**ExxonMobil Chemical Company**

**I/O Facilities**

**Design Specification**

**Version 1.5**

**December 4, 2015**

**Prepared by:**

**ILS Automation Inc.**

REVISION HISTORY

|  |  |  |  |
| --- | --- | --- | --- |
| **Revision** | **Date** | **Initial** | **Description** |
| 1.0 | December 3, 2014 | Pete Hassler | First Draft |
| 1.1 | March 3, 2015 | Pete Hassler | Refined PKS Controller. |
| 1.2 | March 25, 2015 | Pete Hassler | Updated Controller Class Hierarchy and added Lab Data |
| 1.3 | September 25, 2015 | Pete Hassler | Updated API in section 5 |
| 1.4 | November 24, 2015 | Pete Hassler | Updated all screen shots that were out of date. Updated section 5.1.3 - Controller Functions. Updated WriteDatum to remove writeConfirm option. Added flowchart for writeDatum for a PKS controller. |
| 1.5 | December 4, 2015 | Pete Hassler | Added documentation for the PKS ACE controller. |

**Table Of Contents**

1 Introduction 5

2 Design Philosophy 5

2.1 User-Defined-Types (UDTs) 5

2.2 Python Class Definitions 6

2.3 Control Flow 7

2.4 Tag Event Script 7

2.5 Test / Debug Support 7

3 Communication to OPC Servers 9

4 I/O Classes 9

4.1 General Purpose Classes 11

4.2 Controller Classes 13

4.2.1 PKS Controller 15

4.2.1.1 WriteDatum Logic 16

4.2.1.2 ConfirmControllerMode Logic 17

4.2.2 PKS ACE Controller 17

4.3 Recipe Toolkit Classes 18

4.3.1 Recipe Details 18

4.3.2 Download Trigger 19

4.4 Lab Data Toolkit Classes 22

4.4.1 Recipe Details 22

5 API & Wrapper 22

5.1 API 22

5.1.1 General Functions 23

5.1.2 Output Functions 23

5.1.3 Controller Functions 23

5.2 Wrapper 25

# Introduction

This specification describes the I/O facilities provided by the ExxonMobil Ignition platform. These facilities are used by all of the toolkits that are part of the platform.

# Design Philosophy

The previous platform encouraged customization and specialization in a very flexible environment using true object-oriented principles. While the Ignition software was developed using object-oriented principles, when configuring a project, from the end-user’s perspective, the platform is not object oriented. ILS has devised a scheme where the IO facilities are developed and extended in an object-oriented manner using a combination of UDTs and object-oriented Python.

## User-Defined-Types (UDTs)

The purpose of the UDT is to provide a permanent tag instance. A UDT is similar to a class definition but is actually more like a template. It provides a mechanism for organizing a set of tags into a structure that can be instantiated many times. The UDTs are general in nature, and it is expected that there will be situations where not all of the tags in the UDT will be needed for the particular application. In this case the embedded tag can be disabled which will reduce unnecessary load on the OPC server. An example of this is the diagnostic toolkit that writes primarily to SP tags and does not monitor the PV or OP. In this case the generic controller UDT will be used but the PV and OP can be disabled.

The hierarchy of UDTs is shown below.

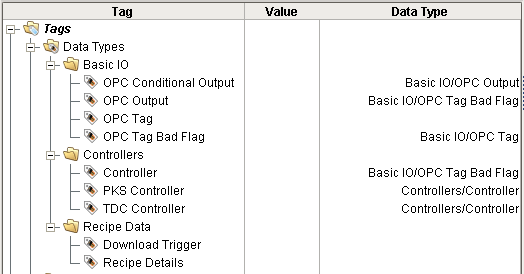


Figure 1 - I/O User Defined Types

## Python Class Definitions

The purpose of these Python classes is to implement an object-oriented class hierarchy of methods. The Python classes and methods are implemented in the *ils.io* package and will be described in detail in section 5.

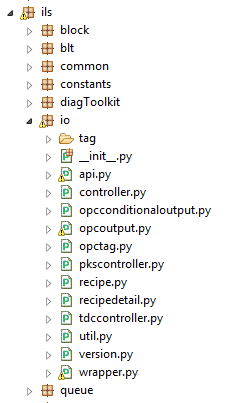


Figure 2 - Python IO Package and Modules

## Control Flow

This section describes the steps required to write to a tag. This highlights the work done in the client versus the work done in the gateway. The gateway work is triggered by the event script on the command tag in the UDT. This sequence shows the logic for an OPC Output and does not include the steps if the client waits for write confirmation. This is the simplest of the write sequences.



