

# **ExxonMobil Chemical Company**

## **I/O Facilities Design Specification**

**Version 1.10**

**April 19, 2021**

**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1.4	November 24, 2015	Pete Hassler	Updated all screen shots that were out of date.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 5.3 - 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.
1.6	December 20, 2016	Pete Hassler	Modified to reflect removal of wrapper layer, interaction is now strictly through API without using tag change scripts.
1.7	June 5, 2017	Pete Hassler	Added section on tag history.
1.8	August 19, 2018	Pete Hassler	Added a reference to the Common Facilities Manual for the Grade UDT.
1.9	October 25, 2018	Pete Hassler	Added support for G-Line/TDC controllers (AM, AutoMan, Digital)
1.10	April, 19, 2021	Pete Hassler	General updates and added class OPC Mode Output.

## **Table of Contents**

1	Introduction .....	5
2	Design Philosophy .....	5
2.1	User-Defined-Types (UDTs).....	5
2.2	Python Class Definitions .....	7
2.3	Test / Debug Support .....	8
2.4	NaN Support .....	9
2.5	Tag History Support.....	9
3	Communication to OPC Servers .....	10
4	I/O Classes .....	11
4.1	General Purpose / Foundation Classes.....	12
4.1.1	OPC Tag .....	12
4.1.2	OPC Tag Bad Flag .....	13
4.1.3	OPC Output .....	13
4.1.4	OPC Conditional Output .....	14
4.1.5	OPC Mode Output .....	16
4.1.6	Grade UDT .....	17
4.2	Controller Classes .....	17
4.2.1	Foundation Classes.....	18
4.2.1.1	Controller.....	18
4.2.1.2	Ramp Controller.....	19
4.2.2	PKS Classes .....	20
4.2.2.1	PKS Controller.....	21
4.2.2.1.1	WriteDatum Logic .....	22
4.2.2.1.2	ConfirmControllerMode Logic.....	23
4.2.2.2	PKS ACE Controller .....	24
4.2.2.3	PKS Ramp Controller .....	25
4.2.2.3.1	WriteRamp Logic .....	25
4.2.2.4	PKS ACE Ramp Controller .....	27

4.2.2.5	PKS Digital Controller .....	28
4.2.3	TDC Classes .....	29
4.2.3.1	TDC AUTOMAN Controller Support .....	29
4.2.3.2	Obsolete TDC Classes.....	29
4.3	Data Transfer Classes.....	31
4.3.1	Data Transfer .....	31
4.3.2	Data Transfer with Count .....	32
4.3.3	Data Transfer with Time .....	32
4.3.4	HDA .....	33
4.3.5	HDA Transfer .....	33
4.3.6	HDA Transfer Simple .....	33
4.3.7	HDA Transfer Driven.....	34
5	API.....	35
5.1	General Functions .....	36
5.2	Output Functions .....	36
5.3	Controller Functions .....	37
5.4	Recipe Toolkit Extensions.....	37
5.5	I/O API Summary .....	38
6	Isolation Support .....	38
6.1	Isolation UDTs .....	38
6.2	Generating Isolation UDT .....	39
6.3	Replicating Tags .....	41

## 1 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.

## 2 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.

The original implementation of the IO facilities performed all of the work in the gateway and utilized a tag change script on a "command" tag in the UDT. Over the fullness of time the advantage of this has become lost and the complexity seems to outweigh the possible benefit. Therefore, the IO facilities support a single API layer that is the only way to interact with the UDTs. The API works equally well from the client or the gateway. Keep in mind that writing to attributes of a controller generally requires several steps and may take several minutes to change the mode of the controller, perform the write, confirm the write and return the controller to the appropriate state.

### 2.1 *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.

Tag	Value	Data Type
<b>Tags</b>		
Data Types		
Basic IO		
Data Transfer		
Data Transfer Tag to HDA		
Data Transfer With Count		Basic IO/Data Transfer
Data Transfer With Time		Basic IO/Data Transfer
HDA		
HDA Transfer		
HDA Transfer Driven		Basic IO/HDA Transfer
HDA Transfer Simple		Basic IO/HDA Transfer
OPC Conditional Output		Basic IO/OPC Output
OPC Mode Output		Basic IO/OPC Output
OPC Output		Basic IO/OPC Tag Bad Flag
OPC Tag		
OPC Tag Bad Flag		Basic IO/OPC Tag
Batch Tracking		
Controllers		
Controller		Basic IO/OPC Tag Bad Flag
HPM Controller		
PKS ACE Controller		Controllers/PKS Controller
PKS Controller		Controllers/Controller
PKS Digital Controller		Basic IO/OPC Tag Bad Flag
TDC AM Controller		Controllers/TDC Controller
TDC AutoMan Controller		
TDC Controller		Controllers/Controller
TDC Digital Controller		Basic IO/OPC Tag Bad Flag
Escorez		
GLine		
Halobutyl		
Lab Bias		
Lab Data		
Misc		
Ramp Controllers		
PKS ACE Ramp Controller		Ramp Controllers/PKS Ramp Controller
PKS Ramp Controller		Ramp Controllers/Ramp Controller
PKS Ramp Setpoint		Basic IO/OPC Output
Ramp Controller		Basic IO/OPC Tag Bad Flag
TDC Ramp Controller		Ramp Controllers/Ramp Controller
TDC Ramp Setpoint		Basic IO/OPC Output

**Figure 1 - I/O User Defined Types**

## 2.2 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.

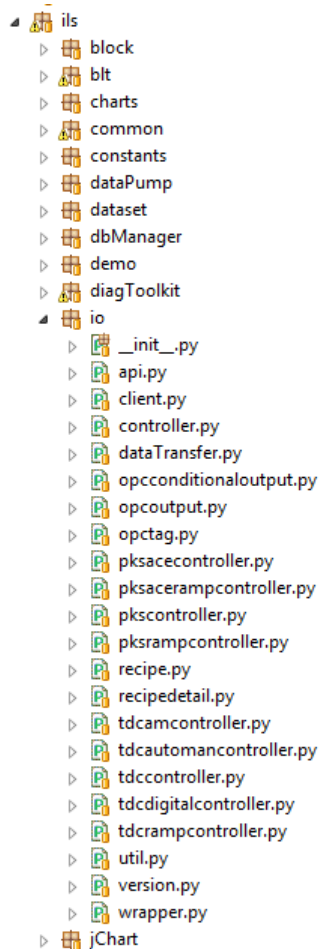


Figure 2 - Python IO Package and Modules

## 2.3 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.

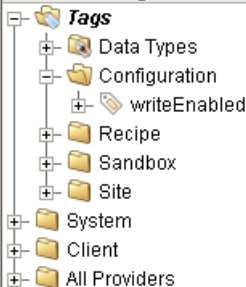
Tag	Value	Data Type
	<input checked="" type="checkbox"/>	Boolean

Figure 3 - OPC Write Enabled Flag



## 2.4 NaN Support

Support for the special value NaN requires careful consideration. Writing NaN requires that the value be coerced using the float("NaN").

Note: It is critical that the datatype of the tag is correctly matched with the datatype in the OPC server. During testing, it was common to define the OPC tags as float in the OPC server and as float8 in Ignition. This was a mismatch because float in the Kepware OPC server corresponds with float4 in Ignition changing the datatype in Kepware to double resolved the issue.

Note: Support for NaN may be dependent on the OPC server. The use of float("NaN") works as desired with Kepware as the OPC server.

## 2.5 Tag History Support

The I/O infrastructure takes full advantage of the built in historian in Ignition. In general, all tags are created without history and then history is enabled whenever needed. There are several reasons for enabling history: viewing a chart showing a trend of tag values, obtaining an average value over a period of time, obtaining a specific value at some time in the past. The recommended settings are shown below:

**Tag History**

Store history for this tag: ☐ No ☒ Yes ●

**History Provider**  
XOMhistory ●

**Historical Scanclass**  
OPC-Fast ●

**Historical Deadband**  
0.0 ●

**Max time between records**  
☒ Unlimited ☐ 1 ● Executions ●

**Historical Deadband Mode**  
Absolute ●

**Timestamp Source**  
System ●

**Value Mode**  
Analog ●

There are several things to note:

- The “Historical Scanclass” should be the same as the tag scan class. If it is slower then values may be missed.
- The “Historical Deadband” is 0.0 so that any change will be logged.
- The “Max Time Between Records” is unlimited so that the storage is optimized when the value does not change.
- The “History Provider” is XOMhistory. (This may be changed when isolation tags are created.)

### **3 Communication to OPC Servers**

All of the communication to OPC servers will use the DA protocol. A server connection needs to be defined for asynchronous and synchronous and then a scan class for each different update rate.

## 4 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.

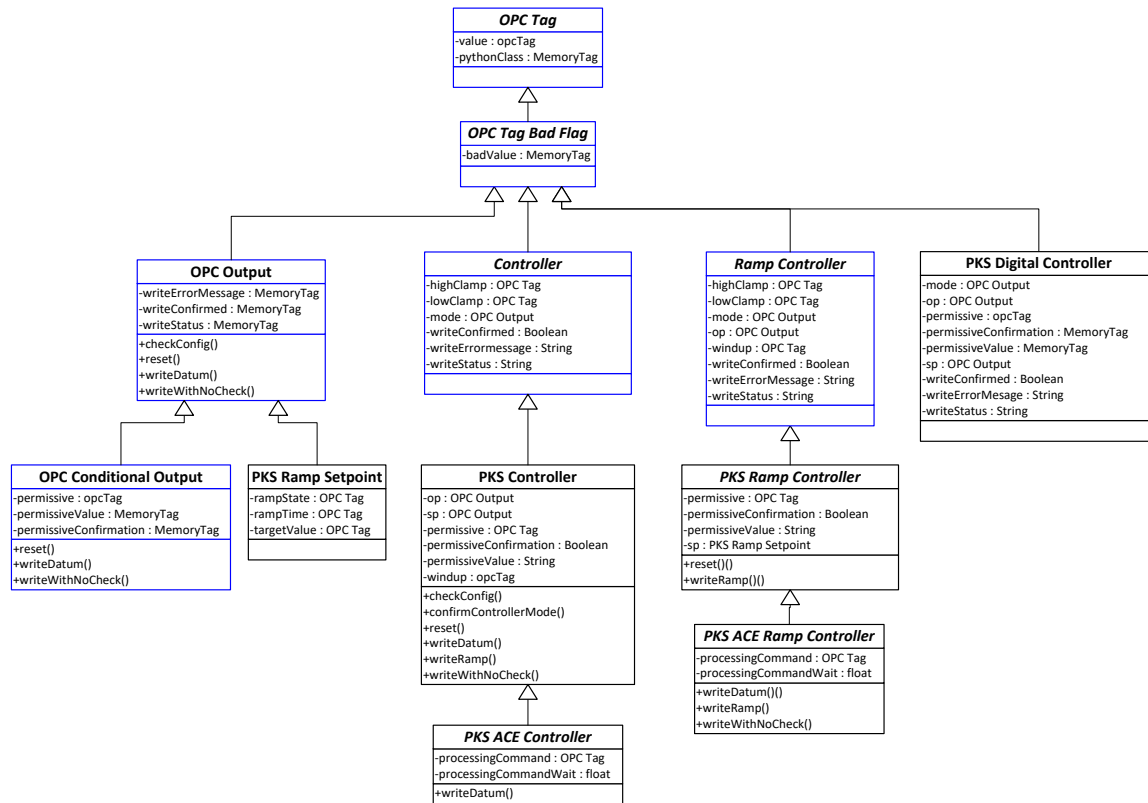
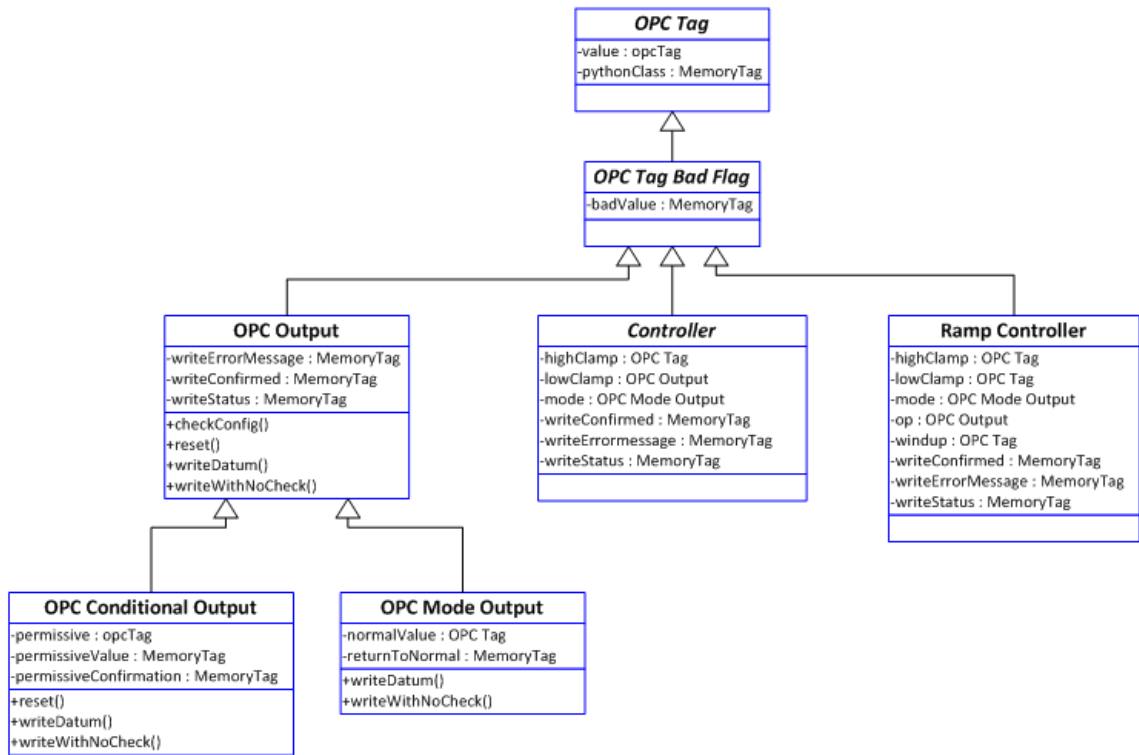


