old name.

### Gateway Synchronization

On gateway startup, the entire chart hierarchy is analyzed (HOPEFULLY)

## Step identification – UUID vs Name

A fundamental assumption of the recipe data system is that a step is uniquely identified by its UUID. Unfortunately this is not the case. Duplicate UUIDs occur when a chart is cloned and also when a chart is imported more than once. Therefore the basis for identifying a step will be changed to use the chartName and step name. Copy and pasting a step gets a new UUID, but copy and paste of the whole chart does not.

Ignition enforces name uniqueness of action steps on a chart. It will not allow a duplicate name to be saved in the property editor. Unique names are not enforced on custom steps (we’d like to but we’re not sure how). Because we can’t enforce it in Designer we can’t enforce it in the database. Originally we had a unique constraint on chart / Name, but is was frequently violated resulting in an aborted save. Once we can validate that step names are unique, perhaps we can put the unique constraint back.

The recipe data API uses chartPath and stepName to avoid the issue with non-unique step UUIDs

## Unit Support

Automatic unit conversion is implicit in recipe data and is based on the generic unit definition module.

Refer to the **Common Facilities User’s Guide** for details.

## Steps to Create a New Type of Recipe Data

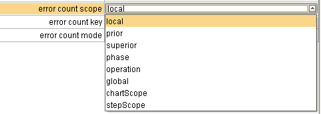
This provides an outline of the steps required to implement a new class of recipe data.

1. Define the new type by adding a record to SfcRecipeDataType
2. Design the new tables(s) in Visio (EMChemicals/Database/EMC Database ER.vsd) and update the Database Design Specification (EMChemicals/Database/EMC Database Design.docx).
3. Create the new tables in SQL\*Server.
4. Create a View patterned after SfcRecipeDataSimpleValueView.
5. Update the RecipeDataEditor Vision Window
   1. Add a dataset property for the a record from the view
   2. Add a new container to the window for the class specific properties.
6. Update Python supporting the Vision Window (ils.sfc.recipeData.visionRecipeDataEditor.py)
7. Add support for s88Set() and s88Get() to ils.sfc.recipeData.core.py
   1. fetchRecipeData()
   2. setRecipeData()

# Step Definitions

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

### Error Reporting

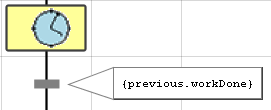


## Common Step Behavior

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

### Long-Running Steps

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.



Long running steps types include: Input, Select Input, Limited Input, Yes/No, Control Panel Message, Notify Dialog, Review Data, Review Data with Advice, Review Flows, Manual Data Entry, Timed Delay, PV Monitor, and Write Output

### Timeout Handling

A number of steps support the notion of a timeout. Timeouts may be appropriate for steps that wait for input from the user or steps that wait for some event or tag value. The Yes/No step is one example of a step that supports a timeout.

The implementation of timeouts is significantly different in Ignition than in the old platform. In the old platform, the timeout behavior was implemented by placing a timer block in parallel with the step of interest and whichever branch completed first satisfied the parallel transition. In Ignition, 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.

### Support for Chart Scope and Step Scope Variables

The toolkit was designed without much consideration given to using chart scope and step scope variables in steps. Chart and step scope variables may be easy substitutes where simple value recipe data is used.

There are several reasons for using chart and step scope variables rather than recipe data:

Convenience –

Performance – the client can directly set the value of the chart and step scope variable. When recipe data is referenced in a transition, the database is queries every second for the value. A simple s88Get generally consists of several queries.

Support for chart and step scope variables was largely added in version 1.2.

#### Java Changes

The RECIPE\_LOCATION enumeration was changed to

This affects 12 steps.

#### Python Changes

The Python

#### Database Changes

The Chart and Step id need to be stored in the datbase so that a client can retrieve the

#### Window Changes

Add the response type, chart id, and step id to the root container.

# Scripting / Action Steps

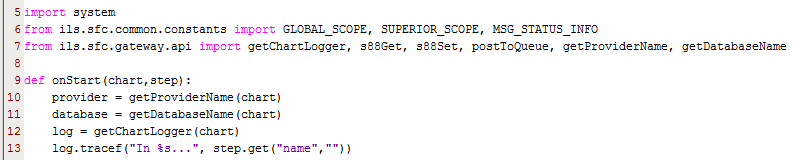
Action steps are a standard feature of IA’s SFC module. Action steps perform actions specified by a Python script.

## Types of Actions

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

## Logging Support

It is important to consider logging, introduced above in section 3, when implementing an action step. The ILS SFC framework will automatically create a logger for each chart that can be set to info or trace mode (debug mode is also a level but is generally not used in favor of trace). There are four logger levels that are typically used: log.info, log.warn, log.error, and log.trace. 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.



## 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.

### 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.

## Scripting API

The external Python module ils.sfc.gateway.api contains methods intended for use from ActionSteps. 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.

