

Technical Document

Niagara^{AX-3.x} kitControl Guide

September 12, 2013



Niagara^{AX} kitControl Guide

Confidentiality Notice

The information contained in this document is confidential information of Tridium, Inc., a Delaware corporation (“Tridium”). Such information, and the software described herein, is furnished under a license agreement and may be used only in accordance with that agreement.

The information contained in this document is provided solely for use by Tridium employees, licensees, and system owners; and, except as permitted under the below copyright notice, is not to be released to, or reproduced for, anyone else.

While every effort has been made to assure the accuracy of this document, Tridium is not responsible for damages of any kind, including without limitation consequential damages, arising from the application of the information contained herein. Information and specifications published here are current as of the date of this publication and are subject to change without notice. The latest product specifications can be found by contacting our corporate headquarters, Richmond, Virginia.

Trademark Notice

BACnet and ASHRAE are registered trademarks of American Society of Heating, Refrigerating and Air-Conditioning Engineers. Microsoft and Windows are registered trademarks, and Windows NT, Windows 2000, Windows XP Professional, and Internet Explorer are trademarks of Microsoft Corporation. Java and other Java-based names are trademarks of Sun Microsystems Inc. and refer to Sun's family of Java-branded technologies. Mozilla and Firefox are trademarks of the Mozilla Foundation. Echelon, LON, LonMark, LonTalk, and LonWorks are registered trademarks of Echelon Corporation. Tridium, JACE, Niagara Framework, Niagara^{AX} Framework, and Sedona Framework are registered trademarks, and Workbench, WorkPlace^{AX}, and ^{AX}Supervisor, are trademarks of Tridium Inc. All other product names and services mentioned in this publication that is known to be trademarks, registered trademarks, or service marks are the property of their respective owners.

Copyright and Patent Notice

This document may be copied by parties who are authorized to distribute Tridium products in connection with distribution of those products, subject to the contracts that authorize such distribution. It may not otherwise, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior written consent from Tridium, Inc.

Copyright © 2011 Tridium, Inc.

All rights reserved. The product(s) described herein may be covered by one or more U.S or foreign patents of Tridium.

CONTENTS

Preface	v
----------------------	----------

About kitControl	1-1
-------------------------------	------------

Application for kitControl components	1-1
--	------------

Types of kitControl components	1-2
---	------------

Location for kitControl components	1-3
---	------------

Extensions and kitControl components	1-3
---	------------

Components that cannot receive extensions	1-4
---	-----

About kitControl Alarm components	1-4
--	------------

About Constant components	1-4
--	------------

About Conversion components	1-5
--	------------

Status value to simple value	1-5
------------------------------------	-----

Simple value to status value	1-7
------------------------------------	-----

Status value to status value	1-7
------------------------------------	-----

About Energy components	1-8
--------------------------------------	------------

About HVAC components	1-8
------------------------------------	------------

About Latch components	1-8
-------------------------------------	------------

Types of Latch Components	1-9
---------------------------------	-----

Types of Latch Component Properties	1-9
---	-----

About the Latch Action	1-9
------------------------------	-----

Latch Examples	1-10
----------------------	------

About Logic components	1-11
-------------------------------------	-------------

About Math components	1-12
------------------------------------	-------------

About Select components	1-13
--------------------------------------	-------------

About String components	1-13
--------------------------------------	-------------

About Timer components	1-13
-------------------------------------	-------------

About Util components	1-13
------------------------------------	-------------

kitControl Component Guides	2-1
--	------------

Alphabetical list of kitControl components	2-1
---	------------

kitControl-AbsValue	2-3
---------------------------	-----

kitControl-Add	2-3
----------------------	-----

kitControl-AlarmCountToRelay	2-3
------------------------------------	-----

kitControl-And	2-4
----------------------	-----

kitControl-ArcCosine	2-5
----------------------------	-----

kitControl-ArcSine	2-5
--------------------------	-----

kitControl-ArcTangent	2-5
-----------------------------	-----

kitControl-Average	2-5
--------------------------	-----

kitControl-BooleanDelay	2-5
-------------------------------	-----

kitControl-BooleanConst	2-6
-------------------------------	-----

kitControl-BooleanLatch	2-6
-------------------------------	-----

kitControl-BooleanSelect	2-6
--------------------------------	-----

kitControl-BooleanSwitch	2-6
--------------------------------	-----

kitControl-BooleanToStatusBoolean	2-6
kitControl-BqlExprComponent	2-6
kitControl-ChangeOfStateCountAlarmExt	2-6
kitControl-Cosine	2-7
kitControl-Counter	2-7
kitControl-CurrentTime	2-8
kitControl-DegreeDays	2-8
kitControl-DigitalInputDemux	2-9
kitControl-Divide	2-11
kitControl-DoubleToStatusNumeric	2-11
kitControl-ElapsedActiveTimeAlarmExt	2-11
kitControl-ElectricalDemandLimit	2-12
kitControl-EnumConst	2-17
kitControl-EnumLatch	2-17
kitControl-EnumSelect	2-17
kitControl-EnumToStatusEnum	2-17
kitControl-EnumSwitch	2-17
kitControl-Equal	2-17
kitControl-Exponential	2-17
kitControl-Factorial	2-17
kitControl-FloatToStatusNumeric	2-18
kitControl-GreaterThan	2-18
kitControl-GreaterThanEqual	2-18
kitControl-IntToStatusNumeric	2-18
kitControl-InterstartDelayControl	2-18
kitControl-InterstartDelayMaster	2-18
kitControl-LeadLagCycles	2-18
kitControl-LeadLagRuntime	2-19
kitControl-LessThan	2-20
kitControl-LessThanEqual	2-20
kitControl-LogBase10	2-21
kitControl-LogNatural	2-21
kitControl-LongToStatusNumeric	2-21
kitControl-LoopAlarmExt	2-21
kitControl-LoopPoint	2-21
kitControl-Maximum	2-25
kitControl-Minimum	2-26
kitControl-MinMaxAvg	2-26
kitControl-Modulus	2-26
kitControl-Multiply	2-26
kitControl-MultiVibrator	2-26
kitControl-Negative	2-26
kitControl-NightPurge	2-26
kitControl-Not	2-28
kitControl-NotEqual	2-29
kitControl-NumericBitAnd	2-29
kitControl-NumericBitOr	2-30
kitControl-NumericBitXor	2-30
kitControl-NumericConst	2-30
kitControl-NumericDelay	2-30
kitControl-NumericLatch	2-31
kitControl-NumericSelect	2-31
kitControl-NumericSwitch	2-31
kitControl-NumericToBitsDemux	2-32
kitControl-NumericUnitConverter	2-32
kitControl-OneShot	2-32
kitControl-OptimizedStartStop	2-33
kitControl-Or	2-37
kitControl-OutsideAirOptimization	2-38
kitControl-Power	2-39
kitControl-Psychrometric	2-39
kitControl-Ramp	2-40
kitControl-Random	2-40
kitControl-Reset	2-41
kitControl-RaiseLower	2-41
kitControl-SequenceBinary	2-43
kitControl-SequenceLinear	2-44
kitControl-SetpointLoadShed	2-46
kitControl-SetpointOffset	2-47
kitControl-ShedControl	2-47
kitControl-Sine	2-48
kitControl-SineWave	2-48
kitControl-SlidingWindowDemandCalc	2-48
kitControl-SquareRoot	2-50
kitControl-StatusBooleanToBoolean	2-50

kitControl-StatusDemux 2-50
kitControl-StatusEnumToEnum 2-50
kitControl-StatusEnumToInt 2-50
kitControl-StatusEnumToStatusBoolean 2-50
kitControl-StatusEnumToStatusNumeric 2-50
kitControl-StatusNumericToDouble 2-50
kitControl-StatusNumericToFloat 2-50
kitControl-StatusNumericToInt 2-50
kitControl-StatusNumericToStatusEnum 2-51
kitControl-StatusNumericToStatusString 2-51
kitControl-StatusStringToStatusNumeric 2-51
kitControl-StringConcat 2-51
kitControl-StringConst 2-51
kitControl-StringIndexOf 2-51
kitControl-StringLatch 2-51
kitControl-StringLen 2-51
kitControl-StringSelect 2-51
kitControl-StringSubstring 2-52
kitControl-StringTest 2-52
kitControl-StringToStatusString 2-52
kitControl-StringTrim 2-52
kitControl-Subtract 2-52
kitControl-Tangent 2-52
kitControl-TimeDifference 2-52
kitControl-Tstat 2-53
kitControl-Xor 2-53

CONTENTS

Preface

[Document Change Log](#)

Document Change Log

Updates (changes/additions) to this *NiagaraAX kitControl Guide* document are listed below.

- Updated: September 12, 2013
Made correction to **Mean Temp** formula description in “[kitControl-DegreeDays](#)” on page 2-9.
- Updated: March 24, 2011
Made AX-3.6-related changes in a few areas of the document, including the “[About Conversion components](#)” section, noting that the automatic “conversion links” feature starting in AX-3.6 should typically make these components unnecessary. Two related subsections were added to the “[Status value to simple value](#)” conversion components section: “[About null values](#)” on page 1-5, and “[Null input handling changes in AX-3.6](#)” on page 1-6. Also added was a summary description for the BQL Expression component (“[kitControl-BqlExprComponent](#)” on page 2-7), available starting in AX-3.6.
- Updated: June 10, 2010
Added missing description for the [SlidingWindowDemandCalc](#) component’s “Meter Rollover” property, available since AX-3.2 and build 3.1.31 and later. The property sheet screen shown for this component ([Figure 2-34](#) on page 53) now includes this property.
- Updated: March 11, 2010
Added summary descriptions for the new [BooleanSwitch](#), [Factorial](#), and [Modulus](#) components starting in AX-3.5, also an entry for the “Earliest Start Time” addition to [OptimizedStartStop properties](#). Corrections were made for loop output formulas shown in the [LoopPoint](#) section, along with a few other minor changes (note that loop behavior itself remains unchanged).
Removed two extraneous components from the [kitControl Component Guides](#) section. This document applies to all NiagaraAX revisions, with any revision-dependent items noted.
- Updated: June 25, 2008
Added descriptions for the following new components: [DigitalInputDemux](#), [RaiseLower](#).
- Revised: April 18, 2008
Minor corrections and additions made to April 7, 2008 revisions based on technical document reviews.
- Revised: April 7, 2008
Added new descriptions for the following components: [AlarmCountToRelay](#), [NumericToBitsDemux](#).
Added more detail to the following component descriptions: [BooleanDelay](#), [ElectricalDemandLimit](#), [OneShot](#), [OptimizedStartStop](#), [ShedControl](#), [TimeDifference](#).
Added more detail and related sections to “[About Latch components](#)”.
Fixed broken online help links, and made minor edits.
- Revised: February 23, 2008
Applied new style to complete document.
Edited links to reflect NiagaraAX document-set reconfiguration.
- Revised: August 23, 2006
Provided setup and operation details for the [LoopPoint](#), and truth tables for Logic components [And](#), [Or](#), [Not](#), and [Xor](#). More details and examples are included for components [EnumSelect](#), [NumericBitAnd](#), [NumericBitOr](#), and [NumericBitXor](#).
Reversed the order of this change log to list newest document changes at the top.
- Revised: November 30, 2005

Minor changes only. Added convenience links in “[Alphabetical list of kitControl components](#)” on page 2-1, plus a link back to this page from online Help “Guide on Target” for any kitControl component. Fixed several screencap figures and links.

- Revised: September 15, 2005

Minor changes only. Fixed links and used newer cover design.

- Revised: June 24, 2005

Minor changes only. Added Copyright and Trademarks to preface, fixed a few links.

- Draft: June 15, 2005

(Initial change log). Added additional descriptions of kitControl components: [ElectricalDemandLimit](#), [EnumSwitch](#), [NightPurge](#), [OptimizedStartStop](#), [OutsideAirOptimization](#), [PsychrometricRandom](#), [SetpointLoadShed](#), [SlidingWindowDemandCalc](#), [StatusEnumToInt](#), [StatusEnumToStatusBoolean](#), [StatusEnumToStatusNumeric](#), [StatusNumericToStatusString](#), [StatusStringToStatusNumeric](#).

CHAPTER 1

About kitControl

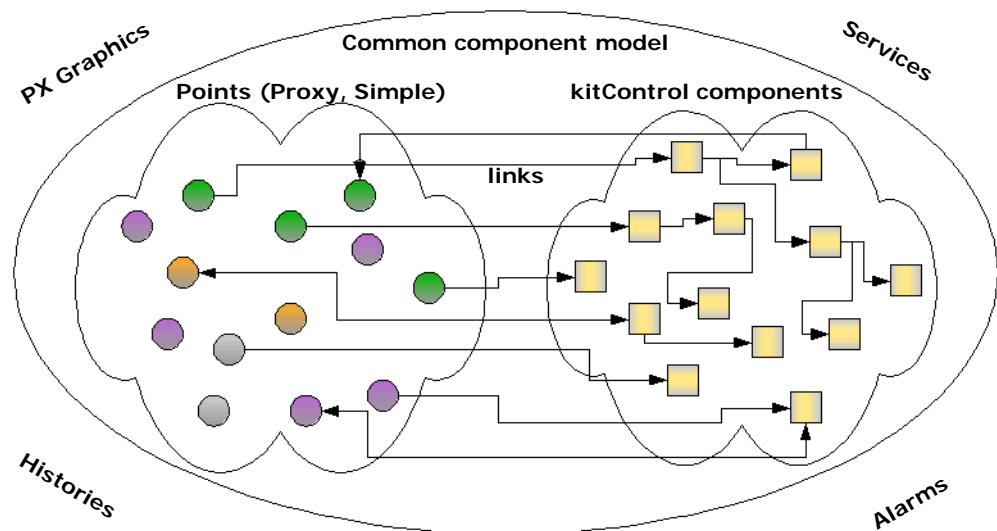
Important main kitControl topics include:

- [Application for kitControl components](#)
- [“Types of kitControl components” on page 1-2](#)
- [“Location for kitControl components” on page 1-3](#)
- [“Extensions and kitControl components” on page 1-3](#)

Application for kitControl components

The kitControl palette contains various components that you can use in combination with points, both *simple* control points and *proxy points*. Whereas proxy points read (and possibly write) data from (and to) remote devices, kitControl provides the “data manipulation” building blocks that let you further process that data. Included are HVAC components like a PID loop and sequencers, a variety of boolean logic components, math components for numeric values, and miscellaneous others. Together with proxy points and schedules, kitControl components provide the basis of the “common component model” for modeling control logic. See “About Scheduling” in the *NiagaraAX-3.x User Guide*.

Figure 1-1 Conceptual use for kitControl in object model

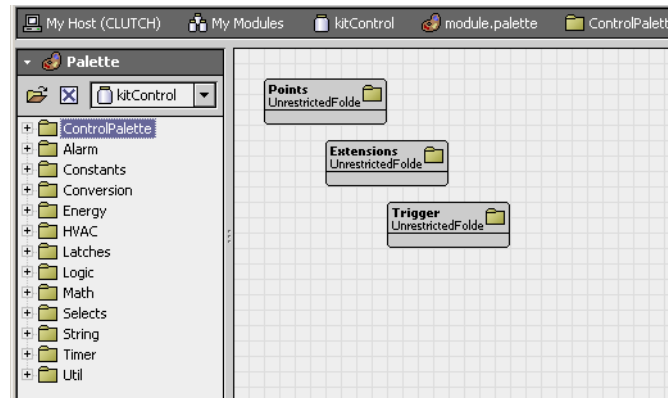


Note: Usage of kitControl components is entirely optional. It is possible to build a monitoring-type application where only proxy points (and perhaps a few simple control points) are used. This would allow real-time data monitoring, plus user-invoked “action” overrides through writable points’ right-click command menus. As needed, you could also add extensions to the proxy points for alarming and history collections.

Types of kitControl components

In total, the kitControl palette contains over 90 unique components across 12 *folder* categories (not counting ControlPalette). As shown in [Figure 1-2](#), folders in the palette reflect the types of components.

Figure 1-2 Folders in palette kitControl



Note: Currently, the kitControl palette includes all **control** palette components, under its “ControlPalette” folder. This is a convenience, allowing you to open kitControl in your palette side bar, and still have access to simple control points, extensions, and timers, in addition to the kitControl components found under its other folders. See “[Alphabetical list of kitControl components](#)” on page 2-1 for a complete list of components found only in kitControl.

By palette folder, types of kitControl components include:

- **ControlPalette**
Equivalent to contents of control palette, including subfolders Points, Extensions, and Triggers. For more details, see the *NiagaraAX-3.x User Guide* sections “About control points”, “About control extensions”, and “About control triggers”.
- **Alarm**
Contains 3 extensions for alarming. One is expressly for a LoopPoint, for “setpoint-deviation” alarming. The others provide alarming options for a Boolean point with DiscreteTotalizerExt. A fourth component provides alarm count monitoring of any Alarm Class, and includes a boolean “relay” output. For more details, see “[About kitControl Alarm components](#)” on page 1-4.
- **Constants**
Contains 4 components, one for each data category. Each provides a linkable status-type output, and a “Set” action for changing value. See “[About Constant components](#)” on page 1-4 for details.
- **Conversion**
Contains 19 components that mainly convert status values to simple values, and vice versa. Also has other special conversion types. See “[About Conversion components](#)” on page 1-5 for details.
Note: Starting in AX-3.6, conversion components may be unnecessary—as linking directly between status values and simple values, or even between different data types, is supported. For details, please refer to the *Engineering Notes II* document NiagaraAX Conversion Links.
- **Energy**
Contains 10 components for typical energy functions, such as degree day calculation and electrical demand limiting. See “[About Energy components](#)” on page 1-8.
- **HVAC**
Contains 9 components for typical HVAC functions, such as for interstart delay, lead-lag control and sequence control. Included are a Tstat (thermostat) and LoopPoint (PID loop) component. See “[About HVAC components](#)” on page 1-8.
- **Latches**
Contains 4 latch components, one for each data category. See “[About Latch components](#)” on page 1-8.
- **Logic**
Contains 10 logic components, each with a StatusBoolean output. Starting in AX-3.6, an example “Expr” (BQL Expression) component is also included—named “ExprLogic”. For more details, see “[About Logic components](#)” on page 1-11.

- **Math**
Contains 23 components for processing one or more *numeric* input values, and producing a Status-Numeric output. Starting in AX-3.6, an example “Expr” (BQL Expression) component is also included—named “ExprMath”. See “[About Math components](#)” on page 1-12 for more details.
- **Selects**
Contains 4 select components, one for each data category. See “[About Select components](#)” on page 1-13.
- **String**
Contains 6 components with one or more StatusString inputs. See “[About String components](#)” on page 1-13.
- **Timer**
Contains 5 components: 3 timer types (BooleanDelay, NumericDelay, and OneShot), and 2 “absolute time” types (CurrentTime, TimeDifference). See “[About Timer components](#)” on page 1-13.
- **Util**
Contains 16 various utility components, including an “Expr” (BQL Expression) component added starting in AX-3.6. See “[About Util components](#)” on page 1-13 for more details.

Location for kitControl components

As with simple control points, you can copy kitControl components to any folder or component needed in the station. (This varies from proxy points—see “Location of proxy points” in the *Drivers Guide*). This includes under any device’s Points extension (container), or any subfolder underneath.

Note: *There are two competing “best practice” philosophies about locating your control logic.*

1. Philosophy A, where you should only have proxy points under a device’s Points container. You can add extensions (control, history, and alarm), as needed, to proxy points. However, other components (kitControl components, simple control points, schedules) are located under a central folder *not* under the station’s Driver architecture—but instead in subfolders under a main “Logic” folder (by convention) created in the root of the station’s Config container.

This method requires many “internal” links between proxy points and kitControl components. It often makes following control logic harder, because you don’t see most links except as “knobs.” In addition, it makes applications much less portable (you can’t copy it all by selecting a single device container, as you might otherwise using philosophy B).

2. Philosophy B, where you add any needed kitControl components, simple control points, and schedules under each device’s Points container (either directly, or in subfolders). This allows you to create more “local” links between the device’s proxy points and other components.

The original intention of philosophy A was to establish a logic “convention” that allows universal support of different application types. However, philosophy B offers more “portability” of each application, allowing easy replication and reuse at a “device level.” In general, use of philosophy B is more common—locate kitControl components where they are needed.

Extensions and kitControl components

You can add point extensions to *many* kitControl components, for example an alarm extension, history extension, or perhaps a control extension. For exceptions, see “[Components that cannot receive extensions](#)” on page 1-4. For general details on extensions, see “About point extensions” in the *User Guide*.

Some examples of using extensions with kitControl components include the following:

- **Average object (Math folder)**
Inputs are linked to multiple proxy NumericPoints, each representing a room temperature. The Average object represents a “Zone” temperature (average). To this object you may wish to add an alarm extension (OutOfRangeAlarmExt) and history extension (NumericInterval).
- **And object (Logic folder)**
Inputs are linked to multiple proxy BooleanPoints, each representing “fan status” (Off or On). The And object is linked to downstream control logic. To track when all fans are running, you may wish to a history extension (BooleanChangeOfValue) and perhaps a control extension to collect runtime (DiscreteTotalizerExt).
- **NumericSwitch object (Util folder)**
Inputs are linked to two proxy NumericWritables, each representing a power rate (kW). The object output is linked to downstream control logic (and represents the current “switched” rate). To totalize this effective rate into energy accumulation, you add a control extension (NumericTotalizerExt) and add proper scaling to collect kWh.

Components that cannot receive extensions

Some kitControl components are *not* based on “simple control points” (ControlPoint). You cannot add any extensions to these components. If you try, you receive an “illegal parent” error message.

Components in kitControl that *cannot* receive extensions include:

- [Constants](#) components (any).
- [Conversion](#) components (any).
- [Energy](#) components (any).
- [HVAC](#) components (except for [LoopPoint](#), [InterstartDelayControl](#), and [Tstat](#), which *can* have extensions)
- [Latches](#) components (any)
- [Selects](#) components (any)
- [String](#) components (any)
- [Timer](#) components (any)
- [Util](#) components: (except for [BooleanSwitch](#), [EnumSwitch](#), [MultiVibrator](#), [NumericSwitch](#), [Ramp](#), [Random](#), and [SineWave](#), which *can* have extensions).

Note: You can quickly tell if a kitControl object can receive an extension, by seeing if it has the frozen “ProxyExt” (proxy extension). See this by expanding the object in the kitControl palette, Nav tree or viewing the object’s property sheet.

If present, you can add other extensions (providing they are the correct type), for example “Boolean-ChangeOfValue” history extension for a Logic-type object, and so forth.

Also, note this is the only use for the proxy extension in a kitControl object (its value is always “null”)—only control points can be proxy points.