Figure 4 – PKS I/O Class Hierarchy

4.1 General Purpose / Foundation Classes

This section describes the general purpose I/O classes. The class hierarchy is shown below:



4.1.1 OPC Tag

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.

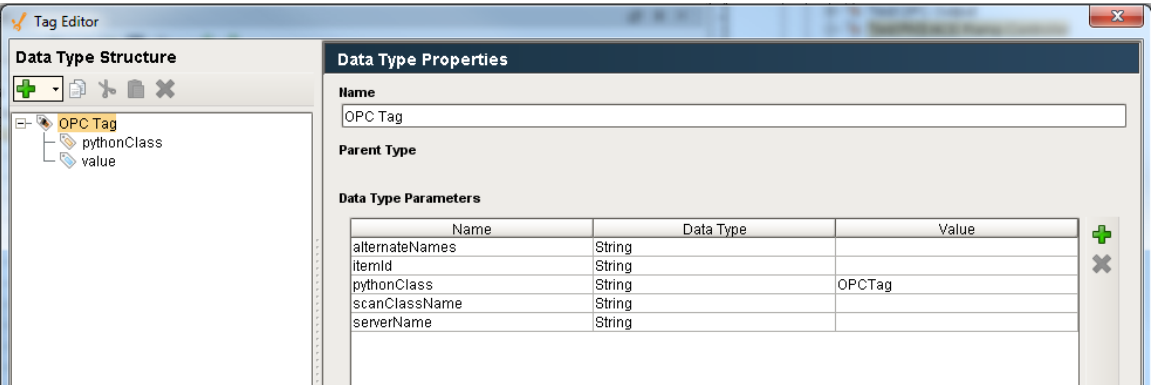


Figure 5 - OPC Tag UDT

4.1.2 OPC Tag Bad Flag

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.

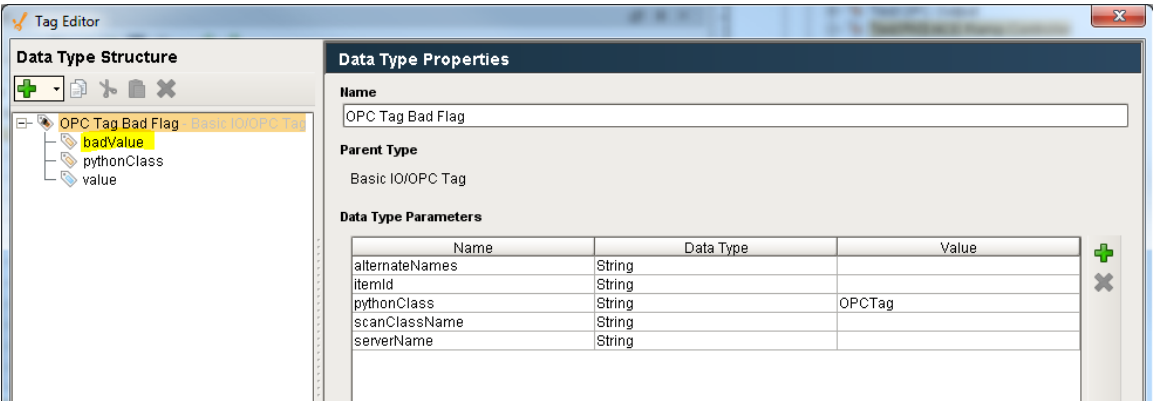


Figure 6 - OPC Tag Bad Flag UDT

4.1.3 OPC Output

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

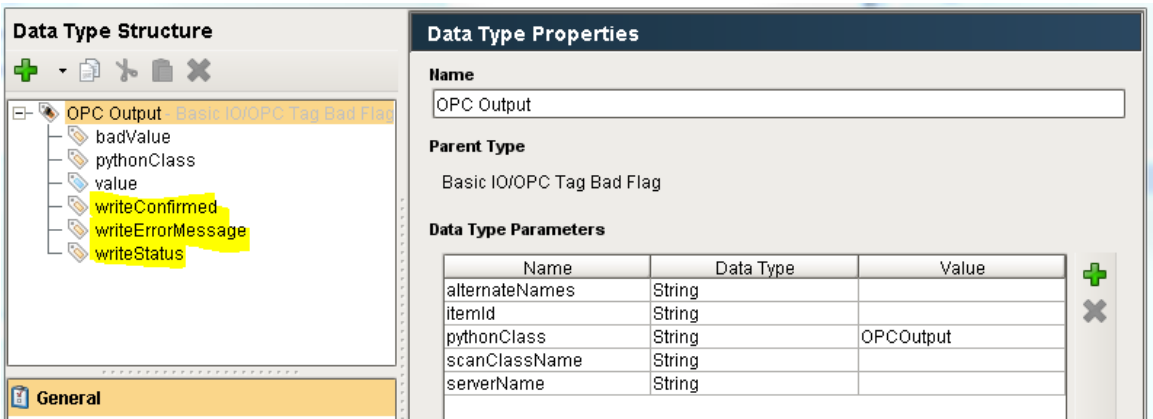


Figure 7 - OPC Output UDT

#### 4.1.4 OPC Conditional Output

The UDT for an “OPC Conditional Output”, which is derived from an “OPC Output” is shown below. It adds a *permissive* OPC tag and two memory tags: *permissiveConfirmation* and *permissiveValue*. 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. This is generally a static value and is configured manually when the UDT is instantiated and configured.
- *permissiveConfirmation*: specifies if the write process needs to confirm the permissive write before writing the value.

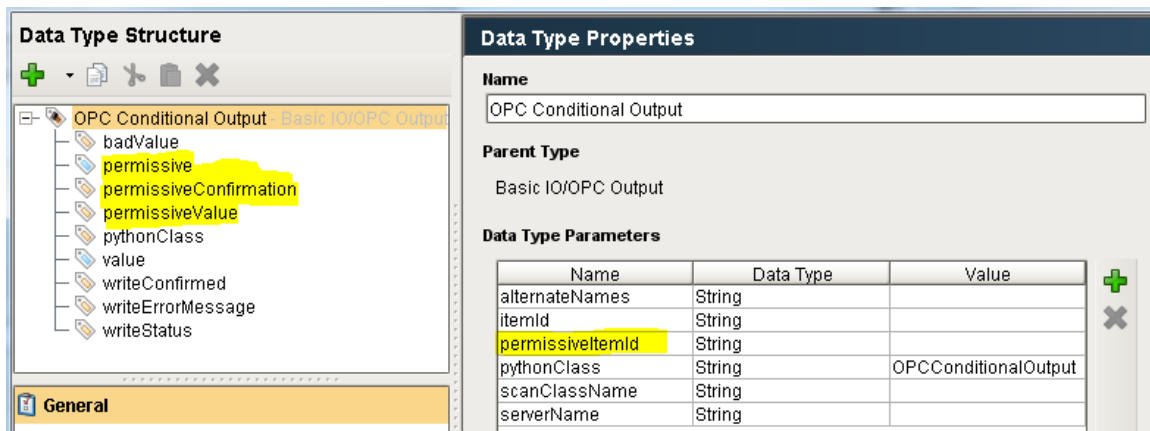
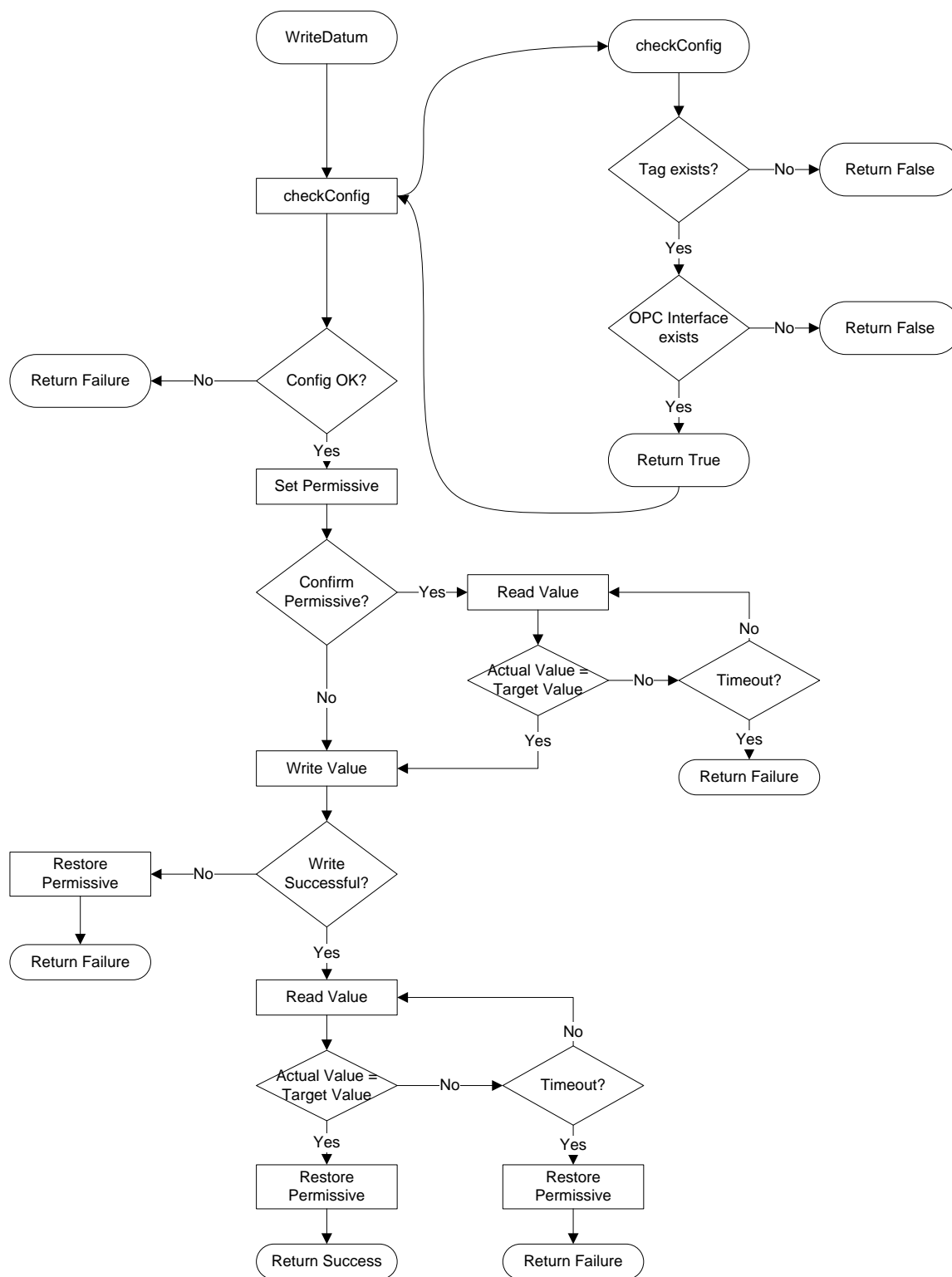