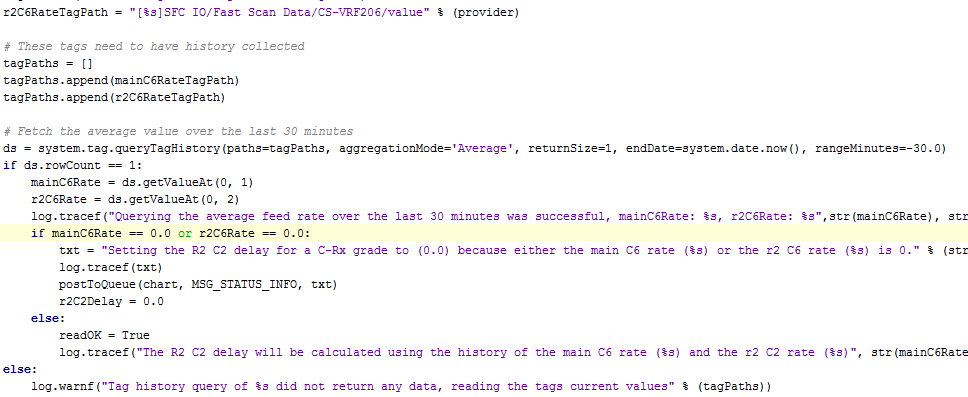
### Argument Dictionary

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 |
|  |  |
|  |  |
|  |  |

### 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.



### Triggerring a Final Diagnosis from a Chart

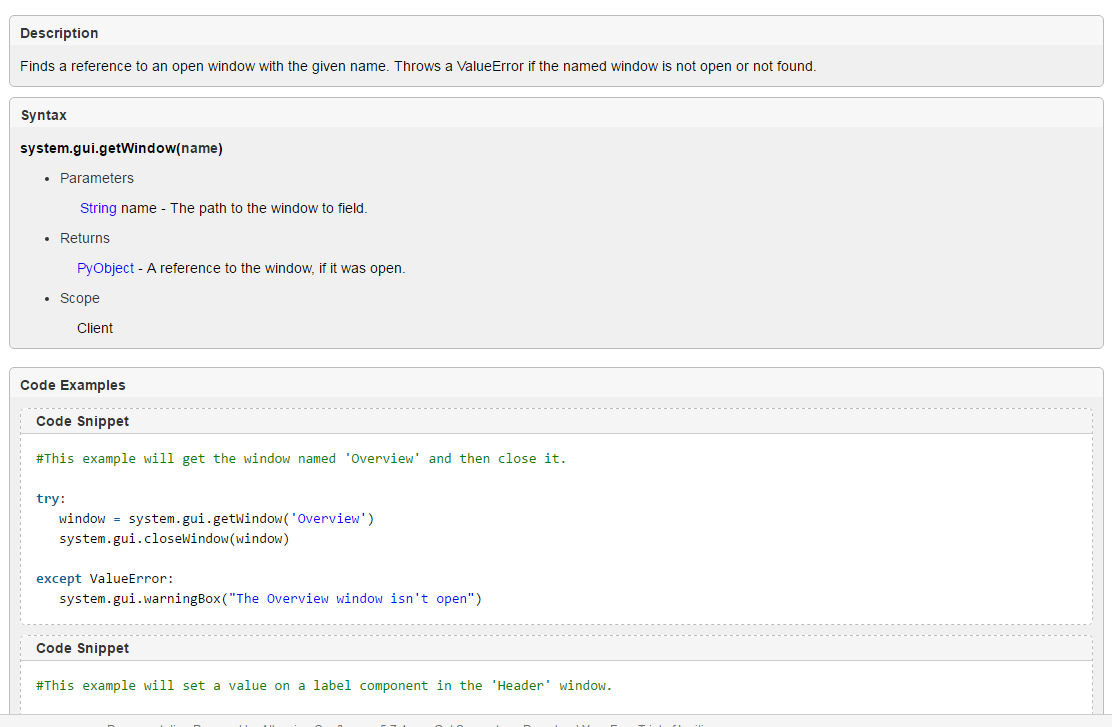
It is common to trigger a final diagnosis from a chart step. Common examples of this are the rate change and flying switch operations. The main reason for triggering a final diagnosis is to take advantage of the framework for writing I/O complete with a common user interface and safety checks on change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

# Error Handling

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

## 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.



## Error Handling

An important aaspect of the error handling scheme is to notify the client that launched the SFC the details of the error that occurred in the gateway as the SFC executes. The major methods that are involved with this is shown below.



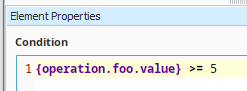
Figure 1 - Error Handling Logic

# Extensions to Transition Expressions

The SFC system has been extended to support recipe data and Isolation mode.

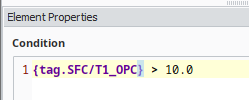
## Recipe Data

Recipe data may be accessed in transitions using the following syntax: {<scope>.key} where scope is one of the standard recipe data scopes. For instance, {global.mydata.value}.



## Tag Values

Ignition supports evaluating tag values in transition expressions. ILS has extended the transition expression language to be aware of the isolation and production tag providers and the mode in which the chart was launched. In order to support isolation mode settings, tag values may be accessed by the syntax: {**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 of the chart. This mode is currently limited to the “value” attribute of the tag, but could be extended to other attributes (e.g. AlarmActiveAckCount).



# Designer Extensions

This section describes features that have been added or extened in the Designer.

## Find/Replace

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

### Find/Replace Overview

The ILS SFC module adds these options to the Find/Replace window:

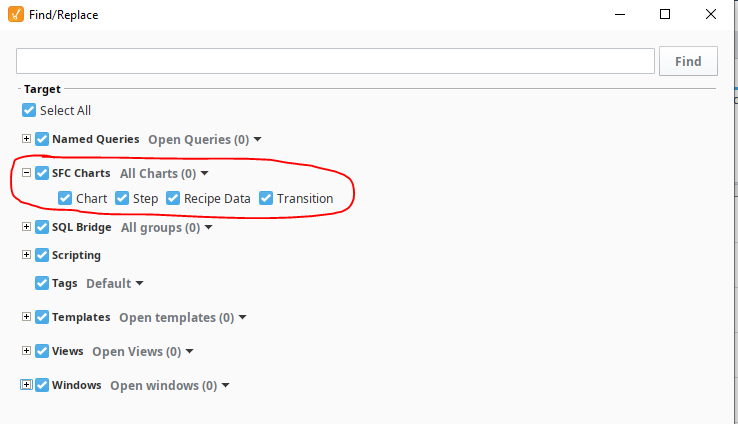


Figure 2 - SFC Extensions to the Find/Replace Framework

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

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.

