**Block Language Toolkit**

**High Level Design**

**ExxonMobil - Baton Rouge Chemical Plant**

**Document Version: 1.9**

**Mar 24, 2015**

Table of Contents

1. Change History 4

1.1 References 4

1.2 Disclaimer 4

2. Introduction 5

2.1 Licensing 5

2.1.1 Ignition 5

2.1.2 Toolkit 5

2.2 Prerequisites 5

2.2.1 Java 5

2.2.2 Eclipse 6

2.2.3 Ignition 6

2.3 Internationalization 6

3. Architecture 7

3.1 Prototypes 8

3.2 Serialization 9

3.3 Custom Blocks 11

3.4 Model Definition 11

3.5 Auxiliary Data 12

3.5.1 Normal operation/Editing 12

3.5.2 Transfer a project 12

3.5.3 Export/Import 12

3.5.4 Isolation Mode 13

3.5.5 Fresh Migration 13

3.5.6 New Blocks 13

3.6 Gateway Scope 13

3.7 Designer 15

3.7.1 Navigation Tree 15

3.7.2 Saving 15

3.7.3 Visual Synchronization 17

3.8 Client 17

4. Gateway 18

4.1 Gateway Functions 18

4.1.1 Dispatcher 18

4.1.2 Resource Changes 18

4.1.3 Block Execution 18

4.1.4 Tag Changes 18

4.1.5 Block State 19

5. Designer 20

5.1 NavTree 20

5.1.1 Root Node 20

5.1.2 Application Nodes 21

5.1.3 Family Nodes 21

5.1.4 Folder Nodes 21

5.1.5 Problem Nodes 22

5.2 Menu 22

5.3 Palette 22

5.4 Icons 23

5.5 Diagrams 23

5.5.1 Diagram State 24

5.5.2 Dirty 25

5.5.3 Reset Action 25

5.6 Blocks 26

5.6.1 Block Behavior 26

5.6.2 Block Definitions 26

5.6.3 Bad Data Handling 57

5.6.4 Block Locking 58

5.6.5 Block Reset 58

5.6.6 Timestamp Handling 58

5.6.7 Connection Type Change 58

5.6.8 Type Coercion 59

5.6.9 Block Programming Interface 59

5.7 Block Properties 63

5.7.1 Data Types 63

5.7.2 Binding 63

5.7.3 Activity Log 64

5.8 Property Editor 64

5.9 Connections 65

5.10 Transmit/Receive 65

5.10.1 Signals 66

5.10.2 Standard Signals 66

5.11 Find/Replace 66

6. Client 68

6.1 Operator 68

6.2 Engineer 68

6.3 Communication with the Gateway 68

7. Python 69

7.1 Custom Blocks 69

7.1.1 Constraints 69

7.1.2 Callbacks 69

7.1.3 Installation 73

7.1.4 Samples 73

7.2 Scripting Interface 73

7.3 Extension Functions 78

Appendix A. Debugging 80

A.1 Logging 80

Appendix B: Functional Testing 81

B.1 Block Test Project 81

B.2 Block Language Toolkit Test Module 81

B.2.1 Mock Diagram 81

B.2.2 Scripting Interface 82

Appendix C: Application Testing 87

C.1 Introduction 87

C.2 Time Acceleration 87

Appendix D: Adding Math Functions 88

D.1 Arithmetic Block 88

D.2 Math Library 88

D.3 Example Modification 88

D.3.1 Function List 88

D.3.2 Import Statement 88

D.3.3 Accept Function 89

Appendix E: Custom Actions 90

E.1 Action Block 90

E.1 Call Signature 90

E.2 Request Handler 90

E.3 Example 90

# Change History

| Version | Date | Author | Changes |
| --- | --- | --- | --- |
| 0.1 | Jan 27, 2013 | C. Coughlin (ILS) | Initial version |
| 0.2 | Jan 28, 2014 | C. Coughlin (ILS) | Application based on block & connector interface |
| 0.3 | April 5, 2014 | C. Coughlin (ILS) | Major expansion. Added sections on migration, block definitions, python interfaces. |
| 0.4 | Aprill 30,2014 | C. Coughlin (ILS) | General updates, added section on functional test |
| 0.5 | July 1,2014 | C. Coughlin (ILS) | Milestone II updates |
| 0.6 | Oct 1,2014 | C. Coughlin (ILS) | Milestone III updates |
| 0.7 | Nov 4,2014 | C. Coughlin (ILS) | Milestone III updates – clarifications |
| 1.0 | Jan 5, 2015 | C. Coughlin (ILS) | Minor block behavior clarifications |
| 1.2 | Mar 27, 2015 | C. Coughlin(ILS) | Block description updates |
| 1.3 | Apr 16, 2015 | C. Coughlin(ILS) | Misc block description updated |
| 1.4 | May 1, 2015 | C. Coughlin(ILS) | Auxiliary data |
| 1.5 | May 18, 2015 | C. Coughlin(ILS) | Block behavior changes |
| 1.6 | June 1,2015 | C. Coughlin(ILS) | Scripting changes |
| 1.7 | Sept 15,2015 | C. Coughlin(ILS) | Compare block handles dates |
| 1.8 | Sept 23,2015 | C. Coughlin(ILS) | Add lab data, pulsed and property setter blocks |
| 1.9 | Mar 20,2016 | C. Coughlin(ILS) | Trend/timer blocks, activity log |

## References

| Version | Date | Title | Author |
| --- | --- | --- | --- |
| 2.0r1 | May 2006 | ExxonMobil – BRCP – Toolkit Migration Project | ILS Automation for EM – BRCP |
| 2.0r1 | March 2007 | ExxonMobil – BRCP – Diagnostics Toolkit, Engineer’s Documentation | ILS Automation for EM – BRCP |

## Disclaimer

The application and its documentation are works in-progress. At times, this document may describe features not-yet-implemented. In these cases the text will be annotated in *italics*.

# Introduction

The ExxonMobil Chemicals Diagnostic Toolkit is a collection of applications designed for the detection, management, annunciation, and response to events. The output of any of the constituent tool is a diagnosis, a recommended response to the problem. Details of existing applications may be found in the references.

ExxonMobil Chemicals requires that the existing Diagnostic Toolkit be ported from the current platform. Ignition™ from Inductive Automation has been chosen as the target platform. Ignition is an extensible platform that provides OPC connectivity, database integration and a graphics library for interface development. It is designed to be extensible via Python scripting and custom Java modules.

The Block Language Toolkit (BLT), the subject of this document, is just such a module. It has been designed as a hosting platform for the Diagnostics applications. The overarching requirement of the BLT project is to achieve operational equivalency with the current Diagnostic Toolkit on a G2™ platform. Additionally, because of the large number of legacy applications, a conversion utility is required. This utility must export individual legacy diagrams into a form ready for import into the new platform.

## Licensing

### Ignition

The application requires a commercial Ignition license to be obtained by ExxonMobil from Induction Automation. ExxonMobil is responsible for its installation on application servers.

### Toolkit

The Block Language Toolkit is the joint property of ILS Automation and ExxonMobil Chemicals. Any applications based on the toolkit in collaboration with ExxonMobil Chemicals remain the property of ExxonMobilChemicals.

The application is heavily dependent on various open-source packages. The packages are listed in the license statement that is displayed when loading the module. Open source packages have been carefully selected to included only those with licenses that allows free and unfettered use of the package.

## Prerequisites

### Java

Ignition 7.7 requires Java JDK1.8. It is the customer’s responsibility to install the Java JDK on any system that will be running the application.

### Eclipse

For development support, compilation of Java 1.8 code requires Eclipse Luna 4.4 or newer. Any systems to be used for development require a Java SDK installation.

### Ignition

This application requires Ignition 7.7.0 or newer.

## Internationalization

There is no requirement for localized text. The application will be presented in English.

# Architecture

An Ignition project implements, by its very nature, a client-server architecture. The server is called the “Gateway”. It supports autonomous processing without need for clients.

Client views are provided for observation and/or control of the application. Clients can either co-reside on the server platform or be remote.

The Ignition platform is typically customized using Python scripting combined with Ignition-provided user-interface widgets. The specific Python implementation is 2.6 via a Java-based translator called Jython 2.5. Since Jython is Java-, it has the ability to utilize Java modules, classes and methods in a very straight-forward manner. Python/Jython scripts execute either within a client/designer or gateway scopes. Client scripts are intended for functionality associated with user-interaction. Gateway scripts execute without user-interaction.

Additionally, the Ignition platform is extensible via customized Java packages called modules. Based on the Ignition Software Development Kit (SDK), modules are tightly integrated with other parts of the platform. For the purposes of the toolkit, a module was developed to take advantage of Ignition’s block-and-connector interface and to implement the execution engine as a completely autonomous entity. The BLT executes completely within the Ignition gateway scope, but offers hooks via RPC calls to designer and client scopes as well. It also provides classes that support the Designer-view of a diagram.

The following diagram shows a functional breakdown and communications between the application scopes. Code written as part of the Java module is shown in blue, Python in green

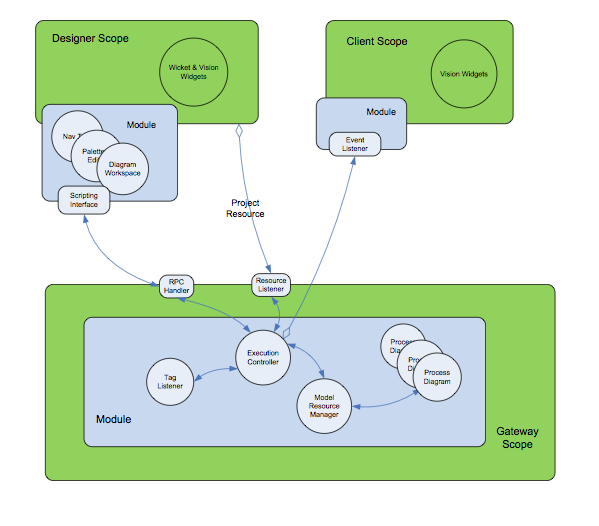


Figure 1 – Collaboration Diagram

## Prototypes

The design decision for the current architecture came about after testing of a series of prototypes.

An initial prototype was developed based on the Vision™ graphics module that is distributed as part of the Ignition platform. This prototype ran into serious, time-consuming obstacles. Namely that the concept of connections was not supported and that the graphics classes were not easily extensible.

A second prototype was developed based on the JgraphX open-source graphics package. It was highly extensible and supported the concept of connections but only at a very primitive level. A high level effort would have been required to complete the customizations necessary to meet the toolkit requirements.

The final architecture makes use of a new feature of Ignition 7.7, blocks-and-connectors. This interface has proven to be easily customizable and is well-integrated into the Ignition framework. It has the additional advantage of being supported by Inductive Automation. It is made available as part of the Ignition core platform at no additional cost.

## Serialization

Serialization refers to the process of converting Java objects into a format suitable for storage and/or network transmission. The process of recovering the Java objects from storage is called “ccordingl”.

Serialization is an important consideration in the toolkit design because, unlike the legacy application, the Ignition scopes (Designer, Client and Gateway) are (or can be) hosted on separate systems. They do not share virtual machines. Serialization of the model is required to synchronize between scopes. It is also used when projects are saved.

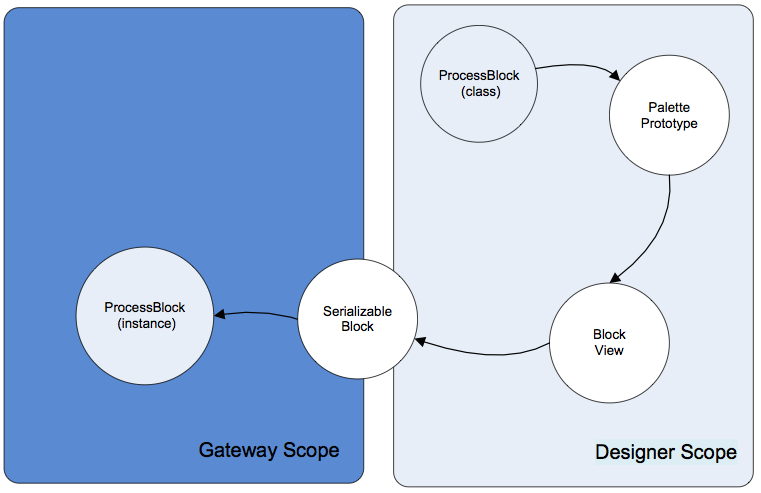


Figure 2 – Serialization Cycle

From a macro level, an Ignition project follows a standard Model-View-Controller paradigm. Objects in the Designer support the view, objects in the Gateway are the model, and Ignition, itself, is the controller.

The figure above illustrates the sequence of operations involved in following a processing block through its different environments.

1. Palette Prototype. When the Designer creates the block palette it queries the Gateway for a list of block classes that are annotated as “*ExecutableBlock*”. From these it obtains prototype objects that contain enough information to create viewable blocks in the Designer view of a diagram.

Each prototype contains:

* 1. Block class
  2. Icon to use in the palette
  3. A label
  4. Basic rendering style (Square ,Diamond ,Entry, Circle . . .)
  5. Anchor points (stubs)
  6. Icon embedded in drawing, if any
  7. Label rendered on drawing, if any

1. Block View. Selection from the palette creates a *ProcessBlockView*. This is the visible rendering of a process block. When editing a block, the model in the Gateway is queried to obtain a list of properties for that block. The property list is not transferred via the palette prototype.
2. Serializable Block. When a “save” operation is requested, the *ProcessBlockView* objects are converted into *SerializableBlock* instances and transferred to the Gateway. Serialization is required because the transfer may span different systems. If a project is exported, the export file contains *SerializableBlock* instances.
3. Process Block. When the Gateway receives a project update, the *SerializableBlock* instances contained in the project resource are converted into *ProcessBlock* instances and added to the “live” diagram.

Ignition provides an XML-based mechanism for serialization. However, this was not compatible with Java-generics (strongly typed lists) and was abandoned for that reason.

JavaScript Object Notation (JSON) is an alternative text-based solution. Serialization/deserialization is handled by an open-source package named “Jackson”. It is distributed with an Apache license. JSON is used for all serialization within the toolkit.

## Custom Blocks

The design supports user-extension of the block repertoire built into the BLT module. These user-designed blocks are written in Python and stored within a separate Ignition project that can be viewed as a library of custom blocks and may be re-used in multiple applications.

## Model Definition

In a Model-View-Controller design, the model contains the definition of the core data structures, in our case, a block diagram. For the purposes of this application, the *model* is a *ProcessDiagram* object. It completely describes a diagram and can be used to render its display. It becomes serialized into a project resource. This resource is the single unifying structure that binds knowledge of a diagram among all three Ignition scopes.

In the Designer, the *File->Export* dialog lists all the resources connected with the current project. This dialog has been modified to include the custom toolkit model resource.

## Auxiliary Data

The toolkit may require accompanying information that is not used in the execution of the blocks themselves (e.g. database entries). As appropriate, a block definition may include specification of a custom editor to define and modify these properties. Auxiliary data are not visible except through these editors. We assume that these data are present in both the toolkit blocks and an external system (database). The question arises as to how to keep the two entities in-synch with each other. Issues arise when moving a toolkit diagram from one system to another.

“ExternalData” is a Python-friendly dictionary structure that can be part of any block. This auxiliary configuration information in no way affects the execution of the block. The block is simply used as a convenient repository when moving a diagram from one system to another.

Consider the following cases:

### Normal operation/Editing

Under normal circumstances, the external system (database) and toolkit are presumed to be in-synch. There is no special mechanism to keep them that way. The operating premise is that the normal way to edit these attributes is with the custom editors provided as part of the toolkit. On “Save” these editors will update both the external system and the auxiliary data stored in the toolkit. If the external system is edited directly, this is considered to be “out-of-bounds”. The person making the edits is also then expected to select a “Refresh Auxiliary Data from Database” menu choice on each affected application in the toolkit to synchronize the toolkit.

### Transfer a project

A project containing toolkit resources is moved to a different environment. Perhaps this is from production to off-site for test/debug/analysis. Perhaps this is from a development area to a production system. Perhaps the database connection has been redefined in the Gateway.

The toolkit the module has no way of “knowing” when the project is started whether or not this is a different environment from when it was last saved. In this scenario, the user must explicitly select a “Refresh Database from Auxiliary Data” menu choice on each application in the toolkit where a refresh is desired.

### Export/Import

In a similar manner to the situation described above, an ccordingly is moved between environments. However instead of the transfer being inside an Ignition project, the application has been exported in .json format from one system and imported into a second. In this instance, the toolkit framework automatically refreshs the database on the second system.

### Isolation Mode

On a transition to “Isolation” mode, the toolkit framework automatically updates the isolation-mode database from the information stored in the diagram. When returning to “Production” mode, no action is taken. The production database remains unchanged.