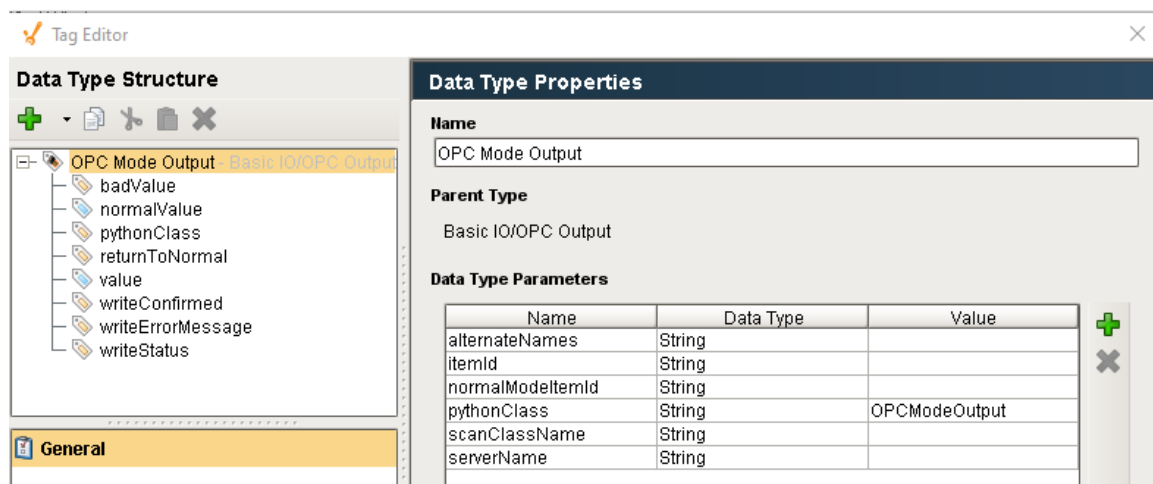
Figure 8 - OPC Conditional Output UDT

The WriteDatum logic for a Conditional Output is shown below:



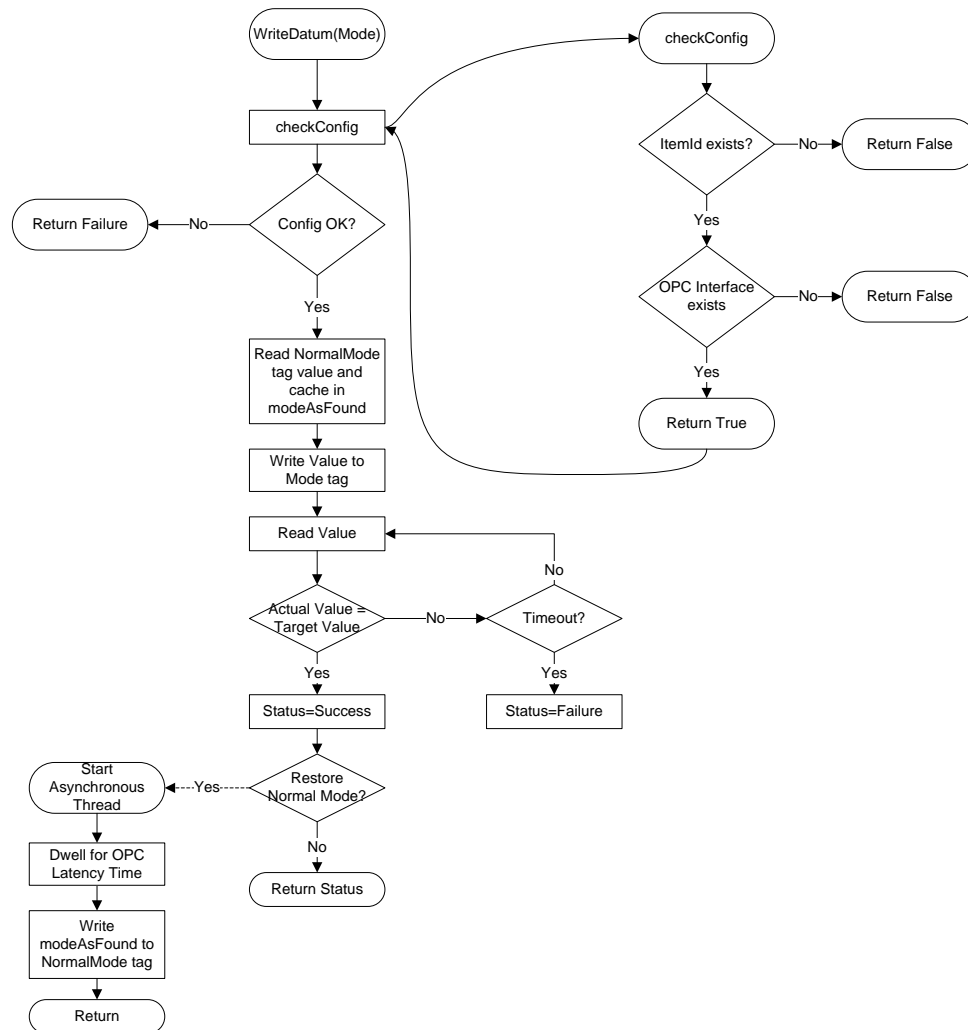
### 4.1.5 OPC Mode Output

The UDT for an “OPC Mode Output”, which is derived from an “OPC Output” is shown below. It adds a *normalMode* OPC tag and a *returnToNormal* memory tag. This UDT is used as the mode tag in all controller UDTs. This class was implemented to compensate for certain DCS systems that have an annoying "feature" that whenever a value is written to the mode attribute it also sets the value into the "normal mode" attribute. This makes life difficult for the operator when they press the "Return to Normal Mode" button. This class changes the normal write behavior to first read the normal mode value, then process the write as normal. After the write confirm completes, regardless of the state of the confirm, if the *returnToNormal* flag is set, in an asynchronous thread, wait for the OPC latency time, then write the value as found directly into the *normalMode* attribute.





The writeDatum() method is specialized as follows:



#### 4.1.6 Grade UDT

The Grade UDT is discussed in the Common Facilities Design Specification.

### 4.2 Controller Classes

The UDTs that implement the controller I/O that support both the EPKS and the TDC3000 system are described in this section. The generic controller classes that are described below may contain more tags than are needed for a specific instance. Individual tags that are not needed may be disabled in order to reduce unnecessary overhead on the OPC server.

4.2.1 Foundation Classes

4.2.1.1 Controller

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.

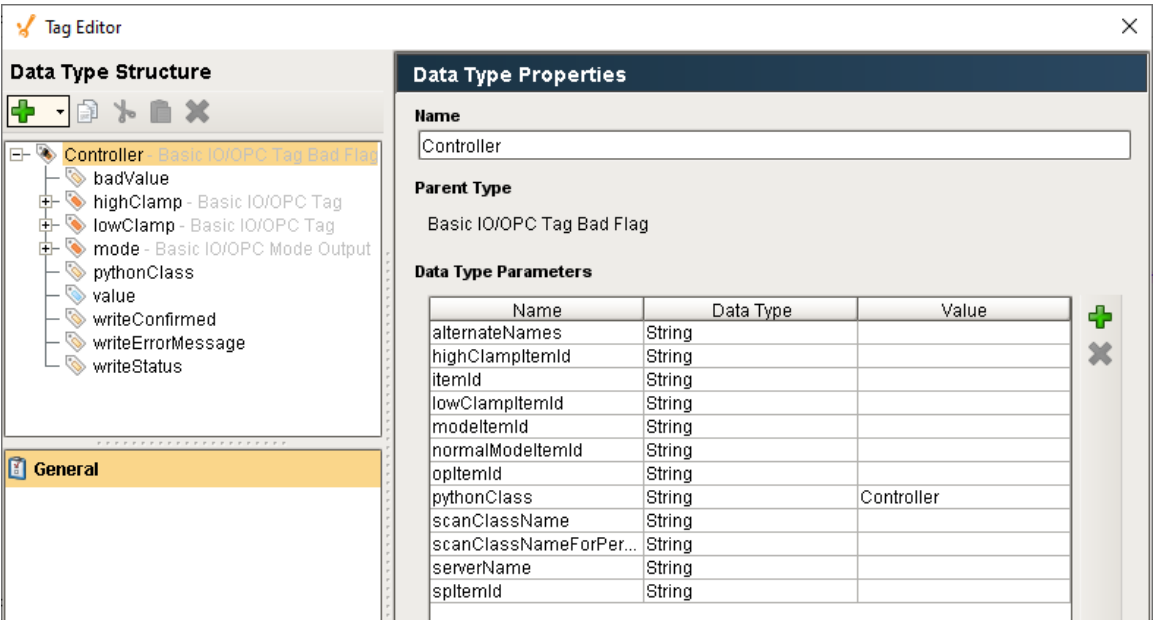


Figure 9 - Controller UDT

### 4.2.1.2 Ramp Controller

The DCS provides a special class of controllers known as ramp controllers which provides a means for gradually changing the setpoint of a controller over a time gradient to achieve the final setpoint rather than just changing it in a single step. Ignition provides a number of UDTs and Python classes for implementing the various ramp controllers.

The UDT for a base ramp controller is shown below. It is an abstract UDT and should not be instantiated.