## Tag Event Script

The UDT defines a script that runs when the command field of the UDT receives a value. This script is conveniently defined on the UDT definition, not on each instance. This is a new feature in Ignition 7.7. Refer to Figure 8 for more details.

## Test / Debug Support

The I/O layer provides test/debug support by preventing I/O writes at a very low level via a single setting. This allows the various toolkits to be run in a development mode while communicating with the production DCS and ensuring that the toolkit will not write to the production DCS. When writes are disabled, writes will fail, which may cause the operation to fail, but it still allows the toolkit to run. In order for the system to write to OPC, the *writeEnabled* memory tag shown below must be *True*. This can be set and cleared via the Designer. It is recommended that this flag is NOT set or cleared programmatically.

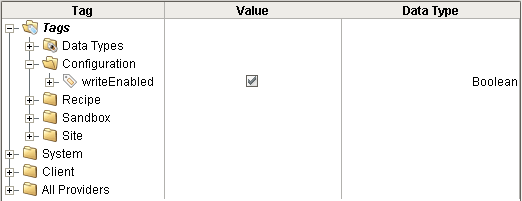
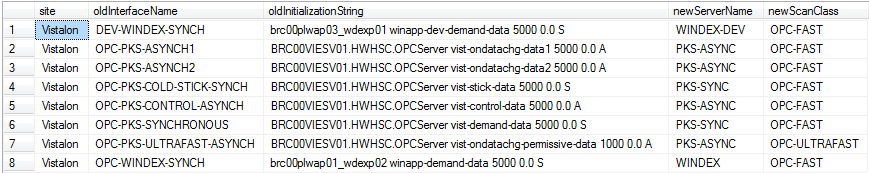


Figure 3 - OPC Write Enabled Flag

# Communication to OPC Servers

All of the communication to OPC servers will use the DA protocol. There needs to be a mapping from the old gsi-interface objects to the OPC server / scan class. In G2 a new interface object was needed for each synchronous/asynchronous and update rate combination. In Ignition, a server connection needs to be defined for asynchronous and synchronous and then a scan class for each different update rate. The following table shows the translation from G2 to Ignition.



# I/O Classes

This section describes the various classes that have been defined. There is a parallel hierarchy of Ignition UDTs and Python classes. The UDT hierarchy uses Object-Oriented like inheritance but should not be considered a pure Object-Oriented design. There is a parallel hierarchy implemented in Python that is a pure Object-Oriented design.



Figure 4 - I/O Class Hierarchy

Note: In the old Vistalon application, the output names do not say SP, which makes me think that they are PVs, but they aren’t! If there isn’t a suffix, it is assumed to be a SP.

## General Purpose Classes

This section describes the general purpose I/O classes.

The UDT for the “OPC Tag” base type is shown below. This class defines basic capabilities of an OPC tag. It is generally used for read only tags although it can be used for writing since OPC tags are inherently bi-directional. It includes an embedded OPC tag and an embedded memory tag that specifies the name of the Python class that will be used for method dispatching.

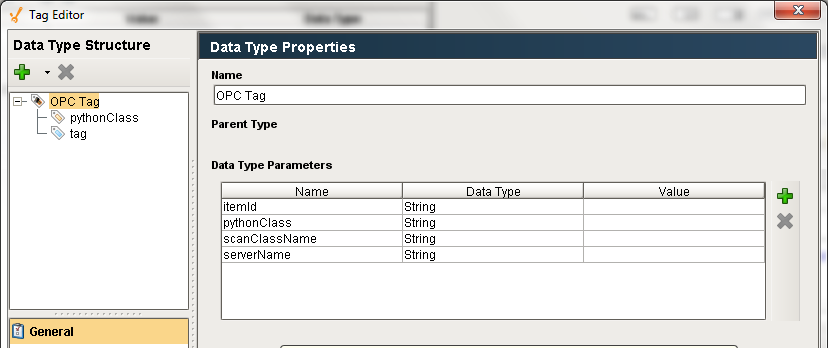


Figure 5 - OPC Tag UDT

The UDT for an “OPC Tag Bad Flag” is shown below. Its parent is “OPC Tag”. This type adds a memory tag: *badValue*. Currently there are no methods defined on this class nor is there any logic anywhere that manipulates the badValue flag.