There are three aspects to the Find/Replace framework:

* 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.

### Find/Replace Implementation

The Java extensions are contained in the *com.ils.sfc.designer.search* package in the *sfc-designer* project. There is a SearchObject class for each type of SFC resource being searched. Each of these represents the resource type as a text string.

### Recipe Data Search Implementation

Recipe Data is stored in the database. Searching recipe data implemented in Python, called from Java. The fundamental technique of the Ignition search is to represent a resource as a text string and search it for the desired text. A recipe data “record” contains numerous fields across several tables. The search operates on a chart at a time. Each class of recipe data has a view which is used to find all of the recipe data for the target chart path. The query is returned in a Python dataset. Each record in the data set is converted into a text string and returned to the Java for processing by the Ignition search engine. The Python does not search for matches, it returns all recipe data for the chart and Ignition searches for matches. It is worth pointing out that each chart involves nine queries, one for each type of recipe data. The nine queries are executed for each chart. So if there are 100 charts, there would be 900 queries. Initial testing showed that for a system with 348 chart the find took six seconds. Processing could be optimized by querying all recipe data at the beginning of the search and then caching the results without hitting the database for each chart.

## Running Charts From Designer

Running a chart from Designer is a normal step in the development of an SFC. Ignition provides a control panel for starting a chart from Designer. However, the standard Ignition control panel does not insert the special entries in the chart dictionary that need to be added to the paramater dictionary that specify the tag provider, database provider, and time factor. The chart dictionary also contains a call stack that is necessary to determine the scope for recipe data access.

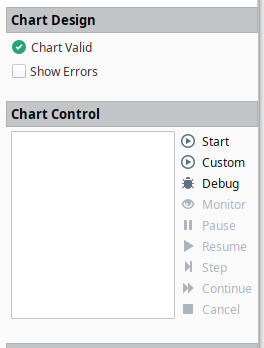


Figure 3 - Standard Ignition Control Panel for Running SFCs

To overcome these limitations, the ILS SFC module adds the following menu choices:

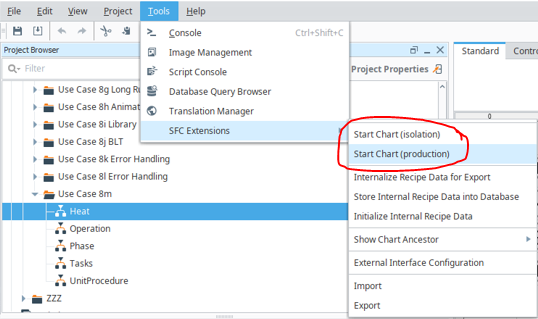


Figure 4 - Custom Menus for Running SFCs from Designer

These menus are added in the *getModuleMenu()* method in the *IlsSfcDesignerHook* class. When either menu is selected, the action that is called creates a *ChartRunner* class and then calls the *start()* method. The *ChartRunner* class is a pretty lightweight class. The *run()* method gets the current chart name, and passes it and the isolation preference to Python. The Python function that is called is specified in the PythonCall class by the *RUN\_CHART* definition. It specifies *ils.sfc.designer.runChart.run().*

### Custom Chart Parameters

In order to take advantage of the various facilities in the ILS framework, such as Isolation mode for both tags and the database, message queues, and time factor acceleratiopn, the following parameters are added to the top level dictionary. These parameters are added when a chart is started from the SFC control panel in a client, the ILS API for starting a chart, or from Designer.

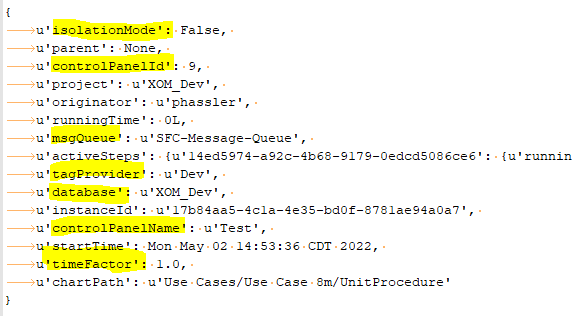


Figure 5 - Additional Chart Parameters

### Mock Call Hierarchy

In addition to the parameters inserted by ILS mentioned above, Ignition adds parent, enclosingStep and activeSteps properties into the dictionary. These properties are used by the framework to access recipe data at the appropriate scope. Use case 8m is a good test case for demonstrating these properties. The charts and call hierarchy are shown below.



Figure 6 - Test Chart

When the unit procedure is run from the control panel, the chart structure for the Operation chart is shown below:



When a chart that is not a unit procedure, i.e., is not a top level chart, is run from Designer, the runner creates a mock call hierarchy using the chart hierarchy in the database. So if the *Heat* chart is run from the Designer using the *Start Chart (production)* menu choice, the runner creates the following chart dictionary shown below. The dictionary is created by *getChartParameters()* in the Python module *ils.sfc.designer.runChart.* The designer menu is implemented in Java. The menu is added in the designer hook class: *IlsSfcDesignerHook.java* in the *com.ils.sfc.designer* package in the *sfc-designer* Java project.



Figure 7 - Mock Chart Hierarchy

## Context Sensitive Help

ILS has implemented context sensitive help in the designer. The user can right click on a block, both on a palette or on a diagram canvas, select “Help” and the appropriate documentation for the block will be opened in a browser. The full User’s Guide is also available from the Gateway Web page under Config, Modules and the selecting Documentation to the right of the module.

### User Manual Preparation

The User’s manual is written in Microsoft Word. It is saved as a word documenta and as a PDF. The User’s Guide covers many topics but section 4 contains a detailed description of each step type. Each step is documented in its own word/pdf document and then inserted into the main document. In order to update the descrption of a step follow the following procedure.