### Fresh Migration

In a delivery of a freshly-migrated applications, the applications are pre-populated with pertinent auxiliary data. The import of the application’s .json file is equivalent to the import case described above. No additional user action is required. The database refresh is performed automatically.

### New Blocks

A required part of creating new block instances in the toolkit is to visit any associated the custom editors. Other than completing configuration through these editors, there is no additional work required on the users’ part to synchronize the external database with the diagram.

## Gateway Scope

For the purposes of the toolkit, the *Gateway* contains the engine that executes the logic blocks. Block logic is retained in separate Java or Python class instances. These classes are the focus of any custom development to extend the toolkit.

This design allows the block logic to be saved and restored just as any other project resource. The gateway is a listener on project resource changes. This is the mechanism for remaining in synch with the Designer. A project update is transmitted to the Gateway upon a “save” action in the Designer.

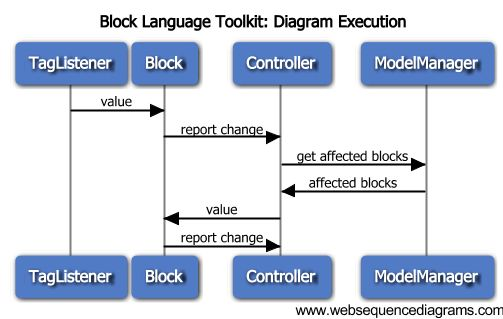


Figure 3 – Sequence Diagram

The diagram above depicts the sequence of operations when the tag listener detects a change that is bound to a block property. The activities all take place within the Gateway scope. The primary actors are:

* TagListener – The TagListener is configured to listen for changes to all tags that are bound to properties of blocks within a diagram. On detection of a change, the appropriate block is notified.
* Block – a process block. On detection of a new value on its input, the block processes the new value and, if appropriate, places a new value on its output.
* Controller – Block Execution Controller. The controller is the core dispatcher, accepting inputs and deciding what happens next. In this case, the controller asks the ModelResourceManager which blocks are connected to the output of the block that reported the change. It then notifies those blocks of the new value on their inputs.
* ModelManager = Model Resource Manager. This instance maintains collections of diagrams. The diagram instances are generated whenever a block-model project resource change (or addition) is detected. On request from the controller. The manager asks a diagram for a list of blocks downstream from a given block.

It should be noted that, in the sequence above, the block that receives the initial value update from the tag listener, is probably not to be the same block that receives the value change from the controller.

## Designer

The *Designer* contains all code for creating and modifying diagrams.

### Navigation Tree

Whenever the BLT module is loaded into the Gateway, a “Diagnostic Toolkit” node appears in the project resource area. This is the root of a tree structure that supplies access to the components (Applications, Families, Problems, Folders) of a Diagnostic Toolkit application, or set of applications,

When any component of the navigation tree is selected, the BLT workspace is displayed. This workspace is distinct and separate from the workspace that presents ccord components.

The dirty state of components of the

### Saving

Saving refers to synchronization of the diagram view in the designer with the “live” version of that diagram and its blocks in the gateway scope. As shown in the diagram below, there are 3 components, each representing the diagram, which must be kept in synch.

* Diagram View – the Designer scope picture of the diagram
* Diagram Resource – the resource that is saved and restored with the project
* Diagram – “live” executing diagram, Gateway scope

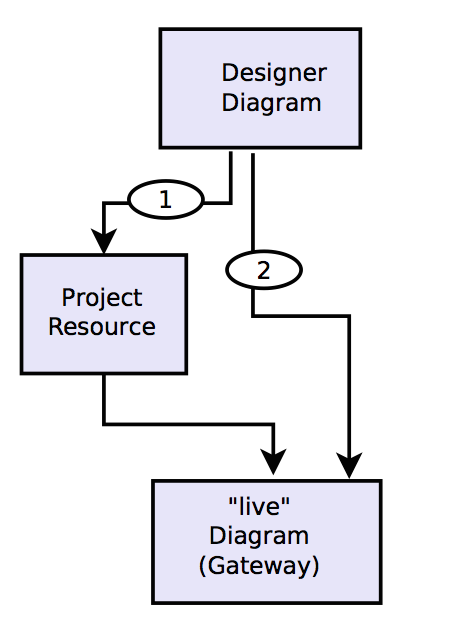


Figure 4 – Save Paths

The normal save path is the route indicated as (1). A “save” action in the Designer converts the view into a project resource, saving it in the project file. The same action generates a event in the gateway prompting the model manager to update the currently executing diagram object.

The alternate update route (2) is used when the user makes non-structural changes to the diagram or its blocks, such as property value changes. These are transmitted directly to the executing copy of the diagram, but leave the project resource out-of-synch.

The need to save is indicated with italicized names in the navigation tree. This indicates a mismatch between the designer view of the node and the project resource. In the case of a diagram view, a mustard background indicates that the designer view is out-of-synch with the actual diagram executing in the gateway.

There are several menu selections ccordi “Save”, each with different behaviors.

* “Save and Publish”. This menu on the main menu bar saves all project resources, including those resources associated with diagnostic toolkit blocks. Unlike the saving of Vision resources, the toolkit resources are saved whether or not they are currently displayed on a workspace tab.
* “Save All”, is a menu selection on the top node of the Diagnostic Toolkit navigation tree. It results in a save of all applications and diagrams to the project. Note that this action has no effect on Vision or other resources not associated with the Block Language Toolkit, nor is any action taken if the designer and project resource are in synch. The menu selection is always enabled. Open diagrams are saved as they currently appear in the editor.
* “Save”, on an Application, saves of all diagrams and blocks associated with that application. As with the previous, only resources associated with the Block Language Toolkit are ccordin. This selection is enabled only if one or more resource in the Application tree has a pending change.
* “Save”, on a Diagram updates the project with properties of that Diagram and of its all blocks. This selection is disabled until the parent application has been saved for the first time. The reason for this behavior is that, until the parent Application is saved, the controller has no knowledge of that diagram. The menu selection is also disabled if there has been no structural change to the diagram or any of its blocks.
* “Save”, on a Block updates properties of that Block directly to the controller. This selection is disabled if there has been no change to the block’s properties. It is also disabled if the diagram is not known to the engine (it has never been saved).
* On closing a dialog tab the user is asked whether or not the diagram should be saved as shown or reverted to its state before the tab was opened..
* Editing block properties in the generic block editor has an immediate effect on the “live” block, but is not permanently saved with the project until the block, parent diagram or application is saved.
* When a node (application,folder, family or diagram) is added or deleted, the change is propagated to the gateway immediately and also updated in the project resource.
* A newly imported dialog is set to “disabled” to allow a review of its contents before going “live”. A newly imported application is immediately “live”.

### Visual Synchronization

In order to create the illusion that the user is viewing the “real” blocks in the Designer, the Designer view must be carefully synchronized with the Gateway, both when initially displayed and in real time.

The mechanism to accomplish this coordination is as follows:

1. In the designer there is a push notification listener and cache. It detects changes as a diagram executes in the gateway. The listener is notified whenever a value is placed on the input of a block. In the designer this is equivalent to a connection receiving a value.
2. When a diagram is open in the designer, each of its connections subscribes to the push listener/cache for relevant changes.
3. When a diagram is opened in the designer,  each of its connections queries the push listener/cache for current value.
4. When the designer is opened, the push listener/cache queries every diagram in the gateway for the current state of each of its blocks.
5. Each block in the gateway, when queried, responds with the last-known value on each of its outputs.
   1. Some blocks have a “value” property bound to the ENGINE that persists when the block is saved (e.g. an input). These blocks are able to respond with state even after a gateway restart.
   2. Some blocks (like observation blocks) retain state as an internal (transient) variable.  After a gateway reset, these blocks have no current state (until a new value arrives)
   3. Some blocks (like junction blocks, test points) are simply pass thru and don’t retain any state. After a designer restart, these blocks cannot report connection state to the drawing.

## Client

*Several Client views are provided for monitoring the state of logic blocks and other results.*

# Gateway

The *Gateway* “runs” diagrams defined in the *Designer* scope. It is the keeper of the “model”, which is a description of the blocks in the diagrams, their attributes and states, and the connections between them.

While it may be tempting to think of the Gateway as a “running engine”, in fact, the Gateway code merely listens to asynchronous events and responds accordingly. This is shown in the sequence diagram, Figure 3.

There are two live elements of the engine, a watchdog thread used for block input synchronization, and a tag subscription thread. The engine itself runs a bounded buffer that collects input and processes the resultant output.

## Gateway Functions

The subsections below describe the major controller classes in the Gateway scope.

### Dispatcher

The *GatewayRpcDispatcher* registers on startup as the receiver of RPC requests from client or designer components.

### Resource Changes

The *ModelResourceManager* is a project change listener. It detects updates to project resources that hold diagram model definitions. On resource change, it deserializes the model and informs the engine of the changes. Resource changes are propagated by Ignition whenever the user selects “Save” on an Application, or “Save All” on the browser root node. In the case of “Save All”, a synchronization takes place to guarantee that there are no missing or extraneous resources in the Gateway due to some abnormal condition.

### Block Execution

The *BlockExecutionController* follows the Singleton design pattern. It is the “engine”. Being a Singleton provides a well-known address for the object from anywhere in the Gateway. The engine is called when a block completes evaluation. Its function is to determine the block or blocks that are next to execute. The selected blocks are provided with the new output value, their input, then the *evaluate()* method is invoked.

### Tag Changes

The Gateway *TagListener* subscribes to tags that are identified as block inputs. When the tags change, the handler informs the appropriate block instance of a property change.

### Block State

All blocks have a state, a truth-value. A truth-value has possible values of UNSET, UNKNOWN, TRUE, or FALSE. In practice, the state is meaningful only for those blocks that conclude an output truth-value. For other blocks the value remains UNSET.

Block state is accessible through the scripting interface and is visible in the user-interface through the “View Internals” menu selection on an individual block.

# Designer

The *Designer* is the only scope where changes to a diagram are supported.

## NavTree

The Designer’s navigation tree contains a “Diagnosis” node. Use this root node to create new applications and under them, new diagrams.

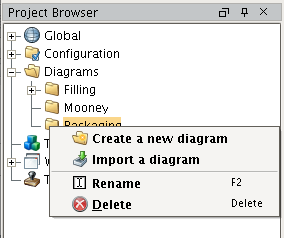


Figure 5 – Navigation Tree

The sections below summarize the available menu options for each node type.

### Root Node

The root node is a single top-level entry in the project browser.

* *Create a new application* – create a folder node that will contain a collection of diagram nodes.
* *Start engine/Shutdown* – only one of these options is enabled at a time. Starting the engine enables tag subscriptions for bound properties, starts the watchdog timer thread and establishes the bound buffer that accepts block output actions. Shutting down unsubscribes to tags, terminates the watchdog thread and shuts down the processing buffer within the engine.
* *Clear controller*. This selection is present purely for debugging. It deletes any diagrams in the controller. An application “Save” can then be used to add a single application to the execution engine. This expedient makes debugging easier as it removes confusion due to other live application that may even belong to different projects. This action has no permanent effect – a Gateway restart will restore the prior state.
* *Debug to log* – write a description of current project resources to the Ignition designer log. Write a similar description of resources known to the Gateway, This comparison is useful only during development.

### Application Nodes

Children of the root node are “Applications”.

* *Create a new family* – create a family node. A family is a container for problem nodes. Multiple problem nodes may be created under the same application.
* *Clone diagram* – *clone the current active diagram. This differs from an import in that the states of the blocks in the cloned diagram are identical to those in the original.*
* *Rename* – change the name of the application.
* *Delete* – remove the application node and all diagrams below it.

### Family Nodes

Family nodes appear between “Application” and “Problem” nodes. There may be an arbitrary number of intervening folder nodes on either side of a family.

* *Create a new diagram* – create a new problem node and accompanying diagram. A workspace is a container for blocks that will make an executable diagram. Multiple diagrams may be created under the same application.
* *Import a diagram* – display a file browser that allows entry of a diagram name. On selection of a file, attempt to marshal it and create a new diagram.
* *Clone diagram* – *clone the current active diagram. This differs from an import in that the states of the blocks in the cloned diagram are identical to those in the original.*
* *Rename* – change the name of the application.
* *Delete* – remove the application node and all diagrams below it.

### Folder Nodes

Folder nodes are simple containers.

* *Create a new diagram* – create a new problem node and accompanying diagram. A workspace is a container for blocks that will make an executable diagram. Multiple diagrams may be created under the same application.
* *Import a diagram* – display a file browser that allows entry of a diagram name. On selection of a file, attempt to marshal it and create a new diagram.
* *Clone diagram* – *clone the current active diagram. This differs from an import in that the states of the blocks in the cloned diagram are identical to those in the original.*
* *Rename* – change the name of the application.
* *Delete* – remove the application node and all diagrams below it.

### Problem Nodes

Each Problem node contains a diagram that schematically defines the problem analysis.

* double-click – opens the diagram workspace.
* *Export diagram* – display a file. On selection of a file, serialize the diagram and write it to the specified file path.
* *Rename* – change the name of the diagram.
* *Delete* – remove the diagram node and associated workspace.

## Menu

The Designer main menu has been enhanced with an additional function. When any node in the “Diagrams” tree is selected, the “Reset Panels” sub-menu will reset the visible workspace for the Block Language Toolkit. This includes hiding Vision windows, showing a tabbed pane center area that holds diagrams and also showing the block palette.

## Palette

The palette is a Designer (Wicket) panel made viewable by a double-click action anywhere in the toolkit Nav tree or by the menu selection described above. The palette is a tabbed pane. The tab designations are deteremined by whatever is specified in the block prototype definitions.

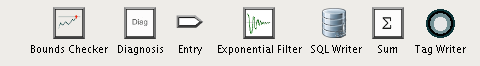


Figure 6 – Palette

## Icons

All icons used in the toolkit reside in the Gateway (they are not project-specific). They may be accessed via the Ignition Designer “Icon Management” tool and, through it, be freely imported from and exported to files. This provides for easy user-modification.

The icon path is of the form: Block/icons/*nn*/name\_*nn*.png, where nn is the icon dimension in pixels. In general icons that look good at a 32x32 pixel resolution are appropriate for the application. Icons of different pixel dimensions will be translated as needed.

## Diagrams

The diagram window is tabbed pane located in the center of the Designer workspace area.

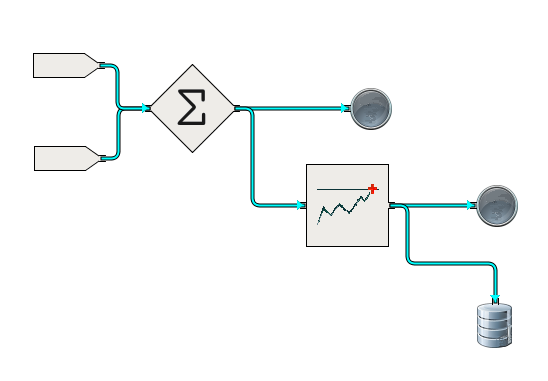


Figure 7 – Sample Diagram

### Diagram State

The underlying block execution engine is, by default, “running”. In this condition, tag subscriptions are active and inter-block traffic is delivered to ccording blocks.

The engine is started when the BLT module is loaded and shutdown when the module is unloaded (usually during Ignition startup/shutdown). The engine running state may be also controlled via a designer menu selection. This action is typically used only during development.

Individual diagrams exist in one of three states:

1. Active. This is the normal state. The blocks in the diagram respond to tag value updates that modify bound properties. They respond to incoming data at their inputs, execute ccordingly and post results on their outputs.

When gateway and designer versions of that diagram are “in-sync”, that is there have been no changes in the designer version since the last “save” action, then block state changes are transmitted to the Designer scope via the Ignition “push notification” mechanism. These are used to animate the diagram view.

1. *Disabled.* The user may explicitly set the state of a diagram to “disabled” via a menu selection in the Navigation tree.When a diagram is disabled, block properties bound to tags cease to be updated and no data/messages are delivered to blocks within that diagram. *When the diagram is moved out of the” disabled” state, the diagram is reset.*

The disabled state in no way inhibits the ability of a user to edit the diagram.

1. Isolated. When in a “isolated” state, blocks in the diagram interact with a completely separate database and set of tags from production. In this way a diagram in this state is prevented from making “live” updates. The isolation database connection and tag provider are configured in the Designer *View* menu. When the Block Language Toolkit module is loaded this menu will contain a custom entry for “External Interface Configuration”. This allows definition of production and isolation database connections and tag providers. Once set, these values persist in the Ignition internal database.

### Dirty

A diagram becomes “dirty” when an edit action changes block and/or connection configuration. In a “dirty” state, the version of the diagram executing in the Gateway does not correspond to the version in the Designer. In this state, notifications coming from the Gateway are ignored. Additionally, the diagram background is muddied to indicate this state to the user.