About kitControl Alarm components

The Alarm folder in the kitControl palette contains 3 special-purpose alarm extensions, two of which are for Boolean type points that also have one or more [DiscreteTotalizerExt](#) extensions, as follows:

- [ChangeOfStateCountAlarmExt](#)
- [ElapsedActiveTimeAlarmExt](#)

They provide alarm capability based upon COS count and runtime (elapsed active time). If using either (or both) extensions above, in the parent Boolean point (BooleanPoint, BooleanWritable), you must add (order) them *under* the corresponding DiscreteTotalizerExt.

Also, a [LoopAlarmExt](#) is available. This extension provides a “sliding limit” type alarm for a [LoopPoint](#), based upon control deviation from setpoint.

Finally, an [AlarmCountToRelay](#) component provides configurable monitoring of the number of alarms (alarm count) of a linked Alarm Class component, and features a timed boolean “relay” output.

Note: Starting in AX-3.6, a new alarm extension is available in the “Extensions” folder of the alarm palette: a [StringChangeOfValueExt](#). See the User Guide section “alarm-StringChangeOfValueExt” for details.

About Constant components

The four constant kitControl components are:

- [BooleanConst](#)
- [EnumConst](#)
- [NumericConst](#)
- [StringConst](#)

Each object simply provides a linkable status-type output value, and offers a default “Set” action for changing that value (in the case of the BooleanConst object, *two* default actions are “Active” and “Inactive”). A constant object may be handy when the same target property of multiple components needs to be adjusted simultaneously. Constants may also help clarify logic on a wire sheet.

Constant components are far simpler than writable control points (no priorities, status processing, and so forth).

About Conversion components

In most cases, a kitControl conversion object takes an input value and makes it available on the object's output as a different data *type*. Prior to AX-3.6, you typically use them as a “go-between” to allow linking between different data types.

For example, you use a conversion object to allow a link between a:

- [Status value to simple value](#), or
- [Simple value to status value](#), or
- [Status value to status value](#)

Note: *Starting in AX-3.6, in most cases you no longer need conversion components. Instead, you simply link directly between component slots of different data types, and a “conversion link” is automatically made. Sometimes, further editing of such a link is also possible. For complete details, see the Engineering Notes II document NiagaraAX Conversion Links.*

Also starting in AX-3.6, all of the “Status value to simple value” components in kitControl were extended with optional properties. See the “Status value to simple value” section for details.

Also, a [NumericUnitConverter](#) is available. It is unique in that it does not change the data type (both input and output are StatusNumeric), but changes the actual *numeric value*, based upon unit conversions going from the configured “In Facets” to “Out Facets.” To produce a valid output, you must configure both facets to be under the same category (such as temperature or power, as examples). Otherwise, the NumericUnitConverter has a fault status.

Status value to simple value

Here, these conversion object types are often used in a AX-3.5 or earlier host to permit a link between the:

- Source “Out” of a point, kitControl, or Schedule object (status value), to a
- Target “non status” value of a property in an extension or other component.

For example, to link the “Out” of a Schedule object to the “Enabled” property of a history extension (of a point or object), you link a StatusBooleanToBoolean object between the two.

Conversion components in this category include:

- [StatusBooleanToBoolean](#)
- [StatusEnumToEnum](#)
- [StatusEnumToInt](#)
- [StatusNumericToDouble](#)
- [StatusNumericToFloat](#)
- [StatusNumericToInt](#)

Note: *Starting in AX-3.6, additional properties were added to the conversion components listed above. These properties allow specifying a pre-defined output value in case of a “null” input. Previously (and by default), upon input change to “null”, the conversion output uses the “null value” from the source, typically from a writable point’s “Fallback” slot with null status.*

For more details see the following two sections:

- [“About null values”](#)
- [“Null input handling changes in AX-3.6”](#)

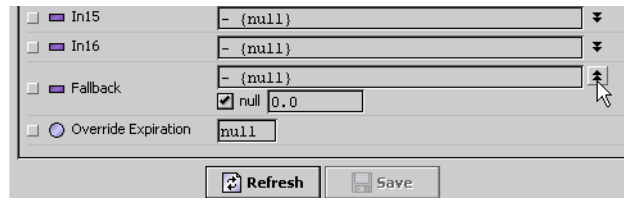
About null values

The four “status value” NiagaraAX data types: StatusBoolean, StatusEnum, StatusNumeric, and StatusString, each hold two pieces of data:

- status
In normal operation, status is “ok”, meaning no status flag or flags are set. Status flags include alarm, overridden, fault, and others, including one for “null”.
- value
The data value portion. If StatusBoolean: value is either true or false, if StatusNumeric: value is a number, if StatusEnum: value is a state (or ordinal integer), or if StatusString: value is a text string. If status is null, this data value is *ignored* (not processed) by any other linked “status value” properties. *However*, a value remains that does correspond with null, utilized only if data conversion from a “status value” to “simple value” occurs.

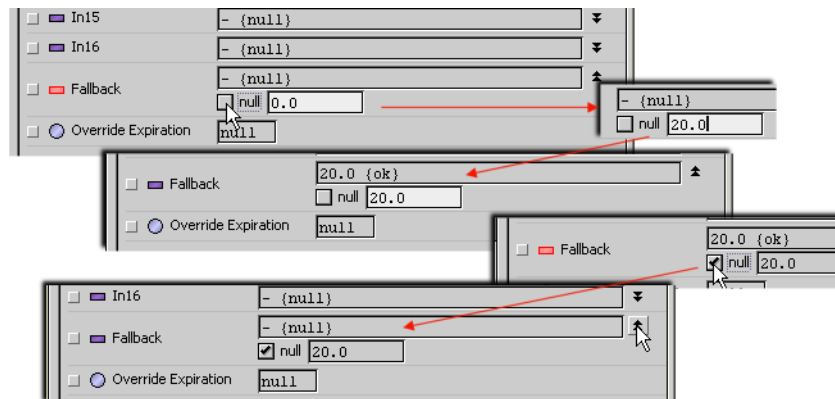
Consider a NumericWritable point, in this case also a proxy point, that positions a damper from 0 to 100%. In the point's configuration, its “Fallback” property has a *default* status of null, with 0.0 value. See [Figure 1-3](#).

Figure 1-3 Default null value for NumericWritable is 0.0



If during configuration the null status is unchecked, and another value entered (and saved), this will now become the new “null value” for this point. In other words, if the null status checkbox is set again this value is now the null value. See Figure 1-4 below.

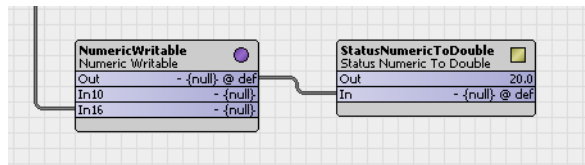
Figure 1-4 Non-default null value for NumericWritable can occur if Fallback has been configured



In this example, the Fallback property was changed from default: “null, 0.0” to 20.0. Then a subsequent change was made to re-select (set) null for Fallback. Note the 20.0 null value remains, as read-only. If only linking between other “StatusNumeric” properties, this is moot, as the null value is ignored—essentially “dropping through” priority inputs.

However, when converting from StatusNumeric to a simple number data type (Double, Float, or Integer), the current null value is used, as shown in Figure 1-4 below.

Figure 1-5 Null input to conversion component (or conversion link) can result in unexpected value



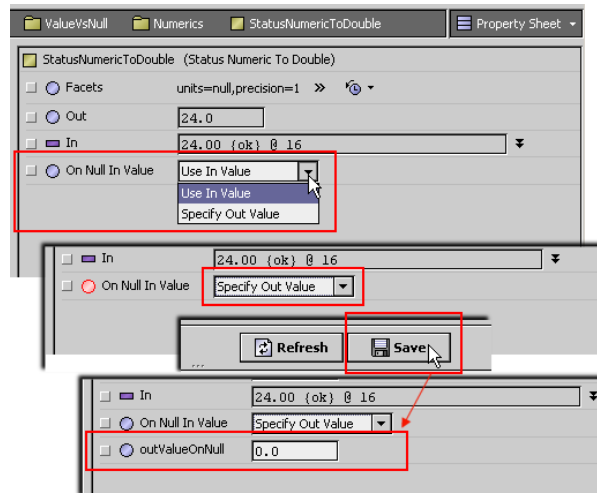
In this case the linked StatusNumericToDouble component has a value of 20.0, sourced by the “null value” coming from the Fallback property of the NumericWritable. More typical—and perhaps even expected, would be a value of 0.0, from Fallback defaults.

To avoid this type of ambiguity, new properties were added to all the “Status value to simple value” conversion components in kitControl, starting in AX-3.6. For more details, see “Null input handling changes in AX-3.6” on page 1-6.

Null input handling changes in AX-3.6

Starting in AX-3.6, all of the “Status value to simple value” conversion components have extended properties that allow optional handling of a “null” input.

Figure 1-6 Example “On Null In Value” property and “outValueOnNull” property in AX-3.6 and later



These new properties are:

- **On Null In Value**
By default, this property is set to the first of two possible values:
 - **Use In Value**
If saved this way, the “Out Value On Null” property below does not appear. Operation remains the same as in previous NiagaraAX revisions. The *value* portion of the “null status” input is used, which is often (if a source NumericWritable) a value of 0.0, or if a BooleanWritable, false. However, note that *other* null values may result, in cases where the source “Fallback” slot was previously set to a specific *non-default* value, and then set back to “null”.
 - **Specify Out Value**
With this selection, following a Save, the property listed below appears. Configure it with a desired out value for the conversion object, used whenever its input has a “null” status.
- **Out Value On Null**
This property shows only if the component is saved with “On Null In Value” as “Specify Out Value”. You can specify a pre-defined out value for the conversion object when its input has a null status. For example, a certain numeric value if a StatusNumericToFloat, StatusNumericToDouble, or StatusNumericToInt, or one of two Boolean values (false, true) if a StatusBooleanToBoolean, and so on. This removes any ambiguity about the conversion component’s output value, in case the component’s input sees a null status.

Simple value to status value

Here, the conversion object is often used in a AX-3.5 or earlier host to permit a link between the:

- Source “non status” value of a property in an extension or other component, to a
- Target “In” of a point or kitControl object (status value).

Conversion components in this category include:

- [BooleanToStatusBoolean](#)
- [DoubleToStatusEnum](#)
- [EnumToStatusEnum](#)
- [FloatToStatusEnum](#)
- [IntToStatusNumeric](#)
- [LongToStatusNumeric](#)
- [StringToStatusString](#)

For example, to link the “changeOfStateCount” property of a DiscreteTotalizer extension of a Boolean-Point to Math object input, you would link an IntToStatusNumeric between the two.

Status value to status value

Here, the conversion object is often used in a AX-3.5 or earlier host to permit a link between the:

- Source “status” value of a property in an extension or other component, to a
- Target “status” *different type* value of a point or kitControl object.

Conversion components in this category include:

- [StatusEnumToStatusBoolean](#)

- [StatusEnumToStatusNumeric](#)
- [StatusNumericToStatusEnum](#)
- [StatusNumericToStatusString](#)
- [StatusStringToStatusNumeric](#)

About Energy components

Energy components include a degree-days calculation object as well as various objects used for electrical demand limiting. Additional energy-saving functions are also represented as components.

Components in the Energy folder include:

- [DegreeDays](#)
- [ElectricalDemandLimit](#)
- [NightPurge](#)
- [OptimizedStartStop](#)
- [OutsideAirOptimization](#)
- [Psychrometric](#)
- [SetpointLoadShed](#)
- [SetpointOffset](#)
- [ShedControl](#)
- [SlidingWindowDemandCalc](#)

About HVAC components

HVAC components provide various control functions used in commercial HVAC applications. Included are the following components:

- [InterstartDelayControl](#)
- [InterstartDelayMaster](#)
- [LeadLagCycles](#)
- [LeadLagRuntime](#)
- [LoopPoint](#)
- [RaiseLower](#)
- [SequenceBinary](#)
- [SequenceLinear](#)
- [Tstat](#)

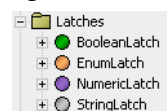
Note: *InterstartDelayControl* components are like **BooleanWritable** control points, only they provide three additional slots for use in an interstart delay control scheme. You use them with an *InterstartDelayMaster*, then link outputs of *InterstartDelayControl* objects, as needed, to control corresponding **BooleanWritable** points (typically proxy points) for the final interstart control.

About Latch components

Latch components allow you to capture an input value by using either the component's Clock property or by using the component's Latch action. In either case, "latching" means setting the value of the latch component "Out" property to whatever the value of the latch component "In" property is at the time that the "latch" occurs. The value of the latch component "In" property is ignored at all times other than the when a latch occurs.

Latch components are available in the kitControl palette "Latches" folder, as shown in Figure 1-7.

Figure 1-7 Latch components



Each latch component type has the same properties and function; they vary only to accommodate different point data types, as described in the following sections.

- [Types of Latch Components](#)
- [Types of Latch Component Properties](#)
- [About the Latch Action](#)
- [Latch Examples](#)

Types of Latch Components

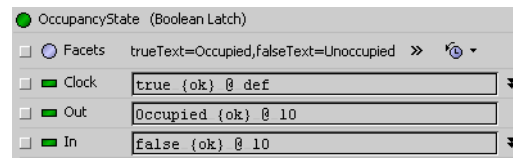
The following types of Latch Components are available:

- **BooleanLatch**
The BooleanLatch component provides a “latch” for a status boolean input and is located in the “Latches” folder of the kitControl palette. It has the same properties and actions as all the latch components, which are described in “Types of Latch Components” and “About the Latch Action”.
- **EnumLatch**
The EnumLatch component provides a “latch” for a status Enum input and is located in the “Latches” folder of the kitControl palette. It has the same properties and actions as all the latch components, which are described in “Types of Latch Components” and “About the Latch Action”.
- **NumericLatch**
The NumericLatch component provides a “latch” for a status Numeric input and is located in the “Latches” folder of the kitControl palette. It has the same properties and actions as all the latch components, which are described in “Types of Latch Components” and “About the Latch Action”.
- **StringLatch**
The StringLatch component provides a “latch” for a status String input and is located in the “Latches” folder of the kitControl palette. It has the same properties and actions as all the latch components, which are described in “Types of Latch Components” and “About the Latch Action”.

Types of Latch Component Properties

The following illustration shows the latch property sheet view of a BooleanLatch component. While the facets in this graphic are configured for a specific boolean usage, the properties are the same for all types of latch component data types.

Figure 1-8 Latch property sheet view



Latch components have the following properties that are common to all latch component data types:

- **Facets**
This property allows you to configure how the component value displays. For example, on a NumericLatch component you can set the units, precision, minimum and maximum value for the Out property value.
- **Clock**
This is a boolean status property that has either a True or False state for all latch components. This property “latches” the input property to the output property on the “rising edge”. This means that a single input property is captured and sent to the output property at the instant that the Clock status changes from a False to a True state and NOT when the property changes from a True to a False state.
- **Out**
This standard component property provides the actual latched value that is captured from the input property at “latch” time. Link to this property to display the value on a graphic or to process the value with another component.
- **In**
This is the standard component input property that you link into from a data source. For example, you can link into this property from a control point or a Schedule output.

About the Latch Action

Latch components have a Latch action that you can invoke by:

- Selecting the Latch command under the Actions popup menu.
In this case, you right-click on the latch component and select “Latch” from the popup menu.
- Linking a boolean value to the latch action slot on a Latch component.
In this case, any change of the boolean status value invokes a latch action (False to True or True to False). This is in contrast to using the Clock property, which latches only when the Clock property status changes from False to True.

The Latch action captures the input value any time that the Latch command is invoked.

Invoke a latch using the Latch command on the popup menu

Figure 1-9 Invoke a latch using the Latch command on the popup menu

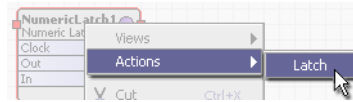
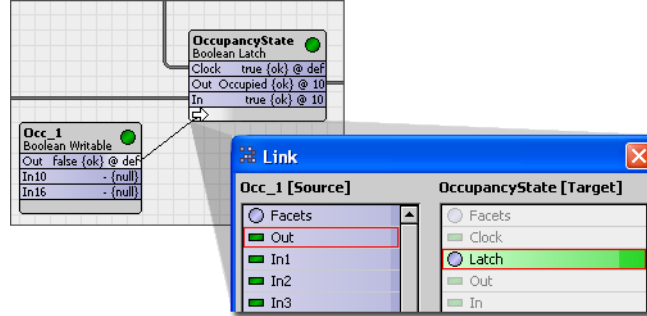


Figure 1-10 Invoke a latch by linking to the latch action



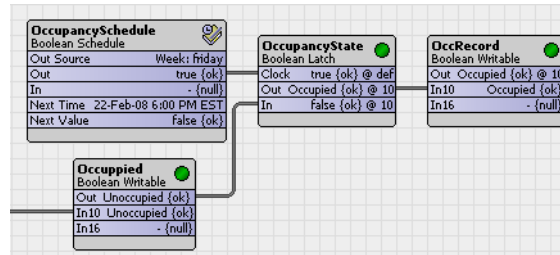
Latch Examples

The following examples are similar, in that they all use a Schedule component to invoke a latch. Other components may be used to invoke a latch, however, any latch that is invoked using the Clock property must include a method for setting the Clock property status back to False before the Clock is available for latching again. Example 3 illustrates the use of a latch component's latch action instead of using the Clock property.

Example 1: BooleanLatch Component

In this example, a building manager wants a record of days when the building has occupants that arrive before scheduled opening time of 6:00AM. This involves collecting the occupancy status from a building security system at the scheduled opening time once a day. The following illustration shows a Boolean-Latch component (OccupancyState) being used to capture the occupancy status value at a 6:00AM (scheduled occupancy time) every day.

Figure 1-11 Using the NumericLatch component Clock property



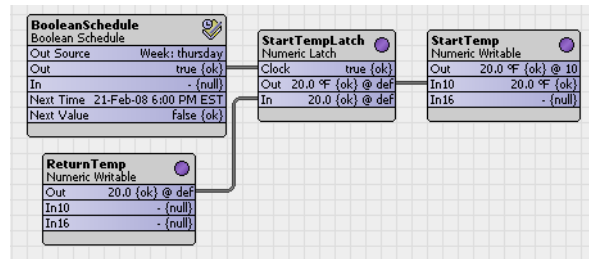
Note the following about this example:

- A Boolean Schedule (OccupancySchedule) Out value is linked to the BooleanLatch Clock property and a Boolean Out value from the "Occupied" boolean point is linked to the BooleanLatch In property.
- At 6:00AM, the OccupancySchedule Out value changes to True and sets the Clock property to True, causing the BooleanLatch component to "latch" the 6:00AM In value into the Out property.
- The BooleanLatch component Out property is linked to the BooleanWritable (OccRecord) component which can record the value using a history extension.
- At 6:00PM the Schedule Out value changes to False and sets the Clock property to False from True. No latch occurs here since the Clock property only latches on the "rising edge" (False to True). No change is made in the Out value until the Clock property status changes again from False to True - scheduled for 6:00AM the next day.

Example 2: NumericLatch Component

This example involves collecting the return air temperature value once a day according to a scheduled building opening time. The following illustration shows a NumericLatch component (StartTempLatch) being used to capture the return air temperature value at a specific time (scheduled occupancy time) every day.

Figure 1-12 Using the NumericLatch component Clock property



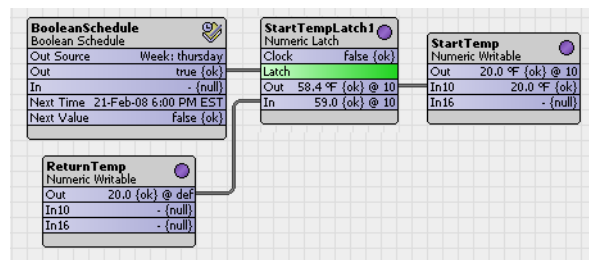
Note the following about this example:

- A Boolean Schedule Out value is linked to the NumericLatch Clock property and a Numeric Out value from the ReturnTemp numeric point is linked to the NumericLatch In property.
- At 6:00AM, the Schedule Out value changes to True and sets the Clock property to True, causing the NumericLatch component to "latch" the 6:00AM In value into the Out property.
- The NumericLatch component Out property is linked to the NumericWritable (StartTemp) component which can record the value using a history extension.
- At 6:00PM the Schedule Out value changes to False and sets the Clock property to False from True. No latch occurs at this time and no change is made in the Out value until the Clock property status changes again from False to True (scheduled for 6:00AM).

Example 3: NumericLatch Component Latch Action

This example involves collecting the return air temperature value twice a day according to scheduled building opening and closing times. The following illustration shows a NumericLatch component (StartTempLatch1) being used to capture the return air temperature value at two specific times (scheduled occupancy time and scheduled un-occupancy) Monday through Friday. This example is similar to Example 1 except that the Schedule Out value is linked to the NumericLatch "Latch" action.

Figure 1-13 Using the NumericLatch component latch action



Note the following about this example:

- A Boolean Schedule Out value is linked to the NumericLatch Clock property and a Numeric Out value from the ReturnTemp numeric point is linked to the NumericLatch In property.
- At 6:00AM and at 6:00PM, the Schedule Out value changes to True or to False, respectively. In each case this status change invokes the latch action, causing the NumericLatch component to "latch" the In value into the Out property.
- The NumericLatch component Out property is linked to the NumericWritable (StartTemp) component which can record the values using a history extension.
- A Trigger Schedule may be simpler to use in this example. Using a Trigger Schedule, you can simply define the two time in the day that you want trigger the latch. The Boolean Schedule requires start and stop times for each event.

About Logic components

All 10 of the logic components process input values and provide a StatusBoolean output. Logic object types vary by *input* types.

- Four types have *StatusBoolean* inputs:
 - [And](#)
 - [Or](#)
 - [Xor](#)
 - [Not](#)
- Six types have *StatusNumeric* inputs:
 - [Equal](#)
 - [GreaterThan](#)
 - [GreaterThanEqual](#)
 - [LessThan](#)
 - [LessThanEqual](#)
 - [NotEqual](#)

Starting in AX-3.6, an “ExprLogic” component was added to the Logic folder in the kitControl palette, however, it is unlike all others—and technically not a “logic component”. Instead it is a demonstration example of an Expr component ([BqLExprComponent](#)) that provides a 4-input logic AND function.

If needed, you can add alarm and history extensions to any logic component, in addition to the control extension **DiscreteTotalizerExt**. If configured with a **DiscreteTotalizerExt**, you can also add special-purpose alarm extensions [ChangeOfStateCountAlarmExt](#) and or [ElapsedActiveTimeAlarmExt](#).