1. Open the word document for the step, make the desired change and save as a word document
2. Save as a PDF
3. Open the User’s Guide (SFCUsersGuide.docx),
   1. Find the appropriate section and delete the text but not the section header.
   2. On the Insert menu, select Object -> Text from File and select the file updated in #1.

*This inserts the text of the file, it does NOT insert a link to the source document. There is a way to insert a link, but the referenced document is limited to a single page and any page and section breaks do not work well.*

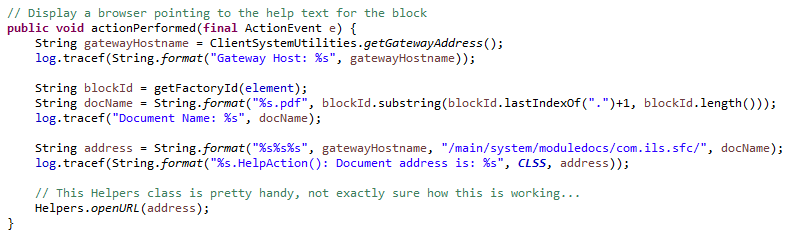
* 1. Save as a word document.

1. Save as a PDF
2. Build the SFC module. The build script copies all of the PDF files into the module.



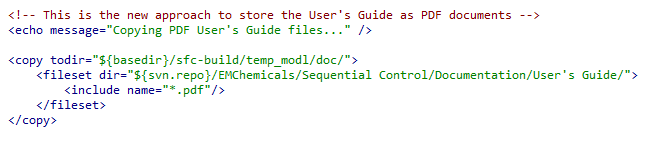
### Java Module Implementation

The core code that launches the context sensitive help is shown below. This is in class *IlsStepEditor.java* in the *com.ils.sfc.designer.stepEditor* package.



### Build Script

The portion of the ANT build script that copies the PDF files into the module is shown below:



# Substitution Expressions in Messages

In certain step types, 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 a scope, key, and attribute separated by periods, e.g. “{local.rdata.value}”. Allowable scopes are tag, chart, step, and any recipe data scope (from ils.sfc.common.constants.py). These constants should also be used when used in a script; however when used in-line in a property of a step, the legal values are:

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

*How does this work?*

# SFC KPI

The intent is to track the performance of unit procedures and operations. There are a couple of things for implementing KPI metrics for SFC unit procedures and operations that are already implemented. First is a database table *SfcRunLog*. Currently only the start of procedures and operations is logged. There is a rudimentary UI provided as of 0.1r60b8.

This enhancement is to capture the completion time and status.

## Data Collection enhancements

Several alternatives were investigated for capturing the completion time and status of unit procedures and operations. First of all, a unit procedure and operation are complete when the chart it starts is complete. Unit Procedure, Operation, and Phase all inherit from IA’s EnclosingStep. The implementation of IA’s enclosing step dictates how our steps operate. In IA’s paradigm, the enclosing step is considered done when the subchart is called. Therefore the deactivation method is called nearly immediately after the step starts. It would be ideal if this method were called just before execution passes to the step after the enclosing step. There is a cancelAction() method that is called when a chart is cancelled. So this was a dead end but is important to understand.

The approach that was used was entirely in Python utilizing the system.sfc.getRunningCharts(). The chartPath of the called subChart is available and when passed to this system procedure the status of the running instance can be monitored. The activation method of the unit procedure and operation each call monitorCalledChart() in ils.sfc.gateway.steps.commonEncapsulation. This starts an asynchronous thread that monitors the chart until a terminal state is reached. It is designed to handle multiple instances of the chart path in the data structure because a chart stays in the results for several minutes after the chart completes.

# Architecture

This section presents information intended for engineers that will be involved with enhancement of the software.

## 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, each input window has one record in SfcInput, and each Select Input window has several rows in SfcInputChoices in addition.

When implementing a new persistent window, a particular pattern should be followed. Use the SelectInput Vision window, gateway python ils.sfc.gateway.steps.commonInput.py, and ddl for SfcInput and SfcInputChoices in svn/EMChemicals/Database/createSfcTables.sql as a template. The general sequence in gateway-scope step code is to 1) create window records 2) message clients that the window has been created 3) wait for a response from the client (if appropriate) 4) delete window records.

In gateway-side step scripting it is essential that the window records be removed even if the chart is canceled or an error occurs. Arrange exception and pause handlers so this is the case, and do not use “return” statements that will bypass the record removal.

Each window has custom properties of windowId (String) and data (dataset). The data property is bound to a sql query of that window’s table, e.g. "select \* FROM SfcInputs WHERE windowId = '{Root Container.windowId}'". All bindings using column values can be accessed from expressions referencing this dataset using the dataset access syntax. Such expressions should protect against a 0-length dataset. For example, the Yes/No window text property is bound to the expression “if(len({Root Container.data}) > 0, {Root Container.data}[0,'prompt'], '')”

Never use the default datasource for SQL bindings! As noted above under Isolation Mode, the datasource should always be bound to the [Client]Database tag.

If a window contains tabular data, this data will be held in a second database table. The table or combo box component can probably be bound directly to a SQL Query.

By default, SQL bindings have a polling interval. This is unnecessary overhead for most windows with static content, so the polling mode should be set to “Off” unless there is dynamic content (e.g. in Download GUI).

Avoid referencing the “data” custom property in visionWindowOpened scripts; it may not have been read from the database yet.

Timeouts should function correctly even if no client windows are open—this means that timeout logic should be written in the gateway step code. When scripting client-side messages associated with time delays or timeouts, do NOT use the time the window opened as a reference, as the window may have opened in the middle of an operation.

A step-specific default button label should be set in the gateway step code.

## Step Implementation

This section describes how to implement a step.

### SDK model for Custom Steps

The Ignition SDK contains an example for defining a custom SFC step. Our implementation follows that model. At runtime, the main point of control is the IlsAbstractChartStep class which handles activation/deactivation/pause/resume/cancel logic and calling the Python methods for the step type.