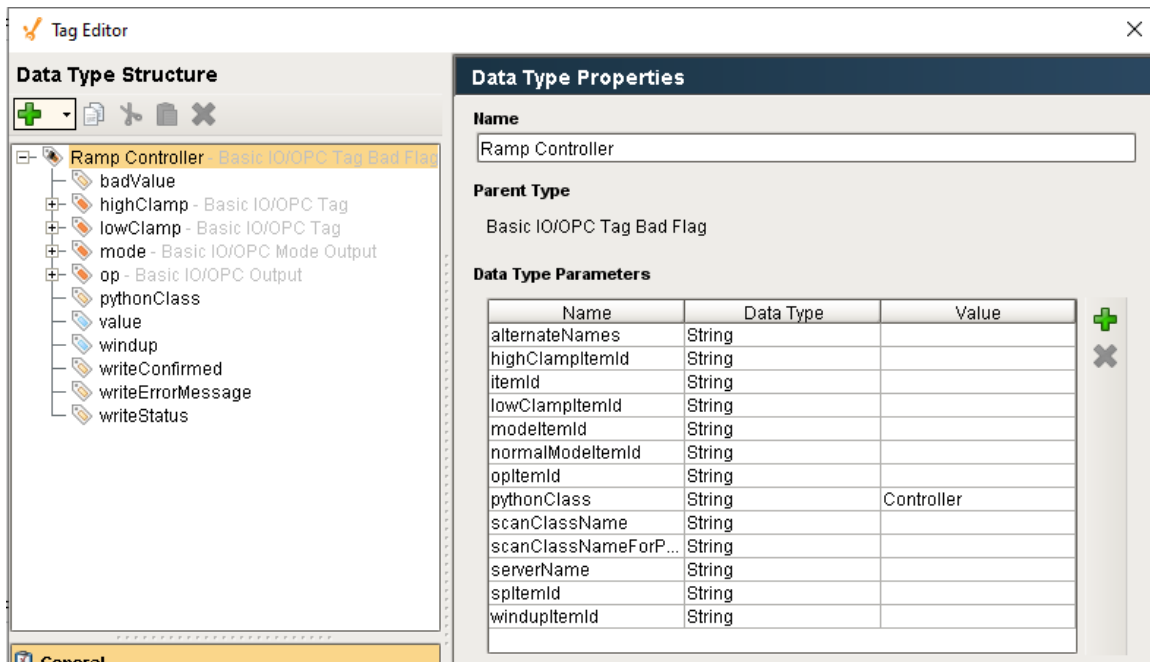
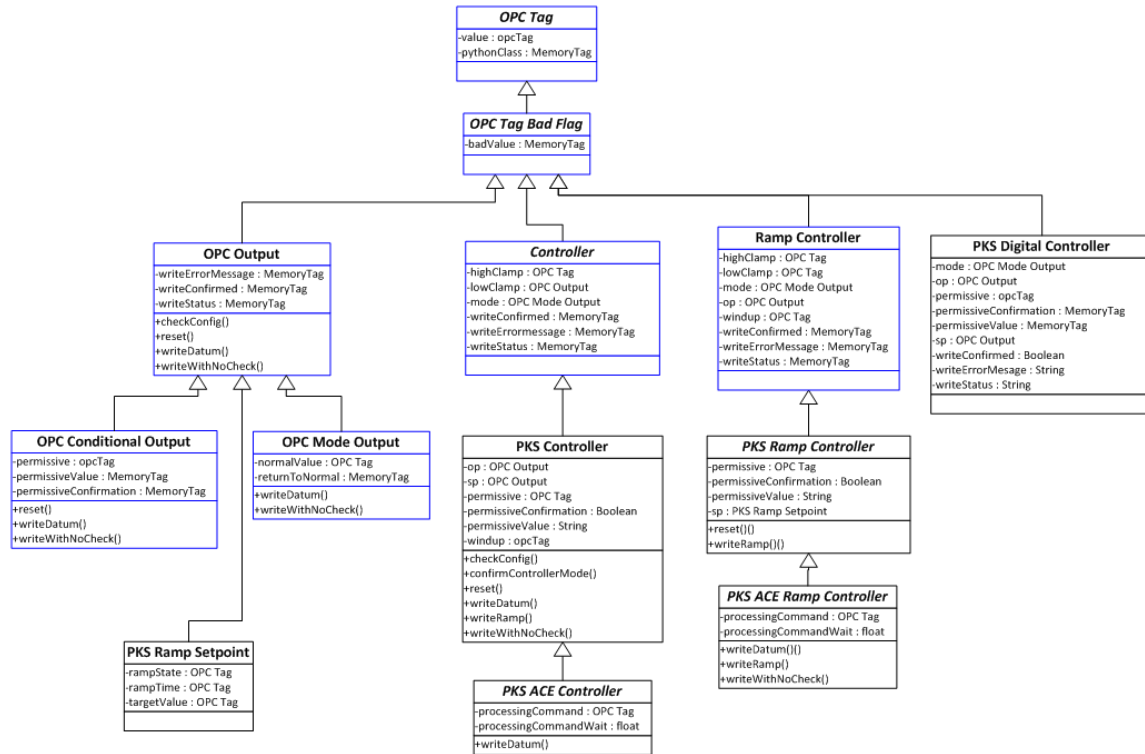


Figure 10 - Base Ramp Controller

## 4.2.2 PKS Classes

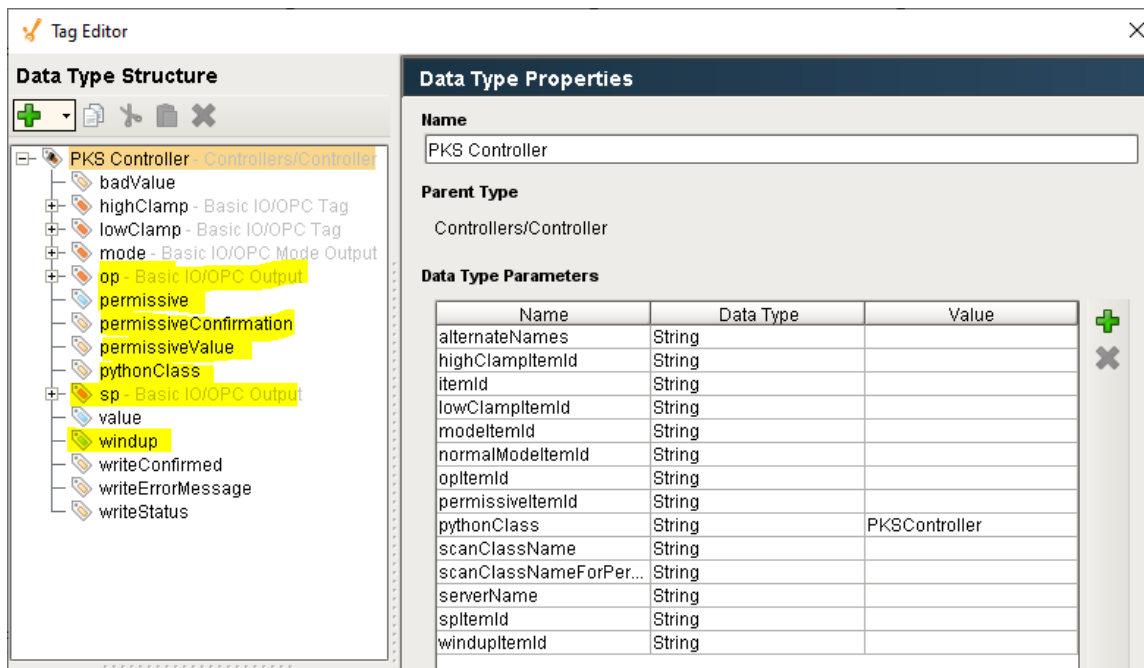
The class hierarchy is shown below:



### 4.2.2.1 PKS Controller

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

Note: the *op* and *sp* tags technically belong in the superior controller UDT.



The screenshot shows the Tag Editor interface with two main panes: Data Type Structure and Data Type Properties.

**Data Type Structure:** A tree view showing the hierarchy of the PKS Controller. The root is 'PKS Controller' under 'Controllers/Controller'. It contains several sub-items: 'badValue', 'highClamp - Basic IO/OPC Tag', 'lowClamp - Basic IO/OPC Tag', 'mode - Basic IO/OPC Mode Output', 'op - Basic IO/OPC Output', 'permissive', 'permissiveConfirmation', 'permissiveValue', 'pythonClass', 'sp - Basic IO/OPC Output', 'value', 'windup', 'writeConfirmed', 'writeErrorMessage', and 'writeStatus'. The items 'op', 'permissive', 'permissiveConfirmation', 'permissiveValue', 'pythonClass', 'sp', and 'windup' are highlighted in yellow.

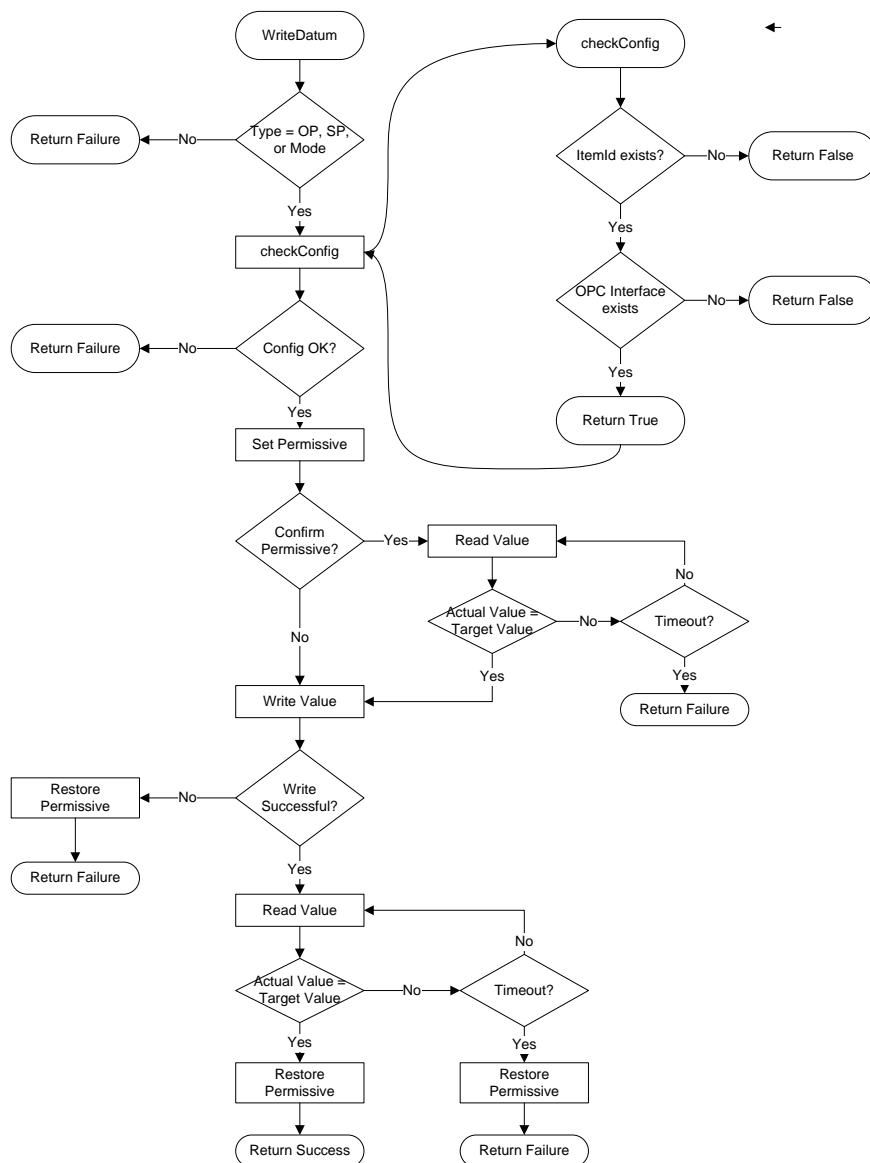
**Data Type Properties:** A form for configuring the PKS Controller. The 'Name' field is 'PKS Controller' and the 'Parent Type' is 'Controllers/Controller'. Below is a table for 'Data Type Parameters'.