The “dirty” condition is exited when the user “saves” the diagram either individually, via a parent application or via the main menu (which saves all project resources/diagrams).

New diagrams and newly imported diagrams are treated as “dirty” because, until a “save” action, there is no corresponding entity in the Gateway to actually execute.

A workspace tab that shows a “dirty” diagram cannot be closed without the user agreeing to save or discard the changes. In this instance, “save” means that the diagram changes are recorded into the project resource.

### Reset Action

A “reset” action on a diagram results in the following sequence of actions:

1. Disable: The diagram is disabled, thereby cancelling all tag subscriptions related properties of blocks within the diagram.
2. Block reset: A “reset” signal is sent to all blocks within the diagram. Each block is expected to appropriately set its internal state. This usually results in setting its state to unknown and clearing any internal data caches.
3. Connection reset: All blocks with outgoing truth-value connections propagate an UNKNOWN state along those connections.
4. Enable: The diagram is finally re-enabled (returning to its previous active or isolated state). Tag subscriptions are restored and each subscribing property receives the current value. Input blocks propagate the current tag value on their outputs.

Note: If the diagram is currently in a “disabled” state, a reset action has no effect.

## Blocks

This section describes characteristics of blocks within the block language.

### Block Behavior

With few exceptions, all blocks respond to the following action commands:

* evaluate – force propagation of the latest values on all outputs
* force – place a value on the selected output. Once propagated, the block is placed into a locked state.
* lock/unlock – a *locked* block will not propagate a value on its output(s)
* reset – initiate a *reset* action on the block. Reset behavior is dependent on the block type.

*These actions can be triggered via a user menu associated with the block or via a signal connected directly to the block. The signal stub is enabled/disabled interactively by the operator. By default it is not displayed.*

### Block Definitions

This section describes specific characteristics of each block type.

| Block | Palette | Description |
| --- | --- | --- |
| Action | Misc | Execute a configurable Python script |
| And | Logic | Propogate the logical “and” of the inputs |
| Arithmetic | Arithmetic | Perform one of meny arithmetic functions on the input |
| Bias | Arithmetic | Add a specified constant to the input |
| Command | Connection | Send a specified signal |
| Compare | Arithmetic | Compare two data values and return a truth-value |
| Compare Absolute | Arithmetic | Compare the absolute values of two data inputs and return a truth-value |
| Compare Deadband | Arithmetic | Compare the values of two data inputs considering a deadband zone. |
| Data Conditioner | Misc | Apply additional quality constraints to the input |
| Data Pump | Connection | Emit a value on a timed interval |
| Data Shift | Control | Buffer incoming data for a fixed sample size |
| Delay | Control | Introduce a delay into the data flow |
| Difference | Arithmetic | Subtract one input from the other |
| EdgeTrigger | Control | Detect a truth-value change and hold for a specified interval |
| Encapsulation | Connection | Route inputs to configured connection posts. Accept connection post values and transfer to output. |
| Entry Connection | n/a | Connection post that accepts input on an EncapsulatedDialog |
| Equality Observation | Observation | Compare the input to a fixed target value |
| ExitConnection | n/a | Connection post that propagates output on an EncapsulatedDialog |
| Exponential | Arithmetic | Compute ”e” to the value of the input. |
| Exponential Filter | Arithmetic | Perform an exponentially weighted moving average on the input. |
| Final Diagnosis | Analysis | A speciality block for determining a recommendation based on the sense of the input |
| Gain | Arithmetic | Multiply the input by a specified constant |
| High Limit Observation | Observation | Compare incoming data with a configured upper limit and optional deadband |
| High Limit Sample Count | Observation | Return true if m of the last n samples are greater than a specified limit |
| High Limit Time Window | Observation | Return true if n samples within a specified time window are greater than a specified limit |
| High Selector | Analysis | Pass the maximum value observed across a number of inputs |
| Inhibit | Control | Discard input older that a specified date |
| In Range Observation | Observation | Compare incoming data with a configured range and optional deadband |
| In Range Sample Count | Observation | Return true if m of the last n samples were outside a specified range |
| In Range Time Window | Observation | Return true if n samples within a specified time window are within than a specified range |
| Input | Connection | Subscribe to a tag and propagate its value changes |
| Input Median | Arithmetic | Return the median of all inputs |
| Inverse | Arithmetic | Compute the inverse of the input (1/x). |
| Junction | Connection | A block that passes information through without change, a null operation |
| Lab Data | Connection | A special input block that reads value and timestamp from separate tags |
| Ln | Arithmetic | Compute the natural ogarithm of the input |
| Log10 | Arithmetic | Compute the ogarithm base 10 of the input |
| Logic Filter | Analysis | Monitor the ratio of time true to time false over a specified interval. |
| Logic Latch | Logic | Hold a the most recent true/false through a reset |
| Low Limit Observation | Observation | Compare incoming data with a configured lower limit and optional deadband |
| Low Limit Sample Count | Observation | Return true if m of the last n samples are less than a specified limit |
| Low Limit Time Window | Observation | Return true if n samples withn a specified time window are less than a specified limit |
| Moving Average | Statistics | Maintain an average of all points that have arrived at the block |
| Moving AverageSample | Statistics | Maintain a rolling average of the most recent n points that have arrived at the block |
| Moving AverageTime | Statistics | Maintain a time-weighted rolling average of points that have arrived within a time interval |
| N True | Logic | Return true if a the number of true inputs meets a threshold |
| Not | Logic | Invert a truth-value |
| Note | Misc | A text box for commentary on the diagram |
| Or | Logic | Propogate the logical “or” of the inputs |
| Out of Range Observation | Observation | Compare incoming data with a configured range and optional deadband |
| Output | Connection | Write value received on input to a tag. |
| Quotient | Arithmetic | Divide the value on one input by the value on the other |
| Parameter | Connection | Encapsulate a tag. |
| Persistence Gate | Control | Guarantees that its input values are unchanged for a specified period |
| PID |  | Perform PID control on an input |
| Product | Arithmetic | Compute the product of numerical inputs |
| Property Setter | Misc | Set a specified property on a connected downstream block. |
| QualifiedValue | Connection | Create a qualified value from eparate value, quality, time streams |
| Quotient | Arithmetic | Compute the quotient of two inputs |
| Random | Statistical | Inject random noise into the input |
| Range Observation | Observation | Compare incoming data with a configured range and optional deadband |
| Readout | Misc | Display the current value of a data connection |
| Receiver | Connection | Accept a signal and propagate on an output connection |
| Reset | Control | Send a reset signal |
| Sink Connection | Connection | Accept a data or truth value for routing to source connections |
| Source Connection | Connection | Receive data or truth values from “the system” and propagate on the output. |
| SQC | Analysis | Perform a SPC calculation. Execute one of the Westinghouse rules. |
| SQL | Connection | Write incoming value to a database |
| Sum | Arithmetic | Add values on a variable number of inputs |
| TimeReadout | Misc | Display the time of the last value passing through the block |
| Timer | Control | On reset, transmit a configured truth for a specified interval |
| TimeFork | Misc | Pass the timestamp of the incoming data on the output |
| Trend Detection | Analysis | Apply SQC-like rules to detect trends |
| Transmitter | Connection | Transmit a signal to other blocks, either in the same diagram or not |
| Truth Value Pulse | Control | Transmit a momentary truth value |
| Unknown | Logic | Return TRUE if th input is UNKNOWN |
| Zero Crossing | Observation | Detect a sign change in the inpuit. |

Note that in the observation blocks, a TRUE value indicates an abnormal situation. Time intervals are always expressed in seconds as a *double* value. This allows fractional seconds to be used, usually for testing.

#### Action

* When a triggering condition is detected on the input, this block executes a custom Python script. Specifications for the script are contained in Appendix E. On execution, the script is given the originating block as its single argument.

Properties:

Script – full package and module name of the script to be executed.

Trigger – truthvalue that, when received, causes the block to execute the script

Connections:

* in – incoming truth value.
* out – truth-value, pass the input through the block.

#### And

* Propagate the logical “and” of the inputs. This block expects multiple connections on its input port. It accomodates any number. On a reset, the block clears all information regarding its input connections and sets its current state to UNKNOWN. The output is given a GOOD quality unless the block state is UNKNOWN. In that case the quality is set to the “worst” quality of any of the inputs. On a reset, existing values are retained, and an UNKNOWN value is assigned the output state. However, no new output is emitted until the next new value is received.

Attribute Display:

Value – the current state of the block

Properties:

SyncInterval – a “coalescing interval”. This is the time allowance for multiple values that have originated in the same scan to be evaluated at the same time. ~ secs. The default value is 0 (no synchronization).

Connections:

* in – truthvalues. Multiple upstream blocks may be connected to the single input.
* out – truthvalue representing the “and” of the inputs. The timestamp is updated.

#### Arithmetic

* Apply a custom arithmetic function. The function is written in Python and must accept a qualified value for its input. It must generate a qualified value for the output. The custom code has full access to the Apache commons-math library. There is no reset behavior. The code sample shown in Appendix D is an initial implementation encompassing the following functions:
  + *abs*
  + *round*
  + *floor*
  + *ceiling*
  + *sine*
  + *cosine*
  + *tangent*
* The function is designed to be user-modifiable. The current version is found in external python, in *xom.block.arithmetic.py.*

Properties:

Function – name of the function to be executed. This is a string value chosen from the list of available functions

*Connections:*

* *in – incoming data value.*
* *out – double value, the function result. The output timestamp is updated.*

#### Bias

* Change the value of the input by a configured constant and propagate the result. Only a single input connection is expected. There is no synchronization issue, nor reset behavior.

Properties:

Bias – the value offset.

Connections:

* in – data value.
* out – data value that is the input plus a specified factor. The timestamp is not altered from the incoming value.

#### Command

* When a triggering condition is detected on the input or the signal connection receives a START, this block propagates a configurable command on the output.

Properties:

Command – the command to be sent. By default the signal is “reset”.

Trigger – the value to be matched on input in order to trigger the output. Default value is TRUE.

Connections:

* in – a truth value. This triggers the output if it matches the trigger value
* out – the signal.

#### Compare

* Compare two numeric or date values, “x” and “y”. The output is a truth-value, true if x>=y. On reset the inputs are set to not-a-number. This requires that new values must be received on both inputs before the block will emit a new result.
* The inputs may be either numeric values or dates. Comparing a numeric value to a date is not allowed. For a date, “greater than” implies more recent.

Properties:

Offset – an offset added to “Y” before the comparison. Default value is 0. If dates are being compared, this value is interpreted as milliseconds.

SyncInterval – a “coalescing interval”. This is the time allowance for multiple values that have originated in the same scan to be evaluated at the same time. ~ secs. The default value is 1 sec.

Connections:

* x – data connection, the first input.
* y – data value, the second input
* out – truthvalue connection, the result.

#### Compare Absolute

* This block is identical to the Compare block woth the exception that a comparison is made of the absolute values of the inputs.

#### Compare Deadband

* Compare two input data values, “x” and “y”. The output is a truth-value, true if x>=y and false if x<y – deadband, where the deadband is a positive configured constant. If the value lands within the deadband zone, then return the previous value. On reset the inputs are set to not-a-number. This requires that new vaules must be received on both inputs before the block will emit a new result.

Properties:

Deadband – a value that defines a tolerance region below “Y”. Default value is 0.

SyncInterval – a “coalescing interval”. This is the time allowance for multiple values that have originated in the same scan to be evaluated at the same time. ~ secs. The default value is 0 sec.

Connections:

* x – data connection, the first input.
* y – data value, the second input
* out – truthvalue connection, the result.

#### Data Conditioner

* Allow only good quality values to propagate. The quality is determined by the quality of the incoming value and by the state of a secondary quality line. A Boolean value of FALSE (no trouble found) implies a good quality. A result is propagated only if the main value is of good quality and the secondary input is itself of good quality and not TRUE. On initialization and reset, the secondary input is internally set to UNKNOWN.
* The timestamp of the output is the same as the incoming data value.

Properties:

SyncInterval – a “coalescing interval”. This is the time allowance for multiple values that have originated in the same scan to be evaluated at the same time. ~ secs. The default value is 0 sec.

Connections:

* in – data value. The data value to be considered
* b – truth value. The additional quality value to apply.
* out – data value, a qualified value with always “good” quality
* status – truth-value. TRUE if a problem is detected regarding the data quality.

#### Data Pump

* Emit a fixed (or bound) value on a configured interval. The output is stamped with the current time. The block is initially inactive waiting until the interval property receives a value. Additionally, the block will not cycle if the the interval is less than or equal to zero.
* The value property may be updated at any time. An new value is propagated immediately. On reset, the block restarts its cycle (assuming that both interval and value are valid. If the value is bound to a tag, then the block essentially polls the value of the tag.

Properties:

Interval – interval at which a value is generated on the output ~ secs. The default value is Double.NaN. The block will not begin polling until a valid interval is configured.

LiveOnStart – If the interval and value are defined, and this parameter is TRUE, the block will begin emitting values when it is started. Otherwise the value will not be emitted until there is a change to either value or interval properties.

Value – the value to be emitted by the block. Datatype is OBJECT, meaning it is alterable based on the connection type.

Connections:

* out – output anchor. The connection type may be set after the block is placed on the workspace. The emitted value is coerced to a datatype compatible with the connection.

#### Data Shift

* Store and hold incoming data values in a fixed size FIFO buffer. When the buffer fills, additional incoming data are placed at the buffer head and the oldest data are placed on the output connection. A reset action clears the buffer.
* Values with bad quality are ignored.

Properties:

SampleSize – the number of input values to store. The default value is 0 – which results in pass-through behavior.

Connections:

* in –data to be processed.
* out – data released when the buffer is full. The original timestamps of the data values are retained.

#### Delay

* Store and hold incoming data values for a specified period. The block allocates memory as necessary in order to hold values waiting for their respective timeout. This block accepts any input type. A reset action deletes all queued information.

Properties:

SampleDelay – the length of time for which input values are delayed. ~ secs. The default value is 0.

Connections:

* in – raw data to be analyzed. Any datatype is accepted.
* out – data, truthvalue or string. The output is the input value without change (just delayed). The original timestamps of the data values are retained.

#### Difference

* Compute the difference of two inputs. Report the value a port “a” minus the value of the port labeled “b”. On a reset, all existing values are forgotten.
* If the data types of both inputs are Date, then the result will be the difference in time between the inputs, expressed as seconds. If only one of the input data types is a date, the output will be *NaN* with a bad quality.

Properties:

SyncInterval – a “coalescing interval”. This is the time allowance for multiple values that have originated in the same scan to be evaluated at the same time. ~ secs. The default value is 1 sec.

Connections:

* a – data value. The number to subtract from
* b – data value. The number to subtract.
* out – double value representing the difference of the inputs. The output timestamp is updated.

#### Edge Trigger

* When an incoming truth-value matches the trigger value, then the output reflects that truth-value for the specified interval before returning to the inverse value. There is no synchronization issue. On reset the timer is cancelled and the prior value is propagated to the output. Initially the value is unset until a new value arrives on the input (and then that value becomes the output).
* New values arriving during the hold period are ignored. Changing the hold interval does not affect any interval in progress.

Properties:

HoldInterval – The interval ~secs during which the output is delayed. Default is zero, which results in a short pulse.

Trigger – the incoming value which starts the hold interval.

Connections:

* in – truth value.
* out – input value held for the specified interval, then returned to its inverse.

#### Encapsulation

* *An Encapsulation block represents an EncapsulatedDiagram in the daigram superior to it. An Encapsulation is the only block that has runtime-configurable anchor points. On creation of an Encapsulation an associated EncapsulationDiagram is created and linked into the Navigation tree. When anchor points are configured on the encapsulation, associated Entry and/or Exit Connections are created on the sub-diagram.*

*Connections: There are a variable number of connections, configured on the encapsulation at run-time.*

#### Entry Connection

* *This is a connection post that is automatically generated when an Encapsulation block is created. It appears on the associated EncapsulatedDiagram associated with the encapsulation. There is one EntryConnection for each incoming connection to the encapsulation.*

*Properties:*

Label – This label is read-only. It matches the name of the anchor point of the parent encapsiulation associated with this block.

*Connections:*

* out –value derived from the parent encapsulation.

#### Equality Observation

* Validate the input against a configured target value and within a specified deadband as shown by the truth-table below. There are no synchronization considerations, nor reset behavior.

|  |  |
| --- | --- |
| TRUE | FALSE |
| X >= Nominal – Deadband/2  and  X <= Nominal + Deadband/2 | X < Nominal – Deadband/2  or  X > Nominal + Deadband/2 |

Properties:

Nominal – the nominal target value. The default is zero.

Deadband – a tolerance region around the target (½ above and ½ below). Default value is 0.

.

Connections:

* in – data connection, the input.
* out – truthvalue connection, the result.