Note: *As with math components, you can individually configure logic components to “propagate” status flags received on linked inputs (by default, status propagation does not occur). For more details, see “How status flags are set” in the User Guide.*

About Math components

Math components process one or more *StatusNumeric* input values and provide a *StatusNumeric* output. Each component type provides a specific math function like Add, Average, Divide, Minimum, Maximum, Reset, AbsValue, and so on.

Math object types vary by number of inputs used, in addition to math operation.

- The following Math types perform an operation on from *one to four* inputs:
 - [Add](#)
 - [Average](#)
 - [Maximum](#)
 - [Minimum](#)
 - [Multiply](#)
- The following Math types perform an operation using *two* inputs:
 - [Divide](#)
 - [Modulus](#)
 - [Power](#)
 - [Subtract](#)
- The following Math types perform an operation on a *single* input:
 - [AbsValue](#)
 - [ArcCosine](#)
 - [ArcSine](#)
 - [ArcTangent](#)
 - [Cosine](#)
 - [Exponential](#)
 - [Factorial](#)
 - [LogBase10](#)
 - [LogNatural](#)
 - [Negative](#)
 - [Reset](#) (using 4 values of high and low limits, both input and output, also linkable as inputs)
 - [Sine](#)
 - [SquareRoot](#)
 - [Tangent](#)

Starting in AX-3.6, an “ExprMath” component was added to the Math folder in the kitControl palette, however, it is unlike all others—and technically not a “math component”. Instead it is a demonstration example of an Expr component([BqLExprComponent](#)) that provides a 4-input Add math function.

If needed, you can add alarm and history extensions to any math component, in addition to the control extension **NumericTotalizerExt**.

Note: *As with logic components, you can individually configure math components to “propagate” status flags received on linked inputs (by default, status propagation does not occur). For more details, see “How status flags are set” in the User Guide.*

About Select components

A select object allows one of multiple inputs to be selected (passed to the output) upon selection by the value at its “Select” (StatusEnum) input. From 3 to 10 inputs can be specified.

Note that all select objects require an enumerated input to perform the selection—the four select object types differ only by the type of input data selected and passed to the “Out” slot.

The Selects folder of the kitControl palette includes the following components:

- [BooleanSelect](#)
- [EnumSelect](#)
- [NumericSelect](#)
- [StringSelect](#)

About String components

String components process one or more status string inputs, and produce some form of output. String object types vary by outputs.

- Three String types perform a string function with a StatusString output:
 - [StringConcat](#)
 - [StringSubstring](#)
 - [StringTrim](#)
- Two String types provide a logic function with a StatusBoolean output:
 - [StringIndexOf](#)
 - [StringTest](#)
- Finally, [StringLen](#) simply provides a StatusNumeric output equal to the number of non-null characters in the input string.

About Timer components

Components in the kitControl “Timer” folder include 3 timer types, and 2 components for working with Baja absolute time (AbsTime). Two of the timers are boolean-types, as follows:

- [BooleanDelay](#) provides a configurable delay for an input transition passed to the output.
 - [OneShot](#) provides an adjustable “one shot” timed output upon a false-to-true input transition.
- The third timer is a [NumericDelay](#), a numeric-type. It provides an input-to-output delay that is effectively a “stepped ramp,” based upon configured properties.

For working with AbsTime, the Timer folder provides the following two components:

- [CurrentTime](#) — for display or reference to the current system time.
- [TimeDifference](#) — subtracts one AbsTime value from another, with output result in milliseconds.

About Util components

Components in the Util folder of kitControl range from a Counter object with boolean input and a numeric output (counts active transitions) to a boolean-controlled NumericSwitch that selects one of two numeric values. A StatusDemux provides a method to logic test (true/false) a linked object for the presence of one or more status flags.

Util components also include “generator” components useful for simulation/logic testing, such as the SineWave and Ramp (each with a numeric output) and MultiVibrator (boolean output).

Note: *Starting in AX-3.6, an “Expr” (BqlExprComponent) component was added to the Util folder, available to create custom math and logic functions based upon one or more BQL expression statements. For complete information, refer to the Engineering Notes II document BQL Expression component.*

The following components are included in the Util folder of the kitControl palette:

- [BooleanSwitch](#)
- [Counter](#)
- [DigitalInputDemux](#)

- [EnumSwitch](#)
- [Expr](#) ([BqlExprComponent](#), AX-3.6 and later only)
- [MinMaxAvg](#)
- [MultiVibrator](#)
- [NumericBitAnd](#)
- [NumericBitOr](#)
- [NumericBitXor](#)
- [NumericSwitch](#)
- [NumericToBitsDemux](#)
- [Ramp](#)
- [Random](#)
- [SineWave](#)
- [StatusDemux](#)

CHAPTER 2

kitControl Component Guides

This Component Guides section provides summary information on all kitControl components. Some component topics include detailed descriptions with property information and examples. See the next section “[Alphabetical list of kitControl components](#)” for a complete list of all kitControl components. For details on the folder-based organization in the kitControl palette, see “[Types of kitControl components](#)” on page 1-2.

Alphabetical list of kitControl components

For an overview of kitControl see “[About kitControl](#)” on page 1-1. For details on the folder-based organization of the kitControl palette, see “[Types of kitControl components](#)” on page 1-2.


The following list includes all kitControl components in alphabetical order, by name:

- [AbsValue](#) on page 3 (folder [Math](#))
- [Add](#) on page 3 (folder [Math](#))
- [AlarmCountToRelay](#) on page 3 (folder [Alarm](#))
- [And](#) on page 4 (folder [Logic](#))
- [ArcCosine](#) on page 5 (folder [Math](#))
- [ArcSine](#) on page 5 (folder [Math](#))
- [ArcTangent](#) on page 5 (folder [Math](#))
- [Average](#) on page 5 (folder [Math](#))
- [BooleanDelay](#) on page 5 (folder [Timer](#))
- [BooleanConst](#) on page 6 (folder [Constants](#))
- [BooleanLatch](#) on page 6 (folder [Latches](#))
- [BooleanSelect](#) on page 6 (folder [Selects](#))
- [BooleanSwitch](#) on page 6 (folder [Util](#))
- [BooleanToStatusBoolean](#) on page 6 (folder [Conversion](#))
- [ChangeOfStateCountAlarmExt](#) on page 7 (folder [Alarm](#))
- [Cosine](#) on page 7 (folder [Math](#))
- [Counter](#) on page 7 (folder [Util](#))
- [CurrentTime](#) on page 8 (folder [Timer](#))
- [DegreeDays](#) on page 9 (folder [HVAC](#))
- [DigitalInputDemux](#) on page 10 (folder [Util](#))
- [Divide](#) on page 12 (folder [Math](#))
- [DoubleToStatusEnum](#) on page 12 (folder [Conversion](#))
- [ElapsedActiveTimeAlarmExt](#) on page 12 (folder [Alarm](#))
- [ElectricalDemandLimit](#) on page 12 (folder [Energy](#))
- [EnumConst](#) on page 18 (folder [Constants](#))
- [EnumLatch](#) on page 18 (folder [Latches](#))
- [EnumSelect](#) on page 18 (folder [Selects](#))
- [EnumToStatusEnum](#) on page 18 (folder [Conversion](#))
- [EnumSwitch](#) on page 18 (folder [Util](#))
- [Expr](#) ([BqlExprComponent](#) on page 7, folder [Util](#)) also “[ExprLogic](#)” and “[ExprMath](#)” components
- [Equal](#) on page 18 (folder [Logic](#))
- [Exponential](#) on page 19 (folder [Math](#))
- [Factorial](#) on page 19 (folder [Math](#))
- [FloatToStatusEnum](#) on page 19 (folder [Conversion](#))
- [GreaterThan](#) on page 19 (folder [Logic](#))
- [GreaterThanEqual](#) on page 19 (folder [Logic](#))

- [IntToStatusNumeric](#) on page 19 (folder [Conversion](#))
- [InterstartDelayControl](#) on page 19 (folder [HVAC](#))
- [InterstartDelayMaster](#) on page 19 (folder [HVAC](#))
- [LeadLagCycles](#) on page 19 (folder [HVAC](#))
- [LeadLagRuntime](#) on page 21 (folder [HVAC](#))
- [LessThan](#) on page 22 (folder [Logic](#))
- [LessThanEqual](#) on page 22 (folder [Logic](#))
- [LogBase10](#) on page 22 (folder [Math](#))
- [LogNatural](#) on page 22 (folder [Math](#))
- [LongToStatusNumeric](#) on page 22 (folder [Conversion](#))
- [LoopAlarmExt](#) on page 22 (folder [Alarm](#))
- [LoopPoint](#) on page 23 (folder [HVAC](#))
- [Maximum](#) on page 27 (folder [Math](#))
- [Minimum](#) on page 27 (folder [Math](#))
- [MinMaxAvg](#) on page 27 (folder [Util](#))
- [Modulus](#) on page 28 (folder [Math](#))
- [Multiply](#) on page 28 (folder [Math](#))
- [MultiVibrator](#) on page 28 (folder [Util](#))
- [Negative](#) on page 28 (folder [Math](#))
- [NightPurge](#) on page 28 (folder [HVAC](#))
- [Not](#) on page 30 (folder [Logic](#))
- [NotEqual](#) on page 31 (folder [Logic](#))
- [NumericBitAnd](#) on page 31 (folder [Util](#))
- [NumericBitOr](#) on page 32 (folder [Util](#))
- [NumericBitXor](#) on page 32 (folder [Util](#))
- [NumericConst](#) on page 32 (folder [Constants](#))
- [NumericDelay](#) on page 33 (folder [Timer](#))
- [NumericLatch](#) on page 33 (folder [Latches](#))
- [NumericSelect](#) on page 33 (folder [Selects](#))
- [NumericSwitch](#) on page 34 (folder [Util](#))
- [NumericToBitsDemux](#) on page 34 (folder)
- [NumericUnitConverter](#) on page 35 (folder [Conversion](#))
- [OneShot](#) on page 35 (folder [Timer](#))
- [OptimizedStartStop](#) on page 36 (folder [HVAC](#))
- [Or](#) on page 40 (folder [Logic](#))
- [OutsideAirOptimization](#) on page 41 (folder [Energy](#))
- [Power](#) on page 42 (folder [Math](#))
- [Psychrometric](#) on page 42 (folder [HVAC](#))
- [RaiseLower](#) on page 44 (folder [HVAC](#))
- [Ramp](#) on page 43 (folder [Util](#))
- [Random](#) on page 43 (folder [Util](#))
- [Reset](#) on page 44 (folder [Math](#))
- [SequenceBinary](#) on page 46 (folder [HVAC](#))
- [SequenceLinear](#) on page 47 (folder [HVAC](#))
- [SetpointLoadShed](#) on page 49 (folder [Energy](#))
- [SetpointOffset](#) on page 50 (folder [Energy](#))
- [ShedControl](#) on page 50 (folder [Energy](#))
- [Sine](#) on page 51 (folder [Math](#))
- [SineWave](#) on page 51 (folder [Util](#))
- [SlidingWindowDemandCalc](#) on page 51 (folder [HVAC](#))
- [SquareRoot](#) on page 53 (folder [Math](#))
- [StatusBooleanToBoolean](#) on page 53 (folder [Conversion](#))
- [StatusDemux](#) on page 53 (folder [Util](#))
- [StatusEnumToEnum](#) on page 53 (folder [Conversion](#))
- [StatusEnumToInt](#) on page 53 (folder [Conversion](#))
- [StatusEnumToStatusBoolean](#) on page 53 (folder [Conversion](#))
- [StatusEnumToStatusNumeric](#) on page 53 (folder [Conversion](#))
- [StatusNumericToDouble](#) on page 53 (folder [Conversion](#))
- [StatusNumericToFloat](#) on page 54 (folder [Conversion](#))
- [StatusNumericToInt](#) on page 54 (folder [Conversion](#))
- [StatusNumericToStatusEnum](#) on page 54 (folder [Conversion](#))
- [StatusNumericToStatusString](#) on page 54 (folder [Conversion](#))


- [StatusStringToStatusNumeric](#) on page 54 (folder [Conversion](#))
- [StringConcat](#) on page 54 (folder [String](#))
- [StringConst](#) on page 54 (folder [String](#))
- [StringIndexOf](#) on page 54 (folder [String](#))
- [StringLatch](#) on page 55 (folder [Latches](#))
- [StringLen](#) on page 55 (folder [String](#))
- [StringSelect](#) on page 55 (folder [Selects](#))
- [StringSubstring](#) on page 55 (folder [String](#))
- [StringTest](#) on page 55 (folder [String](#))
- [StringToStatusString](#) on page 55 (folder [Conversion](#))
- [StringTrim](#) on page 55 (folder [String](#))
- [Subtract](#) on page 55 (folder [Math](#))
- [Tangent](#) on page 56 (folder [Math](#))
- [TimeDifference](#) on page 56 (folder [Timer](#))
- [Tstat](#) on page 56 (folder [HVAC](#))
- [Xor](#) on page 56 (folder [Logic](#))

kitControl-AbsValue

 AbsValue performs the operation $out = abs(inA)$ (absolute value of inA). The AbsValue is available in the [Math](#) folder of the kitControl palette.


See also [Alphabetical list of kitControl components](#)

kitControl-Add

 Add performs the operation $out = (inA + inB + inC + inD)$. The Add is available in the [Math](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-AlarmCountToRelay

 The AlarmCountToRelay component allows you to link from an Alarm Class component to monitor alarm count and send an associated boolean output to a relay whenever there is an increase in the alarm count. The alarm count type that you choose to monitor is optional and includes the alarm states or statuses, as described under the “Alarm Count Type”, below. This component may be used in security applications or situations where you want to connect the occurrence of a new alarm with an event such as a light, horn, or other signal which requires a relay connection. When an alarm count increases, the component’s Relay Out property is set to True from False and maintained at that status for the amount of time that is specified by the Timer property value.

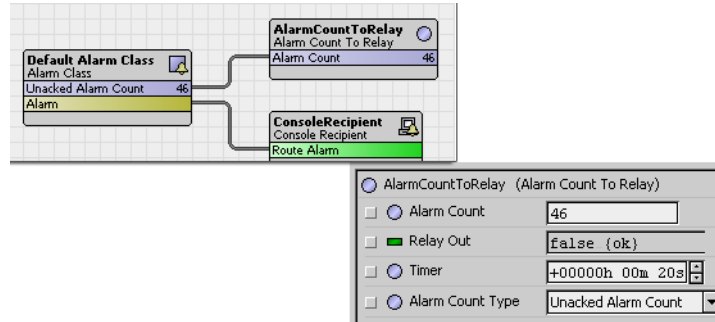
AlarmCountToRelay component properties are described below:

- **Alarm Count**
When the AlarmCountToRelay component is linked to an alarm class component, this property displays the current alarm count for that alarm class component. The numeric value of this property dynamically displays the number of alarms of the type specified in the Alarm Type Count property.
- **Relay Out**
This property provides the boolean output value for linking into a relay control component. The default value is false and the active value is true. When this property transitions to true, it stays in the true state for a time equal to the value of the Timer property.
- **Timer**
This property allows you to set a value that specifies how long the Relay Out value is to be held in the active (true) state.
- **Alarm Count Type**
This property allows you to choose one of the following alarm types to monitor:
 - **Any Count**
When you select this option, you can link from any alarm type to the Alarm Count property. In this case, an increase in any type alarm count from this alarm class invokes a status change at the Relay Out property for a time equal to the value of the Timer property.
 - **Unacked Alarm Count**
When you select this option, you can link from the Unacked Alarm Count property on an Alarm Class component to the Alarm Count property. Any increase in the Unacked Alarm Count invokes a status change at the Relay Out property for a time equal to the value of the Timer property.

- Open Alarm Count**
 When you select this option, you can link from the Open Alarm Count property on an Alarm Class component to the Alarm Count property. Any increase in the Open Alarm Count invokes a status change at the Relay Out property for a time equal to the value of the Timer property.
- In Alarm Count**
 When you select this option, you can link from the In Alarm Count property on an Alarm Class component to the Alarm Count property. Any increase in the In Alarm Count invokes a status change at the Relay Out property for a time equal to the value of the Timer property.

Figure 2-1 shows an example of an AlarmCountToRelay component that is linked to the Unacked Alarm Count property. In this example, the Unacked Alarm Count is 46. Any increase in the number of Unacked Alarms causes the Relay Out property to change status from false to true for a time equal to the value of the Timer property.

Figure 2-1 Example AlarmCountToRelay usage



See also [Alphabetical list of kitControl components](#)

kitControl-And

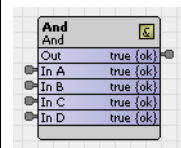
And performs a logical AND on all inputs and writes the result to the out property. It is available in the Logic folder of the kitControl palette. Table 2-1 shows the And object truth table when using two inputs. Table 2-2 shows the And object truth table if using all four inputs. NAND gate logic is accomplished by linking to a Not object.

See also [Alphabetical list of kitControl components](#)

Table 2-1 And object truth table (2 inputs)

In A	In B	Out
false	false	false
false	true	false
true	false	false
true	true	true

Table 2-2 And object truth table (4 inputs)



In A	In B	In C	In D	Out
false	false	false	false	false
false	false	false	true	false
false	false	true	false	false
false	false	true	true	false
false	true	false	false	false
false	true	false	true	false
false	true	true	false	false
false	true	true	true	false
true	false	false	false	false
true	false	false	true	false
true	false	true	false	false
true	false	true	true	false
true	true	false	false	false
true	true	false	true	false
true	true	true	false	false
true	true	true	true	true

kitControl-ArcCosine

☞ ArcCosine performs the operation $out = \text{acos}(inA)$. The ArcCosine is available in the **Math** folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-ArcSine

☞ ArcSine performs the operation $out = \text{asin}(inA)$. The ArcSine is available in the **Math** folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-ArcTangent

☞ ArcTangent performs the operation $out = \text{atan}(inA)$. The ArcTangent is available in the **Math** folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-Average

☞ Average determines the average value of valid inputs and writes that value to out. $out = (inA + inB + inC + inD) / 4$. The Average is available in the **Math** folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-BooleanDelay

☞ The BooleanDelay component provides a way to delay the status change of a boolean status “out” property value by configuring an associated “Delay” property. Delay properties are provided for on (true) and off (false) statuses and are labeled “On Delay” and “Off Delay”, respectively. The delay applies to any transition (status change from on to off or off to on) at the component’s status boolean input. Both delay times are configurable in terms of hours, minutes and seconds.

Types of BooleanDelay component properties include the following:

- **Facets**
Use this property to set the trueText and falseText for the Out property values. For example, you might want to set the facet trueText to display “ON” and the facet falseText to display “OFF”.

- **In**
Typically, you set this property by linking a boolean out value into it. You can manually configure the default state to be `true`, `false`, or `null`, so that when no value is linked into this property, the default value is used. This property value is passed to the Out and Out Not properties (after any On Delay or Off Delay) whenever there is a change in this property's status.
- **On Delay**
This property allows you to set the amount of time (in hours, minutes, and seconds) that you want to expire before sending a `true` (On) value to the Out property. Time begins to expire at the moment that a change in the In property occurs (a transition from `false` or `null` to `true`).
- **Off Delay**
This property allows you to set the amount of time (in hours, minutes, and seconds) that you want to expire before sending a `false` (Off) value to the Out property. The time begins at the moment that a change in the In property occurs (a transition from `True` to `False` or `False` to `true`).
- **On Delay Active**
This read-only property shows whether or not the On Delay time is actively counting down to expiration. This (normally `false`) value changes to `true` anytime that a transition from `false` to `true` occurs at the In property and stays at `true` until any Off Delay time is expired. If the On Delay value is set to "0", then this value does not change to `true`.
- **Off Delay Active**
This read-only property shows whether or not the Off Delay time is actively counting down to expiration. This (normally `false`) value changes to `true` anytime that a transition from `true` to `false` occurs at the In property and stays at `true` until any Off Delay time is expired. If the On Delay value is set to "0", then this value does not change to `true`.
- **Out**
This property has `true`, `false`, or `null` options available. These values are set at the end of any On Delay or Off Delay to reflect the In property value.
- **Out Not**
This property has `true`, `false`, or `null` options available. These values are set at the end of any On Delay or Off Delay to reflect the inverse In value. For example, when the In value is `true`, the Out Not value is set to `false` (after expiration of any "delay" value).

The BooleanDelay component is located in the [Timer](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-BooleanConst

- Provides constant status boolean value, with actions to set. It is available in the [Constants](#) folder of the kitControl palette. See "About Constant components" on page 1-4.

See also [Alphabetical list of kitControl components](#)

kitControl-BooleanLatch

- BooleanLatch provides a latch for a status boolean input, and is found in the [Latches](#) folder of the kitControl palette. See "About Latch components" on page 1-8.

See also [Alphabetical list of kitControl components](#)

kitControl-BooleanSelect

- ▣ BooleanSelect is a boolean select, and is found in the [Selects](#) folder of the kitControl palette. See "About Select components" on page 1-13 for an overview.

See also [Alphabetical list of kitControl components](#)

kitControl-BooleanSwitch

- ▣ (AX-3.5 and later) BooleanSwitch selects one of two StatusBoolean inputs based upon the boolean value at the StatusBoolean input "In Switch." BooleanSwitch is available in the [Util](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-BooleanToStatusBoolean

- ▣ BooleanToStatusBoolean converts a Boolean value to StatusBoolean. See "Simple value to status value" on page 1-7. It is available in the [Conversion](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-BqlExprComponent


■ (AX-3.6 and later) BqlExprComponent (**Expr**) provides the means to create custom math and logic operations based upon manually-added slots and one or more BQL expression statements. Slots can be various baja types such as primitives Double, Float, Integer, Boolean, or String, or status types such as StatusBoolean, StatusNumeric, and so on. Slots are used either as inputs, or as one or more outputs. BQL expressions are entered in the component's "Expr" property.

A "blank" **Expr** component is available in the **Util** folder of the kitControl palette. Additionally, *example* Expr components are in the **Logic** folder (**ExprLogic**) and **Math** folder (**ExprMath**), demonstrating a 4-input logic AND gate and 4-input math ADD component, respectively.

For complete information, refer to the Engineering Notes II document *BQL Expression component*.

See also [Alphabetical list of kitControl components](#)

kitControl-ChangeOfStateCountAlarmExt


 ChangeOfStateCountAlarmExt is a special-purpose alarm extension, especially for use as child of a BooleanPoint or BooleanWritable that has one or more **DiscreteTotalizerExt** extensions. It provides alarming on COS (change of state) counts, using offNormal property errorLimit.