Name	Data Type	Value
alternateNames	String	
highClampItemId	String	
itemId	String	
lowClampItemId	String	
modelItemId	String	
normalModelItemId	String	
opItemId	String	
permissiveItemId	String	
pythonClass	String	PKSController
scanClassName	String	
scanClassNameForPer...	String	
serverName	String	
spItemId	String	
windupItemId	String	

Figure 11 - PKS Controller UDT

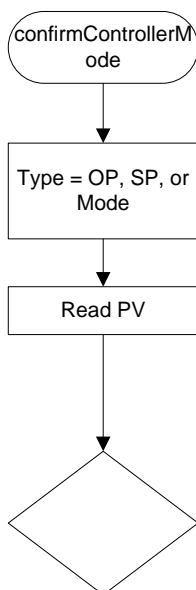
#### 4.2.2.1.1 WriteDatum Logic

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



#### 4.2.2.1.2 *ConfirmControllerMode Logic*

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



4.2.2.2 PKS ACE Controller

A PKS 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*. The background behind these two attributes is that the DCS tags are configured to run slowly, every 5 minutes. These tags change the configuration of the controllers to run almost immediately after a download to avoid - delays to the operator. The *processingCommandWait* is written to the *BPSDelay* of the controller.

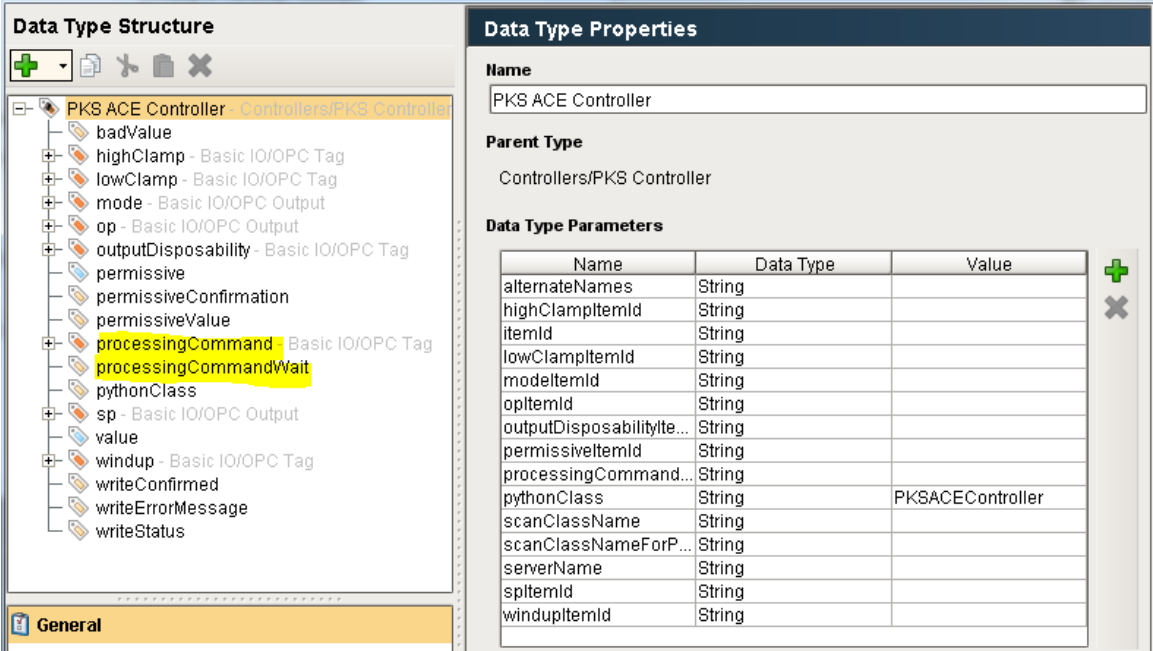


Figure 12 - PKS ACE Controller UDT



### 4.2.2.3 PKS Ramp Controller

The UDT for a PKS Ramp controller is shown below. It adds several additional tags: *permissive*, *permissiveConfirmation*, *permissiveValue*, and *sp*. It also initializes the *pythonClass* property to “PKSRampController”.

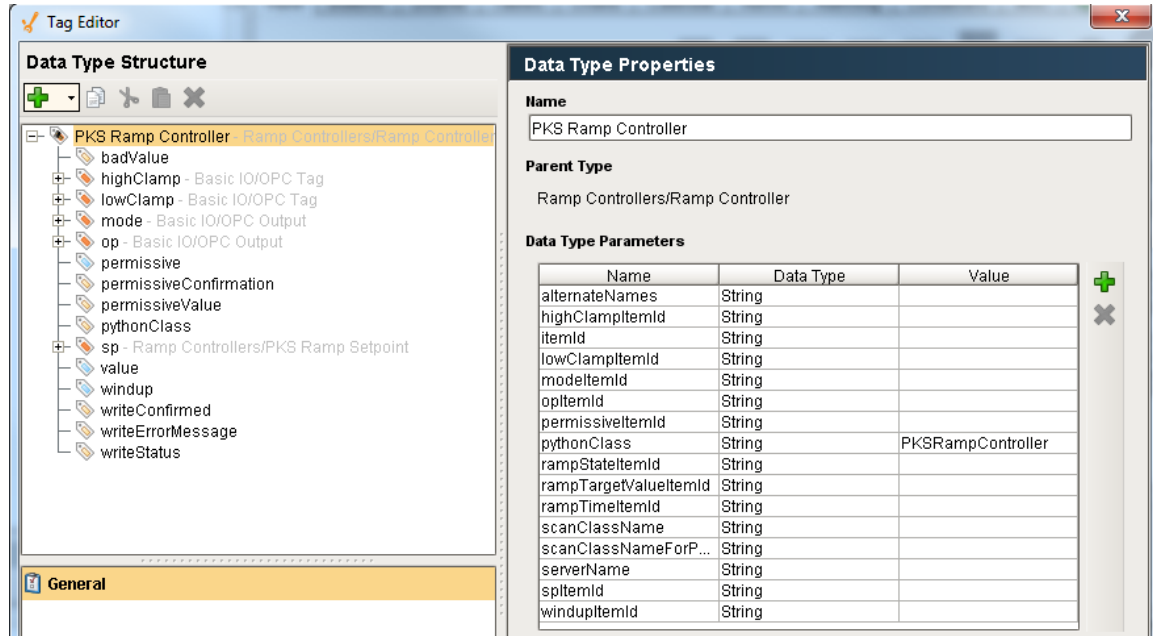


Figure 13 - PKS Ramp Controller UDT

The *sp* is a special UDT designed for controllers is a PKS Ramp Setpoint UDT and is shown below. It has tags for specifying the ramp target value, the ramp time, and the ramp state.

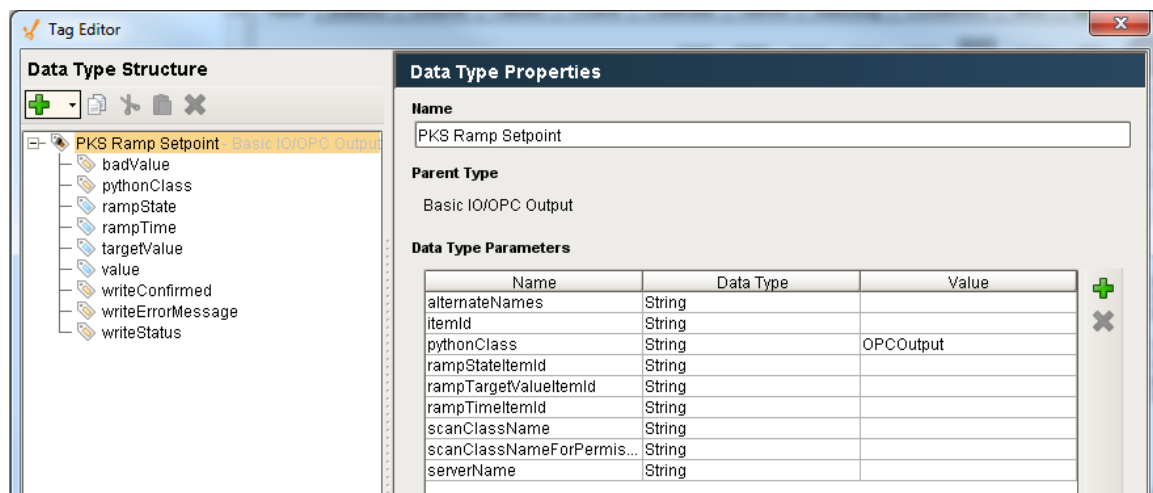
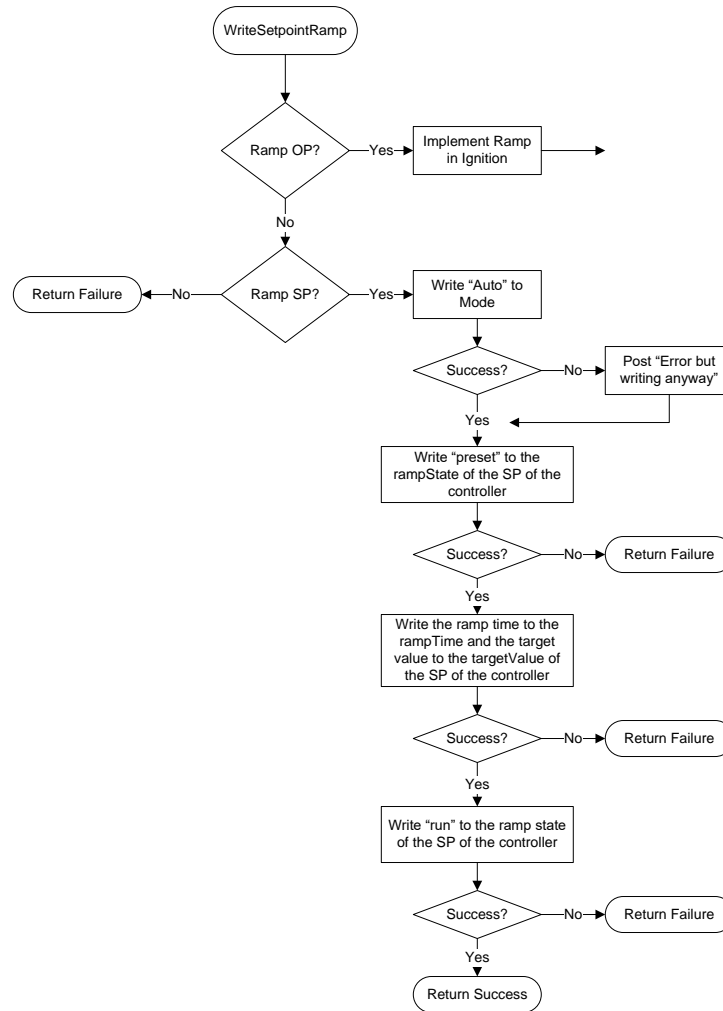