#### Exit Connection

* *This is a connection post that is automatically generated when an Encapsulation block is created. It appears on the associated EncapsulatedDiagram associated with the encapsulation. There is one EntryConnection for each outgoing connection on the encapsulation.*

*Properties:*

Label – This label is read-only. It matches the name of the anchor point of the parent encapsiulation associated with this block.

*Connections:*

* in –value to be routed to the parent encapsulation.

#### Exponential

* Raise “e” to the value of the input and propagate the result. Only a single input connection is expected. There is no synchronization issue, nor reset behavior.

Connections:

* in – data value.
* out – data value that is e to the power of the input. The timestamp is not altered from the incoming value.

#### Exponential Filter

* Smooth the input with an exponentially-weighted-moving-average. There is one output for each input. On reset, the average calculation is initialized. Bad data is ignored in the moving average. Entries are weighed by the time-interval between inputs.

Properties:

TimeWindow – a time constant for the average ~ secs.

Connections:

* in – incoming data value.
* out – double value representing the input filtered as a moving average. The output timestamp is updated.

#### Final Diagnosis

* Run a configured procedure to determine a recommendation based on a TRUE value on the inpit.

Properties:

*??? – the multiplier.*

Connections:

* in – incoming truth value.
* out – passthru of the input
* diagnosis – text value that is the recommended course of action.

#### Gain

* Multiply the value of the input by a configured constant and propagate the result. Only a single input connection is expected. There is no synchronization issue, nor reset behavior.

Properties:

Gain – the multiplier.

Connections:

* in – data value.
* out – data value that is a specified factor times the input. The timestamp is not altered from the incoming value.

#### High Limit Observation

* Validate incoming input data values against a configured upper limit. The output is a truth-value determined per the table below.

|  |  |
| --- | --- |
| TRUE | FALSE |
| X > Limit | X < Limit - Deadband |

* For values that fall within the deadband range, retain the most recent past value. On reset the past value is set to UNKNOWN.

Properties:

Deadband – the value of the deadband, a non-negative number. Default = 0.

Limit – the value of the limit. Default = 0.

Connections:

* in – data connection, the raw input.
* out – truthvalue connection, the result.

#### High Limit Sample Count

* Test the last “n” samples input for values greater than a specified limit. For the period of time that the buffer is not filled and a FALSE cannot be assured, the result is dependent on the *FillRequired* property. If TRUE then the conclusion is UNKNOWN; otherwise the conclusion is FALSE.
* A hysteresis property controls the behavior with regard to deadband processing. Refer to the “LogicFilter” block for details.

Attribute Display:

Value – the current true ratio

Properties:

Deadband – the value of the deadband, a non-negative count. If the value is zero, then there is no deadband consideration. Default = 0.

Hysteresis – TRUE, FALSE, ALWAYS, NEVER. Default is NEVER.

Limit – the value of the limit, default is zero.

FillRequired – if TRUE, the state of the block is UNKNOWN until the buffer fills. Default is TRUE.

TriggerCount – the minumum number of samples that must be grater than the limit in order to evaluate as TRUE. Default = 0.

SampleSize – the number of points to include in the analysis. When the sample size is changed, the buffer is cleared.

Connections:

* in – truth value, the input being monitored.
* out – truth value, true if the input is true for more than the limit.

#### High Limit Time Window

* Test inputs within a specified time window for values greater than a specified limit. For the period of time that the buffer is not filled and a FALSE cannot be assured, the result is dependent on the *FillRequired* property. If TRUE then the conclusion is UNKNOWN; otherwise the conclusion is FALSE.
* If a new value is not received within the scan interval, then a value for that interval is generated as a copy of the last good value. Values of bad quality are ignored.
* A hysteresis property controls the behavior with regard to deadband processing. Refer to the “LogicFilter” block for details.

Attribute Display:

Value – the current count

Properties:

Deadband – the value of the deadband, a non-negative count. If the value is zero, then there is no deadband consideration. Default = 0.

Hysteresis – TRUE, FALSE, ALWAYS, NEVER. Default is NEVER.

Limit – the value of the limit, default is zero.

ScanInterval – the sampling interval ~ secs

TimeWindow – the period over which the analysis is to take place. ~ secs.

TriggerCount – the minumum number of samples that must be less than the limit in order to evaluate as TRUE. Default = 0.

Connections:

* in – truth value, the input being monitored.
* out – truth value, true if the count is greater than the limit.

#### High Selector

* Pass the maximum value observed across multiple inputs and propagate the result. Only values with good quality are considered. A *SyncInterval* property provides for synchronization across the different entry connections.. Data arriving when the block is locked are ignored. The timestamp of the output is taken from the maximum input. If multiple inputs have the same maximum the timestamp is indeterminate.

Properties:

SyncInterval – a “coalescing interval”. This is the time allowance for multiple values that have originated in the same scan to be evaluated at the same time. ~ secs. The default value is 0.5 sec.

Connections:

* in – data value.
* out – data value that is the maximum of all input values. When a maximum is propagated, the timestamp is equal to the timestamp of the input that is the maximum.

#### In Range Sample Count

* Test the last “n” samples input for values within a specified range. Return TRUE if the number of values outside the range exceeds a trigger count. For the period of time that the buffer is not filled and a FALSE cannot be assured, the result is dependent on the *FillRequired* property. If TRUE then the conclusion is UNKNOWN; otherwise the conclusion is FALSE.
* A hysteresis property controls the behavior with regard to deadband processing. Refer to the “LogicFilter” block for details.

Attribute Display:

Value – the current true ratio

Properties:

Deadband – the value of the deadband, a non-negative count. If the value is zero, then there is no deadband consideration. Default = 0.

Hysteresis – TRUE, FALSE, ALWAYS, NEVER. Default is NEVER.

LowerLimit – the value of the lower limit, default is zero.

FillRequired – if TRUE, the state of the block is UNKNOWN until the buffer fills. Default is TRUE.

SampleSize – the number of points to include in the analysis. When the sample size is changed, the buffer is cleared.

TriggerCount – the minumum number of samples that must be in-range in order to evaluate as TRUE. Default = 0.

UpperLimit – the value of the upper limit, default is zero.

Connections:

* in – truth value, the input being monitored.
* out – truth value, true if the input is true for more than the limit.

#### In Range Time Window

* Test inputs within a specified time window for values within a specified range. Return TRUE if the number of values outside the range exceeds a trigger count.For the period of time that the buffer is not filled and a FALSE cannot be assured, the result is dependent on the *FillRequired* property. If TRUE then the conclusion is UNKNOWN; otherwise the conclusion is FALSE.
* If a new value is not received within the scan interval, then a value for that interval is generated as a copy of the last good value. Values of bad quality are ignored.
* A hysteresis property controls the behavior with regard to deadband processing. Refer to the “LogicFilter” block for computation details.

Attribute Display:

Value – the current count

Properties:

Deadband – the value of the deadband, a non-negative count. If the value is zero, then there is no deadband consideration. Default = 0.

Hysteresis – TRUE, FALSE, ALWAYS, NEVER. Default is NEVER.

LowerLimit – the value of the lower limit, default is zero.

ScanInterval – the sampling interval ~ secs

TimeWindow – the period over which the analysis is to take place. ~ secs.

TriggerCount – the minumum number of samples that must be less than the limit in order to evaluate as TRUE. Default = 0.

UpperLimit – the value of the upper limit, default is zero.

Connections:

* in – truth value, the input being monitored.
* out – truth value, true if the count is greater than the limit.

#### Inhibit

* Discard any input that has a time-stamp more recent than a specified target. This block responds to an “inhibit” signal that starts the inhibiting period. The target date-time is computed based on an increment from the current time. A “reset” action resumes normal pass-thru behavior.
* A typical use of this block is for discarding laboratory measurements that may not apply to the current control regime. This block accepts any input type.

Properties:

InhibitInterval – time period measured from receipt of a triggering signal during which to discard input ~ secs. The default value is 0. A negative value sets the gating time in the past. This may still be appropriate to discard data arriving with time-stamps in the past.

Connections:

* in – data connection.
* out – data connection. When the block is operating in pass-thru mode, incoming data are propagated without change to the timestamp.

#### Input

* Subscribe to a tag and transmit new values.

Attribute Display:

Value – the most recent value read from the tag

Properties:

TagPath – to make any sense, this value should be bound to a tag.

Connections:

* out – data connection, the tag value.

#### Input Median

* Analyze values received on multiple input connections, compute the median and propagate to the output. This block uses the median() function from Apache commons math library.There is no special reset behavior.

Properties:

SyncInterval – a “coalescing interval”. This is the time to allow data to arrive on multiple inputs. It is also the no-data-change wait interval. The sync interval must be less than the scan interval.~ secs. The default value is 1 sec.

Connections:

* in – data. The expectation is that multiple connections will terminate at the port.
* out – data, the median value of the inputs.

#### Inverse

* Divide one by the input and propagate the result. Only a single input connection is expected. There is no synchronization issue, nor reset behavior. An incoming value of zero is ignored.

Connections:

* in – data value.
* out – data value that is one divided by the input. The timestamp is not altered from the incoming value.

#### Junction

* A junction block simply takes what appears on its input and transfers to its output. The purpose of this block is to provide a target during migration for those blocks from the legacy platform that have no meaning in the current setup.

Connections:

* in –value, any datatype is allowed.
* out – the input value is propagated immediately.

#### Lab Data

* This is a block that creates a qualified value from separate inputs, a value and a timestamp. The datatype of the output is the same as that of the value input. The input determines the default quality and timestamp as well. The block will emit an output for every value received on its input.
* The timestamp input has a datatype of either a date or a string. If a string, its format is specified by the FORMAT property of the block. This is a string date-time format as specified by the Java *SimpleDateFormat* class. The default value is: “YYYY/MM/dd hh:mm:ss”. If the format does not include a date (i.e. is a time only), the current date is assumed. If it does not include a time, the current time is assumed. The timestamp is taken as the last value received on the time input. If no input has ever arrived on the time channel, the timestamp of the output is assigned the timestamp of the value input.
* A synchronization interval should be specified to guarantee proper pairing of readings between the separate input channels. The synchronization is driven by the “value” input only. This prevents re-issuance of a value on receipt of quality or timestamps only.

Properties:

Format – string format for the date-time. If the “time” input is textual (as opposed to a date) it is converted to a timestamp via this format.

SyncInterval – a “coalescing interval”. This is the time allowance for multiple values that have originated in the same scan to be evaluated at the same time. ~ secs. The default value is 1 sec.

TimePath – tag path of the timestamp input. The data type may be either a date-time or string.

ValuePath – tag path of the value input. To make any sense, this value should be bound to a tag.

Connections:

* out – qualified value constructed from the three input channels.

#### Ln

* Take the natural logrithm of the input and propagate the result. There is no synchronization issue, nor reset behavior. Input values less than or equal to zero are ignored.

Connections:

* in – data value.
* out – data value that is the natural logrithm of the input. The timestamp is not altered from the incoming value.

#### Log10

* Take the logrithm base 10 of the input and propagate the result. There is no synchronization issue, nor reset behavior. Input values less than or equal to zero are ignored.

Connections:

* in – data value.
* out – data value that is the logrithm base 10 of the input. The timestamp is not altered from the incoming value.

#### Logic Filter

* Test the input for amount of time TRUE versus amount of time FALSE. Report TRUE if this ratio is greater than a specified ratio within a specified time interval. If sufficient time to conclude a TRUE result has not yet passed since the last reset, the block state is UNKNOWN.
* A “hysteresis” property controls the behavior with regards to the deadband as follows:

ratio = sum of time true / timeWindow

case (hysteresis) of

true:

if currentOutput = True then

threshold = limit - deadband

else

threshold = limit

false:

if currentOutput = True then

threshold = limit

else

threshold = limit + deadband

always:

if currentOutput = True then

threshold = limit - deadband

else

threshold = limit + deadband

never:

threshold = limit

end;

if ratio > threshold then

output = True

else

output = False

Attribute Display:

Ratio – the current true ratio

Properties:

Deadband – the value of the deadband, a fractional, non-negative number. If the value is zero, then there is no deadband consideration. Default = 0.

Hysteresis – TRUE, FALSE, ALWAYS, NEVER. Default is NEVER.

Limit – the minumum ratio of time that the input is true versus false Default = 0.

Ratio – the current fraction of time-true to the total time. This property has an ENGINE binding and is, therefore, visible in the designer by subscription.

ScanInterval – the sampling interval ~ secs

TimeWindow – a sliding time window over which the ratio is calculated ~ secs

Connections:

* in – truth value, the input being monitored.
* out – truth value, true if the input is true for more than the limit.

#### Logic Latch

* The function of the latch block is to preserve TRUE or FALSE values following a diagram reset. The block does not respond to UNKNOWN or UNSET. It will propagate only TRUE or FALSE values that are of good quality. Upon reset, the block will re-tranmiit its latest TRUE or FALSE value.

Connections:

* in – truthvalue.
* out – the latest TRUE or FALSE value.

#### Low Limit Observation

* Validate incoming input data values against a configured lower limit. The output is a truth-value determined per the table below.

|  |  |
| --- | --- |
| TRUE | FALSE |
| X < Limit | X > Limit + Deadband |

* For values that fall within the deadband range, retain the most recent past value. On reset the past value is set to UNKNOWN.

Properties:

Deadband – the value of the deadband, a non-negative number. Default = 0.

Limit – the value of the limit. Default = 0.

Connections:

* in – data connection, the raw input.
* out – truthvalue connection, the result.

#### Low Limit Sample Count

* Test the last “n” samples input for values less than a specified limit. For the period of time that the buffer is not filled and a FALSE cannot be assured, the result is dependent on the *FillRequired* property. If TRUE then the conclusion is UNKNOWN; otherwise the conclusion is FALSE.
* A hysteresis property controls the behavior with regard to deadband processing. Refer to the “LogicFilter” block for details.

Attribute Display:

Value – the current true count

Properties:

Deadband – the value of the deadband, a non-negative count. If the value is zero, then there is no deadband consideration. Default = 0.

Hysteresis – TRUE, FALSE, ALWAYS, NEVER. Default is NEVER.

Limit – the value of the limit,.

FillRequired – if TRUE, the state of the block is UNKNOWN until the buffer fills. Default is TRUE.

SampleSize – the number of points to include in the analysis. When the sample size is changed, the buffer is cleared. Default is 1.

TriggerCount – the minumum number of samples that must be less than the limit in order to evaluate as TRUE. Default = 0.

Connections:

* in – truth value, the input being monitored.
* out – truth value, true if the input is true for more than the limit.

#### Low Limit Time Window

* Test inputs within a specified time window for values less than a specified limit. For the period of time that the buffer is not filled and a FALSE cannot be assured, the result is dependent on the *FillRequired* property. If TRUE then the conclusion is UNKNOWN; otherwise the conclusion is FALSE.
* If a new value is not received within the scan interval, then a value for that interval is generated as a copy of the last good value. Values of bad quality are ignored.
* A hysteresis property controls the behavior with regard to deadband processing. Refer to the “LogicFilter” block for details.

Attribute Display:

Value – the current count

Properties:

Deadband – the value of the deadband, a non-negative count. If the value is zero, then there is no deadband consideration. Default = 0.

Hysteresis – TRUE, FALSE, ALWAYS, NEVER. Default is NEVER.

Limit – the value of the limit, default is zero.

ScanInterval – the sampling interval ~ secs

TimeWindow – the period over which the analysis is to take place. ~ secs.

TriggerCount – the minumum number of samples that must be less than the limit in order to evaluate as TRUE. Default = 0.

Connections:

* in – truth value, the input being monitored.
* out – truth value, true if the count is greater than the limit.

#### Moving Average

* Compute the average of all values that have arrived on the block’s input over its entire existence. Report the average on the arrival of each new point. Values of bad quality are ignored for the purpose of the average. When bad value is received, then the output is given a bad quality and the buffer is cleared. The average may, optionally, be cleared on reset.

Properties:

ClearOnReset – If TRUE, the average computation will re-start when the block is reset. By default this property is FALSE.

Connections:

* in – data connection, the raw input.
* out – data, the accumulated average.

#### Moving Average Sample

* Compute a rolling average of a specified number of the most recent values that have arrived on the block’s. Once the buffer is filled, report the average on the arrival of each new point. Values of bad quality are ignored for the purpose of the average. When bad value is receivd, then the output is given a bad quality. The average may, optionally, be cleared on reset.

Attribute Display:

Value – the current moving average

Properties:

ClearOnReset – If TRUE, the average computation will re-start when the block is reset. By default this property is FALSE.

SampleSize – the number of points to include in the average. When the sample size is changed, the buffer is cleared.

Connections:

* in – data connection, the raw input.
* out – data, the rolling average.

#### Moving Average Time

* Compute the average of all values that have arrived on the block’s input within a specified time window. Report the average on the sample interval. Values of bad quality are ignored for the purpose of the average. When bad value is received, then the output is given a bad quality. The average may, optionally, be cleared on reset.