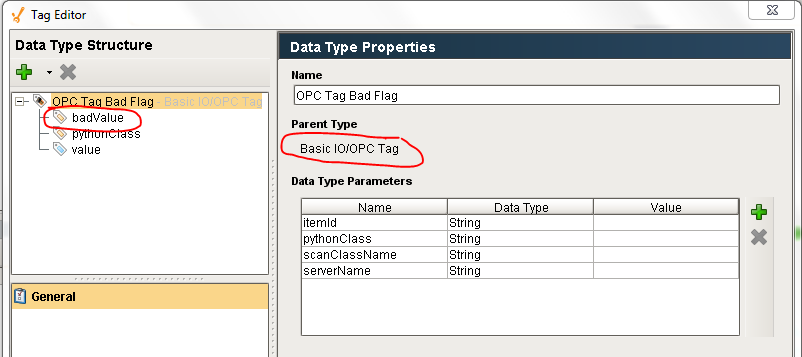


Figure 6 - OPC Tag Bad Flag UDT

The UDT for an “OPC Output” is shown below. Its parent is “OPC Tag Bad Flag”. This type adds several memory tags: command, writeConfirmed, writeErrorMessage, writeStatus, and writeVal. This class is intended to support OPC tags that are written to.

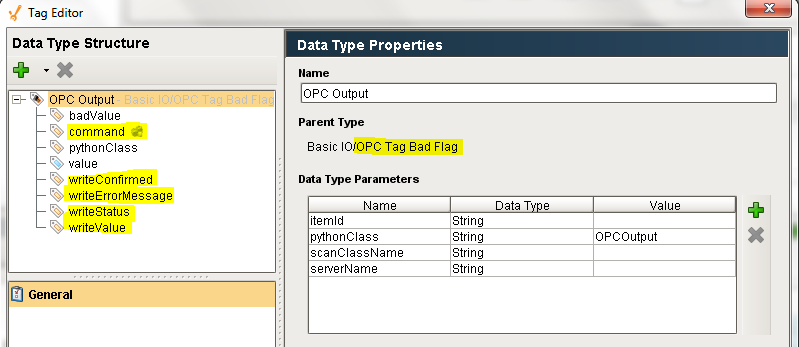


Figure 7 - OPC Output UDT

There is an event script for the *command* tag which is shown below:

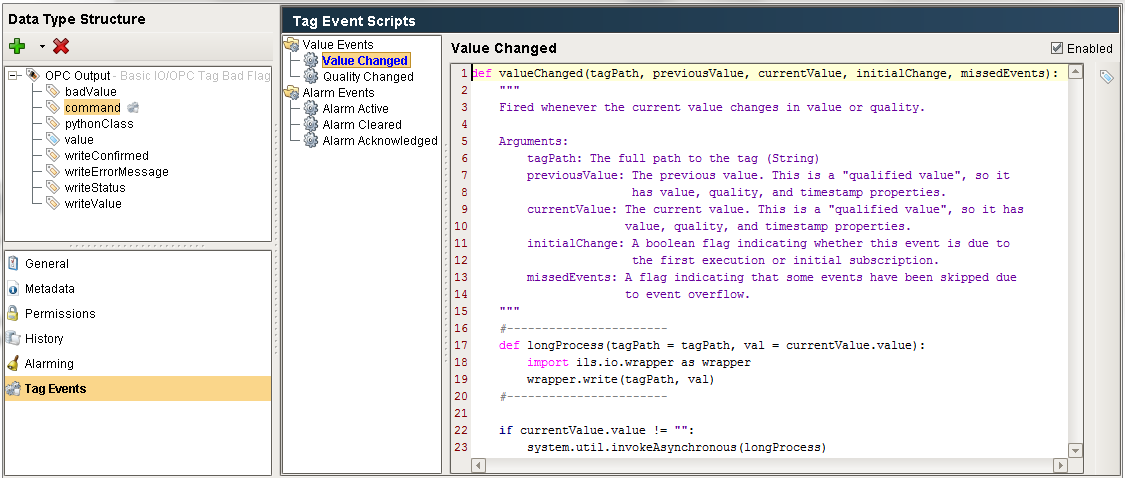


Figure 8 - Command Event Trigger

The UDT for an “OPC Conditional Output”, which is derived from an “OPC Output” is shown below. It adds a *permissive* OPC tag and several memory tags: *writePermissiveAsFound, permissiveConfirmation, permissiveValue*. To the tags already provided by the “OPC Output” UDT. This UDT is generally used when the OPC tag is actually an attribute of a controller in the DCS and when the controller needs to be in a specific mode before the tag can be written. The permissive is a generic term for what is generally the mode of the controller. The following memory tags are generally statically configured at design time and then are not changed:

* *permissiveValue*: the mode the controller needs to be in to accept the write
* *permissiveConfirmation*: specifies if the write process needs to confirm the permissive write before writing the value

Once configured, a new value can be written to a conditional output by doing the following:

* Wrting the value to be written to the writeValue memory tag
* Writing “WriteDatum” to the command tag.

The gateway implements the write process in the following steps:

* Reads the current permissive and stores it in *permissiveAsFound*
* Writing *permissiveValue* to the *permissive*
* Writing *writeValue* to the value
* Writing *permissiveAsFound* to the *permissive*

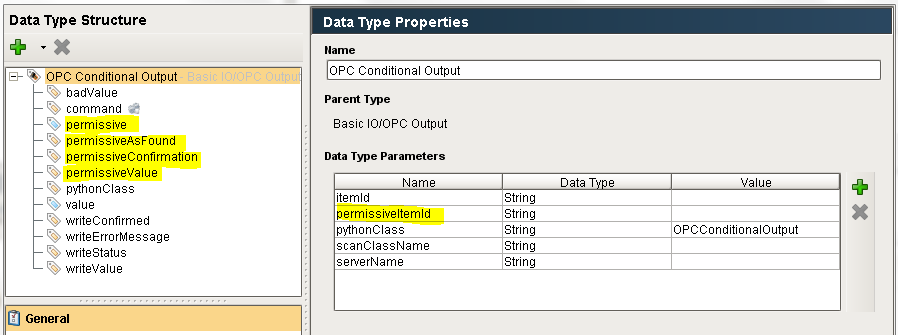


Figure 9 - OPC Conditional Output UDT

## Controller Classes

The UDTs that implement the controller I/O are described in this section. A major departure from the previous system is a standard generic implementation of a controller for both the EPKS and the TDC3000 system. The current implementation uses a different approach for each system. The generic controller classes that are described below may contain more tags than are needed for a specific application. Individual tags that are not needed may be disabled in order to reduce unnecessary overhead on the OPC server.

The basic controller defines OPC tags for the PV (named *value* and inherited from OPC Tag Bad Flag), mode, OP, SP and Output Disposability. The mode, OP, and SP are all embedded OPC Output UDTs because these tags support writing with confirmation. The UDT provides individual properties for the each embedded OPC tag in the controller, as shown below.

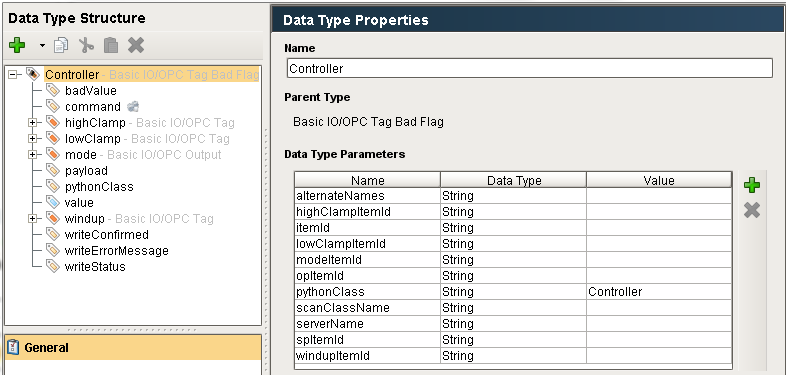


Figure 10 - Controller UDT

### PKS Controller

The UDT for a PKS controller is shown below. It adds several additional tags: highClamp, lowClamp, and windup. It also initializes the *pythonClass* property to “PKSController”.