Note: *In the parent Boolean point, order the ChangeOfStateCountAlarmExt slot below the DiscreteTotalizerExt slot that it references. In the ChangeOfStateCountAlarmExt's Offnormal container, use the Discrete Totalizer Select property to reference the DiscreteTotalizerExt.*

ChangeOfStateCountAlarmExt is available in the **Alarm** folder of the kitControl palette, along with a **ElapsedActiveTimeAlarmExt** (for runtime-based alarms). You can use *both* extensions to reference the same DiscreteTotalizerExt.

See also [Alphabetical list of kitControl components](#)

kitControl-Cosine

 Cosine performs the operation $out = \cos(inA)$. The Cosine is available in the **Math** folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-Counter

■ The Counter component will count boolean inactive to active transitions. It supports counting up, counting down, presetting, and clearing. The Counter is available in the **Util** folder of the kitControl palette. The following sections provide more details:

The Counter component includes the following properties:

- **Facets**
This is used to set the units and number precision of the Out property.
- **Propagate Flags**
Specifies which status flags will propagate from the Count Up, Count Down, Preset In, and Clear In properties to the Out status flags.
- **Out**
This is the current count output.
- **Count Up**
This is a StatusBoolean input. When this input makes inactive to active transition the value of the Out property increments by the Count Increment value.
- **Count Down**
This is a StatusBoolean input. When this input makes inactive to active transition the value of the Out property will be decremented by the Count Increment.
- **Preset In**
This is a StatusBoolean input. When this input makes inactive to active transition the value of the Out property will be decremented by the Count Increment.
- **Clear In**
This is a StatusBoolean input. When this input makes inactive to active transition the value of the Out property will be set to 0.0.
- **Preset Value**
This defines the value that will be set in the Out property when the Preset In changes to active, or when the Preset action is invoked.
- **Count Increment**
This is the value that the Out property will change for a single count up or count down active transition.

- **Preset**
This action sets the Out property value to the Preset Value.
- **Clear**
This action sets the Out property value to 0.

Figure 2-2 shows an example of a Counter component property sheet.

Figure 2-2 Counter component example property sheet

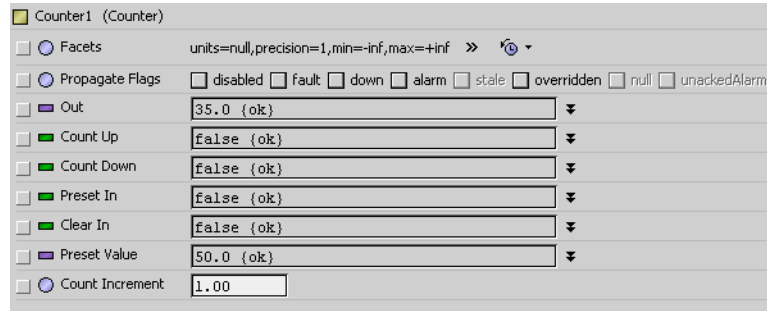


Figure 2-3 shows an example application that ramps between the RampMaxValue and the RampMinValue. The period of the MultiVibrator object sets how fast the ramp counts. The Clock input of the BooleanLatch object config flags is set to allow fan-in.

Figure 2-3 Counter component example ramp up and down

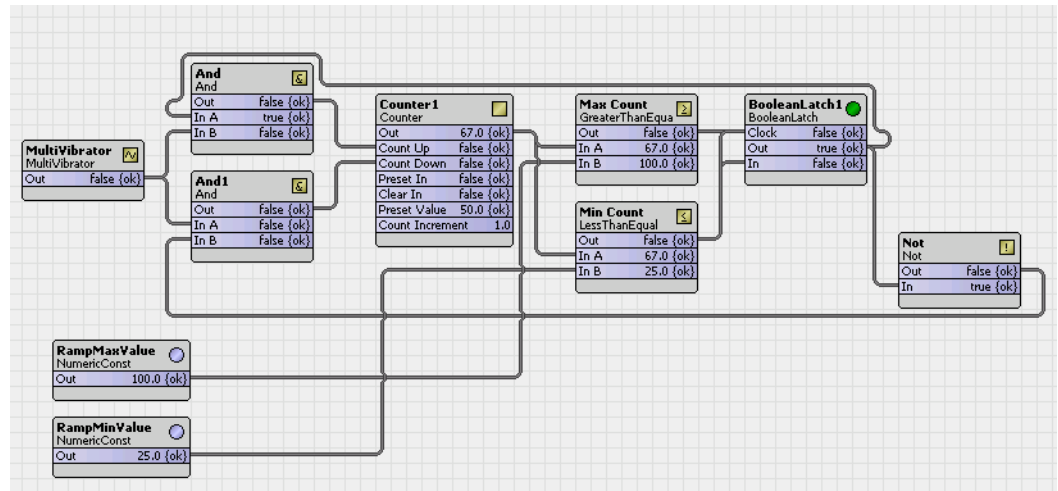
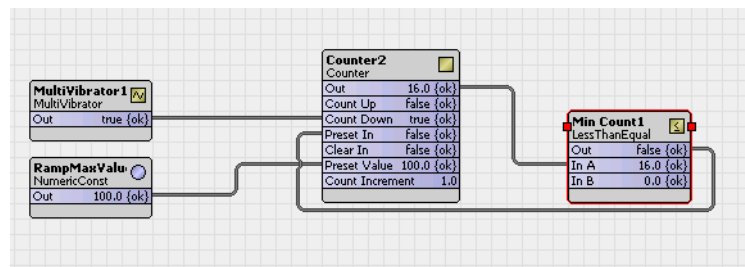


Figure 2-4 shows an example count from the RampMaxValue down to 0 and then reset back to RampMaxValue and repeat.

Figure 2-4 Counter component example ramp down



See also [Alphabetical list of kitControl components](#)

kitControl-CurrentTime

CurrentTime provides the current system time formatted in Baja absolute time (AbsTime). Use it directly for graphics display, or with a TimeDifference object for other applications. CurrentTime is in the Timer folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-DegreeDays

- DegreeDays provides degree day calculations, based upon temperature received at the Temp In slot and values of various other properties.

Note: *Definition of Degree Days: Degree Days is a unit of measure that may be expressed as either Heating Degree Days (HDD) or Cooling Degree Days (CDD). You calculate Degree Days by taking the difference between the average temperature during a given time period (month, season, year) and a reference point, usually 65 degrees Fahrenheit.*

Both cooling and heating degree day values are available, including totalized values. A Reset Totals action is available to clear (zero) totalized values.

DegreeDays is available in the [Energy](#) folder of the kitControl palette. The following sections provide more details:

The DegreeDays component includes the following properties and one action:

- **Facets**
This is used to set the units and number precision of the Temp In, Min Temp, Max Temp, and Mean Temp properties.
- **Base Temperature**
Specifies the base temperature used in the degree-day calculation.
- **Temp In**
This is the input for the outside air temperature used in the degree-day calculation. Note: If this input is not valid the no calculations will be done.
- **Min Temp**
The minimum temperature recorded for the current day. Tested and set on each calculation.
- **Max Temp**
The maximum temperature recorded for the current day. Tested and set on each calculation.
- **Mean Temp**
The mean temperature recorded for the previous day. Calculated when the day changes. $\text{Mean Temp} = (\text{Max Temp} + \text{Min Temp}) / 2.0$
- **Clg Deg Days**
This is the cooling degree-day calculated for the previous day. Calculated when the day changes.
 $\text{If } (\text{Mean Temp} - \text{Base Temperature}) > 0$
 $\text{Clg Deg Days} = \text{Mean Temp} - \text{Base Temperature}$
else
 $\text{Clg Deg Days} = 0.0$
- **Clg Deg Days Total**
This is the totalized cooling degree-days since last Reset Totals action was invoked. Calculated when Clg Deg Days changes.
- **Htg Deg Days**
This is the heating degree-day calculated for the previous day. Calculated when the day changes.
 $\text{If } (\text{Mean Temp} - \text{Base Temperature}) < 0$
 $\text{Htg Deg Days} = \text{Base Temperature} - \text{Mean Temp}$
else
 $\text{Htg Deg Days} = 0.0$
- **Htg Deg Days Total**
This is the totalized heating degree-days since last Reset Totals action was invoked. Calculated when Htg Deg Days changes.
- **Reset Totals**
This action will clear the Clg Deg Days Total and Htg Deg Days Total properties to zero when invoked.

[Figure 2-5](#) shows an example DegreeDays property sheet.

Figure 2-5 DegreeDays example property sheet

DegreeDays (Degree Days)	
<input type="checkbox"/> Facets	units=°F,precision=1,min=-inf,max=+inf >>
<input type="checkbox"/> Base Temperature	65.0 °F
<input type="checkbox"/> Temp In	60.0 °F {ok}
<input type="checkbox"/> Min Temp	0.0 °F {ok}
<input type="checkbox"/> Max Temp	61.0 °F {ok}
<input type="checkbox"/> Mean Temp	0.0 °F {ok}
<input type="checkbox"/> Clg Deg Days	0.0 {ok}
<input type="checkbox"/> Clg Deg Days Total	0.0 {ok}
<input type="checkbox"/> Htg Deg Days	0.0 {ok}
<input type="checkbox"/> Htg Deg Days Total	0.0 {ok}

See also [Alphabetical list of kitControl components](#)

kitControl-DigitalInputDemux

- The Digital Input Demux (Demultiplexer) object provides four status boolean outputs from one StatusNumeric input. This component is available in the [Util](#) folder of the kitControl palette.

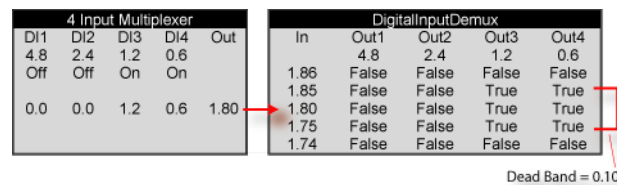
A typical application for this demultiplexer object is in association with a multiplexer module to expand the IO capacity of a system. The multiplexer creates a single analog voltage output to represent the state of up to four digital inputs. The analog voltage is then demultiplexed into four status boolean outputs by the DigitalInputDemux object.

The DigitalInputDemux component has the following properties:

- **Propagate Flags**
By default, this object maintains independent status flags from input-linked points. However, as a configuration option you can specify “out” status to propagate from the input status. The propagate flags property specifies which status flags propagate from the “In” property to the “Out” status flags. The PropagateFlags property allows you to select any combination of the following status types for propagation:
 - disabled
 - fault
 - down
 - alarm
 - overridden
- **In**
This is a StatusNumeric value for this object and is the input analog value from the multiplexer which represents the state of the four digital inputs. This input must be valid for the object to function.
- **Out1**
This is a status boolean value which is set to `true` if the object determines that the “In” property contains a value equivalent to the value set as the Out1 Value property.
- **Out2**
This is a status boolean value which is set to `true` if the object determines that the “In” property contains a value equivalent to the value set as the Out2 Value property.
- **Out3**
This is a status boolean value which is set to `true` if the object determines that the “In” property contains a value equivalent to the value set as the Out3 Value property.
- **Out4**
This is a status boolean value which is set to `true` if the object determines that the “In” property contains a value equivalent to the value set as the Out4 Value property.
- **Out1 Value**
This should be set to a value which corresponds with the equivalent setting in the multiplexer device to represent the status of digital input 1. The default is 4.80.
- **Out2 Value**
This should be set to a value which corresponds with the equivalent setting in the multiplexer device to represent the status of digital input 2. The default is 2.40.
- **Out3 Value**
This should be set to a value which corresponds with the equivalent setting in the multiplexer device to represent the status of digital input 3. The default is 1.20.

- **Out4 Value**
 This should be set to a value which corresponds with the equivalent setting in the multiplexer device to represent the status of digital input 4. The default is 0.60.
- **Dead Band**
 The Dead Band allows you to set a tolerance value to prevent ‘chatter’ of the outputs due to fluctuations of the input value. The Dead Band function operates purely on the input “In” value.
 In the example shown in Figure 2-6, the DigitalInputDemux object is fed from a multiplexer device on site which is connected to four digital inputs (DI). DI1 and DI2 are in an open (off) condition and DI3 and DI4 are closed (on). The combined voltage weighting of DI3 and DI4 is 1.8v which is transmitted to the DigitalInputDemux object via an NDIO universal input. The DigitalInputDemux object then faithfully demultiplexes this signal so that “Out3” and “Out4” are both set to ‘True’.
 In practice, a voltage drop occurs on the received signal and the Dead Band property allows you to engineer in some protection for this fluctuation to prevent it adversely upsetting your control strategy. The Dead Band property in this example is set to 0.10 which is applied to the “In” value. The Dead Band function operates equally in both positive and negative sense on the “In” value. In this example therefore, all values from 1.75 through to 1.85 are valid.
 The default value of the Dead Band property is 0.10.

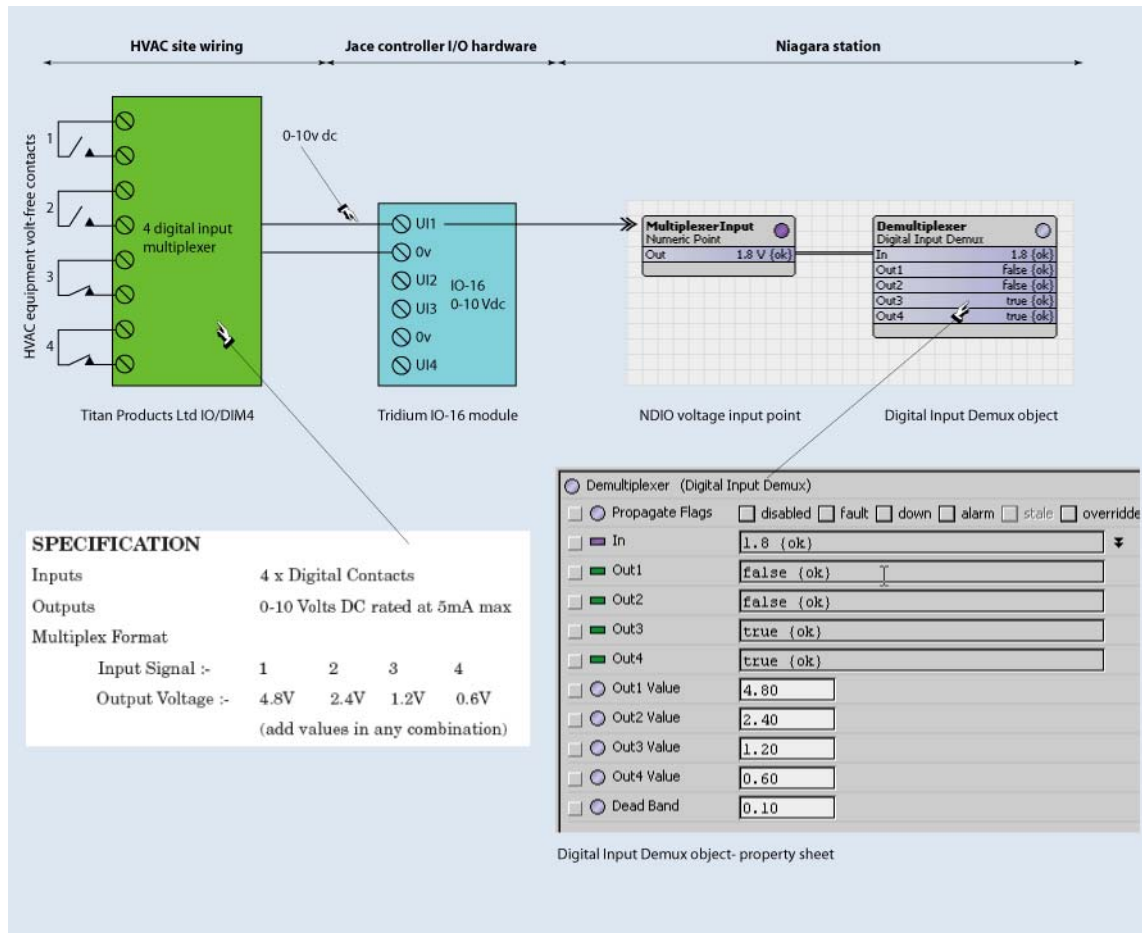
Figure 2-6 Operation of the Dead Band property



Note: The DigitalInputDemux object has no actions.

Examples In the example shown in Figure 2-7, four volt-free contacts, such as door status, are connected to a 4 channel digital input multiplexer. The IO/DIM4 multiplexer device shown is manufactured by Titan Products Ltd., although similar suitable devices are available from other equipment suppliers. The IO/DIM4 multiplexer is microprocessor based and is designed to convert 4 separate digital input signals into a single analogue voltage output. Each combination of digital input signal is converted to an output voltage level which is then connected to one universal input of an NDIO IO-16 module. The NDIO universal input is configured as a “Voltage Input Point” and its output feeds the DigitalInputDemux object. Finally, the DigitalInputDemux object provides 4 status boolean outputs which represent the status of the original 4 contacts.

Figure 2-7 DigitalInputDemux object application



See also [Alphabetical list of kitControl components](#)

kitControl-Divide

Divide performs the operation $out = (inA / inB)$. If either input is Numeric.NaN, the output will be Numeric.NaN. The Divide is available in the **Math** folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-DoubleToStatusNumeric

DoubleToStatusNumeric converts a Double value to StatusNumeric. See “Simple value to status value” on page 1-7. It is available in the **Conversion** folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-ElapsedActiveTimeAlarmExt

ElapsedActiveTimeAlarmExt is a special-purpose alarm extension, especially for use as child of a BooleanPoint or BooleanWritable that has one or more **DiscreteTotalizerExt** extensions. It provides alarming on runtime (elapsed active time), using offNormal property errorLimit.

Note: In the parent Boolean point, order the ElapsedActiveTimeAlarmExt slot below the DiscreteTotalizerExt slot that it references. In the ElapsedActiveTimeAlarmExt’s Offnormal container, use the Discrete Totalizer Select property to reference the DiscreteTotalizerExt.

ElapsedActiveTimeAlarmExt is available in the **Alarm** folder of the kitControl palette, along with a **ChangeOfStateCountAlarmExt** (for COS-based alarms). You can use both extensions to reference the same DiscreteTotalizerExt.

See also [Alphabetical list of kitControl components](#)

kitControl-ElectricalDemandLimit

ElectricalDemandLimit provides load shedding calculations based upon a projected electrical demand averaging using a configurable sliding window interval.

To use the ElectricalDemandLimit component, as a minimum, you need to do the following:

- configure the component properties
- setup any Power Input links
- setup the output links
- enable the EDL component

The Shed Out slot (output) is typically linked to a [ShedControl](#) object, which actually performs the equipment shed and restoration control. Based on how you have configured the EDL component, the calculations direct that load shedding, load restoration, or no action be taken. With each calculation, the projected average demand is updated and displayed. Execution of this object can be enabled or disabled (default) either by linking or manually setting the Prediction Enabled property value.

ElectricalDemandLimit is available in the [Energy](#) folder of the kitControl palette, along with related objects [SetpointOffset](#) and [ShedControl](#).

The following sections provide an overview and description of this component:

- [Overview of the Electrical Demand Limit component](#)
- [Determining shed levels](#)

See also [Alphabetical list of kitControl components](#)

Overview of the Electrical Demand Limit component Electrical Demand Limiting (EDL) is an energy management tool that allows you to level-out fluctuations that may occur in daily energy demand levels. Energy providers often set billing rates based on periodic maximum demand levels, so it is possible that a single day (anomaly) could dramatically increase a monthly billing rate. Reducing peak demand levels can significantly lower energy costs - even if the total energy consumption does not change. The ElectricalDemandLimit component is used to monitor and control building or enterprise energy usage in order to avoid costly spikes in demand level.

The EDL component monitors instantaneous electrical power, calculates a “projected demand average over a specified demand interval, and directs the shedding of specified loads whenever the projected demand average is higher than a specified demand limit. As projected demand levels recede, the component invokes a prioritized restoration of loads. Also, the EDL component records and saves peak demand times, dates, and values for both the current month and the previous month.

- **Projected Demand Average**

The EDL component logic executes at a minimum of once per minute using a single Power Input value collected at the current time to calculate a “projected demand average”. It totalizes the Power Input every time the Power Input property value changes, but it only calculates projected demand average and Shed Level once a minute. This projected average is calculated using a combination of *projected* and *historical* samplings that are averaged over a specified interval (configurable in the EDL property sheet view). You can influence the value of the Projected Demand Average by using the Demand Interval and the Percent Interval Elapsed properties.

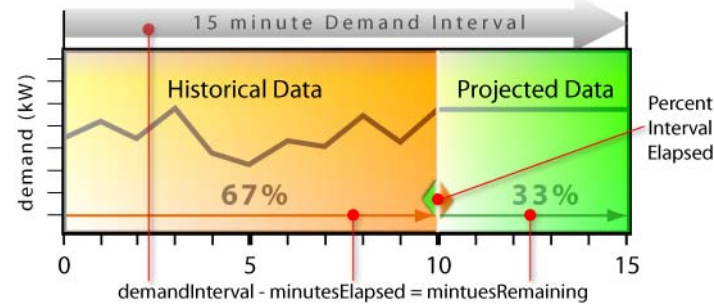
- **Demand Interval**

You can set the demand interval time window using the EDL Demand Interval property. This property value sets the length (in minutes) of the demand window that is used for calculating the average. The default demand interval value is 15 minutes and may be set to 30 minutes—any other entry results in 15 minutes being used. The larger demand interval has more data points (one per minute) than the smaller interval. Depending on the value of the Percent Interval Elapsed property, these data points may be comprised of mostly sampled historical demand values, mostly values that are projected (based on current demand), or half and half.

- **Percent Interval Elapsed**

You can control the weighting of projected demand data usage versus historical demand data usage by setting the Percent Interval Elapsed property value. A Percent Interval Elapsed value of “50” uses half of the actual sampled (or historical) values and half of the projected values in a demand interval to calculate the projected demand average. The projected demand value that is used to figure the “projected” data is the current energy demand.

Figure 2-8 shows how actual historical data and projected data are proportionally used across a 15 minute demand interval by setting the Percent Interval Elapsed property. In this example, the Percent Interval Elapsed property is set to 67, so the actual “minutes elapsed” over the 15 minute demand interval equals 10 minutes and the “minutes remaining” equals 5 minutes. The actual historical demand data is averaged over 10 minutes and the instantaneous demand (taken at the 11 minute point) is used to calculate an average “Projected Data” value. These two numbers are then used to calculate the Projected Demand Average.