Attribute Display:

Value – the current moving average

Properties:

ClearOnReset – If TRUE, the average computation will re-start when the block is reset. By default this property is FALSE.

ScanInterval – the sampling interval ~ secs

TimeWindow – the period over which the average is to take place. ~ secs.

Connections:

* in – data connection, the raw input.
* out – data, the accumulated average.

#### N True

* Propagate a “true” if the number of input true values meets or exceeds a specified threshold. This block expects multiple connections on its input port. It accomodates any number. The output is given a GOOD quality unless the block state is UNKNOWN. In that case the quality is set to the “worst” quality of any of the inputs. On a reset, existing values are retained, and an UNKNOWN value is assigned the output state. However, no new output is emitted until the next new value is received.

Attribute Display:

Value – the current state of the block

Properties:

SyncInterval – a “coalescing interval”. This is the time allowance for multiple values that have originated in the same scan to be evaluated at the same time. ~ secs. The default value is 0 (no synchronization).

NTrue – the number of inputs which must carry a “true” value in order to propagate a “true” on the output. The default value is 0 (always succeeds).

Connections:

* in – truthvalues. Multiple upstream blocks may be connected to the single input.
* out – truthvalue representing the “or” of the inputs. The output timestamp is updated.

#### Not

* Propagate the logical inversion of the input. Only a single input connection is expected. There is no synchronization issue, nor reset behavior.

Connections:

* in – truthvalue.
* out – truthvalue representing the “not” of the input. The timestamp is not altered from the incoming value.

#### Note

* A note block is a place to store commentary regarding the diagram. A note block has no behavior, has no connections and does not participate in the execution of the diagram.
* If the text is legal HTML, then the HTML is used for formatting. Like any other attribute, the text may be linked to a tag for live updating.
* A specialized editor is available via right-click on the block. This editor allows editing of the plain-text HTML, simultaneously displaying the formatted output. On save, the block dimensions are automatically updated to accomodate the formatted text.
* See <https://docs.oracle.com/javase/tutorial/uiswing/comonents/html.html> for a comprehensive discussion regarding the use of HTML in a component.

Properties:

Height – height of the text block in screen units

Text – information to be displayed in the block.

Width – width of the text block in screen units

#### Or

* Propagate the logical “or” of the inputs. This block expects multiple connections on its input port. It accomodates any number. The output is given a GOOD quality unless the block state is UNKNOWN. In that case the quality is set to the “worst” quality of any of the inputs. On a reset, existing values are retained, and an UNKNOWN value is assigned the output state. However, no new output is emitted until the next new value is received.

Attribute Display:

Value – the current state of the block

Properties:

SyncInterval – a “coalescing interval”. This is the time allowance for multiple values that have originated in the same scan to be evaluated at the same time. ~ secs. The default value is 0 (no synchronization).

Connections:

* in – truthvalues. Multiple upstream blocks may be connected to the single input.
* out – truthvalue representing the “or” of the inputs. The output timestamp is updated.

#### Output

* Write data received on the input connection to an Ignition tag. Any type of incoming connection is allowed. The specified tag type must match the selected output connection type.

Attribute Display:

Value – the simple value being written to the tag

Properties:

TagPath – a fully qualified (i.e. includes provider name in brackets) path to a text tag

Connections:

* in – data, truthvalue, signal or text connection.

#### Parameter

* Achieve data persistence by encapsulating an Ignition tag. Incoming values are written to the tag. Tag changes are propagated on the output. There is no special reset behavior. Because *Parameter* blocks are associated with a tag, it is possible to have the same block repicated in several places in the collection of diagrams.

Properties:

TagPath – a fully qualified (i.e. includes provider name in brackets) path to a text tag

Connections:

* in – data, truthvalue, signal or text connection.
* out - data, truthvalue, signal or text connection

#### Persistence Gate

* The persistence gate gurantees that the input value has remained unchanged for a specified interval before that value is propagated. The block displays a count-down of the interval on its icon.
* When the block receives a value of good quality that matches the trigger property, it begins a countdown and passes the value when the countdown reaches 0. If the block receives a different value during the countdown, it aborts the countdown and passes the new value immediately.
* To specify how often the icon updates the countdown, set the ScanInterval attribute. The unit of time it displays is appropriate to the time remaining.

Properties:

ScanInterval – the interval between updates of the block ~ secs. The default value is 10. The scan interval must be at least 1 second.

TimeWindow – the length of time for which input values are monitored for no change ~ secs. The default value is 0 which effectively disables the block

Trigger – the state (or ANY state) that starts the monitoring process.

Connections:

* in – truthvalue to be monitored.
* out truthvalue as the behavior of the block dictates.

#### PID

* Perform PID control on an incoming data stream.

Properties:

SetPoint – the target setting.

Interval – the time interval at which control calculations are made ~ milliseconds.

Kd – the derivative constant.

Ki – the integral constant.

Kp – the proportional constant .

Connections:

* in – raw data stream to be controlled.
* out – truthvalue, initialized to UNKNOWN. Value is emitted on a state change.
* recv – signal. If the command is “RESET”, then any accumulated data will be cleared and the calculations will procede from scratch.

#### Product

* Compute the product of the inputs. This block expects multiple connections on its input port. It accomodates any number. On a reset, existing values are retained, and an UNKNOWN value is assigned the output state. However, no new output is emitted until the next new value is received..

Properties:

SyncInterval – a “coalescing interval”. This is the time allowance for multiple values that have originated in the same scan to be evaluated at the same time. ~ secs. The default value is 1 sec.

Connections:

* in – data values. Multiple upstream blocks may be connected to the single input.
* out – double value representing the prouct of the inputs. The output timestamp is updated.

#### Property Setter

* When a new data value is detected on input, this block propagates a CONFIGURE signal on the output. This signal is universally understood by every block. If that block has a property whose name matches the *argument* parameter of the signal, then the value of the property in the block will be set to the *payload* of the signal. The block will be appropriately notified of the change.
* The signal will contain the name of the property (configured in the block), plus a value that is newly arrived on the input.

Properties:

Property – the name of the property to be configured.

Connections:

* in – a data value. This value is sent to the output as a signal.
* out – a “configure” signal.

#### Qualified Value

* Create a qualified value from individual components: value, quality and timestamp. The datatype of the output is the same as that of the value input. The input determines the default quality and timestamp as well. The block will emit an output for every value received on its input.
* The quality input is a string where “good” is interpreted as a good quality and all other value are bad. This sets the quality of the output. However, if the “value” input has bad quality, it will drive the output to a bad quality as well. Quality is taken as the last value received on the quality input. If no input has ever arrived on the quality channel, its contribution is considered to be good.
* The timestamp channel is either a date or a string input. If a string, its format is specified by the FORMAT property of the block. This is a string date-time format as specified by the Java *SimpleDateFormat* class. The default value is: “YYYY/MM/dd hh:mm:ss”. If the format does not include a date, the current date is assumed. If it does not include a time, the current time is assumed. The timestamp is taken as the last value received on the time input. If no input has ever arrived on the time channel, the timestamp of the output is assigned the timestamp of the value input.
* A synchronization interval should be specified to guarantee proper pairing of readings between the separate input channels. The synchronization is driven by the “value” input only. This prevents re-issuance of a value on receipt of quality or timestamps only.

Properties:

Format – string format for the date-time. If the “time” input is textual (as opposed to a date) it is converted to a timestamp via this format.

SyncInterval – a “coalescing interval”. This is the time allowance for multiple values that have originated in the same scan to be evaluated at the same time. ~ secs. The default value is 1 sec.

Connections:

* value – value of any data type. This forms the value component of the output
* quality – quality, a string. If present, this forms the quality component of the output.
* time – a date or a string in the format specified. If present, this forms the timestamp component of the output.
* out – qualified value constructed from the three input channels.

#### Quotient

* Compute the quotient of two inputs. Report the value a port “a” divided by the value of the port labelled “b”. On a reset, all existing values are forgotten.

Properties:

SyncInterval – a “coalescing interval”. This is the time allowance for multiple values that have originated in the same scan to be evaluated at the same time. ~ secs. The default value is 1 sec.

Connections:

* a – data value. The dividend
* b – data value. The divisor.
* out – double value representing the quotient of the inputs. The output timestamp is updated.

#### Random

* This block is a noise generator. It alters the input value by selecting a random sample from one of several statistical distributions, adding it to the input and propagating the result. There is no reset behavior.

Properties:

Distribution – the random distribution, one of EXPONENTIAL, NORMAL or UNIFORM.

Lower – least possible value when using a UNIFORM distribution.

Upper – greatest possible value when a UNIFORM distribution is selected.

Mean – when NORMAL distribution is used.

StandardDeviation – when NORMAL distribution is used

Connections:

* in – incoming data value.
* out – double value representing the input perturbed by addition of a random sample selected from the specified distribution. The output timestamp retains the value of the input.

#### Range Observation

* Validate incoming input data values against a configured set of upper and lower limits with optional deadband ranges. The output is a truth-value according to the truth-table below.

|  |  |
| --- | --- |
| TRUE | FALSE |
| X < UpperLimit – UpperDeadband  and  X > LowerLimit + LowerDeadband | X > UpperLimit  or  X < LowerLimit |

* For values that in ranges not specified above, retain the most recent past value. On reset the past value is set to UNKNOWN.

Properties:

LowerLimit – the value of the lower limit, default is zero.

UpperLimit – the value of the upper limit, default is zero.

LowerDeadband– tolerance around the lower limit, default is zero.

UpperDeadband – tolerance around the upper limit, default is zero.

Connections:

* in – data connection, the raw input.
* out – truthvalue connection, the result.

#### Readout

* This block has no behavior. It simply propagates any value on its input to its output. Its function is to provide a place to display the current value of its input connection.
* The format must contain exactly one form of “%s” (for string data), or “%d” (for integer data), or “%f” (for floating point data). If no format is specified, “%s” is assumed. The format strings may be modified with a width and precision. E.g. “%5.3f” means “display leaving space for at least 5 digits with 3 following the decimal”. Further options can be found in the the Java 8 documentation (http://docs.oracle.com/javase/8/docs/api) for *String.format(*).

Properties:

Format – the format in which to display the value. See java documentation for String.format() for format configuration. The format must accept exactly one argument. The argument will be coerced into the data type specified by the format.

Value – the most recent value to pass through the block. c

Connections:

* in – data value.
* out – data value. The timestamp is not altered from the input.

#### Receiver

* *Accept a signal message from the “engine” and propagate it on the output connection. This block is largely superfluous since any block can be configured to receive signals.*

*Properties:*

*Command – the command that the receiver will accept. All others are ignored.*

*Connections:*

* *recv – data or truth value.*

#### Reset

* Propogate a reset signal to connected blocks at a configurable interval. An interval of zero effectively disables the block. On reset() the timer is reset.

Properties:

Command – the command to be sent at the interval. By default this is “reset”.

Interval – the rate at which the reset signal is emitted ~ secs.

Connections:

* out – reset signal.

#### Sink Connection

* Send values from the input connection to a memory tag for routing to SourceConnection blocks that are configured to receive from this block. SinkConnection blocks share tag paths with SourceConnections with which they are logically connected.

Attribute Display:

Value – the most recent value written to the tag

Properties:

TagPath – the name of the Sink Connection block which is logically connected as the input to this block.

Connections:

* in – data or truth value.

#### Source Connection

* Accept an input value from a memory tag and propagate it on the output connection. The tag is shared, presumably by a corresponding SinkConnection. During migration, this is guaranteed to be the case as the tag path is derived from the names shared by connected sources and sinks.

Attribute Display:

Value – the most recent value read from the tag

Properties:

TagPath – the name of the Sink Connection block which is logically connected as the input to this block.

*Connections:*

* *out – data or truth value.*

#### SQC

* Execute one of the Western Electric/Westinghouse SPC rules on an input. A truth-value output records changes in the rule state. Any changes in either limit or target trigger a new rule computation.
* The rules are one-sided or two-sided x-out-of-y, or x consecutive points on one side of the limit.
* A SQC block has three numeric input paths, the values carried over the paths are the target value (T), the standard deviation (S), and the actual measured value (V). The block has a single boolean output that is true if a SQC rule has been violated and false otherwise. Evaluation will take place whenever a new data value is presented along any one of the three paths (there is no special input synchronization because, normally, the target and standard deviation values are fairly static).
* All three data paths must have GOOD data quality. If not, the output result is UNKNOWN, with a bad quality.
* As data points are propagated to the block on the data path connected to the (V) input port they are stored in a buffer that is sized based on the *sampleSize* property. The algorithm checks the number of data points in the block’s history as follows:
* If the number of out-of-range points equals or exceeds the limit then the output is TRUE.
* If the number of out-of-range points is less than the limit and there are insufficient unfilled slots in the data buffer to reach that limit, then the output is FALSE.
* If neither of the above conditions hold, the output is UNKNOWN.

On *reset(),*if the *clearOnReset* property is TRUE, tne buffer is cleared and the block reports an UNKNOWN status. NOTE: the block does not poll for data. It only detects changed values on its input.

* SQC blocks communicate status. If a one-sided SQC rule has been violated, then the block that detected the violation notifies all remaining SQC blocks on the same diagram. Any one-sided SQC rules on the opposing side must then report FALSE, if currently TRUE. This extra-ordinary block report has no effect on the current state of the block, or its history buffer. Two-sided blocks neither initiate nor participate in this capability. This is important so that the most recent SQC violation is the one that is driving the recommendation and corrective action. The signals that are propagated are: CLEAR\_LOWER\_SQC and CLEAR\_UPPER\_SQC.
* Calculate the control limits as follows. The control limits rarely change but could change between block executions.
* Upper = target + standard-deviation \* no.-of-standard-deviations-to-test
* Lower = target - standard-deviation \* no.-of-standard-deviations-to-test
* Count the number of data values violating the control limits. Compare to the SQC rules.

Properties:

ClearOnReset – If TRUE, the value buffer will clear when the block is reset. By default this property is FALSE.

NumberOfStandardDeviations – the distance above or below the target which signals a violation. This is expressed in standard deviations. This parameter is used for HIGH, LOW or BOTH limit types.

LimitType – CONSECUTIVE, HIGH, LOW or BOTH.

MinimumOutOfRange – minimum number of points exceeding the limit within the stored history in order for a TRUE value to be concluded. In the case of CONSECUTIVE this value refers to the minimum number of consecutive points on the same side of the limit to trigger a violation.

Mean – the nominal process mean.

SampleSize – number of observations to retain for computation of rule state.

Connections:

* t – target value.
* s – standard deviation of the input
* v – the input
* out – truthvalue, result of evaluation of the inputs, initialized to UNKNOWN. A rule violation results in a TRUE. Value is emitted on a state change.

#### SQL

* *Write data received on the input connection to a database. The block specifies the database name and SQL.*

*Properties:*

*Database – Name of the database.*

*SQL – A SQL insert statement that can be compiled into a prepared statement taking exactly one argument.*

*Connections:*

* *in – data, truthvalue, signal or text connection.*

#### Sum

* Total values received on multiple input connections and propagate to the output. On a reset, existing values are retained, and an UNKNOWN value is assigned the output state. However, no new output is emitted until the next new value is received.

Properties:

SyncInterval – a “coalescing interval”. This is the time to allow data to arrive on multiple inputs. It is also the no-data-change wait interval. The sync interval must be less than the scan interval.~ secs. The default value is 1 sec.

Connections:

* in – data. The expectation is that multiple connections will terminate at the port.
* out – data, the sum of the inputs.

#### Time Readout

* This block has no behavior. It simply propagates any value on its input to its output. Its function is to provide a place to display the time of the last value that passed through.
* The format must be a valid SimpleDateFormat. If no format is specified, “YYYY/MM/dd hh:mm:ss” is assumed. Further options can be found in the the Java 8 documentation (http://docs.oracle.com/javase/8/docs/api) for *SimpleDateFormat*.

Properties:

Format – the format in which to display the time. See java documentation for SimpleDateFormat for format configuration.

Value – the most recent value to pass through the block.

Connections:

* in – data value. This can be any data type.
* out – data value. This is identical to the input.

#### Time Fork

* Take the timestamp of the incoming value and pass it on the output.

Connections:

* in – a data value of any type.
* out – a Date value equal to the timestamp of the input.

#### Timer

* When the triggering value is received on the input, this block starts a counter recording the number of seconds in the triggering state. The counter is evaluated at the configured interval. The trigger may be any TruthValue. The StopOn value may not be the same as the trigger. On reset, the counter is cleared.

If a tag path is configured, this block saves the count value in the tag at the same time it is updated, thus making it available for other applications.

Properties:

AccumulateValues –if TRUE, the timer value is NOT reset when the block returns to its triggering value. Otherwise the count is zeroed on start of “on” interval.