Figure 14 - PKS Ramp Setpoint UDT

#### 4.2.2.3.1 WriteRamp Logic

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



#### 4.2.2.4 PKS ACE Ramp Controller

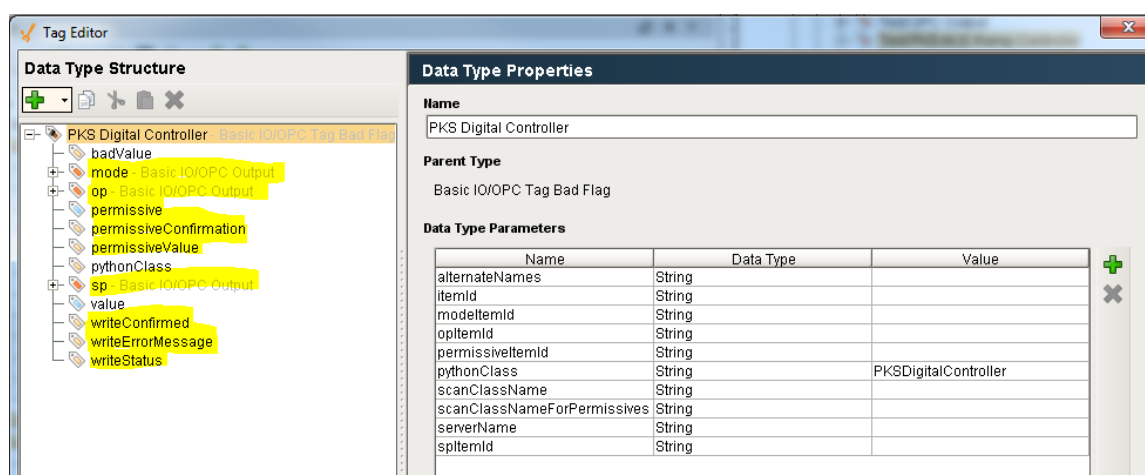
The PKS ACE ramp controller is derived from the PKS ramp controller. The only additions are the *processingCommand* and *processingCommandWait* tags.

The screenshot shows the 'Tag Editor' window. On the left, the 'Data Type Structure' pane displays a tree view of the 'PKS ACE Ramp Controller' data type. The tree includes the following tags: badValue, highClamp (Basic IO/OPC Tag), lowClamp (Basic IO/OPC Tag), mode (Basic IO/OPC Output), op (Basic IO/OPC Output), permissive, permissiveConfirmation, permissiveValue, processingCommand (highlighted in yellow), processingCommandWait (highlighted in yellow), pythonClass, sp (Ramp Controllers/PKS Ramp Setpoint), value, windup, writeConfirmed, writeErrorMessage, and writeStatus. On the right, the 'Data Type Properties' pane shows the 'Name' as 'PKS ACE Ramp Controller' and the 'Parent Type' as 'Ramp Controllers/PKS Ramp Controller'. Below these, the 'Data Type Parameters' table lists various parameters with their data types and values.

Name	Data Type	Value
alternateNames	String	
highClampItemid	String	
itemid	String	
lowClampItemid	String	
modelItemid	String	
opItemid	String	
permissiveItemid	String	
processingCommandItem...	String	
pythonClass	String	PKSACERampController
rampStateItemid	String	
rampTargetValueItemid	String	
rampTimeItemid	String	
scanClassName	String	
scanClassNameForPermi...	String	
serverName	String	
spItemid	String	
targetValueItemid	String	
windupItemid	String	

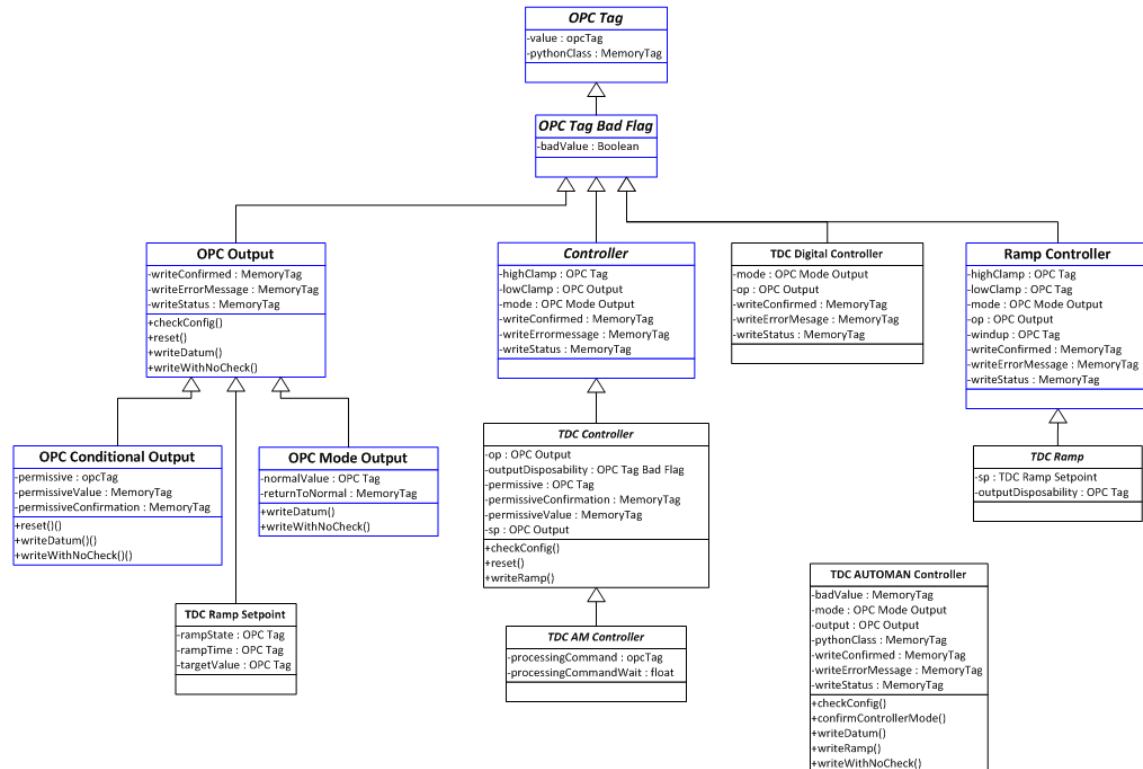
### 4.2.2.5 PKS Digital Controller

The PKS Digital Controller is derived from the OPC Tag Bad Flag. The purpose of this class is to provide a convenient way of writing directly to the OP of a controller using text values such as Open / Close, Start / Stop, etc. Typically, op, sp, and value are all text values. All of their item ids will generally end in "/ENUM". The configuration of the UDT determines which API functions can be used. Some controllers / DCS nodes support the tag type of "GOP". The sp tag would be configured with the GOP tag and the op tag is configured with an "OP" tag type. If an output type of "setpoint" is used then the value will be written to the sp tag and will be confirmed from the op tag. If an output type of "output" is used then the value will be written to the op tag and confirmed from the op tag.



### 4.2.3 TDC Classes

The TDC classes have not been fully implemented or tested. Originally, the G-Line application used a parallel class of TDC UDTs. However, the Ignition platform will utilize the PKS UDTs at G-Line by directing I/O through an Experion node in the TDC / Experion hybrid DCS architecture. The class hierarchy is shown below:



#### 4.2.3.1 TDC AUTOMAN Controller Support

The G-Line application, which uses a TDC DCS, implements a pseudo controller known as an AUTOMAN controller. It is not clear exactly what this class of controller looks like in the DCS, but in Ignition, this class does not extend any existing UFT (not sure why not). One of methods this class implements is a ramp. The ramp is implemented entirely in software. Unlike most controllers, an AUTOMAN controller does not have a PV, SP, limits, or permissives.

#### 4.2.3.2 Obsolete TDC Classes

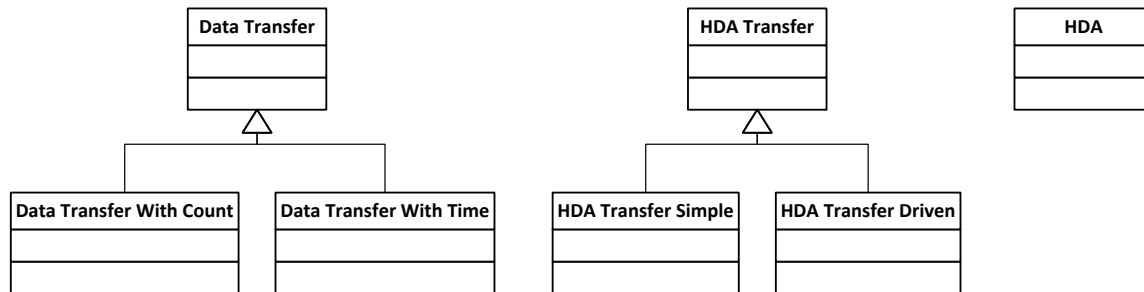
The following table describes how old classes were mapped into the new system:

Old Class	New Class	Description
opc-tdc-controller	Obsolete	Abstract Class
opc-tdc-controller-	TDC Controller	

pm		
opc-tdc-controller-am	Obsolete	Deleted because there were no instances.
opc-controller-tdc-output	Obsolete	These instances were merged into TDC Controller
opc-controller-tdc-am-output	TDC AM Controller	
opc-tdc-digital-controller	Obsolete	Abstract class
opc-tdc-digital-pm-controller	TDC Digital Controller	This class was merged with the abstract parent class.
opc-tdc-digital-am-controller	Obsolete	There were no instances so not implemented
opc-tdc-automan-controller	TDC AUTOMAN Controller	
Opc-tdc-ramp-var	TDC Ramp Controller	
Opc-tdc-sp-ramp	TDC Ramp Setpoint	This UDT is embedded in a TDC Ramp Controller and cannot stand alone
Opc-controller-tdc-ramp-output	Obsolete	The TDC Ramp Controller supports writes to OP, SP, MODE, OP ramp, and SP ramp.