Figure 2-8 Demand Interval Illustration

- Demand Periods and Demand Limits**
 A day may be divided into three periods, based on time-of-day, with each period having a specific demand limit value. The Projected Demand Average is compared to the Demand Limit value that is set for the *current* time-of-day to determine whether “shedding” or “restoring” loads is appropriate. If the projected demand is higher than the demand limit for the current time of day, shedding is invoked. If shedding is active and the projected demand is lower than the demand limit for the current time of day, then the restoring is invoked.
- Power Shed Levels**
 There are 32 Power Shed Level properties available in the EDL component. You can set a value on one or more of these properties to make them available for shedding calculations. Each shed level property value represents an amount of power that is dropped when that shed level is active. These values must be entered manually in the property sheet and are estimates, not “live” data. When the projected demand exceeds the Demand Limit for the current demand period, a calculation is performed to determine how many loads to shed. The Power Shed Level properties should be set to a demand value that is based on loads that are controlled by the specific shed group. The load shed logic calculates how much demand is required to be reduced, and then uses the Power Shed Level values to determine how many of the loads to shed. Shedding or restoring loads occurs in a fixed priority that sheds Power Shed Level1 first and restores it last.

Following, are examples that illustrate how Projected Demand Average calculation can vary.

Example: Projected demand average using mostly Projected Values

Assuming that the Demand Interval is set to 15 minutes and the Percent Interval Elapsed property is set to “7” (7%), then the Projected Data (calculated demand) is based on the current minutes demand reading being projected for the remaining minutes in the Demand Interval window. The following example assumes that the Power Input value is 400.

```
minutesElapsed = 15 * 6.67 / 100 = 1
minutesRemaining = 15 - 1 = 14
calculated total = (current Power Input * minutesRemaining) + Power Input from
each minutesElapsed interval
calculated total = (400 * 14) + 400 = 6000
projectedDemand = calculated total / (minutesElapsed + minutesRemaining)
projectedDemand = 6000 / (1 + 14) = 400
```

Note the following about this example:

- By setting the Percent Interval Elapsed property to a value that corresponds to the first minute of the demand window, the calculated demand is based almost entirely on a projected data value.
- In this case the projected demand calculation uses the Power Input value at minute “2” and averages that demand across the remaining 14 minutes.

Example: Projected demand average using all Recorded Values

Assuming that the Demand Interval property is set to 15 minutes and the Percent Interval Elapsed property is set to 93%, then the calculated demand is based *completely* on recorded demand readings for the current minute and the 14 previous minutes. Following, is an example of using all “recorded” values and no “projected” values.

```
minutesElapsed = 15 * 93.33 / 100 = 14
minutesRemaining = 15 - 14 = 1
calculated total = (current Power Input * minutesRemaining) + Power Input from
each minutesElapsed interval
```

```
calculated total = (600 * 1) + 600 + 600 + 600 + 600 + 600 + 600 + 600 + 600  
+ 600 + 400 + 400 + 400 + 400 + 400 = 8000  
projectedDemand = calculated total / (minutesElapsed + minutesRemaining)  
projectedDemand = 8000 / (14 + 1) = 533
```

- There would be no actual “projected” demand in this case.
- This example assumes that the Power Input is currently 600 and has been at that value for the previous 9 minutes, prior to that the value was 400.
- By setting the Percent Interval Elapsed to a value that corresponds to the last minute of the Demand Interval, the projected output is a sliding window average of the minutely recorded Power Input values.
- In this case actual demand is used in the calculation as opposed to projected demand.

Example: Projected demand average using Recorded and Projected Values

The default operation of the EDL component uses a Percent Interval Elapsed property value of 75%. The calculated demand is then based 75% on actual recorded Power Input property values and 25% on a projection that assumes the demand will remain at the current value for the remaining minutes in the Demand Interval.

Determining shed levels When Projected Demand Average exceeds the Demand Limit value for the current Demand Interval, a calculation is performed to determine how many loads to shed. All available (or desired) Power Shed Level properties (1-32) should be set to a demand value based on the loads that are controlled by the specific shed group. The load-shed logic calculates how much demand is required to be reduced, and then uses the Power Shed Level property values to determine how many the loads to shed.

Example: Estimating the Power Shed Level values

An estimate of the demand associated with a group of equipment can be calculated if the operating voltage and current draw are known for the loads. For example:

- **Single Phase Loads**

$$W = V * A$$
$$W = 120 \text{ Volts} * 30 \text{ Amps} = 3600 \text{ Watts} = 3.6 \text{ kW}$$

- **Three Phase Loads (use square root of 3)**

$$W = V * A * 1.73$$
$$W = 480 \text{ Volts} * 30 \text{ Amps} * 1.73 = 24919 \text{ Watts} = 24.9 \text{ kW}$$

Example: Shed calculation

Using the following list of property values, this example shows a calculation that uses an EDL component configured with three power shed levels. For this example, assume that the current Demand Limit Period is Demand Limit Period1.

- calculated total = 7625 kW
- Projected Demand Average = 533 kW
- Demand Interval = 15
- Percent Interval Elapsed: = 75
- Demand Limit Period1: = 500 kW
- Demand Limiting Deadband = 5 kW
- Power Shed Level: = 20 kW
- Power Shed Level2: = 15 kW
- Power Shed Level3: = 30 kW

The following equations show example calculations:

```
targetIntervalTotal = demandLimit * (minutesElapsed + minutesRemaining)  
targetIntervalTotal = 500 * (11.25 + 3.75) = 7500  
powerChange = (calculated total - targetIntervalTotal) / minutesRemaining  
powerChange = (7625 - 7500) / 3.75 = 33.33 KW that needs to be shed
```

Note the following about this example:

- Since Power Shed Level1 is only expected to reduce the demand by 20 kW, both Power Shed Level1 and Power Shed Level2 must be shed to reduce the demand by an expected 35 kW combined.
- The necessary loads are shed in sequential order during the same execution cycle, without evaluating the actual impact on demand.
- The load shed determination is based on the projected reduction in demand for each group.
- Subsequent executions of the object may result in additional load shedding if the actual demand is not reduced below the demand limit.

The following properties are available in the Electrical Demand Limit component property sheet view.

- **Prediction Enabled**
This property allows you to enable or disable the EDL component by choosing `true` or `false`, respectively. Choosing the `null` option (by selecting the null checkbox from the property sheet view) leaves the “enabled” status in its current state. For example, if this property is currently set to `true` then choosing the `null` option does not stop the execution of the EDL component. This value must be set to `true` for the component to work.
- **Power Input**
This property is a writable field that allows you to link in a numeric value that represents the actual power demand (kW) rate. This property monitors the demand rate and averages it over every minute in order to use the value for comparison to the Projected Demand Average. This property should always represent the total actual demand rate — the total of all meters that are on the energy network being monitored by this component. Whenever a shed or restoration is invoked, this value is expected to change in relation to the estimated values that are set in the Power Shed Level properties.
- **Message**
This property displays information that relates to the status of the shed, restoration, or projected demand values. It also may indicate the status of the EDL component, itself.
- **Shed Out**
This property displays a value that indicates the number of shed levels that are to be shed. For example, a Shed Out value of 3 specifies that a Power Shed Level of 3 is being shed.
- **Billing Start Delay**
This property specifies the first billing day of the month for utility billing. This allows you to align your data with actual energy company billing periods. Each month, on the day specified by this property, the “current month” data moves to “Previous Month” and the current month data becomes “This Month”.
- **Demand Interval**
This property represents the length of time, in minutes, that is used for the demand window portion of the Projected Demand Average calculation. The default value is 15 minutes and may also be set to 30 minutes.
Note: If any value other than 15 or 30 minutes is entered in this field, the value automatically reverts to 15.
- **Percent Interval Elapsed**
This property is used to determine how much of the calculated demand is based on *actual* demand as opposed to how much is based on *projected* demand. This integer value is used to set where in the demand window the “current minute” is. In a 15 minute demand window, a value of 67 would mean that the “current minute” is at 10. Larger numbers in this property increase the amount of historical data that is used and decrease the amount of data that is based on the “current minute” demand.
- **Rotate Level**
This property specifies the maximum Shed Level that may be used. For example, a Rotate Level value of 3 limits load shedding to Shed Level 3.
- **Demand Limiting Deadband**
This property allows you to set a deadband value that is used when activating restoration levels. The deadband value is used only in determining whether or not to invoke a restoration action; it is not used for invoking shed actions.
- **Demand Period (1, 2, and 3) Start**
These three properties allow you to split-up a 24 hour day into three different time-periods in order to assign a separate demand limit for each distinct time.
- **Demand Limit Period(1,2,3)**
These three property fields allow you to set a desired demand limit value to correspond to each of the three demand periods. When the Demand Limit value for a period is exceeded, load shedding is invoked.
- **Power Shed Level (1-32)**
These properties allow you to set up to 32 estimated power shed levels. Each property represents the amount of demand that you expect to shed when the associated shed level is invoked. The numbers in these properties are used to calculate how many shed levels need to be invoked in order to lower the demand level below the current Demand Period limit. Once a shed level is invoked, the actual power drop is evaluated at the next minute to determine the actual effects of the shed action. If the initial load shed does not actually bring down the demand to below the demand limit level, the next shed level (if any) is invoked.
Note: You can limit the maximum number of shed levels that may be invoked by using the Rotate Level property.

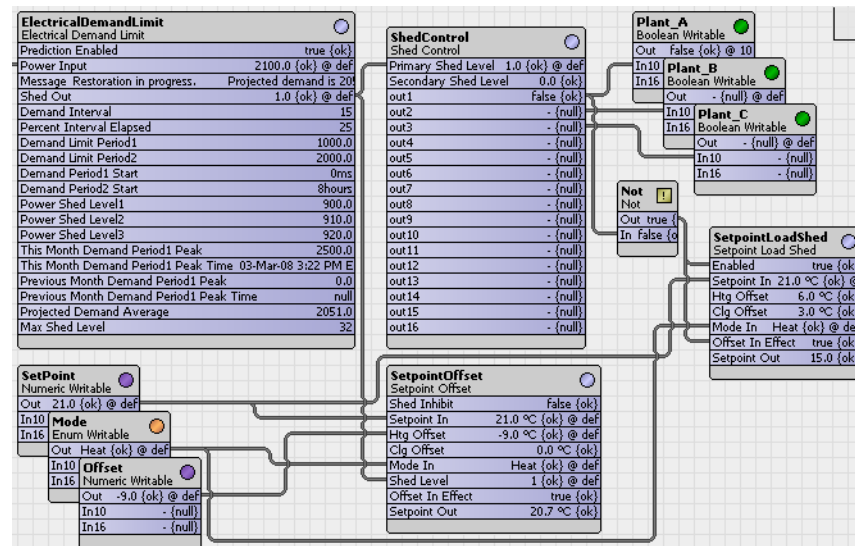
- **This Month Demand Period (1,2, or 3) Peak**
This historical data property displays the value of the highest demand (minute) that has occurred (so far) in the current month.
- **This Month Demand Period (1,2, or 3) Time**
This property is associated with the “This Month Demand Period (1,2, or 3) Peak and displays the time and date that the current month’s peak demand occurred.
- **Previous Month Demand (1,2, or 3) Period Peak**
This historical data property displays the value of the highest demand (minute) that occurred in the previous month.
- **Previous Month Demand (1,2, or 3) Period Time**
This property is associated with the “Previous Month Demand Period (1,2, or 3) property and displays the time and date that the previous month’s peak demand occurred.
- **Projected Demand Average**
This is the read-only display of the average demand that is predicted to occur for the current demand interval. Calculations occur to update this value every minute.
- **Max Shed Level**
This property displays the maximum shed level that has been used in the current month.

The following example illustrates using the Electrical Demand Limit component.

Example: Setting EDL component links and properties

The following example shows a partial wiresheet view of an EDL component configured for shedding energy loads. The current time period in this example is Demand Period2.

Figure 2-9 Wiresheet view of EDL example application



Note the following about this example:

- **EDL Configuration**
 - **Power Input**
Total demand is linked into the EDL component Power Input property. This is a single input property, therefore the power sources need to be totaled before linking because there is more than one meter supplying actual electrical demand data. This demand level value is expected to change in response to load shedding.
 - **Demand Interval**
This property is set to the default value of 15 minutes.
 - **Percent Interval Elapsed**
This property is set to 25 percent, which adds more weight to the “current demand” and less weight to “historical demand” for each calculation of the Projected Demand Average.
 - **Demand Limit**
Demand limits are shown for Period1 and Period2, as 1000 and 2000, respectively. These are the values that specify the demand levels that initiate power shedding.

- **Demand Period**
Demand period start times are shown for Period1 and Period2, as 0000 (midnight) and 0800 (8:00 am), respectively. Demand Period3 start time is not shown. These values specify the start time for each of the three Demand Periods.
- **Power Shed levels**
In this example, it is estimated that by shedding loads associated with Power Shed Level1, that the amount of demand will decrease by 900 kW. Power Shed Levels 2 and 3 are set at 910 and 920, respectively. These properties are only the estimated amount of demand that is reduced by shedding at each respective level. If they are exactly correct, then shedding at level 3 reduces demand by the sum of all three shed levels: $(900+910+920)=27030\text{kW}$.
- **Projected Demand Average**
The current value for this property is shown as 2051, so shedding is initiated, as shown in the message property “SHEDDING REQUIRED! Projected demand is 2051” and in the Shed Out property value of “1.0” (Shed Level 1).
- **EDL Linking**
 - **Shed Out**
The EDL Shed Out property value is linked to a Shed Control component that allows you to set specific Shed Level(1-16) links into boolean controls. In the example, these controls are configured to shut off power to “Plant_A”, “Plant_B”, and “Plant_C”, with Shed Level(1, 2, and 3), respectively. In addition, the Shed Control component “out1” value is linked to a SetpointLoadShed component that uses a configurable setpoint offset to reduce power usage. You can also link from the Shed Out property to other energy components, such as a Setpoint Offset component (also shown here).
 - **Power Input**
In the example, with a Shed Level1 in effect, Power Input is at 2100, and the Projected Average Demand value is 2051, still greater than the Demand Limit Period2 value of 2000. The Power Shed Level2 value (estimate) indicates that invoking a Power Shed Level2 will yield a decrease of 910kW and bring the demand down below the limit. If this estimate is fairly accurate, actual power usage should drop and the Power Input value lower to below the Demand Limit Period2 value.

kitControl-EnumConst

- Provides constant EnumStatus value, with available action to Set. See “About Constant components” on page 1-4. EnumConst is available in the [Constants](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-EnumLatch

- EnumLatch provides a latch for a StatusEnum input, and is available in the [Latches](#) folder of the kitControl palette. See “About Latch components” on page 1-8.

See also [Alphabetical list of kitControl components](#)

kitControl-EnumSelect

- ▣ EnumSelect is an Enum select, and is available in the [Selects](#) folder of the kitControl palette. See “About Select components” on page 1-13 for an overview.

See also [Alphabetical list of kitControl components](#)

kitControl-EnumToStatusEnum

- ▣ EnumToStatusEnum converts an Enum value to StatusEnum, and is available in the [Conversion](#) folder of the kitControl palette. See “Simple value to status value” on page 1-7.

See also [Alphabetical list of kitControl components](#)

kitControl-EnumSwitch

- ▣ EnumSwitch selects one of two StatusEnum inputs based upon the boolean value at the Status-Boolean input “In Switch.” EnumSwitch is available in the [Util](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-Equal

- ▣ Equal performs the operation $A == B$. Numeric.NaN values are never equal. Equal is available in the [Logic](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-Exponential

- Exponential performs the operation $out = e^{inA}$ (e raised in the inA power). The Exponential is available in the [Math](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-Factorial

- (AX-3.5 and later) Factorial provides a factorial math output, based upon the value present at its statusNumeric input. Only the integer portion of the input value is evaluated—for example, either value of 1.03 or 1.9999 is evaluated as 1. Factorial is available in the [Math](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-FloatToStatusNumeric

- FloatToStatusNumeric converts a Float value to a StatusNumeric. See “Simple value to status value” on page 1-7. FloatToStatusNumeric is available in the [Conversion](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-GreaterThan

- GreaterThan performs the operation $A > B$ with a boolean result. It is available in the [Logic](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-GreaterThanEqual

- GreaterThanEqual performs the operation $A \geq B$ with a boolean result. It is available in the [Logic](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-IntToStatusNumeric

- IntToStatusNumeric converts an Int (integer) value to StatusNumeric. See “Simple value to status value” on page 1-7. IntToStatusNumeric is in the [Conversion](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-InterstartDelayControl

- InterstartDelayControl objects are just like BooleanWritables, but with 3 additional slots for use in interstart delay sequences, as follows:

- Delay — Delay before next object in delay sequence is started.
- Master — Specifies the [InterstartDelayMaster](#) component in the station that acts as delay master.
- Start Pending — Read-only Boolean status of whether a start is pending (true) or not (false).

No other [InterstartDelayControl](#) object using the same delay master can start for delay time after this object starts. If delay is not defined, the default delay on the master will be used. InterstartDelayControl is available in the [HVAC](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-InterstartDelayMaster

- InterstartDelayMaster defines the master in an interstart delay sequence. Use it in conjunction with one or more [InterstartDelayControl](#) objects. An available action is **DelayTimerExpired**. The InterstartDelayMaster is available in the [HVAC](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

- **DelayTimerExpired**
DelayTimerExpired is an available action of an [InterstartDelayMaster](#).

kitControl-LeadLagCycles

- LeadLagCycles provides lead-lag control of from 2 to 10 BooleanWritables based upon their accumulated COS (change of state) counts. This object balances the number of change of states cycles of each of the devices. Only one of the controlled devices will be active at a time based on cycle count.

LeadLagCycles is available in the [HVAC](#) folder of the kitControl palette, along with a similar [LeadLagRuntime](#) object.

Setup of the object involves the following properties:

- **In**
A StatusBoolean input that controls whether any control device should be on. If this input is true, one of the outputs will be active based on the cycle count of each controlled device.
- **Number Outputs**
Specifies the number of devices (outputs) that are controlled.
- **Max Runtime**
Specifies the maximum amount a given output will be true before switching to another output.
- **Feedback**
A StatusBoolean input, to provide positive feedback that a controlled device actually started. If the feedback value does not show true within the Feedback Delay time, the current controlled output will show alarm, and the LeadLagCycles switches to the next controlled output. Setting this value to true (and not linking) disables this alarm feature.
- **Feedback Delay Time**
Specifies the delay time used to evaluate the feedback link (if any)
- **Out A—J**
StatusBoolean outputs, each typically linked to a BooleanWritable control point with a DiscreteTotalizerExt. Outputs are typically used to control loads of some type, such as 2 or more pumps.
- **Cycle Count A—J**
These are Integer inputs that are used for cycle count feedback for the corresponding Out A - J. These inputs will typically be linked to the ChangeOfStateCount property of the DiscreteTotalizerExt that is measuring the cycles of the corresponding Out A - J.

Example: Using the LeadLagCycle component

A simple example LeadLagCycle object that controls 3 pumps is shown in Figure 2-10 and Figure 2-11.

Figure 2-10 LeadLagCycle example property sheet

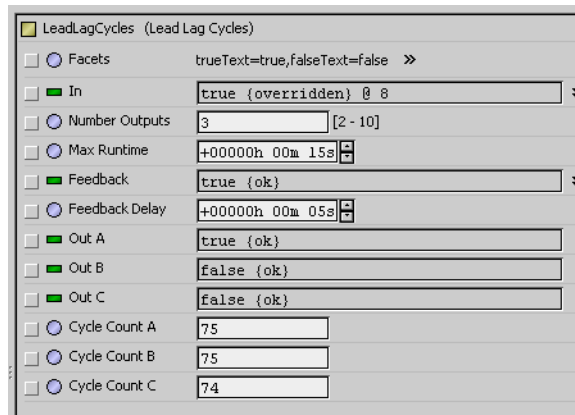
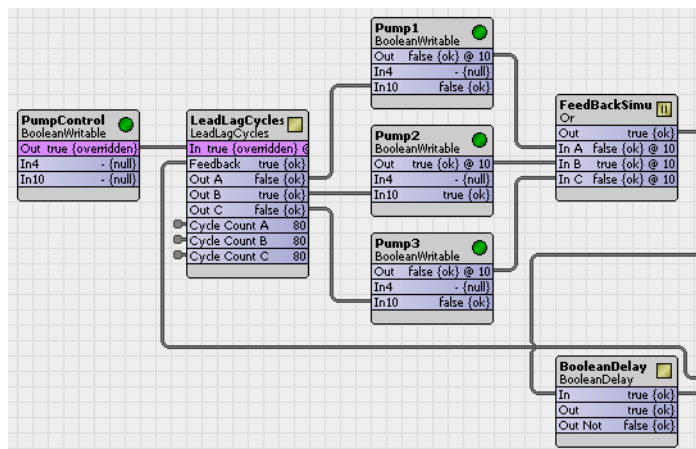


Figure 2-11 LeadLagCycle example with linked objects



Note that in this example, each of the three BooleanWritable points has a DiscreteTotalizerExt, with its changeOfStateCount slot linked back to a Cycle Count x input on the LeadLagCycles object. The “feedback” or object simulates feedback, fed through a BooleanDelay object.

See also [Alphabetical list of kitControl components](#)

kitControl-LeadLagRuntime

LeadLagRuntime provides lead-lag control of from 2 to 10 BooleanWritables based upon their accumulated runtimes (elapsed active time). This object balances the active runtime of each of the devices. Only one of the controlled devices will be active at a time based on runtime.

LeadLagRuntime is available in the HVAC folder of the kitControl palette, along with a similar [LeadLag-Cycles](#) object.

Setup of the object involves the following properties (also see [LeadLagRuntime usage](#)), as follows:

- **In**
A StatusBoolean input that controls whether any control device should be on. If this input is true, one of the outputs will be active based on runtime.
- **Number Outputs**
Specifies the number of devices (outputs) that are controlled.
- **Max Runtime**
Specifies the maximum amount a given output will be true before switching to another output.
- **Feedback**
A StatusBoolean input, to provide positive feedback that a controlled device actually started. If the feedback value does not show true within the Feedback Delay time, the current controlled output will show alarm, and the LeadLagRuntime switches to the next controlled output. Setting this value to true (and not linking) disables this alarm feature.
- **Feedback Delay Time**
Specifies the delay time used to evaluate the feedback link (if any)
- **Out A—J**
StatusBoolean outputs, each typically linked to a BooleanWritable control point with a DiscreteTotalizerExt. Outputs are typically used to control loads of some type, such as 2 or more pumps.
- **Runtime A—J**
These are RelTime inputs that are used for runtime feedback for the corresponding Out A - J. These inputs will typically be linked to the ElapsedActiveTime property of the DiscreteTotalizerExt that is measuring the runtime of the corresponding Out A - J.