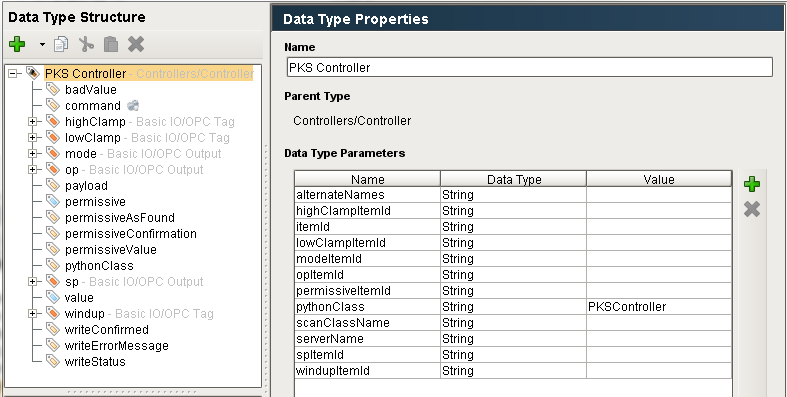


Figure 11 - PKS Controller UDT

#### WriteDatum Logic

The writeDatum() logic for the PKS controller is shown below.



#### ConfirmControllerMode Logic

The confirmControllerMode() logic for the PKS controller is shown below.



### PKS ACE Controller

APKS ACE controller is an Experion controller that exists in an Application Controller Environment. The UDT for a PKS ACE controller is shown below. It adds two attributes: *processingCommand* and *processingCommandWait*. It also initializes the *pythonClass* property to “PKSACEController”. These two properties are used in the WriteDatum method which first calls the superior method, inherited from PKS controller, then reads the memory tag *processingCommandWait value* and writes it to the OPC tag  *tagprocessingCommand/value*.

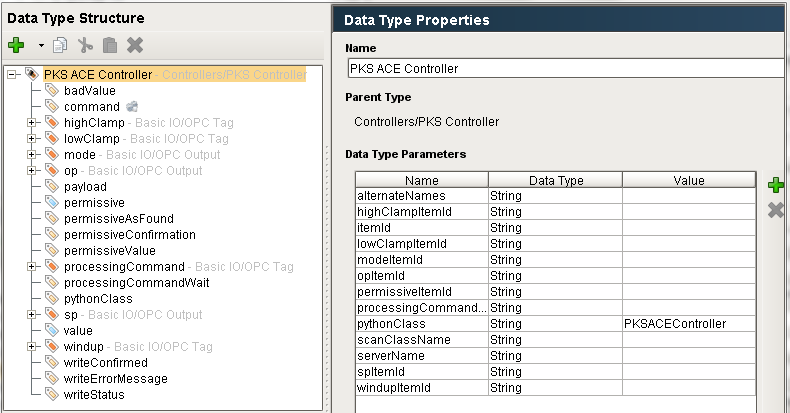


Figure 12 - PKS ACE Controller UDT

## Recipe Toolkit Classes

The Recipe Toolkit uses the classes described above. In addition it defines two classes: *Recipe Details* and *Download Trigger*.

### Recipe Details

This UDT is used to coordinate writes to a controller in the DCS when there are high and/or low limits and an associated setpoint where the writes need to be coordinated in order to not violate the limits. The recipe toolkit creates instances of the UDT as needed based on the configuration in the recipe database.

The UDT definition is shown below:

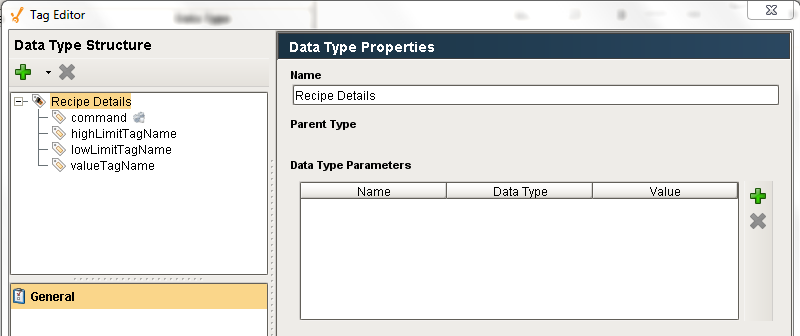


Figure 13 - Recipe Details UDT

The UDT defines a script that will be called whenever the command of the tag receives a value:

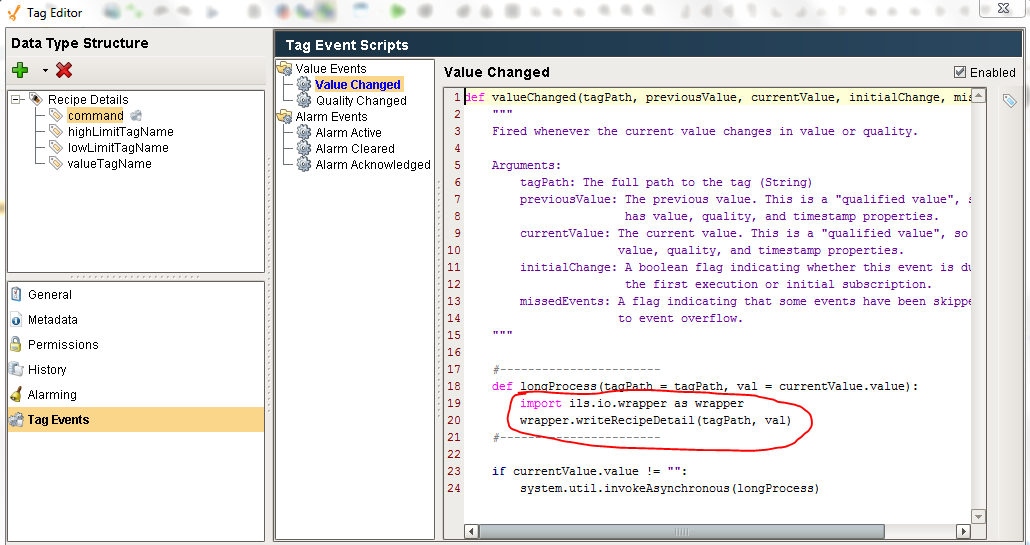


Figure 14 - Recipe Details Command Processing

### Download Trigger

The Download Trigger UDT is used to implement an automated download for a unit. These instances must be manually instantiated and configured using the Ignition Designer0.

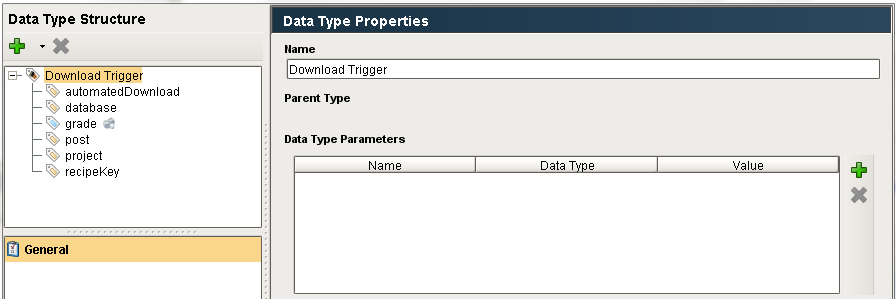


Figure 15 - Download Trigger UDT

The memory tags are:

|  |  |
| --- | --- |
| Tag | Description |
| automatedDownload | If True, then the download will be completely automatic without a user interface. If False, then the Recipe Download will be automatically displayed on the console and wait for the operator to initiate the actual download. |
| database | Because an automated download runs in the gateway, it does not have a default database. |
| Grade | An OPC tag that triggers the download, this must be set by something in the DCS. |
| Post | The post that will monitor the download. |
| Project | Because an automated download runs in the gateway, it does not have a project. |
| recipeKey | The recipeKey / unit that determines the type of recipe. Examples: RLA3, VFU, RLA3-Tower. |

The automated processing is implemented by the following trigger:

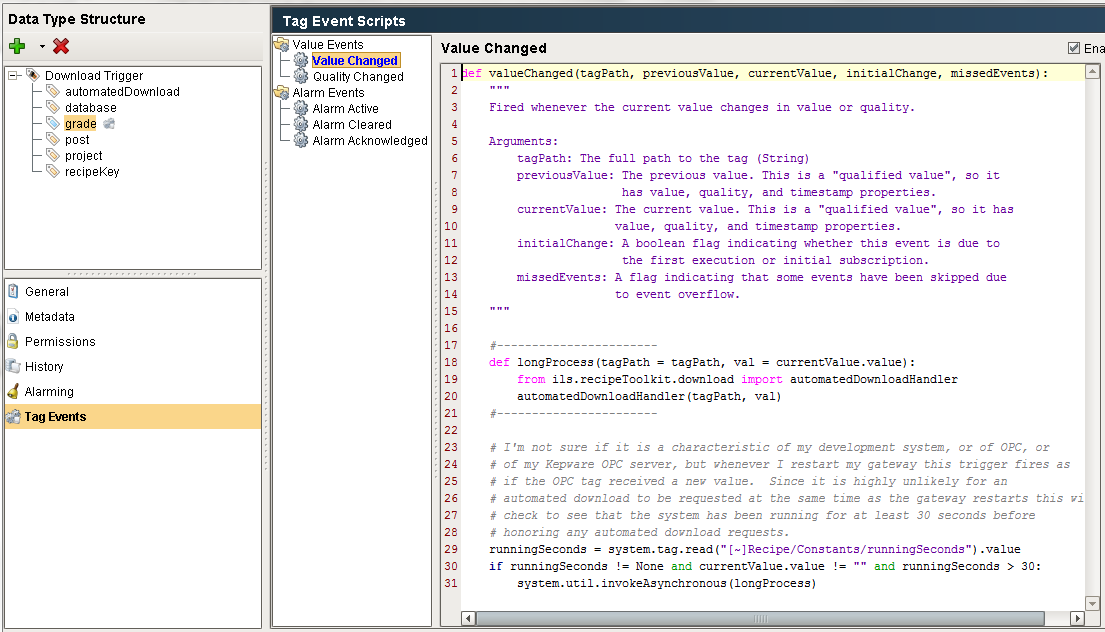


Figure 16 - Download Trigger Event

The three Vistalon recipe triggers used during development are shown below

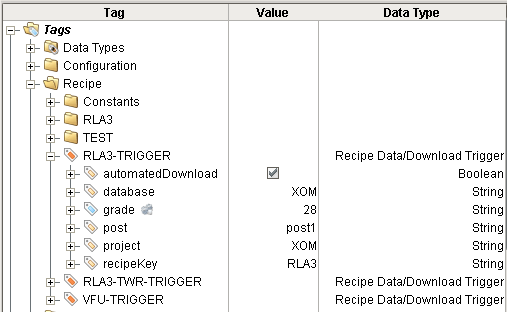


Figure 17 - Automated Download Triggers Used for Testing

## Lab Data Toolkit Classes

The Lab Data Toolkit defines several new classes: *XXX* and *XXX*.

### Recipe Details

This UDT is used to coordinate writes to a controller in the DCS when there are high and/or low limits and an associated setpoint where the writes need to be coordinated in order to not violate the limits. The recipe toolkit creates instances of the UDT as needed based on the configuration in the recipe database.

# API & Wrapper

There is a module that provides an Application Programmer’s Interface (API) around the UDT instances that make it convenient to write to them from custom Python callbacks. There is also a wrapper module that provides an interface to the commands that are written to the command tag of a UDT and called by event scripts.

## API

The following scripts are available as an interface to the I/O module to be called from Python scripts. All of the functions described in this section are provided in the external Python module *ils.io.api*. An example of how to call one of the API functions from a pushbutton is:

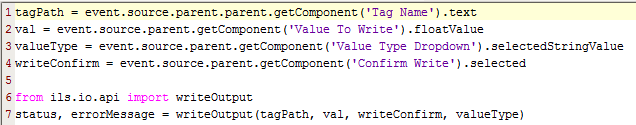


Figure 18 - I/O API Example

### General Functions

These functions apply to both outputs and controllers

**reset**(tagname)

Reset all of the memory tags that are part of the UDT in an OPC output or controller. This should be called by the client before performing a write.

|  |  |
| --- | --- |
| Argument | Description |
| tagname | Full path to the UDT being reset. |

### Output Functions

These functions apply to OPC outputs and its subclasses

**writeDatum**(tagPath, val)

Write a value to an OPC or a memory tag and confirm that the write was successful by reading the value back. The actual write will be performed in the gateway. Depending on the scan class scan rates, the confirmation may take several minutes, because the client is single threaded, this will hang the client.