Interval – the sampling time, ~seconds. Output is generated at this rate. The default value is 1 second.

TagPath – a fully qualified (i.e. includes provider name in brackets) path to a integer tag. If configured, this tag will receive the counter value at the evaluation interval.

Trigger – a TruthValue. The value to be matched on input in order to trigger the counter. Default value is TRUE.

StopOn – the value to be matched on input in order to stop the counter. Default value is FALSE.

Connections:

* in – a truth value. This triggers the counter if it matches the trigger value
* out – the counter value.

#### Trend Detector

The Trend Detection block has the same inputs as an SQC block – target, standard deviation and value. It implements trend detection using a least squares regression algorithm. The evaluator will fire if a new data value is propagated along any one of the three input paths as long as there is good data on each of the three.

In addition to providing a Boolean output signaling that it has detected a trend, the actual slope, slope variance, and projected value are also available as outputs of the block and may feed other blocks if needed.

On reset, the block optionally clears its local history.

Properties:

ClearOnReset – If TRUE, the value buffer will clear when the block is reset. By default this property is FALSE.

NumberOfTrendPointsRequired – number of observations to retain for trend computation. This is the buffer size.

RelativeToTarget – if TRUE, detection is relative to the target value, otherwise it is relative to the actual mean.

SlopeCalculationOption – AVERAGE, or LINEARFIT.

StandardDeviationMultiplicativeFactor – the incoming standard deviation is enhanced by this factor.

TestLabel – the trend test label.

TrendCountThreshold – number of observations in one direction or the other required to compute a trend.

TrendDirection – DOWNWARD, UPWARD, or BOTH.

Connections:

* target – target value.
* standardDeviation – standard deviation of the input
* value – the input
* out – truthvalue, result of evaluation of the inputs, initialized to UNKNOWN. If a trend is detected, the value is TRUE.
* slope – when a trend is detected this output propagates the computed slope
* variance - when a trend is detected this output propagates the computed standard deviation. (The port name was chosen to avoid confusion with “slope” in the block portrayal)
* projection - when a trend is detected this output propagates the computed value

#### Transmitter

* *Broadcast the signal received on the block input. This block is largely superfluous since any block can be configured to transmit.*

*Properties:*

*Scope – a scope which determines to which blocks the signal will be delivered.*

*Connections:*

* *send – signal.*

#### Truth-value Pulse

* On block (or diagram) reset, or when the the signal connection receives a RESET, this block sets its output to momentarily to the configured value, waits for specified interval and then returns its output to the opposite of the pulsed value.

Properties:

Interval – the length of time to hold the output, ~seconds. The default value is 1 second.

PulseValue – the value to be propagated during the pulsing phase. Default value is TRUE.

Connections:

* out – a momentary TruthValue value (TRUE or FALSE)

#### Unknown

* The Unknown detection block emits a TRUE if its incoming value is UNKNOWN.

Connections:

* in – truth value.
* out – truth value, TRUE if its incoming value is UNKNOWN. The timestamp is not altered from the input.

#### Zero Crossing

* This block emits a TRUE whenever the input value goes from positive to negative or nagative to ppositive..

Connections:

* in – value, a numerical value.
* out – truth value, TRUE if the sign of the incoming value is different from the previous value. The timestamp is not altered from the input.

### Bad Data Handling

The information transmiited between blocks is encapsulated in *QualifiedValue* objects. These contain a value, a quality and a timestamp. In general, when a data value of bad quality is received, the resulting output will also be marked as having bad quality. In addition, the result of any calculation involving the bad data will return a value of not-a-number or UNKNOWN as appropriate to its type.

Exceptions to this pattern occur when it is possible to deduce an answer without involving the spurious input. For example, an “And” block can conclude a FALSE with one FALSE input of good quality, independent of the value or quality of other inputs.

### Block Locking

Any block may be placed in a *locked* state. When *locked*, no further results are transmitted on its output, however all input processing occurs as normal. When the block becomes *unlocked*, output is again enabled. The block will output a value during its next normal evaluation. Depending on the block this may be as a result of a timer expiring or receipt of input.

*The locked state is activated via a menu choice that is available on all blocks. A locked block is annoted with a lock badge in the designer. Locked state is always cleared during a save.*

*Even when locked, the block will respond to a “forcedPost” request,. This causes the block to place a specified value on its output. This allows a tester/developer to isolate portions of a diagram during development/debugging.*

### Block Reset

All blocks support a *reset()* command. At a minimum, a reset will place the block in an INITIALIZED state. (Receipt of the next value will return the block to ACTIVE state). If the block retains state of any sort, that state will also be set to its initial value.

### Timestamp Handling

In general, if a data value is passed through a block, even if there has been an operation on it (e.g. exponential), the timestamp of the result will be the same as the timestamp of the input. However, if the value has been combined in any way with other inputs, then the timestamp of the result is set to the time at which the output was propagated.

### Connection Type Change

Several of the block classes operate on any of the possible data types. In order to properly configure connections for each individual situation, a control-click menu option is supplied to “Change Connection Types”. This option alters all of the block’s stubs to the chosen type. The blocks for which this feature is available include:

* Input
* Junction
* Output
* Parameter
* Readout
* Sink
* Source

Once a diagram has been saved, this option is no longer available.

### Type Coercion

Blocks are aware of the types of data they process. The type of the input connector guarantees the data type of an input. (In the case of Input blocks, the incoming tag value is coerced to the expected type). Blocks also guarantee the correct data type on output.

Where coercion is necessary, the following steps are taken:

1. The value is converted to a string. This provides the conversions with a common starting point.
2. For strings no conversion is necessary. However, a null will generate an error.
3. For real numbers, the conversion is Double.parseDouble(*value*).
4. For integers, the conversion is Integer.parseInt(*value*).
5. For booleans, any non-zero numeric value or the string “TRUE” (case-insensitive) is taken as true. All others are false.

If the conversion step fails, then the reading is given a BAD quality and propagated.

### Block Programming Interface

All executable blocks, both those implemented in Python and those implemented in Java, must support the following interface:

/\*\*

\* Notify the block that a new value has appeared on one of its

\* input anchors. The notification contains the upstream source

\* block, the port and value.

\* **@param** vcn

\*/

**public** **void** acceptValue(IncomingNotification vcn);

/\*\*

\* Notify the block that it is the recipient of a signal from

\* "the ether". This signal is not associated with a connection.

\* This method is meaningful only for blocks that are "receivers".

\* **@param** sn

\*/

**public** **void** acceptValue(SignalNotification sn);

/\*\*

\* If true, the "engine" will delay calling the start() method

\* of this block until all other blocks that do not indicate

\* a delay have been started.

\* **@return** true if this block should be ordered at the end

\* of the startup process.

\*/

**public** **boolean** delayBlockStart();

/\*\*

\* In the case where the block has specified a coalescing time,

\* this method will be called by the engine after receipt of input

\* once the coalescing "quiet" time has passed without further input.

\*/

**public** **void** evaluate();

/\*\*

\* Place a value on a named output port of a block.

\* This action does not change the internal state of the block.

\* It's intended use is to debug a diagram.

\* **@param** port the port on which to insert the specified value

\* **@param** value a new value to be propagated along an

\* output connection. The string value will be coerced

\* into a data type appropriate to the connection.

\*/

**public** **void** forcePost(String port, String value);

/\*\*

\* **@return** a list of anchor prototypes for the block.

\*/

**public** List<AnchorPrototype> getAnchors();

/\*\*

\* **@return** the universally unique Id of the block.

\*/

**public** UUID getBlockId();

/\*\*

\* **@return** information necessary to populate the block

\* palette and subsequently paint a new block

\* dropped on the workspace.

\*/

**public** PalettePrototype getBlockPrototype();

/\*\*

\* **@return** the fully qualified path name of this block.

\*/

**public** String getClassName();

/\*\*

\* **@return** information related to the workings of the block.

\* The information returned varies depending on the

\* block. At the very least the data contains the

\* block UUID and class. The data is read-only.

\*/

**public** SerializableBlockStateDescriptor getInternalStatus();

/\*\*

\* **@return** the block's label

\*/

**public** String getName();

/\*\*

\* **@return** the Id of the block's diagram (parent).

\*/

**public** UUID getParentId();

/\*\*

\* **@return** the id of the project under which this block was created.

\*/

**public** **long** getProjectId() ;

/\*\*

\* **@return** all properties of the block. The array may be used

\* to updated properties directly.

\*/

**public** BlockProperty[] getProperties();

/\*\*

\* **@return** a particular property by name.

\*/

**public** BlockProperty getProperty(String name);

/\*\*

\* **@return** a list of names of properties known to this class.

\*/

**public** Set<String> getPropertyNames() ;

/\*\*

\* **@return** the current state of the block

\*/

**public** TruthValue getState();

/\*\*

\* **@return** a string describing the status of the block. This

\* string is used for the dynamic block display.

\*/

**public** String getStatusText();

/\*\*

\* **@return** true if this block is locked for debugging purposes.

\*/

**public** **boolean** isLocked();

/\*\*

\* **@return** true if this block is a candidate for signal messages.

\*/

**public** **boolean** isReceiver();

/\*\*

\* **@return** true if this block publishes signal messages.

\*/

**public** **boolean** isTransmitter();

/\*\*

\* Send status update notifications for any properties

\* or output connections known to the designer.

\*

\* In practice, the block properties are all updated

\* when a diagram is opened. It's the connection

\* notification for animation that is most necessary.

\*/

**public** **void** notifyOfStatus();

//===================== PropertyChangeListener ======================

/\*\*

\* This is a stricter implementation that enforces QualifiedValue data.

\*/

**public** **void** propertyChange(BlockPropertyChangeEvent event);

/\*\*

\* Reset the internal state of the block.

\*/

**public** **void** reset();

/\*\*

\* Set the anchor descriptors.

\* **@param** prototypes

\*/

**public** **void** setAnchors(List<AnchorPrototype> prototypes);

/\*\*

\* Set or clear the locked state of a block.

\* **@param** flag True to lock the block.

\*/

**public** **void** setLocked(**boolean** flag);

/\*\*

\* **@param** name the name of the block. The name

\* is guaranteed to be unique within a

\* diagram.

\*/

**public** **void** setName(String name);

/\*\*

\* **@param** id is the project to which this block belongs.

\*/

**public** **void** setProjectId(**long** id);

/\*\*

\* Accept a new value for a block property. It is up to the

\* block to determine whether or not this triggers block

\* evaluation.

\* **@param** name of the property to update

\* **@param** value new value of the property

\*/

**public** **void** setProperty(String name,Object value);

/\*\*

\* Set the current state of the block.

\* **@param** state

\*/

**public** **void** setState(TruthValue state);

/\*\*

\* **@param** text the current status of the block

\*/

**public** **void** setStatusText(String text);

/\*\*

\* Specify the timer to be used for all block-

\* internal timings.

\*

\* **@param** timer

\*/

**public** **void** setTimer(WatchdogTimer timer);

/\*\*

\* Start any active monitoring or processing within the block.

\*/

**public** **void** start();

/\*\*

\* Terminate any active operations within the block.

\*/

**public** **void** stop();

/\*\*

\* Convert the block into a portable, serializable description.

\* The descriptor holds common attributes of the block.

\* **@return** the descriptor

\*/

**public** SerializableBlockStateDescriptor toDescriptor();

/\*\*

\* **@param** tagpath

\* **@return** true if any property of the block is bound to

\* the supplied tagpath. The comparison does not

\* consider the provider portion of the path.

\*/

**public** **boolean** usesTag(String tagpath);

/\*\*

\* Check the block configuration for missing or conflicting

\* information.

\* **@return** a validation summary. Null if everything checks out.

\*/

**public** String validate();

}

Figure 8 – Block Interface

## Block Properties

Each block possesses a pre-defined list of properties. Properties are persistent attributes specified by a name, a binding type and a value. Additionally a property value may be configured for display in the designer near the block’s icon. Property values do NOT have a quality attribute.

### Data Types

Properties are pre-assigned a data type from one of the following options:

* BOOLEAN
* INTEGER
* HTML (*gets an HTML editor*)
* HYSTERESIS (TRUE, FALSE, ALWAYS, NEVER)
* LIMIT (HIGH, LOW, BOTH, NONE, CONSECUTIVE)
* LIST (*multiple user-entered values*)
* OBJECT (*matches any type*)
* OPTION (*block-defined selection list*)
* SCOPE (GLOBAL, APPLICATION, FAMILY, LOCAL)
* STRING
* TIME
* TRUTHVALUE (TRUE, FALSE, UNKNOWN, UNSET)

### Binding

Block properties may be bound to tags or other elements in the application. The binding options are as follows:

* NONE
* ENGINE
* TAG\_MONITOR
* TAG\_READ
* TAG\_READWRITE
* TAG\_WRITE

Except for properties that are pre-defined as bound to the ENGINE or OPTION, the binding definition can be changed dynamically via the property editor.

The ENGINE designation is determined exclusively by the block and cannot be changed. That is, once a property is designated as ENGINE it cannot be set to a different binding type, nor can a property that is not bound to ENGINE be converted to ENGINE. This behavior is enforced by the property editor.

In a similar way, the OPTION binding contains a fixed list of options known to the block and cannot be changed.

The value of any property can, optionally, be displayed in the Designer in proximity to the block. The relative position of the display to the block is editable via the property editor. Properties that have an ENGINE binding are of particular interest as they reflect some internal state of the block and will update dynamically as the block executes.

### Activity Log

Every block has the potential to record a list of activities in a fixed-size buffer. Initially this capability is disabled. It can be activated simply by editing the block’s ActivityBufferSize property, setting the value to something greater than zero.

Contents of the log are available in the “*View Internal State*” popup menu on the block. Activities that are logged include:

* RESET – marks a reset action on the block
* STATE – records an internal block state change
* *port* – records data propagated on the named output port

Additional activites may be available on specific classes of blocks.

## Property Editor

Whenever a block is selected, an editor for its properties appears in a Designer (Wicket) window. The editor displays read-only block attributes, and then a panel for each custom property. When edited, the block properties are immediately transmitted to the Gateway.

Note: Entries into a text field are not recorded until an ENTER is typed.

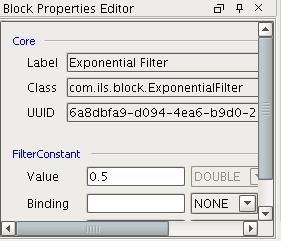


Figure 9 – Properties Editor

## Connections

There are four types of connections:

* Numerical (data)
* Truth-value (true,false,unknown)
* Control (signal)
* Text (diagnoses, recommendations)

## Transmit/Receive

The ability to send and receive signals is a design-time configurable property of a block. Blocks enabled to receive broadcast signals are marked with a receiver “badge”. Blocks that send signals are similarly marked with a transmitter badge.

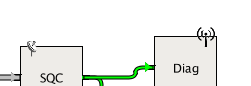


Figure 10 – Receive/transmit Badges

### Signals

Signals are commands that can be used to control block state, e.g. “start” or “reset”. In addition to originating from blocks within a diagram, signals can be sent from the outside, e.g. from a Vision window. Signals have a property of “scope” that determines the collection of blocks to which they may be delivered.

There are four scoping levels:

* Local – the signal is delivered only blocks in the diagram from which it originated
* Family – the signal is delivered blocks in diagrams in the same family as the originator
* Application – the signal is delivered to blocks in diagrams in the same application as the originator
* Global – the signal is delivered to all blocks (with receiving enabled) in the current project

A signal has the following attributes:

* Command – a string value containing a well-known command
* Argument - a string argument or parameter name
* Payload – an optional payload, a string
* Pattern – a string used by a Receiver block to determine whether or not to process the signal. A ‘\*’ configured in the block matches all input. For other types of blocks, the pattern must match the block name.

### Standard Signals

All blocks are expected to respond to the following signal commands as they arrive on the signal input stub that is present for every block.

* configure – use information in the signal to set a property value. Trigger any evaluations would normally occur.
* evaluate – execute the evaluate() method of the block. This may or may not do anything.
* lock – set the block to a locked state. It will not pass any values.
* reset – clear any internal storage, set the current state, if appropriate, to UNKNOWN.
* unlock – set the block to an unlocked state

## Find/Replace

The standard Ignition find/replace capabilities have been extended to include the Block Language Toolkit. There are three elements of this feature: finding, replacing and locating.

The search may be restricted to any or all of the categories: application names, family names, diagram names, block names, block property names and block property values. In the case of bound properties that search is made on the binding rather than the current value. For example we return a tag path rather than the current value of that tag.

The current implementation does not allow editing via the find/replace mechanism. Any edits must be made through the normal Designer interfaces.

Double-clicking on the icon associated with a discovery will display the diagram containing that element. For this display to appear, the block language toolkit workspace must be already selected in the Designer.