Example: LeadLagRuntime usage

A simple example LeadLagRuntime object controlling 3 pumps is shown in [Figure 2-12](#) and [Figure 2-13](#).

Figure 2-12 LeadLagRuntime example property sheet

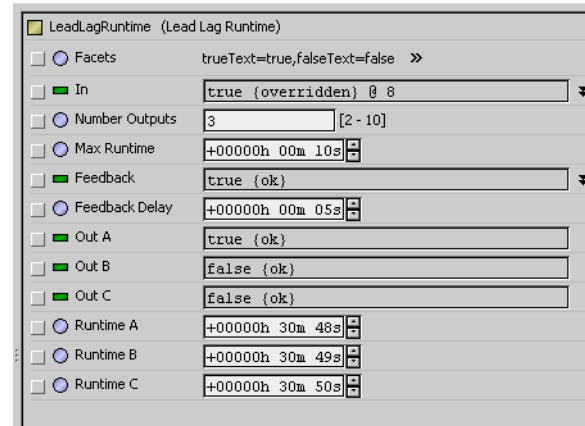
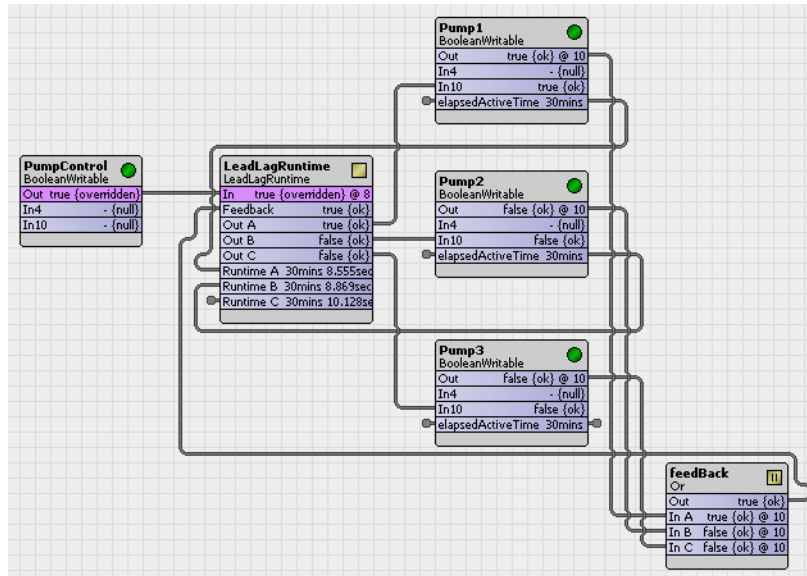


Figure 2-13 LeadLagRuntime example with linked objects



Note that in this example, each of the three BooleanWritable points has a DiscreteTotalizerExt, with its elapsedActiveTime slot exposed up in the composite of the parent point for link clarity. The “feedback” Or object simulates feedback, fed through a BooleanDelay object.

See also [Alphabetical list of kitControl components](#)

kitControl-LessThan

LessThan performs the operation $A < B$ with a boolean result. It is available in the **Logic** folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-LessThanEqual

LessThanEqual performs the operation $A \leq B$ with a boolean result. It is available in the **Logic** folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-LogBase10

LogBase10 performs the operation $out = \log_{10}(inA)$ (log base 10 of inA). It is available in the **Math** folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-LogNatural

LogNatural performs the operation $out = \ln(inA)$ (log base e of inA). The LogNatural is available in the **Math** folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-LongToStatusNumeric

LongToStatusNumeric converts a Long value to StatusNumeric. See “Simple value to status value” on page 1-7. LongToStatusNumeric is available in the **Conversion** folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-LoopAlarmExt

The LoopAlarmExt is a special-purpose alarm extension, especially for use as child of a LoopPoint. It provides alarming as a “deviation-from-current-setpoint” (plus or minus), using offnormal properties errorLimit and deadband. This extension is available in the **Alarm** folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-LoopPoint

● The LoopPoint implements a simple PID control loop, and is available in the [HVAC](#) folder of the kitControl palette. Loop objects provide closed-loop PID control (proportional, integral, derivative) at the station level. Independent gain constants allow the loop to be configured as P-only, PI, or PID.

ResetIntegral is an available action on a LoopPoint. If invoked, this clears the current integral component of the loop's output calculation. If needed, this slot can be linked to another object to provide a quick purge of the integral effect. Typically, the latter would provide more of a “debug” utility, and should not be necessary if the LoopPoint's configuration properties are correctly defined.

The following sections provide more LoopPoint details:

- “[LoopPoint setup](#)” on page 2-23
- “[Loop terms](#)” on page 2-24
- “[Proportional-only control](#)” on page 2-24
- “[Proportional with Integral \(PI\) control](#)” on page 2-25
- “[Proportional with Integral and Derivative \(PID\) control](#)” on page 2-26
- “[LoopPoint Examples](#)” on page 2-27

See also [Alphabetical list of kitControl components](#)

LoopPoint setup Setup of the [LoopPoint](#) component involves setting the following properties:

- **Facets**
Used to set the units and display number precision of the output slot.
- **Loop Enable**
Setting this input to true will enable the PID loop algorithm to execute at the rate selected by the Execute Time property. Setting this input to false will force the PID loop output to a value dependent on the selection in the Disable Action property.
- **Input Facets**
Used to set the units and number precision of the input slot (control variable and setpoint).
- **Control Variable**
Input for the controlled parameter (for example, space temperature). This input must be valid for this object to function.
- **Setpoint**
Input for the setpoint value (for example, space temperature setpoint). This input must be valid for this object to function. The object does not provide an integral command function for the setpoint value when entered on the property sheet. If a commandable setpoint is required, link from a NumericWritable control point to the setpoint slot.
- **Execute Time**
Controls the execution frequency for the PID algorithm, where the default value is 0.5 seconds.
- **Loop Action**
Determines whether the control algorithm is direct or reverse acting.
 - Loops setup for *direct acting* mode increase the loop output as the value of the controlled variable becomes *greater than* the setpoint value. In a temperature loop, this is typically considered to be a *cooling* application.
 - Loops setup for *reverse acting* mode increase the loop output as the value of the controlled variable becomes *less than* the setpoint value. In a temperature loop, this is typically considered to be a *heating* application.
- **Disable Action**
The value that the loop output will be set to when the loop is disabled by setting the Loop Enable property to false.
 - Max Value sets the loop output value to the Max Output property value.
 - Min Value sets the loop output value to the Min Output property value.
 - Hold maintains the loop output at the last calculated value.
 - Zero sets the loop output value to a zero (0.0) value.
- **Proportional Constant**
Defines the value of the proportional gain parameter used by the loop algorithm. Used to set the overall gain for the loop. A starting point for this value is found by output range/throttling range.
- **Integral Constant**
Defines the integral gain parameter, in repeats per minute, used by the loop algorithm. Also called reset rate. Acts on magnitude of the setpoint error. A typical starting point is 0.5.
- **Derivative Constant**
Defines the derivative gain parameter, in seconds, used by the loop algorithm. Acts on the rate of change of the setpoint error.

- **Bias**
Defines the amount of output bias added to the output to correct offset error, normally used only used with proportional control.
- **Maximum Output**
Defines the maximum output value that the loop algorithm can produce.
- **Minimum Output**
Defines the minimum output value that the loop algorithm can produce.
- **Ramp Time**
Defines the minimum time that the output can ramp completely from Minimum Output to Maximum Output, effectively establishing a “rate of change” slope. This rate of change is enforced upon station startup, or whenever the LoopPoint transitions from disabled to enabled.
Once the Ramp Time has expired, it has no effect on the output. Intended use is to prevent the loop from “slamming” a valve or other controlled device to a limit during startup.
Note: The default Ramp Time is 0 : 00 : 00, or disabled. To constrain loop output rate of change when the loop starts or is enabled, enter a reasonable Ramp Time value.

Loop terms The following terms are used when describing the operation of the [LoopPoint](#) component:

- **Process variable**
The controlled process, meaning the value at the setpoint input. (“What you’ve got.”) Abbreviated here as “PV.”
- **Setpoint**
The target for the process variable, meaning the value at the setpoint input. (“What you want.”) Abbreviated here as “setpt.”
- **Setpoint error**
The difference between the process variable and the setpoint, acted upon by the loop algorithm. Abbreviated as “E_S.”
- **Loop output**
The correction signal produced by the loop algorithm. The output should be linked (directly or indirectly) to a NumericWritable component used to position a proportionally-modulated device (such as a valve or damper) that controls the process variable.
- **Proportional gain**
The value of the property Proportional Constant. Abbreviated here as “K_p”. Sets the overall gain of the loop, as in the following ratio:
$$K_p = \text{Output range} / \text{effected process range (sometimes called throttling range)}$$
- **Throttling range**
The amount of *process variable change* expected as a result of throttling the system between the minOutput and maxOutput.
- **Bias**
A value added to the output to correct offset error. It is typically used in proportional-only control as a “pivot” output value, for when the PV = setpt.
- **Action**
Defines the “direction” of the output relative to setpoint error, where:
 - Direct — Loop output increases when PV increases.
 - Reverse — Loop output increases when PV decreases.
- **Integral gain**
The value of the property integralConstant. Abbreviated as “K_i”. Sets the integral or “reset” gain of the loop, expressed in repeats per minute. The K_i component of the loop output reacts to the duration of the setpoint error.
- **Derivative gain**
The value of the property derivativeConstant. Abbreviated as K_D. Sets the derivative or “rate” gain of the loop, expressed in repeats per minute. The K_D component of the loop output reacts to the “rate of change” of the setpoint error, and provides a “dampening” effect.

Proportional-only control P-only control is just reset action, where loop output is directly proportional to the magnitude of the setpoint error (E_S) and the size of the proportional gain (K_p).

The following topics apply to PI loop control with a [LoopPoint](#):

- [Output calculation](#)
- [P-only configuration guidelines](#)

Output calculation P-only loop output is linear, and is calculated as follows:

$$\text{Output} = (K_p \times E_S) + \text{bias} \quad (\text{if action} = \text{direct}), \text{ or}$$

$$\text{Output} = - ((K_p \times E_s) + \text{bias}) \quad (\text{if action} = \text{reverse})$$

where:

$$E_s = [\text{PV} - \text{setpt}]$$

P-only configuration guidelines If using proportional-only loop control, follow these guidelines:

Output limits Define the `maxOutput` and `minOutput` properties for the loop output, noting that the maximum value must be greater than the minimum.

Proportional Gain Calculate and enter a `proportionalConstant` (K_p) property value starting with this formula:

$$[\text{output range} (\text{maxOutput} - \text{minOutput})] / \text{throttling range}$$

where `throttling range` is the corresponding result in the process variable.

For example, for a temperature loop where a 0-to-100% loop output results in a 20 degree swing in the process variable, a starting point K_p is:

$$[(100\% - 0\%) / 20\text{deg.}] = [(100\% / 20\text{deg.})] = 5$$

When tuning the loop, you can try increasing this value (effectively using only a portion of the throttling range) to eliminate the amount of setpoint error. However, if you increase the K_p too much, this typically results in a constant oscillation of the process variable (above and below the setpoint).

Bias Assign the `bias` property an “output-midpoint” value (for example, 50.0). This allows for equal corrections for a process variable above or below setpoint.

Integral and Derivative Gain Set the properties `integralConstant` and `derivativeConstant` to 0.0 (the defaults).

Proportional with Integral (PI) control PI configuration is recommended for most control loops, because the integral term eliminates the setpoint offset inherent in P-only loops. PI control uses proportional gain to adjust the output, and then incrementally continues to “add” (or subtract, if appropriate) from the output value for as long as a setpoint error continues to exist.

The following topics apply to PI loop control with a [LoopPoint](#):

- [Output calculation](#)
- [Repeats per minute](#)
- [Integral overshoot](#)
- [Integral windup prevention](#)
- [PI configuration guidelines](#)

Output calculation PI loop output is calculated as follows:

$$\text{Output} = K_p \times (E_s + K_I \times \text{ErrorSum}) \quad (\text{if action} = \text{direct}), \text{ or}$$

$$\text{Output} = - (K_p \times (E_s + K_I \times \text{ErrorSum})) \quad (\text{if action} = \text{reverse})$$

where:

$$E_s = [\text{PV} - \text{setpt}]$$

$$\text{ErrorSum} = \text{Sum of } E_s \text{ over time}$$

The `integralConstant` property specifies the integral gain (KI) in “[Repeats per minute](#),” sometimes called a “reset rate.”

Repeats per minute To understand repeats per minute, consider the scenario where a loop is controlling at setpoint. If a certain setpoint error occurs, say from a sudden setpoint change, the loop output immediately changes by a level corresponding to its proportional constant (acting on the P-term). During this hypothetical example, assume the controlled process does not react from any loop output change, but stays at the original value (setpoint error stays constant).

The loop’s integral term *immediately* begins increasing the output (or decreasing the output, depending on the direction of setpoint error) at specific rate determined by the integral term. Over the period of one minute, the amount of output change that would occur is defined by the `integralConstant` (repeats per minute). A “repeat” equals the amount of output change initially generated by the P-term. For example, if this loop was configured with an `integralConstant` value of 2.0, and the original output change was +7%, over a period of one minute the integral term would linearly ramp up the output value an additional +14%, or “2 repeats.”

In a real-world PI loop, of course, the process variable *does* respond to an output change, and this continuously-linear ramping of the output would not occur. Instead, the process variable would start moving towards setpoint and the setpoint error would change (changing the proportional and integral terms, thus the loop output).

Integral overshoot The integral term of a PI loop can cause an “overshoot” of setpoint, meaning that the increased loop output may result in a new setpoint error in the opposite direction. In some cases, it is possible for this overshoot to continuously repeat (oscillation), which is typically undesired. However, a small amount of overshoot for an initial correction is not uncommon.

To minimize overshoot, the PI loop’s integralConstant is typically kept small, and sized appropriately for the assigned proportionalConstant.

Integral windup prevention Integral windup is prevented by limiting the ErrorSum value based on the LoopPoint’s Maximum Output and Minimum Output values.

PI configuration guidelines If using PI loop control, follow these guidelines:

Output limits Define the Maximum Output and Minimum Output properties for the loop output, noting that the maximum value must be greater than the minimum.

Proportional Gain Calculate and enter a proportionalConstant (K_p) property value starting with this formula:

$$[\text{output range (minOutput - maxOutput)}] / \text{throttling range}$$

where throttling range is the corresponding result in the process variable.

For example, for a temperature loop where a 0-to-100% loop output results in a 20 degree swing in the process variable, a starting point K_p is:

$$[(100\% - 0\%) / 20\text{deg.}] = [(100\% / 20\text{deg.}] = 5$$

When tuning a PI loop, you typically *reduce* the proportionalConstant value, because the integral effect on the output will correct setpoint error over time.

Bias Assign a value of 0.0 (*no output bias*). A fixed bias is *not desired*, because the integral term of the loop effectively creates an “adjustable bias,” as needed.

Integral Gain Set the integral gain (property integralConstant) to a nominal value, typically less than one (1.0). A value of 0.5 is a good starting point for many loops. Decreasing the integral constant will make the loop respond more slowly.

Derivative Gain Disable derivative by setting the derivativeConstant property at 0.0 (the default).

Proportional with Integral and Derivative (PID) control PID loop control can be difficult to tune and (often for this reason) is seldom used. However, in certain cases, PID control may be needed. An example is the control of a process with a long “reaction time,” such as temperature control of a large mass. For such a lag-oriented system, the derivative component of the PID loop output can help prevent “overshoot” that might otherwise result from PI control.

The derivative gain (K_D) exerts an anticipating “braking” effect on the loop output, based on the rate-of-change of the process.

The following topics apply to PID loop control with a [LoopPoint](#):

- [Output calculation](#)
- [PID configuration guidelines](#)

Output calculation PID loop output is calculated as follows:

$$\text{Output} = K_p \times (E_s + K_I \times \text{ErrorSum} + K_D \times ((E_s - \text{LastE}_s) / \text{deltaT}))$$

(if action = direct), *or*

$$\text{Output} = -(K_p \times [E_s + K_I \times \text{ErrorSum}] + K_D \times ((E_s - \text{LastE}_s) / \text{deltaT}))$$

(if action = reverse)

where:

$$E_s = [\text{PV} - \text{setpt}]$$

ErrorSum = Sum of E_s over time

LastEs = last error

deltaT = time between samples

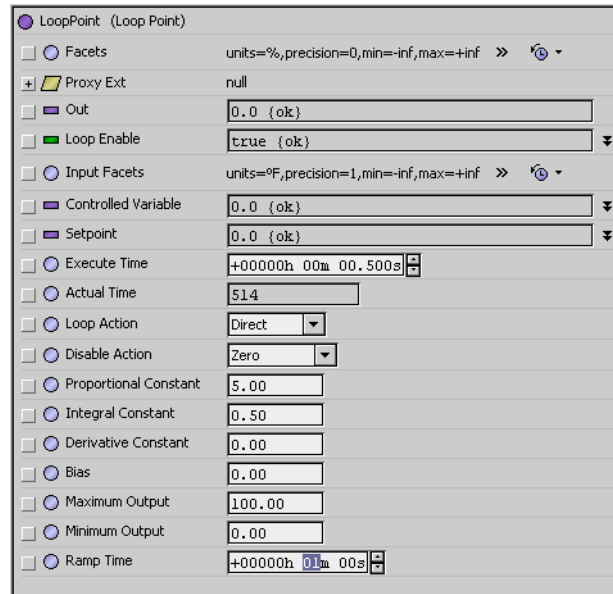
In the **LoopPoint**, the derivativeConstant property specifies the derivative gain (κ_D) directly in seconds (note this differs from some systems using derivative in minutes).

PID configuration guidelines If using PID control, follow the “[PI configuration guidelines](#)” on page 2-26, with the addition of defining a positive value as the derivativeConstant.

In general, a derivativeConstant less than 10 seconds should be tried first, and only then increased (if necessary), providing that the loop output remains stable at steady-state conditions.

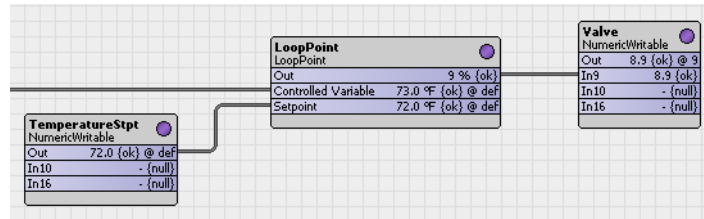
LoopPoint Examples [Figure 2-14](#) shows an example of a **LoopPoint** property sheet.

Figure 2-14 Example LoopPoint property sheet



[Figure 2-15](#) shows an example of a direct-acting **LoopPoint** with a commandable setpoint.

Figure 2-15 Example LoopPoint in wire sheet, linked to other components.



kitControl-Maximum

Maximum determines the maximum value of valid inputs and writes that value to out. Out = max (inA, inB, inC, inD). The Maximum is available in the [Math](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-Minimum

Minimum determines the minimum value of valid inputs and writes that value to out. Out = min (inA, inB, inC, inD). The Minimum is available in the [Math](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-MinMaxAvg

MinMaxAvg has 3 StatusNumeric output slots that provide the current minimum, maximum, and average values of from 2 to 10 linked StatusNumeric inputs. It is available in the [Util](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-Modulus

☒ (AX-3.5 and later) Modulus provides a modulus operation based on values at its two statusNumeric inputs. The output is the remainder of dividing the inA value by the inB value. If the inB value is 0, the output is NaN (not a number). Note that operation is intended for integer input values, such as from the output of a Counter component. Modulus is available in the [Math](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-Multiply

☒ Multiply performs the calculation $out = inA * inB * inC * inD$. The Multiply is available in the [Math](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-MultiVibrator

☒ MultiVibrator provides an oscillating binary pulse output (StatusBoolean) with a period configurable between 200ms to infinity, and a duty cycle configurable from 0 to 100%. It is available in the kitControl palette's [Util](#) folder.

See also [Alphabetical list of kitControl components](#)

kitControl-Negative

☒ Negative simply converts any input status numeric to a negative output value. Negative is available in the [Math](#) folder of the kitControl palette

See also [Alphabetical list of kitControl components](#)

kitControl-NightPurge

● This component is available in the kitControl palette [Energy](#) folder. It uses the two sets of temperature and humidity inputs to find the air supply with the least amount of heat when the purgeEnabled input is true. The freeCooling output will be set to false if `outside >= inside` or set to true if `outside = nightSetpoint`.

For inside and outside comparisons, you can select either temperature or enthalpy comparisons. There is also a low temperature check to protect against freezing.

The NightPurge component includes the following properties:

- **Temperature Facets**
Specifies the units and number precision of the Outside Temp, Inside Temp, and Low Temperature Limit properties.
- **Humidity Facets**
Specifies the units and number precision of the Outside Humidity and Inside Humidity properties.
- **Purge Enabled**
StatusBoolean, must be true to enable night purge operation. Whenever false, the Free Cooling output is set to the opposite of the Free Cooling Command (or null, if Use Null Output is set to true), and the Current Mode slot value is "Disabled."
Often, Purge Enabled is linked to a "Not" object sourced from a BooleanSchedule output.
- **Outside Temp**
Input for the current outside air temperature. This input must be valid for this object to function.
- **Outside Humidity**
Input for the current outside air humidity. This input must be valid for this object to function.
- **Inside Temp**
Input for the current inside air temperature. This input must be valid for this object to function.
- **Inside Humidity**
Input for the current inside air humidity. This input must be valid for this object to function.
- **Low temperature Limit**
This property is used to provide freeze protection.
- **Night Setpoint**
Inside night temperature setpoint, at or below which free cooling is not applied. Instead, the Current Mode is set to "Satisfied."
- **Outside Enthalpy**
This is the calculated outside air enthalpy.
- **Inside Enthalpy**
This is the calculated inside air enthalpy.

- **Free Cooling**
A StatusBoolean output set to value of the Free Cooling Command when it is determined that free cooling should be used. Otherwise, the value is set to the opposite state, or null (if Use Null Output is set to true).
- **Current Mode**
This enumeration indicates which of the following modes this object is currently in:
 - Disabled (Purge Enabled is false)
 - Free Cooling
 - No Free Cooling (free cooling not available)
 - Low temperature (Outside Temp below Low Temperature Limit, free cooling disabled)
 - Input error (A temperature or humidity is invalid (down, fault, etc.), free cooling disabled)
 - Satisfied (Inside temperature below Night Setpoint, free cooling disabled)
- **Setpoint Deadband**
Temperature setpoint deadband applied when inside temperature falls below Night Setpoint, before free cooling can be enabled. Default value is 1.0.
- **Threshold Span**
The difference between the inside enthalpy and the outside enthalpy must be greater than this value before free cooling will be enabled. Default value is 1.0.
- **Use Enthalpy**
Setting this property to true will enable the use of enthalpy for determining if free cooling is available. Otherwise, it will just use outside and inside temperature to decide.
- **Free Cooling Command**
If it is determined that free cooling is available, this is the boolean value that will be set in the Free Cooling output.
- **Use Null Output**
If this property is true, then the null flag will also be set on the Free Cooling output when free cooling is *not* available.