### Java Implementation

The definition of a custom step is specified in Java. The definition includes the name, step properties, icon, and editor configuration. The easiest way to create a new step is to follow the pattern of an existing similar step. The following checklist provides an overview of what is required. This list is for the most recently added step, the “OC Alert” notification step.

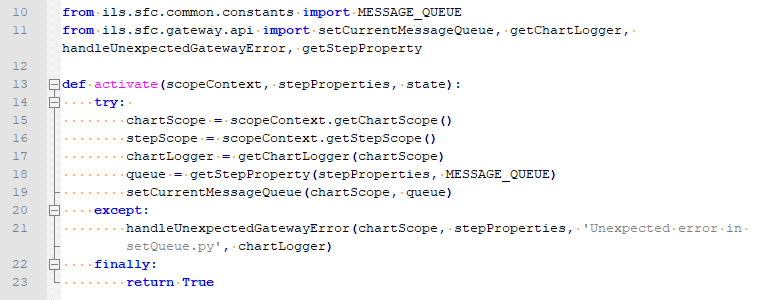
* Create a Python callback in *ils\_python:sfc.gateway.steps*
  + ocAlert.py
* In the sfc-common project:
  + In the folder *com.ils.sfc.common.step*:
    - Create new class *OcAlertStepProperties.java*
    - Create new class *OcAlertStepDelegate.java*
    - Define the new step in *AllSteps.java*
  + In folder *com.ils.sfc.common*
    - Register the callback created above in *PythonCall.java*
* In the sfc-gateway project:
  + In the folder *com.ils.sfc.step*:
    - Create new class *OcAlertStep.java*
    - Create new class *OcAlertStepFactory.java*
  + In folder *com.ils.sfc.gateway*
    - Register the factory with the gateway hook by editing *IlsSfcGatewayHook.java*
* In the sfc-client project:
  + In the folder *com.ils.sfc.client.step*
    - Define the new step in: *AbstractIlsStepUI.java*
    - Create new class *OcAlertStepUI.java*

### Python Implementation

The step specific behavior of a step is implemented in Python. Each step has a module in the ils.sfc.gateway.steps package. The main entry point is the function activate(). The SFC engine calls activate every 0.5 seconds while the block is active. There are several types of step implementations to consider.

#### Short Running Step

The Set Queue step is a typical example of a short running step. A short running step is any step that performs a task without any wait states, extended loops, or waiting for a user action.

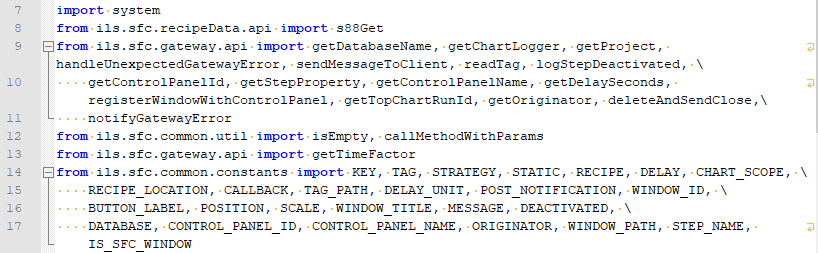


The first thing to examine is the argument signature. The scopeContext and stepProperties provide runtime information about the chart and the step. The scopeContext needs to be separated into chartScope and stepScope information as shown below. One distinction worth pointing out is that the stepProperties provides access to block specific custom properties. The step scope provides access to standard run-time information about the step and is the same for every step.

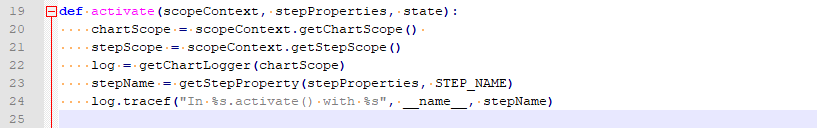
#### Long Running Step

The Time Delay is a good example of a long-running step that does not involve user interaction. As mentioned above, the SFC engine calls activate every 0.5 seconds while the block is active. This means that the block must break it’s work down into discrete chucks each of which executes very quickly. The Python method can be broken down into sections.

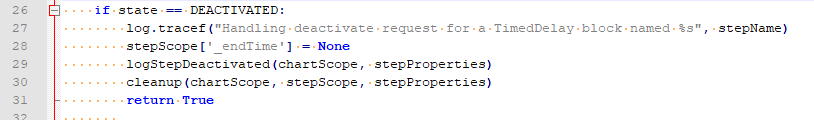
Imports:



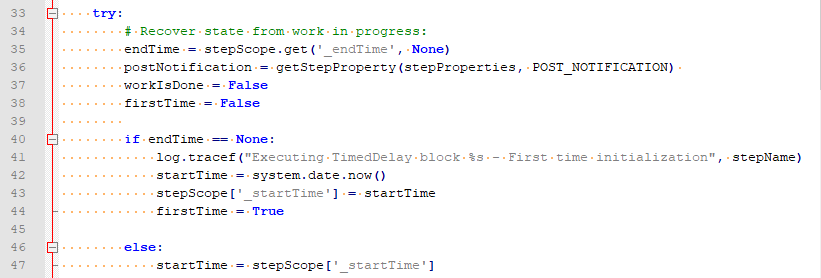
Signiture and initialization:



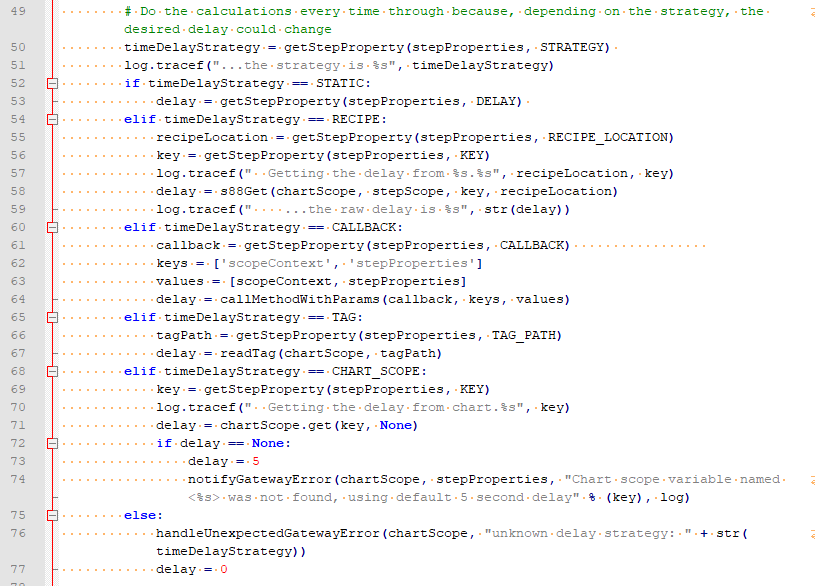
Check if an external signal has ended execution prematurely:



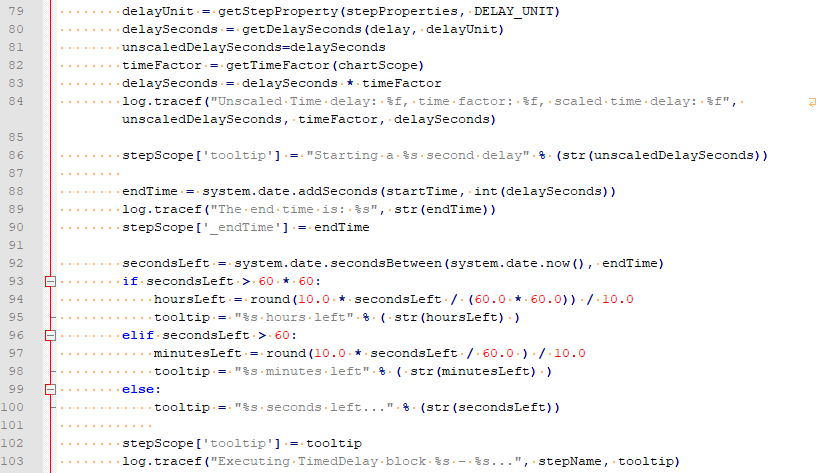
Initialization for the first time through:



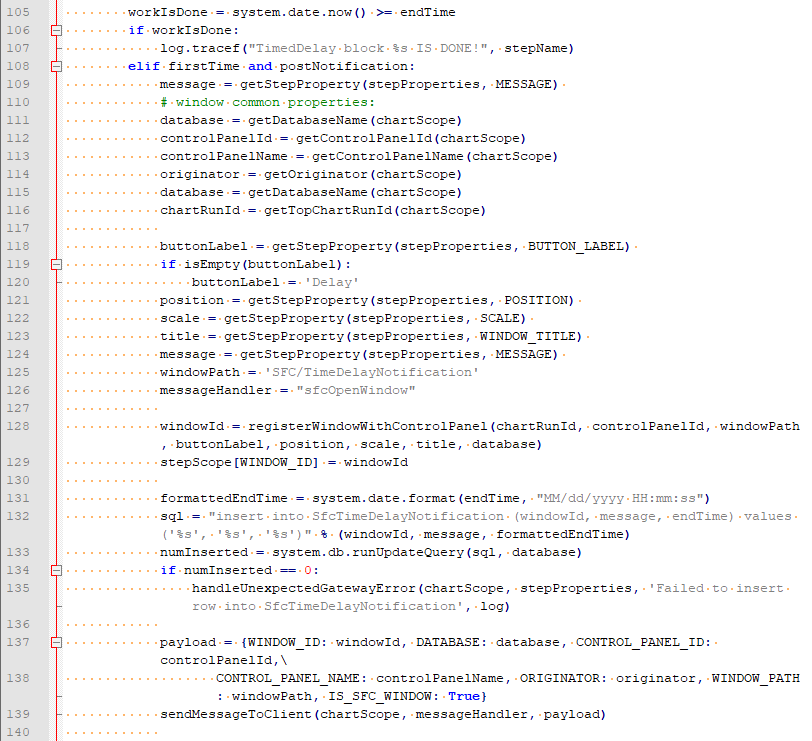
Work:



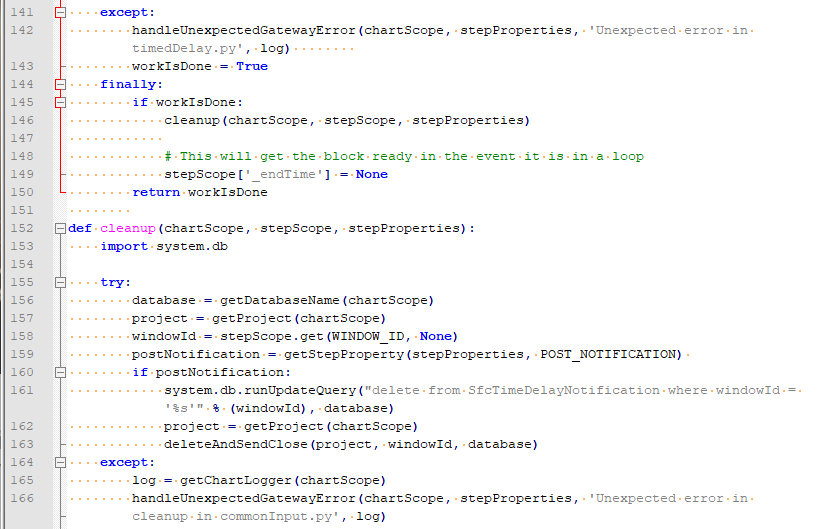
More work:



Check if work is done:



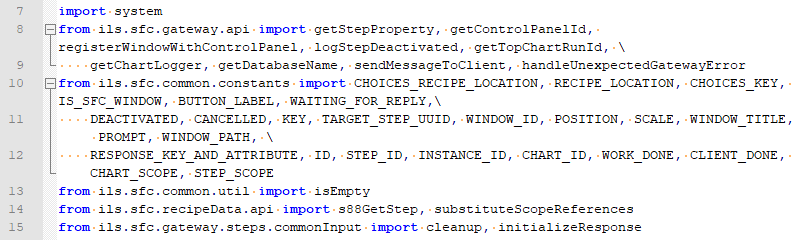
Cleanup:



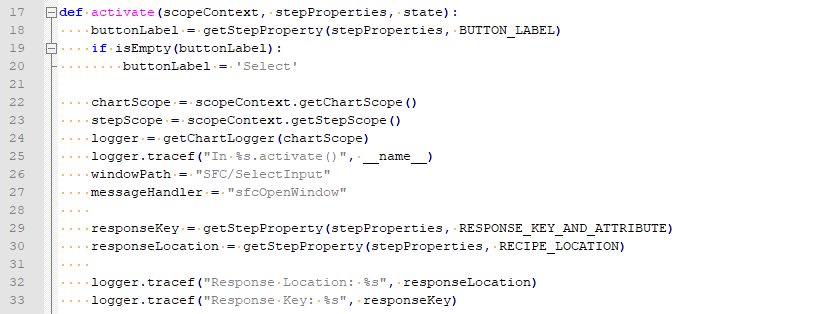
#### Long-Running Step with User Interface

There are a number of factors to consider with long-running steps that include a user interface. The step has components that run in the gateway and in a client and the messaging between the gateway and the client. In general, clients are notified to post a window via a message and the gateway is notified of a response from a client using the database as a blackboard. The Get Input steps is a good example and can be broken down into chunks:

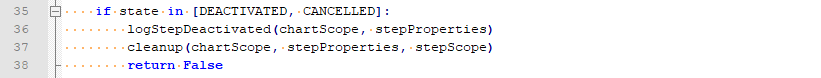
Imports:



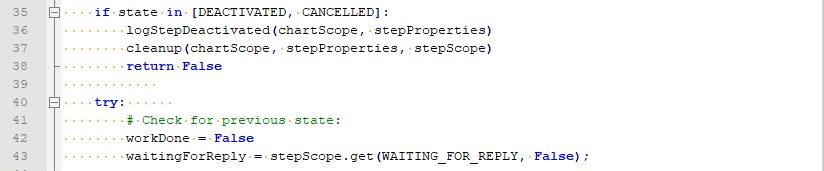
Signiture and initialization:



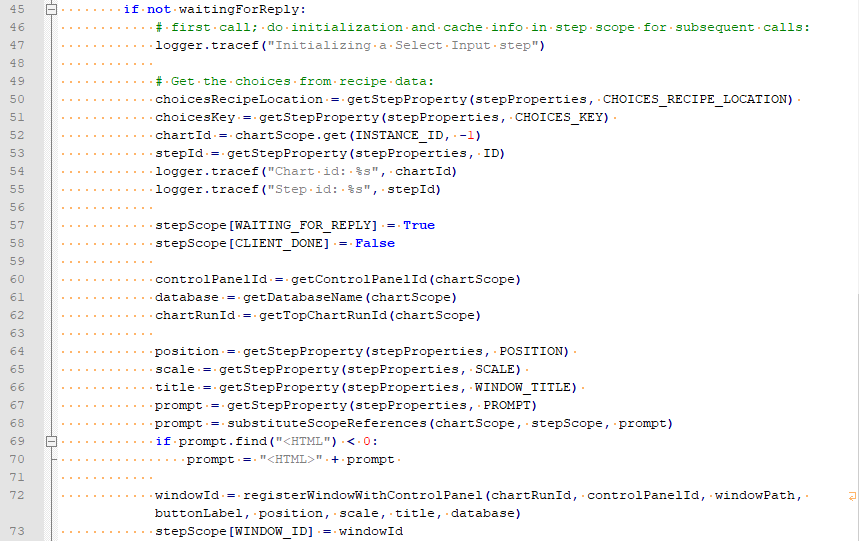
Check if an external signal has ended execution prematurely:

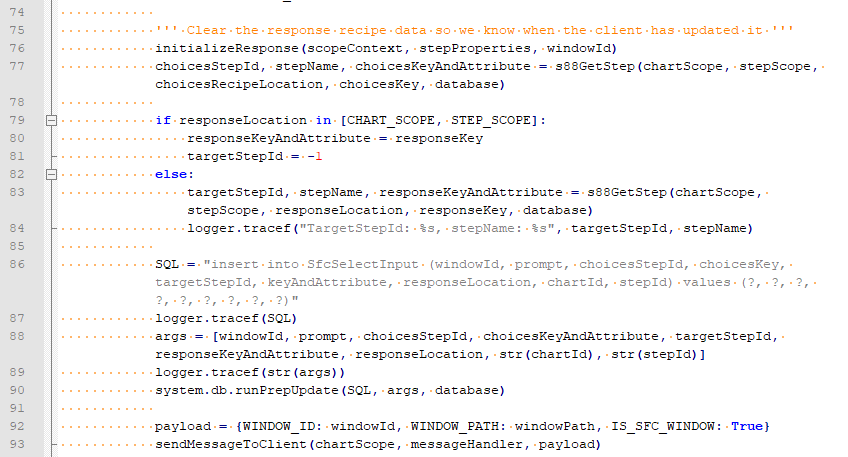


Initialization for the first time through:

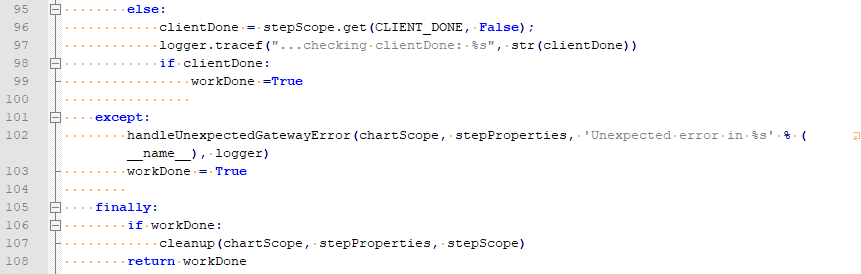


Work to set up the UI:





Wait for the response from the UI and cleanup:



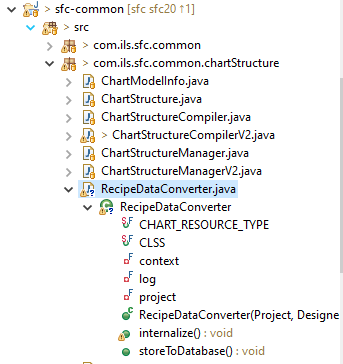
# Export / Import Support

Custom ILS steps are exported and imported without any special consideration. It is entirely handled by the Ignition gateway. However, the implementation of recipe data in the SQL\*Server database does require special consideration. It is important that the export be entirely self contained with everything that is needed to run the SFC after being imported into a remote system. Early versions of the toolkit required an export of SFCs from the Designer and an export of recipe data from the recipe data browser in a client.

As described in section 3, the reason that recipe data is stored in an external database is so that the recipe can be accessed easily from a Vision window when the charts are not running. The design has proved successful with the exception that it becomes difficult to move SFCs from one system to another.

## Recipe Data Converter Class

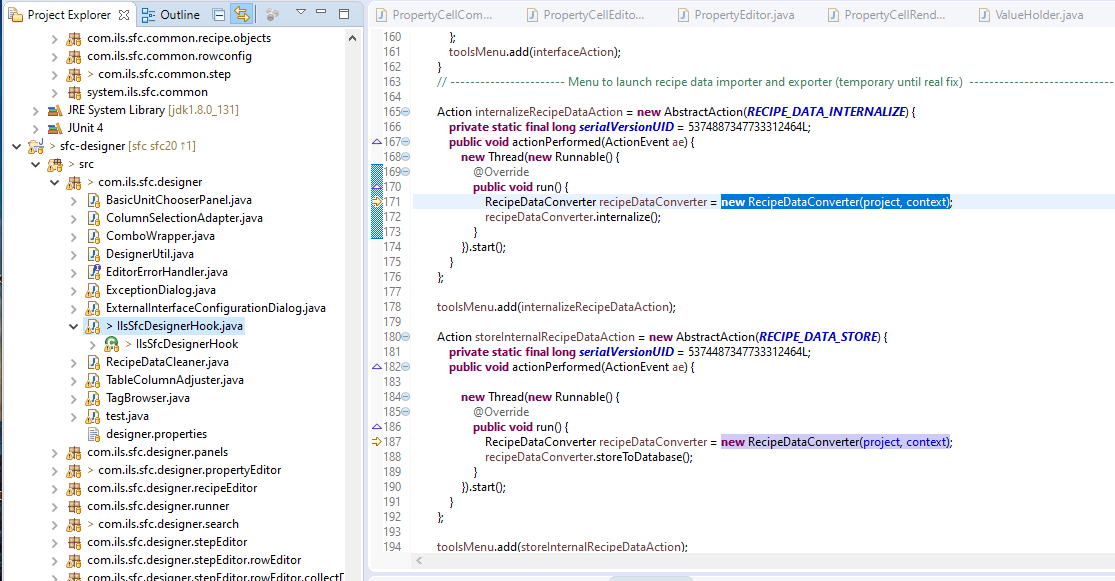
The RecipeDataConverter class provides the high level functionality for import and export support for recipe data.



### Internalize Recipe Data

The first step in exporting an SFC is to internalize the recipe data from the database into the *associated-data* property of the step. Ideally this would be embedded as late as possible, when a chart is exported, however this will require an enhancement by IA to allow a hook into our code when exporting a step. It needs to happen as late as possible because recipe data can be edited in various ways, not just through the designer. Initially this is done from a custom designer menu choice under tools: *Internalize Recipe Data For Export*. Eventually, we hope to do this from a special export hook that IA adds just for us.

The temporary menu choice is added in the IlsSfcDesignerHook as shown below:



### Storing Internalized Recipe Data Into the Database

There is a hook that calls our code when an SFC resource is saved.

There are two ways that a chart can be imported:

1. A single SFC XML chart resource can be imported by right-clicking on any SFC resource in the project tree and selecting: *Import*
2. Second a project file with multiple SFCs can be imported by File Import in the Designer.

In both cases, the resource is not saved during the import. It is loaded into the Designer. It is not saved until the user selects *File->Save*.

## Python Methods

The converter class discussed above is a pretty thin layer. Most of the work, including the actual database logic, is done in Python. The cross-reference from Menu Choice to Java to Python is shown below:

|  |  |  |
| --- | --- | --- |
| Menu Choice | Java | Python |
| Internalize Recipe Data for Export | RecipeDataConverter.intenalize() | ils.sfc.recipeData.internalize.internalize() |
| Store Internal Recipe Data into Database | RecipeDataConverter.storeToDatabase() | ils.sfc.recipeData.save.storeToDatabase() |
| Initialize Internal Recipe Data | RecipeDataConverter.intenalize() | ils.sfc.recipeData.initialize.initialize() |

class

# Testing

One of the perceived shortcomings of the Ignition