# Client

*A Client view is the receiver of state or results for a diagram. The client comes in two flavors. The Engineer view is directed toward simulation and debugging. It features an animated “preview” view of a diagram that displays current status of the blocks. It cannot be edited.*

*The operator view emphasizes the display of conclusions and recommendations from the executing diagram. The operator’s windows feature standard Ignition components.*

## Operator

The operator’s view ...

## Engineer

The engineer’s view ...

## Communication with the Gateway

Event listener ...

# Python

The toolkit may be extended in several important ways via Python. The Python interfaces described in this section are intended as opportunities for the user to develop custom links with the toolkit core. The categories of extensions are as follows:

* Custom Blocks – In addition to the collection of blocks distributed with the toolkit, the user may develop blocks completely in Python, completely external to the official distribution. Blocks that written in this manner behave within the diagram in the same way as the blocks written in Java that are distributed as part of the BLT module.
* Scripting Interface – The toolkit exposes a collection of Python scripts that may be accessed from any Vision widget.
* Extension functions – The toolkit exposes extension points where the engine calls Python scripts (if they are defined) at specified points in the life cycle of an Application or Famly object.

## Custom Blocks

The BLT distribution includes an Eclipse PyDev project with an accompanying *ant* build script that installs the Python blocks directly into the Ignitiion installation, making them instantly accessible to Ignition code.

### Constraints

### Callbacks

The utility package xom.ils.blt.utlil contains all of the methods defined to interface into the Python classes. These are called from a block class in the engine that serves as a proxy in the Gateway “engine” code.

The Python interface includes:

def **createBlockInstance**(className,parent,uid,result):

def **evaluate**(block):

def **getBlockProperties**(block,properties):

def **getNewBlockInstances**():

def **getNewBlockInstance**(className):

def **getBlockPrototypes**(prototypes):

def **setBlockProperty**(block,prop):

def **setValue**(block,port,value,quality):

#### CreateBlockInstance

Return an instance of the specified Python class. The method accepts a single argument, the class of the block to create. The shared variable, ‘result’ is sent to the method as an empty dictionary. On return, it should contain an element ‘instance’ that contains the newly created block.

#### GetBlockProperties

Return the defined properties for a specified block. Typically these are configuration parameters as opposed to real-time state variables. The block is specified as the ‘block’ argument of the method. The shared variable, ‘properties’ is sent to the method as an empty list. On return, the list should contain dictionaries one for each block property . The dictionaries contain the following keys:

* binding - path to the .
* bindingType - type of binding for this property, if any. Options are: NONE, ENGINE, TAG\_READ, TAG\_WRITE and TAG\_MONITOR. Default is NONE. Refer to the JavaDoc for *BindingType* for the most current list.
* editable - TRUE if the property can be edited, otherwise FALSE.
* maximum - a double value specifying the permissible maximum of the property. Default is Double.MAX\_VALUE.
* minimum - a double value specifying the permissible minimum of the property. Default is Double.MIN\_VALUE.
* name - name of the property. This must be unique among the properties for a class.
* quality - current quality of the value of the property.
* type - data type of the property. Options are: STRING, DOUBLE, INTEGER, and OBJECT. Default is STRING. Refer to the JavaDoc for *PropertyType* for the most current list.
* value - current value of the property

#### GetBlockPrototypes

Return the information necessary to create palette prototypes. The method accepts no arguments. The shared variable, ‘prototypes’ is sent to the method as an empty list. On return, the list should contain dictionaries one for each block class implemented in Python. The dictionaries contain the following keys:

* blockClass - class name of the block, e.g. “app.block.Custom.Custom”. The repeated class names are an artifact of storing each Python class in its own file.
* blockStyle - name of the graphical layout of the block in a view. Options are: CLAMP, DIAMOND, ENTRY, ICON, ROUND, and SQUARE. Refer to the JavaDoc for *BlockStyle* for the most current list.
* iconPath - this is the path to the icon used for the block in the palette.
* label - this is the label that appears under the block in the palette
* tabName - name of the palette tab on which this block appears. If the name does not match an existing tab, a new tab will be created.
* tooltip - text that appears when the mouse hovers over this block on the palette.
* viewBlockIcon - path to an icon that is to be used for the entire block.
* viewFontSize - set the size of the font for text embedded in the block view, if any. Default size is 24 points.
* viewHeight - overrides the default height of the block when drawn in a diagram. The default is a function of block style. ~ pixels.
* viewIcon - icon to appear inside a block when drawn in a diagram.
* viewLabel - text to appear inside a block when drawn in a diagram.
* viewWidth - overrides the default width of the block when drawn in a diagram. The default is a function of block style. ~ pixels.
* inports - a list of dictionaries describing the ports that are inputs, that terminate a connection.
* outports - a list of dictionaries describing the ports that are outputs, that originate a connection.

The keys for the dictionaries used to describe ports are:

* name - name of the port
* type - datatype of the connection connected to the port. Options are: ANY, DATA, INFORMATION, SIGNAL or TRUTHVALUE. Refer to the JavaDoc for *ConnectionType* for the most current list.

#### Report Results

Return block output values to the execution engine for dissemination to downstream blocks.

* /\*\*
* \* Handle the block placing a new value on its output.
* \*
* \* **@param** parent identifier for the parent, a string version of a UUID
* \* **@param** id block identifier a string version of the UUID
* \* **@param** port the output port on which to insert the result
* \* **@param** value the result of the block's computation
* \* **@param** quality of the reported output
* \*/
* **void** send(String parent,String id,String port,String value,String quality);

Figure 11 – Block Reporting Functions

#### Gateway Interface

Each block written in Python has access to a *PythonRequestHandler* object. This handler offers the following methods:

**public** PythonRequestHandler();

/\*\*

\* Traverse the parent nodes until we find an Application. If there

\* are none in our ancestry, return null.

\*

\* **@param** nodeId identifier for the node, a string version of a UUID

\* **@return** the ancestrial application

\*/

**public** ProcessApplication getApplication(String nodeId);

/\*\*

\* Find a block given the Id of the parent diagram the block itself.

\*

\* **@param** parent identifier for the diagram, a string version of a UUID

\* **@param** blockId identifier for the block, a string version of a UUID

\* **@return** the referenced block

\*/

**public** ProcessBlock getBlock(String parent,String blockId;

/\*\*

\* Find a block given the Id of the parent diagram the block itself.

\*

\* **@param** diagram container for the blocks

\* **@param** blockName name of the block within the diagram

\* **@return** a string version of the Id of the block with the specified name,

\* else null

\*/

**public** String getBlockId(ProcessDiagram diagram,String blockName;

/\*\*

\* Search the tree of node ancestors until we find one with the project set.

\* **@param** uuidString identifier for a node, a string version of a UUID

\* **@return** the default database for the project containing this node

\*/

**public** String getDefaultDatabase(String uuidString;

/\*\*

\* **@param** parent identifier for the diagram, a string version of a UUID

\* **@return** the default tag provider for the project associated with

\* the specified diagram

\*/

**public** String getDefaultTagProvider(String uuidString;

/\*\*

\* Given an identifier string, return the associated diagram.

\* The parent of a block should be a diagram.

\*

\* **@param** parent identifier for the block, a string version of a UUID

\* **@return** the diagram

\*/

**public** ProcessDiagram getDiagram(String diagramId);

/\*\*

\* Traverse the parent nodes until we find a Family. If there

\* are none in our ancestry, return null.

\*

\* **@param** nodeId identifier for the node, a string version of a UUID

\* **@return** the ancestrial family

\*/

**public** ProcessFamily getFamily(String nodeId;

/\*\*

\* Handle the block placing a new value on its output. The input may be PyObjects.

\*

\* **@param** parent identifier for the parent, a string version of a UUID

\* **@param** id block identifier a string version of the UUID

\* **@param** port the output port on which to insert the result

\* **@param** value the result of the block's computation

\* **@param** quality of the reported output

\*/

**public** **void** sendConnectionNotification(String id, String port, String value;

/\*\*

\* Handle the block placing a new value on its output. The input may be PyObjects.

\*

\* **@param** parent identifier for the parent, a string version of a UUID

\* **@param** id block identifier a string version of the UUID

\* **@param** port the output port on which to insert the result

\* **@param** value the result of the block's computation

\* **@param** quality of the reported output

\*/

**public** **void** postValue(String parent,String id,String port,

String value,String quality) ;

/\*\*

\* Broadcast a result to blocks in the diagram

\*

\* **@param** parent identifier for the diagram, a string version of a UUID

\* **@param** className name of the class of blocks to be signaled

\* **@param** command the value of the signal

\*/

**public** **void** sendLocalSignal(String parent,String command,

String message,String arg);

Figure 12– Request Handler Interface

### Installation

Several of the blocks that are distributed along with the BLT module have been written in Python because they reference significant amounts of custom code. (This allows the user to modify and test without an official build). These blocks are distributed in file *emc\_blocks.zip*. Installation consists of unzipping this file into the user-lib/pylib subdirectory of the Ignition install location.

### Samples

The distribution contains the following classes implemented in Python:

* FinalDiagnosis
* SQCDiagnosis
* Action
* Arithmetic

## Scripting Interface

The following interface is supported to allow Designer- or Client-scope Vision objects to control engine execution and to access or modify values from the executing model in the Gateway. All of these methods are located in the Python package:

system.ils.blt.application.

/\*\*

\* This interface is a common point for managing requests to the gateway dealing with

\* execution engine and block status. It is designed for use by Java code

\* as well as Python scripting. It provides a way to request/set properties of

\* diagrams, blocks and connections.

\*

\* Each request is relayed to the Gateway scope via an RPC call.

\*/

**public** **interface** ToolkitRequestHandler {

/\*\*

\* **@return** a list of resources that are children of the specified resource

\*/

**public** List<SerializableResourceDescriptor> childNodes(String nodeId);

/\*\*

\* Remove all current diagrams from the controller.

\*/

**public** **void** clearController();

/\*\*

\* Determine whether or not the indicated diagram is known to the controller.

\* **@param** diagramId string representation of the diagram's unique id

\*/

**public** **boolean** diagramExists(String diagramId) ;

/\*\*

\* Execute evaluate() on a specified block

\*/

**public** **void** evaluateBlock(String diagramId,String blockId) ;

/\*\*

\* **@param** uuid string representation of the application's unique id

\* **@return** the name of the application that is equal to or

\* superior to the node with the specified UUID

\*/

**public** String getApplicationName(String uuid);

/\*\*

\* Obtain a list of BlockProperty objects for the specified block.

\* CAUTION: If the block is not known to the gateway it will be created.

\* This method is for internal use in the designer.

\*

\* **@param** className class name of the block

\* **@param** projectId

\* **@param** resourceId

\* **@param** blockId

\*

\* **@return** an array of block properties for the subject block

\*/

**public** List<BlockProperty> getBlockProperties(String className,**long** projectId,**long** resourceId,UUID blockId) ;

**public** List<PalettePrototype> getBlockPrototypes() ;

/\*\*

\* **@param** diagramId string representation of the diagram's unique id

\* **@param** blockName name of the block within the diagram

\* **@return** the current state of the specified block.

\*/

**public** String getBlockState(String diagramId, String blockName) ;

/\*\*

\* Determine whether or not the engine is running.

\*/

**public** String getControllerState() ;

/\*\*

\* Find the parent application or diagram of the entity referenced by

\* the supplied id. Test the state and return the name of the appropriate

\* database.

\* **@param** uuid the uniqueId (string) of any node in the nav tree.

\* **@return** database name

\*/

**public** String getDatabaseForUUID(String uuid);

/\*\*

\* It appears that there is no way in the client to obtain a list of data sources

\* (database connection names). Consequently, we implement our own.

\*

\* **@return** a list of data sources configured and enabled in the gateway.

\*/

**public** List<String> getDatasourceNames();

/\*\*

\* blockId String representation of the block's internal Id.

\* **@return** the diagram that is a parent of the specified block.

\*/

**public** SerializableResourceDescriptor getDiagramForBlock(String blockId) ;

/\*\*

\* **@return** the current state of the specified diagram.

\*/

**public** DiagramState getDiagramState(Long projectId, Long resourceId) ;

**public** DiagramState getDiagramState(String diagramId);

/\*\*

\* **@return** the name of the family that is equal to or

\* superior to the node with the specified UUID

\*/

**public** String getFamilyName(String uuid);

/\*\*

\* **@return** internal details of a block for debugging purposes.

\*/

**public** SerializableBlockStateDescriptor getInternalState(String diagramId,String blockId) ;

/\*\*

\* **@param** diagramId identifier of the diagram owning the block, a String

\* **@param** blockId identifier of the block within the diagram, a String

\* **@param** propertyName name of the property for which a value is to be returned

\* **@return** the value of a specified block property.

\*/

**public** Object getPropertyValue(String diagramId,String blockId,String propertyName) ;

/\*\*

\* Acquire a value from the HSQL database table associated with the toolkit. A

\* null is returned if the string is not found.

\* **@param** propertyName name of the property for which a value is to be returned

\* **@return** the value of the specified property.

\*/

**public** String getToolkitProperty(String propertyName) ;

/\*\*

\* Determine whether or not the engine is running.

\*/

**public** **boolean** isControllerRunning() ;

/\*\*

\* Query a diagram in the gateway for list of its blocks that are downstream

\* of the specified block.

\* **@param** diagramId of the parent diagram

\* **@param** blockName name of the block within the diagram

\* **@return** a list of blocks belonging to the diagram.

\*/

**public** List<SerializableBlockStateDescriptor> listBlocksDownstreamOf(String diagramId,String blockName);

/\*\*

\* List all blocks that have properties bound to the supplied tag path.

\* **@param** tagpath the path for the tag of interest.

\* **@return** a list of blocks associated with the tag.

\*/

**public** List<SerializableBlockStateDescriptor> listBlocksForTag(String tagpath) ;

/\*\*

\* Query a diagram in the gateway for list of its blocks.

\* **@param** diagramId of the parent diagram

\* **@return** a list of blocks belonging to the diagram.

\*/

**public** List<SerializableBlockStateDescriptor> listBlocksInDiagram(String diagramId) ;

/\*\*

\* Query a diagram in the gateway for list of its blocks that are upstream

\* of the specified block.

\* **@param** diagramId of the parent diagram

\* **@param** blockName name of the block within the diagram

\* **@return** a list of blocks belonging to the diagram.

\*/

**public** List<SerializableBlockStateDescriptor> listBlocksUpstreamOf(String diagramId,String blockName);

/\*\*

\* The result is a list of SerializableBlockState descriptors for those

\* blocks in any project that have configuration issues. Descriptor attributes

\* contain at least the project and a path to the block. The descriptor

\* contains a textual description of whatever problem is detected.

\* **@return** a list of blocks that have incomplete or incorrect configuration.

\*/

**public** List<SerializableBlockStateDescriptor> listConfigurationErrors() ;

/\*\*

\* **@param** diagramId identifier of the diagram to be queried, a String

\* **@param** className fully qualified class name of blocks to be listed

\* **@return** a list of ids for blocks owned by a specified diagram that

\* are of a specified class.

\*/

**public** List<SerializableBlockStateDescriptor> listDiagramBlocksOfClass(String diagramId,String className);

/\*\*

\* Query the gateway for list of diagrams belonging to a project.

\*

\* **@param** projectName

\* **@return** a list of tree-paths to the diagrams saved (ie. known to the Gateway).

\*/

**public** List<SerializableResourceDescriptor> listDiagramDescriptors(String projectName) ;

/\*\*

\* Query the gateway for list of resource nodes that the block controller

\* knows about. This should correspond to what is displayed in the designer

\* nav tree for all loaded projects.

\*

\* **@return** a list of resources known to the BlockController.

\*/

**public** List<SerializableResourceDescriptor> listResourceNodes();

/\*\*

\* Query the gateway for list of its sink blocks associated with the

\* specified source. The blocks that are returned are not constrained

\* to be part of the same diagram, family or application.

\* **@param** diagramId of the parent diagram

\* **@param** blockName name of the block within the diagram

\* **@return** a list of blocks logically connected to the source.

\*/

**public** List<SerializableBlockStateDescriptor> listSinksForSource(String diagramId,String blockName) ;

/\*\*

\* Query the gateway for list of its source blocks associated with the

\* specified sink. The blocks that are returned are not constrained

\* to be part of the same diagram, family or application.

\* **@param** diagramId of the parent diagram

\* **@param** blockName name of the block within the diagram

\* **@return** a list of blocks logically connected to the sink.

\*/

**public** List<SerializableBlockStateDescriptor> listSourcesForSink(String diagramId,String blockName) ;

/\*\*

\* **@param** diagramId of the parent diagram

\* **@param** blockName name of the block within the diagram

\* **@return** a colon-separated path to the specified block. The path includes

\* the project name.