The following illustrations show some example property sheet and wiresheet views of the NightPurge component usage.

Figure 2-16 Example NightPurge property sheet

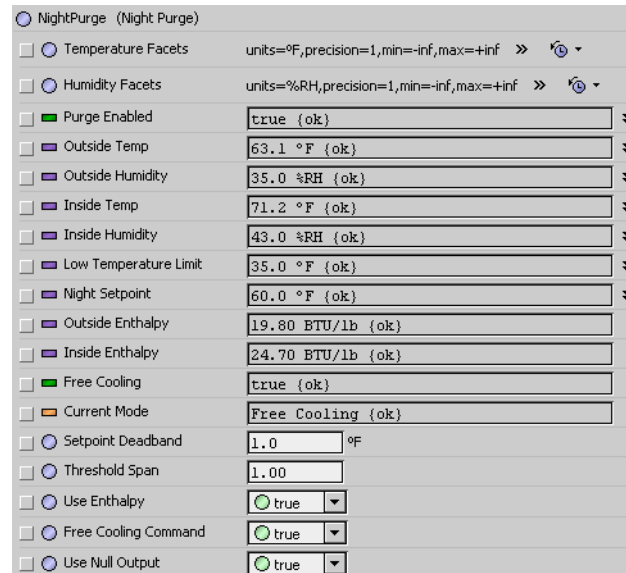


Figure 2-17 Example NightPurge application: Purge enabled and active

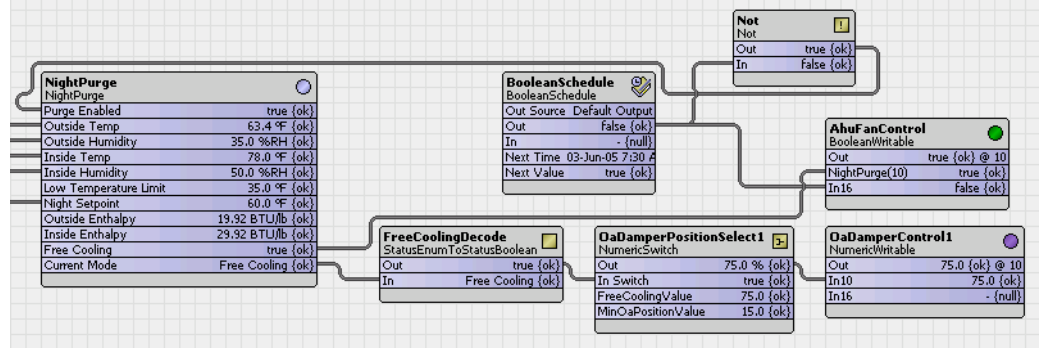
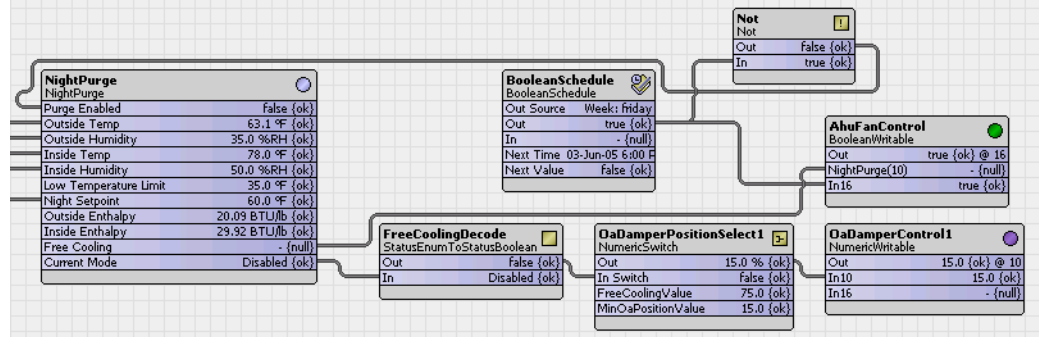


Figure 2-18 Example NightPurge application: Purge disabled



See also [Alphabetical list of kitControl components](#)

kitControl-Not

I The Not out simply inverts the Boolean logic value currently at the (single) object input. It is available in the [Logic](#) folder of the kitControl palette.

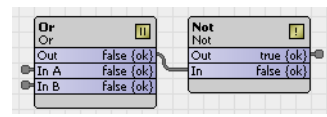
You often link Not objects with other logic objects to make different logic gates. As simple examples, [Table 2-3](#) shows NAND logic, [Table 2-4](#) shows NOR logic, and [Table 2-5](#) shows EQUIV gate logic.

See also [Alphabetical list of kitControl components](#)

Table 2-3 NAND logic using And and Not

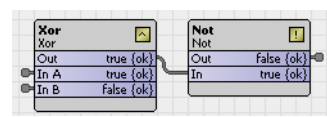
In A	In B	Out
false	false	true
false	true	true
true	false	true
true	true	false

Table 2-4 NOR logic using Or and Not



In A	In B	Out
false	false	true
false	true	false
true	false	false
true	true	false

Table 2-5 EQUIV logic using Xor and Not



In A	In B	Out
false	false	true
false	true	false
true	false	false
true	true	true

kitControl-NotEqual

NotEqual performs the operation $A \neq B$ with a boolean result. It is available in the **Logic** folder of the kitControl palette.

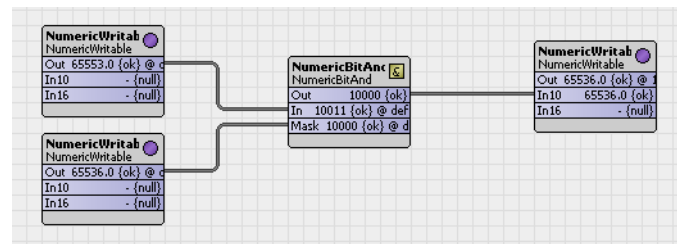
See also [Alphabetical list of kitControl components](#)

kitControl-NumericBitAnd

NumericBitAnd performs a logical AND on the bit equivalent of the StatusNumeric “In” value against the bit equivalent of its StatusNumeric “Mask” slot value. It may be useful in cases where boolean information is mapped into integer values. It is available in the **Util** folder of the kitControl palette, along with the closely-related **NumericBitOr** and **NumericBitXor**

As an example, some manufacturers multiplex binary data into a single numerical point by converting the bits from hexadecimal to decimal format. To obtain the status of the individual binary data, the number must be converted back from decimal to hex format. Each digit of the hex number represents a particular binary parameters state (0 = false, 1 = true). The NumericBitAnd object converts a StatusNumeric input to hex value and compares it against the mask value. Any digits with a value of 1 in the mask *and* the input will result in a corresponding value of 1 in the same digit of the output.

Figure 2-19 NumericBitAnd example



In the example shown in **Figure 2-19**:

- Input decimal 65553 converts to a hex value of 10011
- Mask decimal 65536 converts to a hex value of 10000
- The resulting hex value is 10000

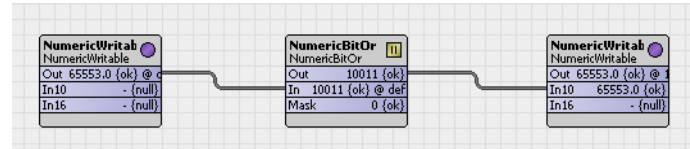
See also [Alphabetical list of kitControl components](#)

kitControl-NumericBitOr

■ NumericBitOr performs a logical OR on the bit equivalent of the StatusNumeric “In” value against the bit equivalent of its StatusNumeric “Mask” slot value. It may be useful in cases where boolean information is mapped into integer values. It is available in the **Util** folder of the kitControl palette, along with the closely-related **NumericBitAnd** and **NumericBitXor**.

As an example, some manufacturers multiplex binary data into a single numerical point by converting the bits from hexadecimal to decimal format. To obtain the status of the individual binary data, the number must be converted back from decimal to hex format. Each digit of the hex number represents a particular binary parameters state (0 = false, 1 = true). The NumericBitOr object converts a StatusNumeric input to a hex value, and compares it against the mask value. Any digits with a value of 1 in the mask *or* the input will result in a corresponding value of 1 in the same digit of the output. Any value on the output slot greater than 1 indicates that at least one of the binary parameters is true.

Figure 2-20 NumericBitOr example



In the example shown in **Figure 2-20**:

- Input decimal 65553 converts to a hex value of 10011
- Mask decimal 0 converts to a hex value of 00000
- The resulting hex value is 10011

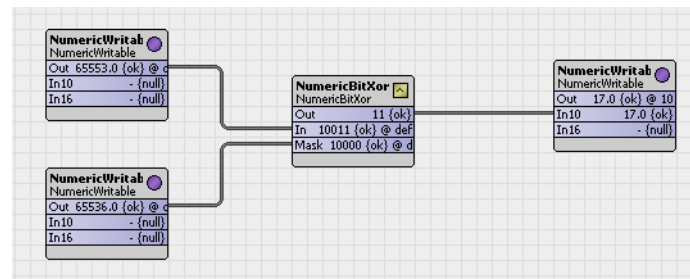
See also [Alphabetical list of kitControl components](#)

kitControl-NumericBitXor

■ NumericBitXor performs a logical XOR on the bit equivalent of the StatusNumeric “In” value against the bit equivalent of its StatusNumeric “Mask” slot value. It may be useful in cases where boolean information is mapped into integer values. It is available in the **Util** folder of the kitControl palette, along with the closely-related **NumericBitAnd** and **NumericBitOr**.

As an example, some manufacturers multiplex binary data into a single numerical point by converting the bits from hexadecimal to decimal format. To obtain the status of the individual binary data, the number must be converted back from decimal to hex format. Each digit of the hex number represents a particular binary parameters state (0 = false, 1 = true). The NumericBitXor object converts a StatusNumeric input to hex value and compares it against the mask value. Each digit is analyzed using exclusive OR (XOR) logic, setting the corresponding digit value to either a 1 or 0.

Figure 2-21 NumericBitXor example



In the example shown in **Figure 2-21**:

- Input decimal 65553 converts to a hex value of 10011
- Mask decimal 65536 converts to a hex value of 10000
- The resulting hex value is 00011

See also [Alphabetical list of kitControl components](#)

kitControl-NumericConst

○ Provides constant StatusNumeric value, with available action to Set. See “[About Constant components](#)” on page 1-4. It is available in the **Constants** folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-NumericDelay

■ The NumericDelay component provides a “soft ramp” delay from StatusNumeric In to Out. The component uses configurable values in properties Max Step Size and Update Time to provide a “stepped” output value. The combination of these two property values determines how quickly and how smoothly the current Out value changes as it approaches the In value.

The NumericDelay component is located in the [Timer](#) folder of the kitControl palette.

Types of NumericDelay component properties include the following:

- **Facets**
Use this property to set the display units, precision, min. and max. values, or other display options, as desired.
- **In**
Typically, you set this property by linking a numeric out value into it. You can manually configure the default state to a numeric value or set it to null, so that when no value is linked into this property, the default value is used. This numeric property value is passed to the component’s Out property in stages or “steps” according to the property values in the Update Time and Max Step Size properties.
- **Update Time**
This property allows you to set a value that determines how often the Max Step Value is added to the current Out value. The greater the Update Time value, the longer it takes for the Out value to match the In value.
Note: An Update Time value that is equal to or less than “0” (zero) does not allow updating. In this case, the NumericDelay component In value is set but no value is passed to the Out property.
- **Max Step Size**
This property allows you to set a number that limits the value that may be added with each “step” that occurs at Update Time. If Update Time is 1 sec., then the Max Step Size value (or a value that is less than that) may be added to the current Out value every 1 sec. until the Out value equals the In value.
- **Out**
This property displays the current output value as it approaches and equals the In property value. The numeric in this property changes at a rate defined by the Update Time and Max Step Size properties until the value equals the In property value.

See also [Alphabetical list of kitControl components](#)

kitControl-NumericLatch

● NumericLatch provides a latch for a status numeric input, and is available in the [Latches](#) folder of the kitControl palette. See “[About Latch components](#)” on page 1-8.

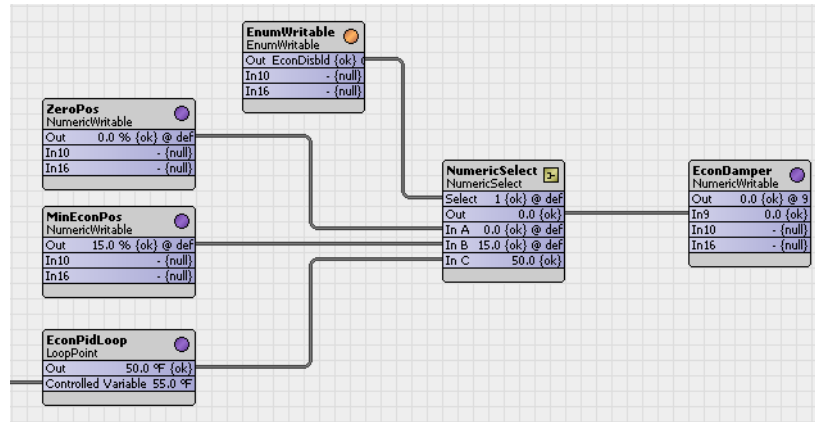
See also [Alphabetical list of kitControl components](#)

kitControl-NumericSelect

▣ NumericSelect is a numeric select, and is available in the [Selects](#) folder of the kitControl palette. See “[About Select components](#)” on page 1-13 for an overview.

See also [Alphabetical list of kitControl components](#)

[Figure 2-22](#) shows an EnumWritable linked to a NumericSelect’s select slot (where enumerated values are 1 = Econ Disabled, 2 = Min Oa Enabled, 3 = Econ Enabled). This sets the output value to one of the input values depending on the select value.

Figure 2-22 NumericSelect example application**kitControl-NumericSwitch**

- NumericSwitch selects one of two StatusNumeric inputs based upon the boolean value at the Status-Boolean input “In Switch.” The NumericSwitch is available in the **Util** folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-NumericToBitsDemux

- NumericToBitsDemux is a component that converts a numeric value into the binary equivalent. Each bit in the component represents the binary bit position of the numeric integer (numerics are truncated to whole numbers for the conversion). This component can express numeric values in bits (up to 32) as well as bytes (up to 4).

Note: This component is not designed to convert negative numbers.

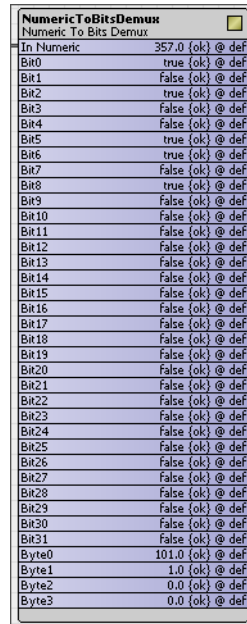
The NumericToBitsDemux component is located in the **Util** folder of the kitControl palette and has the following properties:

- **In Numeric**
This property displays the value of the numeric that is set. Typically you would link a StatusNumeric output to the In Numeric property of this component. The Status portion of the input is propagated to all of the StatusBoolean outputs and StatusNumeric (byte) outputs.
- **Bit0 through Bit31**
These 32 bits are available for representation of the converted numeric as binaries.
- **Byte0 through Byte3**
These 4 bytes are available for expressing the converted numeric input as bytes.

See also [Alphabetical list of kitControl components](#)

Figure 2-23 shows an example of the NumericToBitsDemux component with a numeric value of 357 linked to the In Numeric property. Note that Bits 0 through 8 are set to the binary representation of this number, as well as Bytes0 and 1.

Figure 2-23 Example NumericToBitsDemux component - wiresheet view



NumericToBitsDemux	
Numeric To Bits Demux	
In Numeric	357.0 [ok] @ def
Bit0	true [ok] @ def
Bit1	false [ok] @ def
Bit2	true [ok] @ def
Bit3	false [ok] @ def
Bit4	false [ok] @ def
Bit5	true [ok] @ def
Bit6	true [ok] @ def
Bit7	false [ok] @ def
Bit8	true [ok] @ def
Bit9	false [ok] @ def
Bit10	false [ok] @ def
Bit11	false [ok] @ def
Bit12	false [ok] @ def
Bit13	false [ok] @ def
Bit14	false [ok] @ def
Bit15	false [ok] @ def
Bit16	false [ok] @ def
Bit17	false [ok] @ def
Bit18	false [ok] @ def
Bit19	false [ok] @ def
Bit20	false [ok] @ def
Bit21	false [ok] @ def
Bit22	false [ok] @ def
Bit23	false [ok] @ def
Bit24	false [ok] @ def
Bit25	false [ok] @ def
Bit26	false [ok] @ def
Bit27	false [ok] @ def
Bit28	false [ok] @ def
Bit29	false [ok] @ def
Bit30	false [ok] @ def
Bit31	false [ok] @ def
Byte0	101.0 [ok] @ def
Byte1	1.0 [ok] @ def
Byte2	0.0 [ok] @ def
Byte3	0.0 [ok] @ def

kitControl-NumericUnitConverter

- NumericUnitConverter converts a StatusNumeric value from a definable “In Facets” to definable “Out Facets.” It is available in the [Conversion](#) folder of the kitControl palette.

To produce a valid numeric output, both configured facets must under the same category (such as temperature, power, and so forth). Otherwise, the NumericUnitConverter has a fault status.

See “[About Conversion components](#)” on page 1-5 for related details.

See also [Alphabetical list of kitControl components](#)

kitControl-OneShot

- The OneShot component provides a single, temporary, status boolean output for a specified duration (as set in the Time property). A OneShot action occurs with a False-to-True value transition at the In property, or with an invoked Fire action. When either of these conditions occurs, the Out property value is set to True and the Out Not property value is set to False for a time that is equal to the value of the Time property. When the time expires, these values revert to the previous (default) values.


The following types of properties are used in the OneShot component:

- **Facets**
Use this property to set the display trueText and falseText, or other display options, as desired.
- **In**
Typically, you set this property by linking a boolean Out value into it. You can manually configure the default state to a numeric value or set it to null, so that when no value is linked into this property, the default value is used. This property value is passed to the component’s Out property for the amount of time set in the Time property.
- **Time**
The value of this property determines how long the Out and Out Not properties hold their “one-shot” values. For example, a Time property value of “2” holds the Out property at True for 2 seconds when triggered and the Out Not property value at False for “2” seconds.
- **Out**
This property value displays the current value (display text) that changes with a False to True transition at the In property value or a “Fire” action. Using the Facets property, you can configure the Out value display text, as desired. After a OneShot is triggered and the Time value period expires, this value returns to the default (False) value. If a null value is set, the value does not change with a OneShot “Fire” action or False to True transition at the In property value.
- **Out Not**
This property has true, false, or null options available. The Out value change with a False to True transition at the In property value or a “Fire” action. After a OneShot is triggered and the Time value period expires, this value returns to the default (True) value. If a default null value is set, the value does not change with a OneShot “Fire” action or False to True transition at the In property value.

OneShot is in the [Timer](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-OptimizedStartStop

 The OptimizedStartStop component allows you to use Start Time Optimization and Stop Time Optimization to save energy. This component uses a space temperature input and area characteristics to calculate an optimal amount of lead-time before a scheduled event. It can analyze area temperature changes and adjust the optimization parameters based on the actual temperature change rates after an optimized start or stop.

The OptimizedStartStop component is available in the kitControl [Energy](#) folder.

The two basic optimization types are described, as follows:

- **Start time optimization**
This type of optimization reduces energy consumption by turning on equipment at the latest possible time that still allows for providing a comfortable temperature by occupancy time.
- **Stop time optimization**
This type of optimization turns equipment off at the earliest possible time that allows the building to “drift” and stay within a temperature comfort range until the end of occupancy time.

See the following sections for additional details:

- [OptimizedStartStop operation](#)
- [OptimizedStartStop properties](#)
- [Using the OSS component for optimum start](#)

See also [Alphabetical list of kitControl components](#)

OptimizedStartStop operation The [OptimizedStartStop](#) calculation is performed at 15 seconds after the beginning of every minute, when the appropriate Start Enable or Stop Enable properties are set to `true`, a valid schedule event is linked to the component, and the next scheduled event value is *not already set*.

Note: *For example, if a value is scheduled to be set to “true” in 1 hour but is already set to “true”, no calculation is performed, even if the Start Enable or Stop Enable properties are set to `true`.*

The product of this calculation is the “Calculated Command Time”. The Calculated Command Time applies to both the Start Time and the Stop Time, as appropriate. Therefore, it defines an early start command to achieve a specified temperature range by occupancy time or an early stop command without sacrificing the temperature range by unoccupancy time. After a CalculatedCommand Time is invoked, the actual area response (temperature change rate) is analyzed and weighted adjustments are made to the calculation parameters based on the detected values so that subsequent calculations might be more accurate.

Start time and stop time operations are described below:

- **Calculated Start Time**
Only one optimized start sequence is performed per day. The following factors affect the Calculated Start Time calculation.
 - **Temperature differential**
If the space temperature is outside the range defined by the lower and upper comfort limits, the difference between the space temperature and the closer limit represents the number of degrees the mechanical equipment must make up during the prestart (“optimized”) period.
 - **Run-time minutes**
The run-time heating or cooling factors (depending on the direction the space temperature must move) are multiplied by the temperature differential to determine the number of run-time minutes required to achieve the comfort limit at occupancy time, as defined by the schedule's start time.
 - **Optimum start time**
When the system's time is later than the schedule's time offset by the calculated leadtime, the optimum start outputs are enabled.
Note: *If the calculated leadtime is so large that an optimum start time prior to midnight is the result, the optimum start occurs at midnight. An optimum start is performed only for the first scheduled start for the day.*
- **Calculated Stop Time**
You can perform multiple stop operations but no optimized stop can occur before the time specified by the Earliest Stop Time property.

- **Temperature differential**
If the space temperature is inside the range defined by the lower and upper comfort limits *and* the schedule's status is active, the difference between the space temperature and one of the limits (depending on the mode) represents the number of degrees the temperature can drift between the time the mechanical equipment is stopped and the schedule's inactive event time.
- **Drift time**
The drift (lead-time) calculation is similar to the one for Start Time but using the drift-time heating and cooling factors.
- **Optimum stop time**
Optimum stop time is invoked for each of the schedule's inactive events and is based on the drift time and Next Event Time value.