|  |  |
| --- | --- |
| Argument | Description |
| tagPath | Path to the UDT, not to the OPC tag which is a member of the UDT. |
| val | Value to be written |

**writeWithNoCheck**(tagPath, val)

This function writes a value to an OPC tag without doing any confirmation. It does verify that the OPC write was successful and perform checks to determine if it is ok to write but it does not confirm that the write by reading back the value as is done by WriteDatum.

|  |  |
| --- | --- |
| Argument | Description |
| tagPath | Path to the UDT, not to the OPC tag which is a member of the UDT. |
| val | Value to be written |

### Controller Functions

These functions apply to Controllers and its subclasses

**confirmControllerMode**(controllerTagpath, val, testForZero, checkPathToValve, valueType)

***Not Yet Implemented!*** Checks if a controller is in a mode where a write will succeed. This is often called for each controller when there are a number of controllers to be written to BEFORE any writes are performed. If any controller is not writeable, then no writes are performed.

*Question: If this is operating at the controller level, why do we need to specify the valueType? Also, if checkPathToValve is False then what does this do (perhaps the code has the answer)and explain why this needs the value?*

|  |  |
| --- | --- |
| Argument | Description |
| controllerTagname | Full path to the controller |
| Val | Value to write |
| testForZero | True or False. I don’t really understand this. |
| checkPathToValve | True or False. If True then check if the controller is in a mode to accept the write. |
| valueType | The attribute of the controller to be written to: OP, SP, or mode. |

**writeDatum**(controllerTagpath, val, valueType)

Write a value to one of the attributes of the controller (OP, SP, or mode), which is itself a UDT embedded inside of the controller UDT. It utilizes all of the permissive handling that is implemented at the controller level. The write is always confirmed.

| Argument | Description |
| --- | --- |
| controllerTagpath | Full path to the controller |
| Val | The value to be written. |
| valueType | The attribute of the controller to be written to: OP, SP, or mode. |

**writeRamp**(controllerTagpath, val, testForZero, checkPathToValue, valType)

***Not Yet Implemented!*** This allows for a big step change to be broken up into smaller steps over a specified amount of time to provide a smooth transition. The starting value from the ramp is the current value. Either the SP or OP of a controller can be the target.

|  |  |
| --- | --- |
| Argument | Description |
| controllerTagpath | Full path to the controller |
| val | The value to be written. |
| rampTime | The length of the ramp in seconds |
| updateFrequency | The step size of the ramp in seconds |
| valType | The attribute of the controller to be written to: OP or SP. |

## Wrapper

The wrapper module provides code that implements the I/O infrastructure when the UDT tags are directly manipulating without using the API. These functions are automatically called by tag change scripts on the *command* tag in the UDT. Another way of looking at is that, in general, the API runs in the client and the wrapper runs in the gateway. There are two wrapper functions: *write*, which is for OPC outputs and controllers, and *writeRecipeDetail*, which is for recipe details. These functions could conceivably be called from any Python script but that should be avoided and the API used as the programmatic interface.

**write**(tagPath, command)

This is used for manipulating OPC outputs and controllers. The following commands can be written to the command tag.

| Command | Description |
| --- | --- |
| WriteDatum | This applies to OPC Outputs. Before using this command, it is recommended that the UDT be reset by writing Reset, which is described below. It is also important to write the value to be written to the OPC tag to the *writeVal* memory tag. Regardless of whether the client cares about the results of the write, this will confirm the write by doing a read of the tag. If the client does not care about confirming the write, then it may be more appropriate to use WriteWithNoCheck. |
| WriteWithNoCheck | This applies to OPC Outputs. Same as WriteDatum except a write confirmation is not performed. |
| WriteRamp | This only applies to controllers. This command requires that the payload tag have a valid dictionary of ramp parameters, stored as a text string. |
| Reset | Reset the memory tags in the UDT including clearing the command tag. For a controller, this will also reset all of the embedded OPC Outputs |

**writeRecipeDetail**(tagPath, command)

This is called when any value other than an empty string is written to the command tag of a recipe detail UDT. It will invoke the *writeRecipeDetailmethod* for a RecipeDetail, which manages the order of writes to separate OPC outputs that are actually attributes of the same controller in the DCS.