\*/

**public** String pathForBlock(String diagramId,String blockName);

/\*\*

\* Execute reset() on a specified block

\* **@param** diagramId of the parent diagram

\* **@param** blockName name of the block within the diagram

\*/

**public** **void** resetBlock(String diagramId,String blockName) ;

/\*\*

\* Execute reset() on every block on the diagram

\*/

**public** **void** resetDiagram(String diagramId) ;

/\*\*

\* Determine whether or not the indicated resource is known to the controller.

\*/

**public** **boolean** resourceExists(**long** projectId,**long** resid) ;

/\*\*

\* Send a signal to all blocks of a particular class on a specified diagram.

\* This is a "local" transmission. The diagram is specified by a tree-path.

\* There may be no successful recipients.

\*

\* **@param** diagramId

\* **@param** command string of the signal

\* **@param** message embedded in the transmitted signal

\* **@param** arg also a component of the transmitted signal

\*/

**public** **boolean** sendLocalSignal(String diagramId,String command,String message,String arg) ;

/\*\*

\* Set the state of every diagram that is a member of the application to

\* the specified value.

\* **@param** appname

\* **@param** state

\*/

**public** **void** setApplicationState(String appname, String state);

/\*\* Update all changed properties for a block

\* **@param** duuid diagram unique Id

\* **@param** buuid block unique Id

\*/

**public** **void** setBlockProperties(UUID duuid,UUID buuid, Collection<BlockProperty> props ) ;

/\*\* Update a single changed property for a block

\* **@param** duuid diagram unique Id

\* **@param** buuid block unique Id

\* **@param** property the changed property

\*/

**public** **void** setBlockProperty(UUID duuid,UUID buuid,BlockProperty property ) ;

**public** **void** setDiagramState(Long projectId, Long resourceId, String state) ;

**public** **void** setDiagramState(String diagramId, String state);

/\*\*

\* Save a value into the HSQL database table associated with the toolkit. The

\* table contains name-value pairs, so any name is allowable.

\* **@param** propertyName name of the property for which a value is to be set

\* **@param** the new value of the property.

\*/

**public** **void** setToolkitProperty(String propertyName,String value) ;

/\*\*

\* Start the block execution engine in the gateway.

\*/

**public** **void** startController() ;

/\*\*

\* Shutdown the block execution engine in the gateway.

\*/

**public** **void** stopController() ;

/\*\*

\* Direct the blocks in a specified diagram to report their

\* status values. This is in order to update the UI.

\*/

**public** **void** triggerStatusNotifications() ;

/\*\* Update connections for a block. New connections will be added, old connections

\* may undergo a type conversion.

\* **@param** duuid diagram unique Id

\* **@param** buuid block unique Id

\*/

**public** **void** updateBlockAnchors(UUID duuid,UUID buuid, Collection<SerializableAnchor> anchors ) ;

}

Figure 13 – Application Scripting Interface

Access to the currently configured database and tag provider are available through the *getToolkitProperty*() call. The production and isolation databases are properties: Database and SecondaryDatabase, respectively. Production and isolation tag providers are: Provider and SecondaryProvider.

## Extension Functions

The toolkit provides capabilities for the user to inject python extensions that wiil be called at opportune spots in the life cycle of specific blocks. These functions are identified by the keys listed below. (The argument list is also provided).

* app-add-script (uuid) – this is called with the when an application node is created in the getway. This can either be the result of a user action in the UI, a project load or import.
* app-clone- script (uuid1,uuid2)
* app-delete-script (uuid) – this is called when an application node is deleted in the UI.
* app-get-aux-script (uuid,properties)
* app-set-aux-script (uuid,properties)
* app-update-script (name,uuid) – this is called when the name of an application node is changed. The *name* argument is the former name.
* fam-add-script (uuid) – this is called when a family node is created in the getway. This can either be the result of a user action in the UI, a project load or import. Note that the handler object cn be used to obtain the family’s application.
* fam-clone- script (uuid1,uuid2)
* fam-delete-script (uuid) – this is called with a *ProcessFamily* when a family node is deleted in the UI.
* fam-get-aux-script (uuid,properties)
* fam-set-aux-script (uuid,properties)
* fam-update-script (name,uuid) – this is called when the name of a family node is changed in the gateway. The first argument is the old name. This can either be the result of a user action in the UI, a project load or import.

A “handler” is supplied as a means of obtaining context within the script. The class com.ils.blt.gateway.PythonRequestHandler may be instantiated with a no-argument constructor within the extension scripts.

The “uuid” arguments referenced above are string versions of universal unique identifiers for the application or family.

The scripts which correspond the the keys above may be defined in a right-click popup-dialog associated with the root toolkit node in the Designer’s naviagation tree. The values pertain to all projects. Once defined in this dialog, they permanantly reside in Ignition’s persistent storage.

# Appendix A. Debugging

## A.1 Logging

The Ignition *LoggerEx* class is used throughout to implement logging for purposes of debugging code. This class, based on Log4j, is configurable at run-time by changing log levels in any of Gateway, Designer or Client scopes as appropriate. Separately-configurable loggers are created for each Java package. Available log levels are: ERROR, WARN, INFO, DEBUG, and TRACE. INFO is the default.

# Appendix B: Functional Testing

## B.1 Block Test Project

A functional test project named *blt\_block\_test.proj* is supplied as part of project deliverables. Its goal is to execute PyUnit tests that comprehensively exercise interfaces of each block within the Block Language Toolkit. The test project has the following prerequites:

* BLT Module – this contains the core toolkit as well as the blocks to be tested.
* BLTT Module – this contains a mock diagram, a “test harness” that encapsulates the block under test.
* Unit-Test Module – this module provides facilities for setting tag values from CSV-formatted files.
* SQLTags provider named “TAG”

## B.2 Block Language Toolkit Test Module

The Unit-Test module is a general-purpose Ignition module developed by ILS Automation to assist with PyUnit tests. In particular, it provides a scripting interface for the introduction of test-data into the project.

### B.2.1 Mock Diagram

The core premise of the functional testing is that a block can be completely tested through its well-defined interfaces. This is accomplished using a special “mock” diagram with the following characteristics:

* Accepts exactly one block, the “block under test”
* Stubs connections to all block inputs and output
* Arbitrarily presents data on inputs
* Monitors all “transmissions” from the block
* Injects pertinent “transmissions” for the block to receive
* Reads results at output stubs

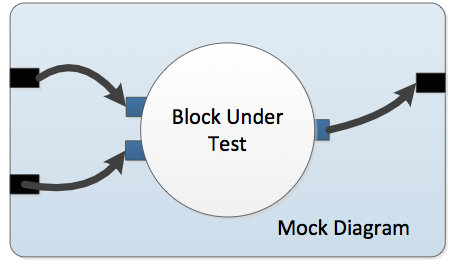


Figure 13 – Mock Diagram

### B.2.2 Scripting Interface

The following functions are provided by the BLT-test module “data” scripting interface. These functions provide for interaction between the Python functional test script and the mock diagram.

system.ils.test.mock. createMockDiagram

D e sc r i p t i o n

Create a new mock diagram and add it to the list of diagrams known to the BlockController. The diagram has no valid resourceId and so is never saved permanently. It never shows in the designer. This call does not start subscriptions to tag changes. Subscriptions are triggered in response to a "start" call. This should be made after all to mock inputs and outputs are defined.

The diagram holds exactly one block, the "Unit Under Test".

Syntax

uuid = createMockDiagram(blockClass)

Parameters

String blockClass – the class of block to be tested.

Return

UUID – the unique ID of the newly created mock diagram.

system.ils.test.mock.addMockInput

D e sc r i p t i o n

Define an input connected to the named port. This input is held as part of the mock diagram. Once defined, the input cannot be deleted. A separate (duplicate) input should be defined for every connection coming into the named port.

Syntax

addMockInput (diagram,path,type,port)

Parameters

UUID diagram – the unique ID of the mock diagram

String path – the fully qualified tag path of the tag that triggers this input.

String type – the data type of the tag (STRING,INTEGER,DOUBLE, BOOLEAN,OBJECT)

String port – name of the port on the block-under-test to which this input must connect.

system.ils.test.mock.addMockOutput

D e sc r i p t i o n

Define an output connected to the named port. This output is held as part of the mock diagram. Once defined, the output cannot be deleted. It does not make sense to define more than one output for each outgoing port on the block-under-test.

Syntax

addMockOutput (diagram,path,type,port)

Parameters

UUID diagram – the unique ID of the mock diagram

String path – the fully qualified tag path of the tag that is set as a result of this output.

String type – the data type of the tag (STRING,INTEGER,DOUBLE, BOOLEAN,OBJECT)

String port – name of the port on the block under test to which this output must connect.

system.ils.test.mock.deleteMockDiagram

D e sc r i p t i o n

Remove the specified test harness from the execution engine (block controller). The harness is stopped before being deleted.

Syntax

deleteMockDiagram (diagram)

Parameters

UUID diagram – the unique ID of the mock diagram

system.ils.test.mock.forcePost

D e sc r i p t i o n

Force the block under test to present a specified value on a specified output port.

Syntax

forcePost (diagram,port,value,)

Parameters

UUID diagram – the unique ID of the mock diagram

String port – name of the output port of the block-under-test on which the value is to be sent.

Object value – the new value. The value should be of type Boolean, Double, Integer, or String. In addition the value must be appropriate to the port.

system.ils.test.mock.getState

D e sc r i p t i o n

Return the execution state of the block-under-test. This is used in conjunction with tests of the *reset* function.

Syntax

getState (diagram)

Parameters

UUID diagram – the unique ID of the mock diagram

Return

String state, the execution state of the diagram. ACTIVE or INITIALIZED.

system.ils.test.mock.isLocked

D e sc r i p t i o n

Determine whether or not the block-under-test is locked.

Syntax

isLocked (diagram)

Parameters

UUID diagram – the unique ID of the mock diagram

Return

boolean value – true if the block is currently locked, otherwise false.

system.ils.test.mock.readValue

D e sc r i p t i o n

Read the current value captured by the named output port.

Syntax

value = readValue (diagram,port)

Parameters

UUID diagram – the unique ID of the mock diagram

String port – name of the output port on the block-under-test on which to read the current value.

Return

QualifiedValue value – the current value held by the specified port

system.ils.test.mock.reset

D e sc r i p t i o n

Reset the block-under-test. This is accomplished by calling its reset() method.

Syntax

reset (diagram)

Parameters

UUID diagram – the unique ID of the mock diagram

system.ils.test.mock.setLocked

D e sc r i p t i o n

Lock or unlock the block-under-test.

Syntax

setLocked (diagram,flag)

Parameters

UUID diagram – the unique ID of the mock diagram

boolean flag – true to lock the block-under-test, false to unlock it.

system.ils.test.mock.setProperty

D e sc r i p t i o n

Set the value of the named property. This value ignores any type of binding. If the property is bound to a tag, then the value should be set by writing to that tag.

Syntax

setProperty (diagram,name,value)

Parameters

UUID diagram – the unique ID of the mock diagram

String name – name of the property of the block-under-test which is to be set.

Integer index – index of the connection into the named port. The index is zero-based.

Object value – the new value. The value should be of type Boolean, Double, Integer, or String.

system.ils.test.mock. startMockDiagram

D e sc r i p t i o n

Start the test harness by activating subscriptions for bound properties and mock inputs.

Syntax

startMockDiagram (diagram)

Parameters

UUID diagram – the unique ID of the mock diagram

system.ils.test.mock. stopMockDiagram

D e sc r i p t i o n

Stop all property updates and input receipt by canceling all active subscriptions involving the harness.

Syntax

stopMockDiagram (diagram)

Parameters

UUID diagram – the unique ID of the mock diagram

system.ils.test.mock.writeValue

D e sc r i p t i o n

Simulate data arriving on the nth connection into the named input port.

Syntax

writeValue (diagram,port,index,value,quality)

Parameters

UUID diagram – the unique ID of the mock diagram

String port – name of the output port on the block-under-test on which to read the current value.

Integer index – index of the connection into the named port. The index is zero-based.

String value – the new value

String qualuty – the quality of the new value

Return

long timestamp – the current system in millisecs (time since start of the Unix epoch) corresponding to the creation of the value being presented on the block input. This value is useful for testing the propogation of timestamps through the block.

# Appendix C: Application Testing

## C.1 Introduction

The functional testing described in the previous section is designed to exercise each interface of each block class in isolation. The application test facility, on the other hand, tests the completed application as a whole. It is designed as an external harness that drives and monitors the production application at an accelerated rate.

This section describes features of the Block Language Toolkit implemented specifically in support of this facility.

## C.2 Time Acceleration

All blocks use the WatchDogTimer class for timing ....

# Appendix D: Adding Math Functions

## D.1 Arithmetic Block

The *Arithmetic* block allows the customer developer to extend its repertoire with additional functions from the Apache Commons-Math library. Functions can also be custom coded. This section presents a discussion of the prototype block provided by ILS Automation.

## D.2 Math Library

The Apache Commons-Math library is an open source collections of mathematical functions. It is already linked into the Block Language Toolkit and available for use. Documentation in the form of Javadoc may be found at

[*http://commons.apache.org/proper/commons-math/javadocs/api-3.3/index.html*](http://commons.apache.org/proper/commons-math/javadocs/api-3.3/index.html)*.*

## D.3 Example Modification

The prototype Arithmetic block can be found in Ignition external python in package *xom.block.Arithmetic.* The steps below highlight areas of the code which must be changed in order to add of new functions. These functions are not currently available with the built-in blocks. Additionally only functions available in the Apache Commons-Math library are available for inclusion in this particular block. Moreover these functions must be of the type that accept a single input and return a single output.

## D.3.1 Function List

The list of available functions is presented to the block-user in a pull-down list inside the property editor.

Extension of the function list is a straightforward modification to the property binding in the following initialization method:

def **initialize**(*self*):

*self*.className = *'xom.block.arithmetic.Arithmetic'*

*self*.properties[*'Function'*] =

{*'value'*:*''*,*'editable'*:*'True'*,*'bindingType'*:*'OPTION'*,

*'binding'*:*'ABS,CEILING,COSINE,FLOOR,SINE,TANGENT,*

*TO\_RADIAN'*}

## D.3.2 Import Statement

Addition of a new function will require a new import statement to include the code from the math library. The sample below shows the statement used for the *Abs()* function.

import org.apache.commons.math3.analysis.function.Abs as Abs

## D.3.3 Accept Function

When a block receives a new value on its input connection, its *acceptValue()* function is executed. The code below shows this code in its entirety for the prototype function. Additional functions involve creation of more *elif* clauses.

# Called when a value has arrived on one of our input ports.

# Compute the result, then propagate on the output.

def **acceptValue**(*self*,invalue,quality,port):

function = *self*.properties.get(*'Function'*,{}).get(*"value"*,*""*)

value = invalue # Default behavior is a pass-through

if len(function)>0:

if function == *'ABS'*:

absolute = Abs()

value = absolute.value(value)

elif function == *'CEILING'*:

ceiling = Ceiling()

value = ceiling.value(value)

elif function == *'COSINE'*:

cosine = Cosine()

value = cosine.value(value)

elif function == *'FLOOR'*:

floor = Floor()

value = floor.value(value)

elif function == *'SINE'*:

sine = Sine()

value = sine.value(value)

elif function == *'TANGENT'*:

tan = Tangent()

value = tan.value(value)

elif function == *'TO\_RADIAN'*:

value = 0.0174532925\*invalue;

*self*.postValue(*'out'*,value,*'good'*)

# Appendix E: Custom Actions

## E.1 Action Block

The *Action* block allows a customer-developer to extend the existing block language with custom python scripts. The configured script is executed when the input to the block newly matches a specified truth-state.

The python package must be in the external python area. The block does not have access to *project* or *shared* package scopes, which are the packages that are available via the Designer editor. Instead the recommended development environment is *Eclipse*. The actions scripts will be accessible to all projects.

## E.1 Call Signature

The script is given the object that represents the originating block as its sole argument. The block is a derivative of class ils.block.basicblock.BasicBlock. A rich environment can be created from this object.

## E.2 Request Handler

Each python block instance is supplied with a *handler* object. The handler is of class com.ils.blt.gateway.PythonRequestHandler. It offers access to a large number of project variables. For details of the methods supported see section 7.1.2.5.

## E.3 Example

The example below demonstrates acquisition of contextual parameters via the block argument. If the *Action* block “script” property held the value:

xom.actions.demo.act

then the file containing the code should be located in;

<*IgnitionInstallation*>/lib/pylibs/xom/actions/demo.py

Each level of the directory hierarchy should contain an empty \_\_init\_\_.py file to aid the python parser in traversing the path.

*'''*

*Demonstration of a custom action module*

*'''*

def **act**(block):

print block.getClassName()

print block.uuid

print block.parentuuid

# The handler is a com.ils.blt.gateway.PythonRequestHandler

print block.handler.getDefaultDatabase(block.parentuuid)

print block.handler.getDefaultTagProvider(block.parentuuid)