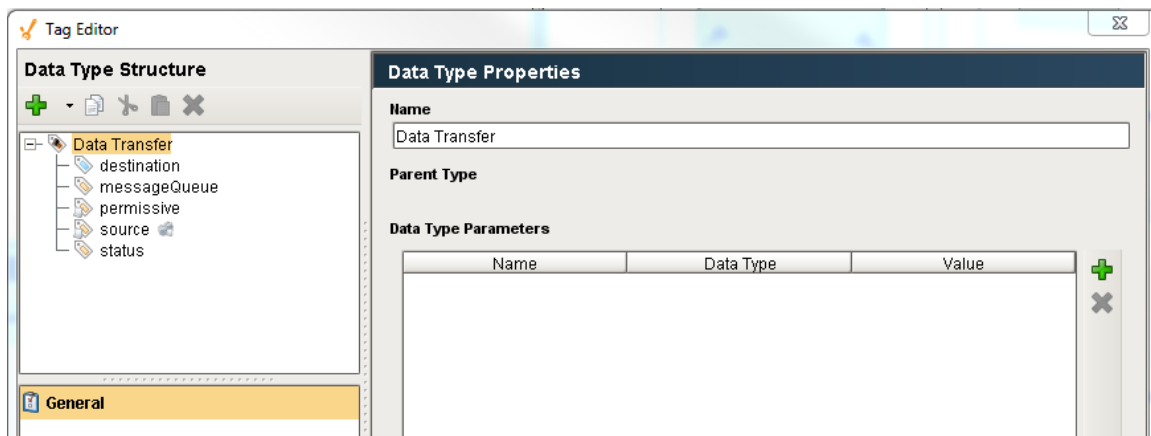
### 4.3 Data Transfer Classes

There are a number of UDTs that are provided to shuttle data between the various systems (Ignition, OPC/DCS, OPC-HDA/PHD). While Ignition provides a set of programmatic functions for reading from and writing to OPC and OPC-HDA servers, these UDTs provide a non-programmatic means of shuttling data. The classes are:



#### 4.3.1 Data Transfer

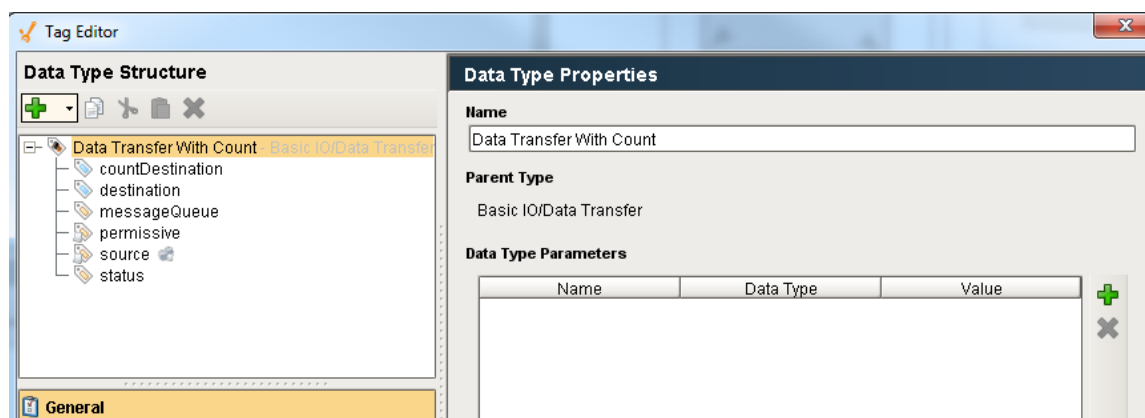
This UDT, and the UDTs derived from it, provide a means to shuttle a value from Ignition to an OPC server. This UDT facilitates the transfer of a single value from Ignition to an OPC server. The UDT definition is shown below:



The UDT does not have any parameters. It is configured by configuring the UDT member tags. The *source* tag is an expression tag that has an event change script. The expression can reference anything in Ignition. The value of this tag is the value that will be shuttled to the OPC tag. If the *permissive*, which is also an expression tag, is True, then the value of the source tag is written to the *destination* tag. If the *messageQueue* tag has a value then the status of the write is posted to the queue named by the value of *messageQueue*. The write logic attempts to write the value three times, using a synchronous write, with a latency delay between each attempt, before setting the *status* tag to failure.

### 4.3.2 Data Transfer with Count

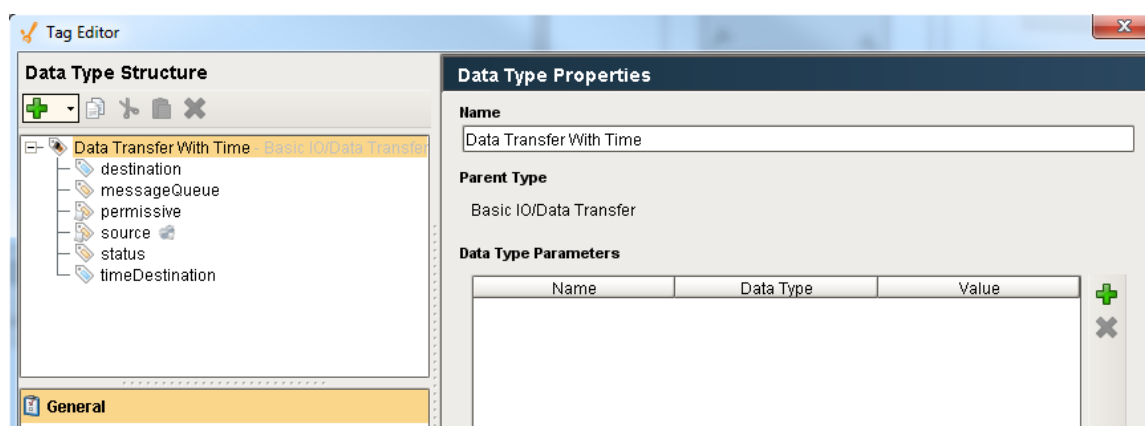
This builds upon the functionality provided by the Data Transfer UDT. It adds an automatically incremented count tag. The UDT definition is shown below:



This follows the same write logic as the Data Transfer UDT with the addition of a second write to the *countDestination* tag.

### 4.3.3 Data Transfer with Time

This builds upon the functionality provided by the Data Transfer UDT. It adds an automatically set date time tag. The UDT definition is shown below:

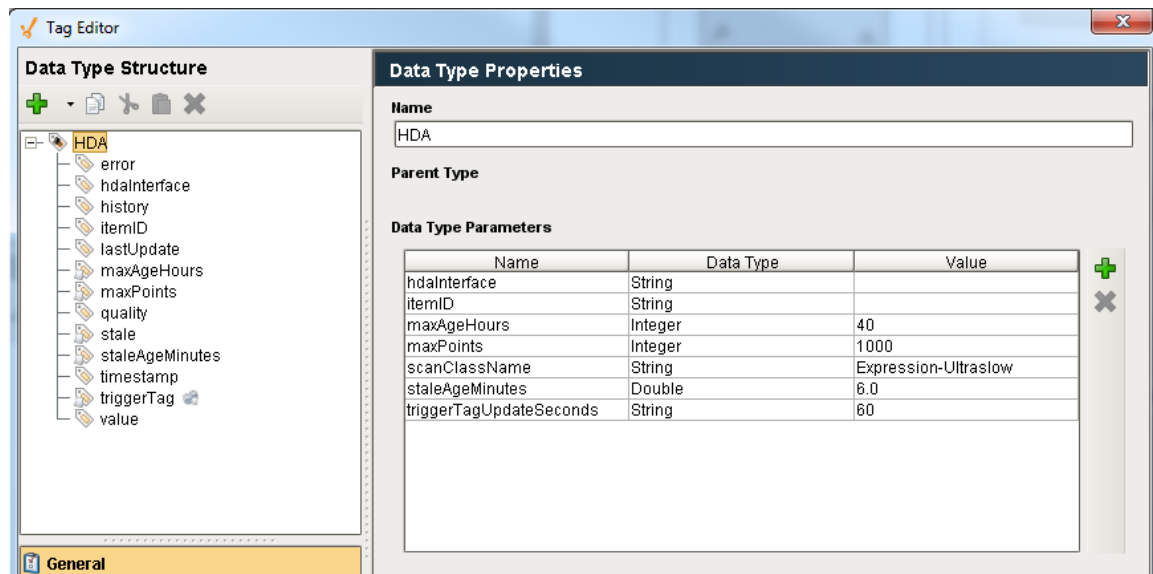


This follows the same write logic as the Data Transfer UDT with the addition of a second write to the *timeDestination* tag.



#### 4.3.4 HDA

This UDT is designed to fetch data from an OPC-HDA in a convenient manner. Unlike the previous UDTs, this UDT is configured via the UDT parameters.

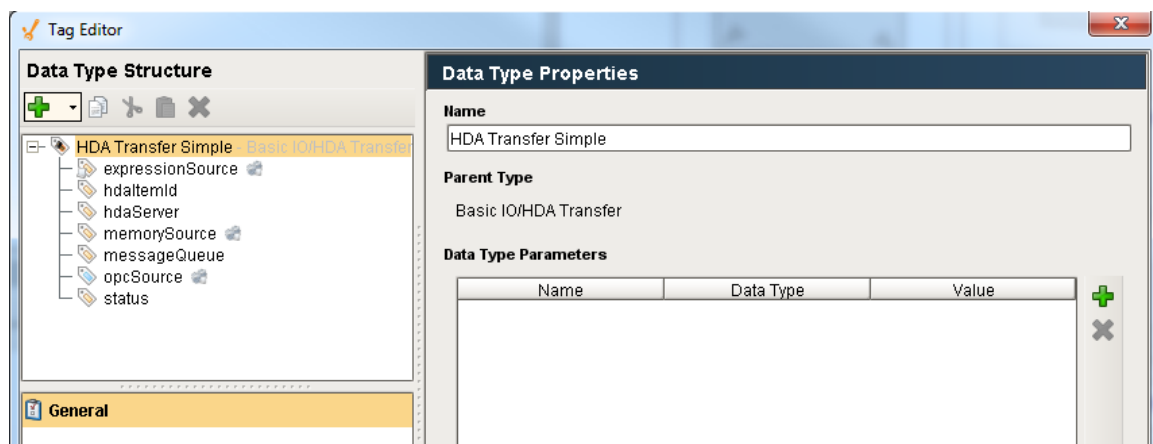


#### 4.3.5 HDA Transfer

This is the root of the HDA Transfer family of UDTs and is abstract and should not be instantiated.

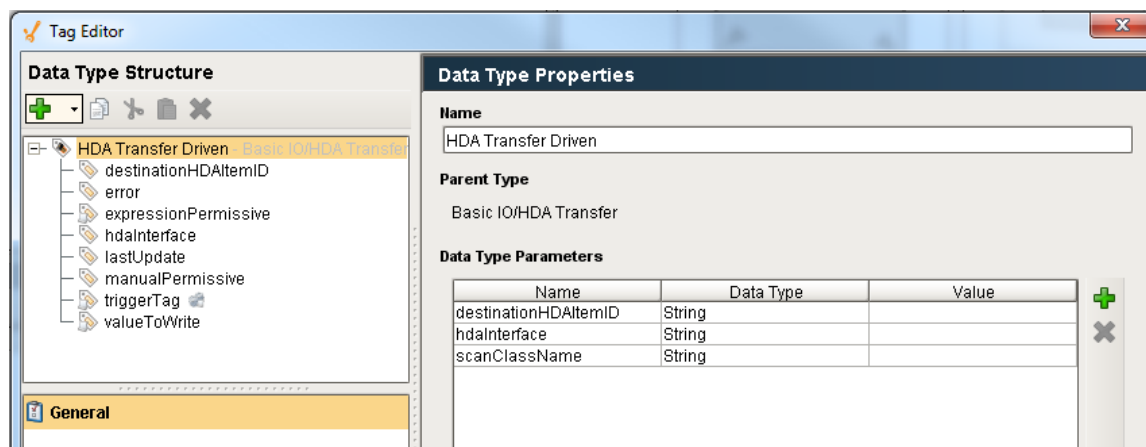
#### 4.3.6 HDA Transfer Simple

This is the simplest UDT that provides integration to an HDA server. Instances of this class are configured by configuring the individual tag members. This UDT will transfer the value from any one of three sources, a memory tag, an OPC tag, or an expression tag, to an HDA server. Rather than providing three different UDTs for each of the different source types, this UDT provides the three different sources in one UDT. It is expected that only one of the three sources be used at any time. This UDT uses the same retry and logging as the Data Transfer UDTs described earlier.

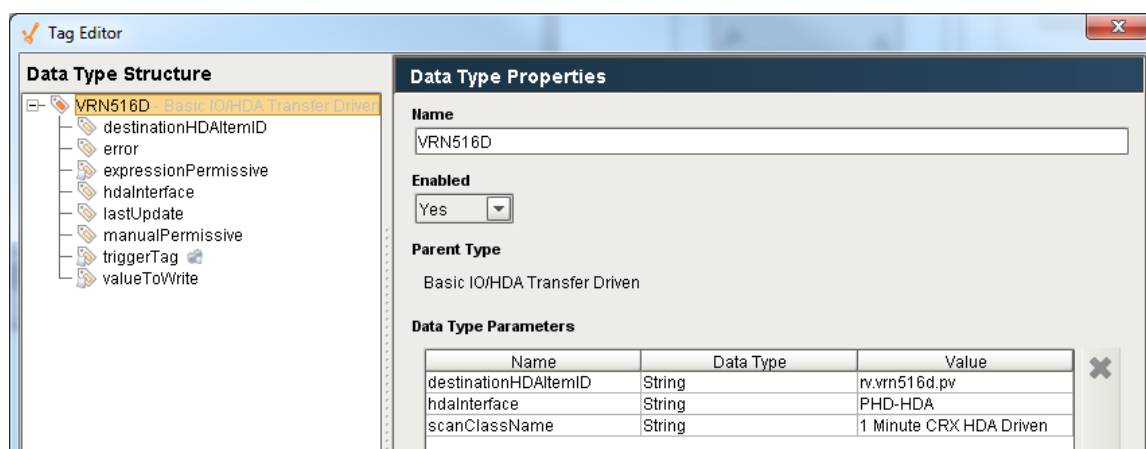


### 4.3.7 HDA Transfer Driven

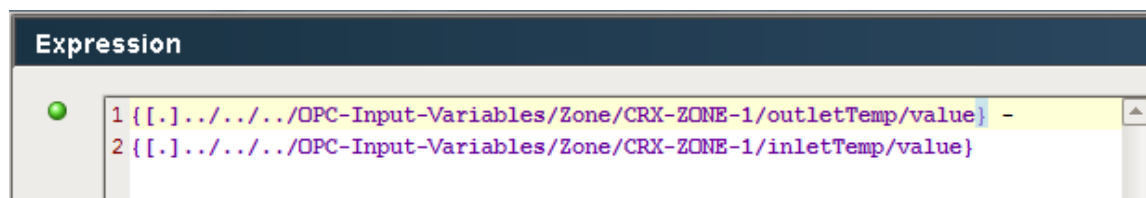
This UDT that provides a means of writing values to a HDA server on a fixed interval using a specialized scan class.



An example of the UDT in use by the Vistalon site is shown below:



The *valueToWrite* expression tag provides the value that will be written to the HDA server. The expression for this UDT is:



The driven scan class that is referenced by the UDT above is shown below. The purpose of the driven scan class is to be able to turn the writes off, in this case when Vistalon is not running the C reactor. It also provides a mechanism to drive a set of tags to write to the HDA sever on a regular interval.

## 5 API

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.

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:

```
1 tagPath = event.source.parent.parent.getComponent('Tag Name').text
2 val = event.source.parent.getComponent('Value To Write').floatValue
3 valueType = event.source.parent.getComponent('Value Type Dropdown').selectedStringValue
4 writeConfirm = event.source.parent.getComponent('Confirm Write').selected
5
6 from ils.io.api import writeOutput
7 status, errorMessage = writeOutput(tagPath, val, writeConfirm, valueType)
```

Figure 15 - I/O API Example

## 5.1 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.

## 5.2 Output Functions

These functions apply to all tags, from simple memory tags to controllers.

**writeDatum**(tagPath, value, valueType)

Write a value and confirm that the write was successful by reading the value back. Depending on the scan class scan rates, the confirmation may take several minutes, because the client is single threaded, this will hang the client. This is supported for everything from memory tags to EPKS controllers.

Argument	Description
tagPath	Path to the UDT, not to the OPC tag which is a member of the UDT.
value	Value to be written
valueType	This applies to controllers. For UDTs that are not controllers use “value”. For controllers the values are sp, op, and mode.

**writeWithNoCheck**(tagPath, value, valueType)

This function writes a value without doing any confirmation. It verifies that the OPC write was successful and performs basic checks to determine if it is ok to write but it does not confirm 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.
value	Value to be written
valueType	This applies to controllers. For UDTs that are not controllers use “value”. For controllers the values are sp, op, and mode.

### 5.3 Controller Functions

These functions apply to Controllers and its subclasses

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

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.

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.

**writeRamp**(controllerTagpath, value, valueType, rampTime, updateFrequency, writeConfirm)

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 valueType.

Argument	Description
controllerTagpath	Full path to the controller
value	The value to be written.
valueType	The attribute of the controller to be written to: OP or SP.
rampTime	The length of the ramp in minutes
updateFrequency	The step size of the ramp in seconds
writeConfirm	Specifies if the final value will be confirmed.

### 5.4 Recipe Toolkit Extensions

There are several extensions to the I/O facilities for the Recipe toolkit. Refer to the **Recipe Toolkit Design Specification** for details.

## 5.5 I/O API Summary

The following table defines which API functions are supported for each of the I/O classes.

	WriteDatum()	WriteWithNoCheck()	ConfirmControllerMode()	WriteRamp()
OPC Output	Yes	Yes	No	No
OPC Conditional Output	Yes	Yes	No	No
PKS Controller	Yes	Yes	Yes	Yes
PKS Ramp Controller	Yes	Yes	Yes	Yes
PKS ACE Controller	Yes	Yes	Yes	Yes
PKS ACE Ramp Controller	Yes	Yes	Yes	Yes
PKS Digital Controller	Yes	Yes	No	No

## 6 Isolation Support

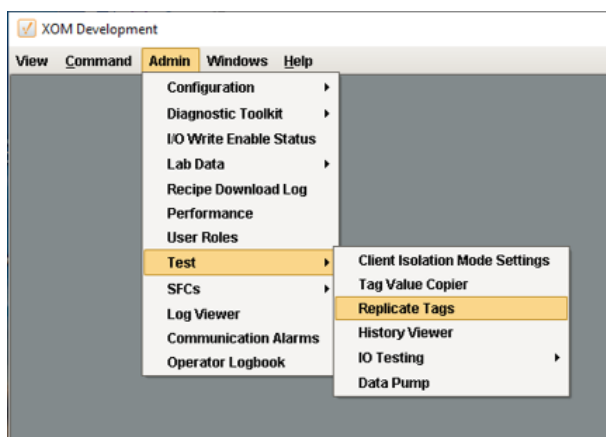
Isolation Mode is ubiquitous throughout the ILS Ignition tools. It is supported in lab data, recipe, SFC extensions, and Symbolic Ai. The purpose of Isolation mode is to “isolate” the production systems from test and development activities. It isolates the database and the DCS. This section describes how the DCS is isolated from development and test activities.

### 6.1 Isolation UDTs

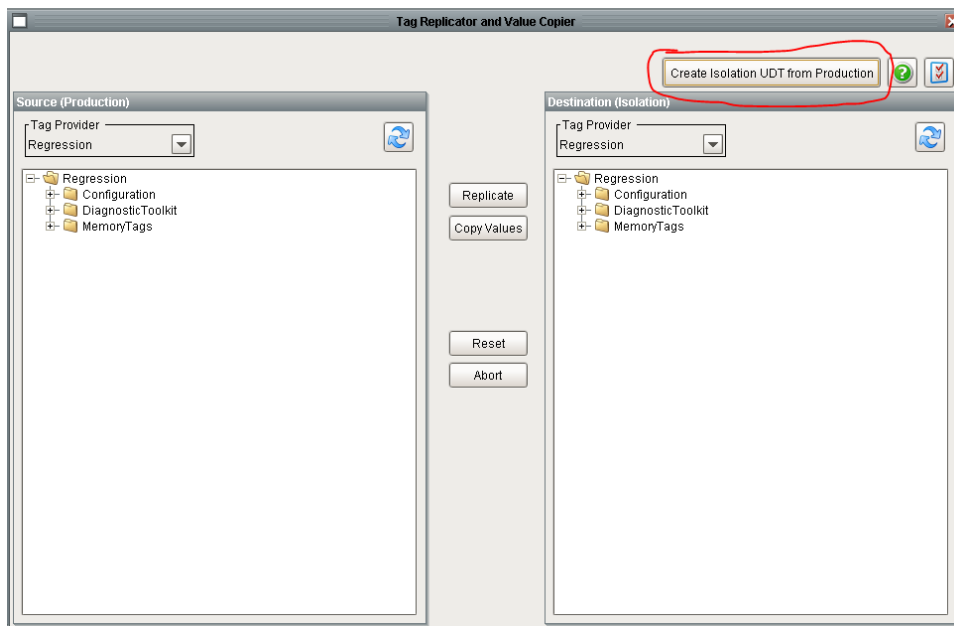
The gateway has a production tag provider and an isolation tag provider. A parallel set of UDTs are provided to support Isolation Mode. The isolation UDTs replace OPC tags with memory tags.

## 6.2 Generating Isolation UDT

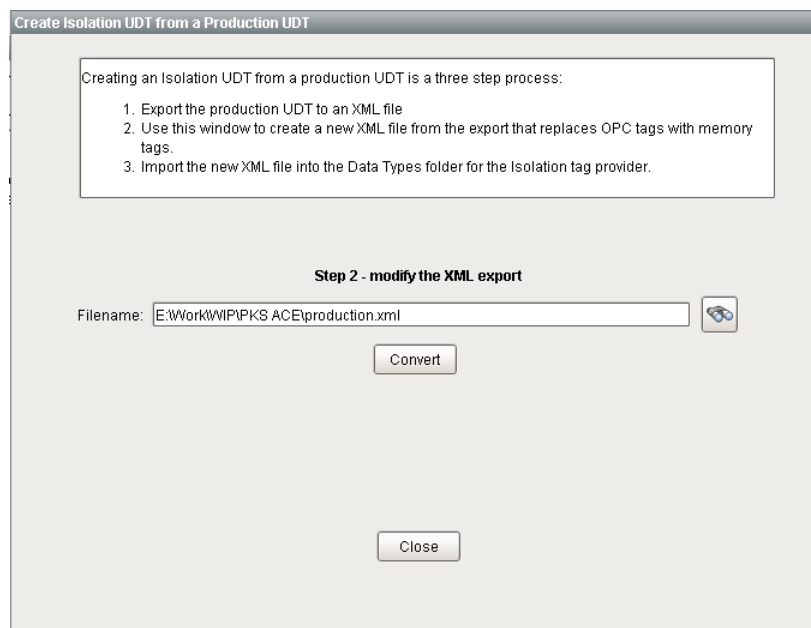
The preferred way to create isolation UDTs is to use the “Replicate Tags” window under the Admin -> Test menu.



Which opens the window shown below:



Use the “Create Isolation UDT from Production” button which displays the following window:



As described at the top of the window, creating Isolation UDTs is a three step process. The first and third steps are performed using the Designer. The second step is performed by this window. It takes the xml file created in step #1, performs string manipulation to replace OPC tags with memory tags. It writes the updated file contents to a file with the same name as the original with “\_isolation” appended to the name.



### 6.3 Replicating Tags

When testing, it is important that the Isolation tags are in sync with Production tags. It is difficult to manually reconcile the production tag provider with the isolation tag provider. The window discussed in the previous section can be used to create isolation UDTs from the production UDTs. The “Replicate” button will create isolation UDTs for the folder selected in the *Source* tag tree. The utility does not update existing UDTs whose type may have changed. If there is doubt, the utility runs fast enough that all isolation instances can be deleted and recreated. The “Copy Values” button can be used to copy the current values of production to tag providers.