OptimizedStartStop properties The [OptimizedStartStop](#) component includes the following properties:

- **Heat Cool Mode**
This boolean property allows you to enable either the `heatMode` or the `coolMode`. The selected option applies only to optimized stop calculations which means that optimized stop calculations are performed only for the selected mode. Optimized start calculations are performed for both heat and cool modes, regardless of this property value.
- **Parameter Reset Time**
This property displays the time when any of the four runtime or drifttime properties change to the User Defined values. The OSS component copies the user defined drifttime and runtime property values to the corresponding actual drifttime and runtime property values.
- **Start Enable**
This property allows you to manually or automatically enable or disable the optimized start function.
- **Stop Enable**
This property allows you to manually or automatically enable or disable the optimized stop function.
- **Schedule Status**
This boolean property monitors and displays the status of the schedule that is linked to it.
- **Next Event Time**
This property is linked to a schedule for the time of the next scheduled event.
- **Next Event Value**
This property is linked to a schedule and reflects the value of the action for next scheduled event.
- **Outside Temp**
This property is linked to outside temperature and displays the value for information only.
- **Space Temp**
This property is linked to a space temperature output and displays the temperature of the area affected by equipment associated with the OSS component.
- **Start Time Command**
This boolean property is an output that you link to a control for invoking an equipment start command. For example, it can be linked to a prioritized input of a boolean writable - or directly to the equipment Start control.
- **Stop Time Command**
This boolean property is an output that you link to a control for invoking an equipment stop command. For example, it can be linked to a prioritized input of a boolean writable - or directly to the equipment Stop control.
- **Message**
This field provides information that indicates the results of the latest start or stop command, the status of an optimized start analysis, or other possible messages. For example, the following message is displayed to indicate that an optimized stop has occurred: "Optimized stop for 14-Jun-07 5:18 PM EDT schedule time. Space temp is 75.0."
- **Upper Comfort Limit**
This property value is the Cooling mode target temperature.
- **Lower Comfort Limit**
This property value is the Heating mode target temperature.
- **Dynamic Parameter Adjust**
This controls whether or not calculation parameters are programmatically adjusted after an execution. After the OSS component completes a start or stop control, if this property value is set to `true`, the component evaluates the actual recovery rate (degrees/hour) and automatically adjusts the Runtime and Drifttime properties values so that they are influenced by actual drift time and run time.

- **Old Parameter Multiplier**

This property is used to weight the dynamic parameter adjustment calculation. The value that you specify in this field affects how much weighting you assign to the previous runtime property value when it is used in the dynamic parameter adjustment calculation. A larger value increases the amount of weighting given to the previous runtime and a smaller value decreases the weighting.
- **Earliest Start Time**

This property allows you to specify a time, before which, no optimized start command may be issued. If this value is set earlier than the Calculated Command Time, the Calculated Command Time is adjusted to equal this time.

Note: Prior to AX-3.5, this property was unavailable, with its default 12:00:10 AM value (around midnight) effectively hardcoded. Now using this property, you can enter a later earliest start time.
- **Earliest Stop Time**

This property allows you to specify a time, before which, no stop command may be issued. If this value is set earlier than the Calculated Command Time, the Calculated Command Time is adjusted to equal this time.
- **Drifttime Per degree Cooling User Defined**

This property allows you to set a default value for calculating the rate of drift in cooling mode. When you save a value to this field, the value is copied to the Drifttime Per Degree Cooling field.
- **Drifttime Per degree Heating User Defined**

This property allows you to set a default value for calculating the rate of drift in heating mode. When you save a value to this field, the value is copied to the Drifttime Per Degree Heating field.
- **Runtime Per degree Cooling User Defined**

This property allows you to set a default value for calculating the runtime value in cooling mode. When you save a value to this field, the value is copied to the Runtime Per Degree Cooling field.
- **Runtime Per degree Heating User Defined**

This property allows you to set a default value for calculating the runtime value in heating mode. When you save a value to this field, the value is copied to the Runtime Per Degree Heating field.
- **Drifttime Per degree Cooling**

This property displays the actual value that is used for calculating an optimized stop time when the equipment is in cooling mode. This value is adjusted automatically if the Dynamic Parameter Adjust value is set to `true`.
- **Drifttime Per degree Heating**

This property displays the actual value that is used for calculating an optimized stop time when the equipment is in heating mode. This value is adjusted automatically if the Dynamic Parameter Adjust value is set to `true`.
- **Runtime Per degree Cooling**

This property displays the actual value that is used for calculating an optimized start time when the equipment is in cooling mode. This value is adjusted automatically if the Dynamic Parameter Adjust value is set to `true`.
- **Runtime Per degree Heating**

This property displays the actual value that is used for calculating an optimized start time when the equipment is in heating mode. This value is adjusted automatically if the Dynamic Parameter Adjust value is set to `true`.
- **Last Start Time**

This is a record of the last Start Time that was used for calculating an optimized start time. Since only one optimized start per day is allowed, this value does not display Start Times (restarts) that are subsequent to the *initial* Start Time for a day.
- **Last Stop Time**

This is a record of the last Stop Time that was used for calculating an optimized stop time. Since multiple Optimized Stops are allowed in a day, this value changes to reflect the latest Optimized Stop time.
- **Outside Temp At Beginning**

This is a record of what the outside air temperature was at the time of the last start or stop command. This is the temperature that was used in calculations for dynamic parameter adjustment.
- **Space Temp At Beginning**

This is a record of what the space temperature was at the time of the last start or stop command. This is the temperature that was used in calculations for dynamic parameter adjustment.
- **Calculated Command Time**

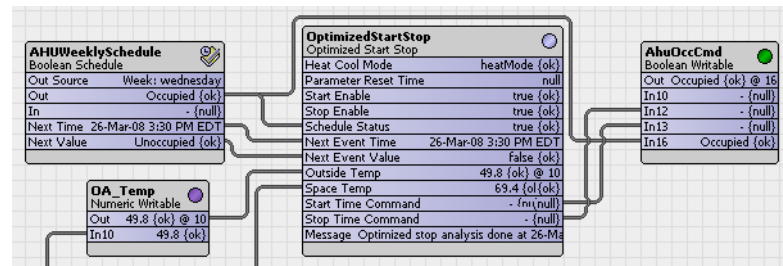
This field shows the calculated time for the next command. This could be a start or a stop command.

- **Program Mode**
As part of the logic that the OSS component uses, there are five “program mode” states. These states serve primarily in logic control, however, they may be informative to the system engineer, as well. The Program Mode value displays the current heating or cooling state for optimized start or stop. The following list describes the possible display values and meanings.
 - **0 (“No” Calculation)**
This value indicates that no calculation is being made
 - **1 (“Start” Calculation)**
This value indicates that the optimized start calculation process is ongoing but that an optimized start or stop is not yet in progress.
 - **2 (“Start” in Process)**
This value indicates that an optimized start has been initiated.
 - **3 (“Stop” Calculation)**
This value indicates that an optimized stop calculation process is ongoing but that an optimized start or stop is not yet in progress.
 - **4 (“Stop” in Process)**
This value indicates that an optimized stop has been initiated.

Example: Using the OSS component for optimum start

Figure 2-24 shows an example wiresheet view of a simple use of an OSS component.

Figure 2-24 Using the OSS component: example - part 1

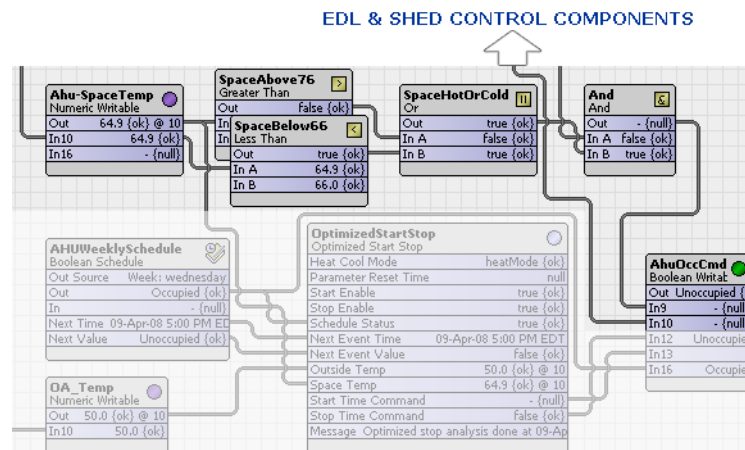


Note the following about this example:

- A weekly schedule specifies occupancy times and is linked to the OSS component for calculations. The schedule is also linked directly (bypassing the OSS component) to the in16 property (lowest priority in this case) of the occupancy control point (AhuOccCmd).
- The OSS component Start Enable and Stop Enable properties are both true, so that both Optimized starts and Optimized stops are enabled.
- The Start Time and Stop Time commands are linked to priority inputs in12 and in13, respectively of the occupancy control point (AhuOccCmd). The Start Time and Stop Time commands are true when an optimal start or stop condition is required, otherwise the outputs are set to null which relinquishes control to the next higher priority level.

Figure 2-25 shows additional logic added to the example.

Figure 2-25 Using the OSS component: example - part 2




- Additional logic is linked into the occupancy command component (AhuOccCmd) to control which

logic has priority on specifying the “AhuOccCmm” boolean point status, as follows:

- in9 temperature control overrides a demand limiting link from an EDL component to in10. This prevents a load shed if the configurable comfort range is exceeded.
- in10 (demand limiting link from EDL component) overrides an OSS Stop link into in12
- in12 (OSS component Stop link) overrides a Start link into in13 (as described above)
- in13 (optimal start) overrides the schedule link to in16 (lowest priority) (as described above)

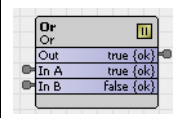
For related information, refer to “[kitControl-ElectricalDemandLimit](#)” on page 2-12.

kitControl-Or

 Or performs a logical OR on all valid inputs and writes the boolean result to the out property. The Or is available in the [Logic](#) folder of the kitControl palette. [Table 2-6](#) shows the Or object truth table when using two inputs. [Table 2-7](#) shows the Or object truth table when using all four inputs. NOR gate logic is accomplished by linking to a [Not](#) object.

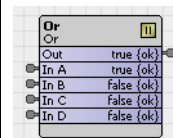
See also [Alphabetical list of kitControl components](#)

Table 2-6 Or object truth table (2 inputs)



In A	In B	Out
false	false	false
false	true	true
true	false	true
true	true	true

Table 2-7 Or object truth table (4 inputs)



In A	In B	In C	In D	Out
false	false	false	false	false
false	false	false	true	true
false	false	true	false	true
false	false	true	true	true
false	true	false	false	true
false	true	false	true	true
false	true	true	false	true
false	true	true	true	true
true	false	false	false	true
true	false	false	true	true
true	false	true	false	true
true	false	true	true	true
true	true	false	false	true
true	true	false	true	true
true	true	true	false	true
true	true	true	true	true

kitControl-OutsideAirOptimization

OutsideAirOptimization is available in the kitControl Energy folder. The OutsideAirOptimization component is used to support applications that need to allow for enthalpy based free cooling. This object is typically used during occupancy periods.

The freeCooling output is set to false if $outside \geq inside$ and set to true if $outside \leq inside - (abs) \text{ thresholdSpan}$. You can select temperature or enthalpy comparisons. There is also a low temperature check to protect against freezing.

Setup of the object involves the following properties (also see Using OutsideAirOptimization), as follows:

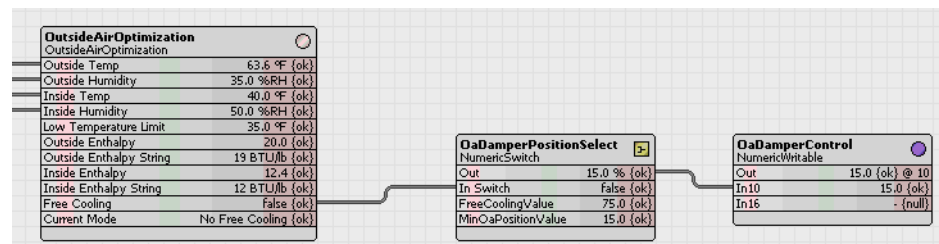
- **Temperature Facets**
 This is used to set the units and number precision of the Outside Temp, Inside Temp, and Low Temperature Limit properties.
- **Humidity Facets**
 This is used to set the units and number precision of the Outside Humidity and Inside Humidity properties.
- **Outside Temp**
 Input for the current outside air temperature. This input must be valid for this object to function.

- **Outside Humidity**
Input for the current outside air humidity. This input must be valid for this object to function.
- **Inside Temp**
Input for the current inside air temperature. This input must be valid for this object to function.
- **Inside Humidity**
Input for the current inside air humidity. This input must be valid for this object to function.
- **Low Temperature Limit**
This property is used to provide freeze protection.
- **Outside Enthalpy**
This is the calculated outside air enthalpy.
- **Outside Enthalpy String**
This provides the outside enthalpy value as a string or possible status/error message.
- **Inside Enthalpy**
This is the calculated inside air enthalpy.
- **Inside Enthalpy String**
This provides the inside enthalpy value as a string or possible status/error message.
- **Free Cooling**
This boolean output value is set to the value of the Free Cooling Command when it is determined that free cooling should be used. Otherwise, the value is set to null.
- **Current Mode**
This indicates what mode this object is currently in.
 - Input out of range
 - Free Cooling
 - No Free Cooling
 - Low temperature
 - Input error
- **Threshold Span**
The difference between the inside enthalpy and the outside enthalpy must be greater than this value before free cooling will be enabled.
- **Use Enthalpy**
Setting this property to true will enable the use of enthalpy for determining if free cooling is available. Otherwise, it will just use outside and inside temperature to decide.
- **Free Cooling Command**
If it is determined that free cooling is available, this is the boolean value that will be set in the Free Cooling property.
- **Use Null Output**
If this property is true, then the null flag will also be set on the Free Cooling output when free cooling is NOT available.

Example: Using OutsideAirOptimization

An example OutsideAirOptimization component usage is shown in [Figure 2-26](#).

Figure 2-26 Example OutsideAirOptimization



See also [Alphabetical list of kitControl components](#)

kitControl-Power

Power performs the operation $out = (inA \wedge inB)$ or a raised to the b power. The Power component is available in the [Math](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-Psychrometric

The Psychrometric component is available in the kitControl [Energy](#) folder. You can use it to support applications that need to calculate the properties of moist air using given temperature and humidity inputs.

The following sections provide more Psychrometric details:

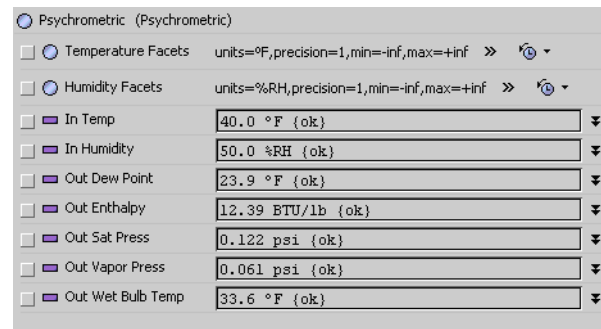
See also [Alphabetical list of kitControl components](#)

Setup of the component involves setting the following properties:


- **Temperature Facets**
Used to set the units and number precision of the Temp In, Min Temp, Max Temp, and Mean Temp properties.
- **Humidity Facets**
used to set the units and number precision of the humidity properties. Currently, only English units are supported.
- **In Temp**
Input temperature
- **In Humidity**
Input humidity
- **Out Dew Point**
Calculated dew point temperature. Requires valid In Temp and In Humidity to calculate.
- **Out Enthalpy**
Calculated enthalpy. Requires valid In Temp and In Humidity to calculate.
- **Out Sat Pressure**
Calculated saturated pressure. Requires valid In Temp to calculate.
- **Out Vapor Pressure**
Calculated vapor pressure. Requires valid In Temp and In Humidity to calculate.
- **Out Wet Bulb Temp**
Calculated wet bulb temperature. Requires valid In Temp and In Humidity to calculate.

A Psychrometric example property sheet is shown in [Figure 2-27](#).

Figure 2-27 Psychrometric example property sheet




kitControl-Ramp

 Ramp provides a StatusNumeric Out with a linear ramping output. Slots define the Period, Amplitude, Offset, and Update Interval. It is available in the kitControl palette's Util folder.

See also [Alphabetical list of kitControl components](#)

kitControl-Random

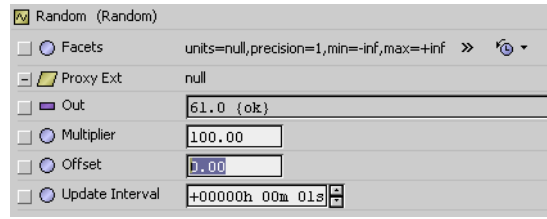
 This component can be used to generate random numbers. The output is derived by multiplying a random number (that is greater than 0 but less than 1) times a variable "multiplier" plus an offset. It is available in the kitControl palette's Util folder. See the next section, [Random setup](#).

See also [Alphabetical list of kitControl components](#)

Random setup Setup of the Random component involves setting the following properties:

- **multiplier:**
this is a double value that is used to multiply by the random number (the random number is ≥ 0.0 but < 1.0). The multiplier is set to 1.0 by default.
- **Offset**
This is the positive or negative distance from zero that the wave's amplitude is centered on. The default offset value is 50.
- **Update Interval**
This is the amount of time between output changes. The default value is set to 01 seconds.

Figure 2-28 Example Random component property sheet



kitControl-Reset

This component performs a linear “reset” on the inA value. Reset is available in the **Math** folder of the kitControl palette. Reset operation is defined by the following four slots:

- Input Low Limit — must be *less* than the Input High Limit
- Input High Limit — must be *greater* than the Input Low Limit
- Output Low Limit — may (or may not) be greater than the Output High Limit
- Output High Limit — may (or may not) be greater than the Output Low Limit

For example, a Reset object is used to establish a hot water control setpoint, based on the outside air temperature at inA. When the outside air temperature is 0°F, the hot water setpoint is 200°F. When the outside air temperature is 75°F, the hot water setpoint is 100°F. The Reset object is configured as:

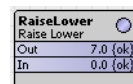
```
Input Low Limit = 0.0
Input High Limit = 75.0
Output Low Limit = 200.0
Output High Limit = 100.0
```

Whenever the inA value is beyond the input limits, the output is limited by the corresponding output limit (in this case, 200 at 0°F or below, 100 at 75°F or above). When the input is at an intermediate value, the output scales linearly. For example, when the outside air temperature is at 38.2°F, the Reset output is 149.1°F.

See also [Alphabetical list of kitControl components](#)

kitControl-RaiseLower

The RaiseLower object provides a staged analog output designed to be used with a third party 2-relay hardware device but also provides for operation of two digital outputs from normal IO hardware such as NDIO as an alternative control method. The actuator should be able to sustain an overdrive at each boundary (for example, a clutch mechanism) as the Raise lower object does not have proportional feedback or limit switch features. The RaiseLower component is available in the **HVAC** folder of the kitControl palette.



A typical application for the RaiseLower component is to control a reversible actuator in order to drive a coupled valve or damper either open or closed. The external hardware device consists of two on-board relays with volt free contacts that switch to provide power to either the “open” command of the actuator or the “close” command of the actuator. Either relay is activated for a proportion of the full scale drive time of the actuator (drive time is pre-defined by the manufacturer). The acceptable input to this device is 0 to 10 volts, with staged control at 0v, 4v, 7v and 10v as detailed in [Table 2-8](#).

Table 2-8 RaiseLower staged control voltage

Volts	Raise	Lower	Function
0	Off	Off	Off
4	Off	On	Lower
7	Off	Off	Static
10	On	Off	Raise

The Raise Lower object has the following properties:

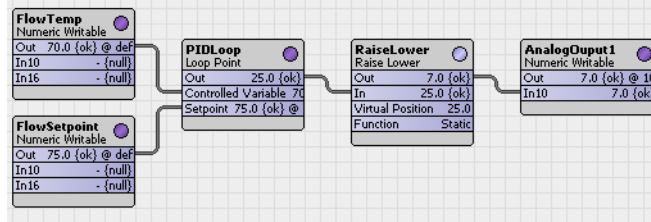
- **Propagate Flags**
By default, this object maintains independent status flags from input-linked points. It is therefore possible to specify the “out” status to propagate from the input status. The Propagate Flags property specifies which status flags will propagate from the “In” property to the “Out” status flags. The object

Propagate Flags property allows for selection of any combination of the following status types for propagation:

- disabled
- fault
- down
- alarm
- overridden
- **Out**
This is a status numeric value and is the analog output value from the object. This output has valid voltage outputs of 0v, 4v, 7v and 10v as illustrated in [Table 2-8](#), depending on the function determined by the object.
- **In**
This property is a status numeric value typically connected to a modulated output of a Control object such as PID Loop. It has an input range of 0 to 100.
- **Virtual Position**
This is a slot that holds the value representing the virtual position of the actuator, as calculated by the RaiseLower object.
- **Raise**
This is a status boolean value which is set to `true` if the object determines that the output should command a “Raise” output. A Raise output is maintained for a period of time determined by the positive differential of “Virtual position” subtracted from the “In” slot, and is a relative proportion of the full scale drive time. In addition, if the “virtual position” of the object is calculated to be 100% (fully raised), then twice the drive time is asserted in order to synchronize the physical actuator with the calculated virtual position to compensate for realtime drift.
- **Lower**
This is a status boolean value which is set to `true` if the object determines that the output should command a “Lower” output. A Lower output is maintained for a period of time determined by the negative differential of “Virtual position” subtracted from the “In” slot and is a relative proportion of the full scale drive time. In addition, if the “virtual position” of the object is calculated to be 0% (Fully Lowered) then twice the drive time is asserted in order to synchronize the physical actuator with the calculated virtual position to compensate for realtime drift.
- **Function**
This property value displays operational information corresponding to the current activity of the object. Valid status values include: Off, Lower, Static, Raise.
- **Dead Band**
This should be set to a value that corresponds to a percentage of full scale drive time. The “In” value would have to exceed the dead band value before the “out” commands to “Raise” or “Lower”. The default value is 0.25, Range is 0 to 5.
- **Drive Time**
This should be set to a value that corresponds to the full scale drive time provided by the manufacturer.
- **Midnight Reset Enabled**
This should be set to `false` to inhibit a reset. A midnight reset invokes a synchronization cycle at midnight in order to compensate for realtime drift that may accumulate during normal operation of the actuator. The reset cycle will override an input signal for a period of twice the full scale drive time.
- **Reset action**
This action can be used to re synchronize the actuator through 0% (Fully Lowered) temporarily overriding an input signal for a period of twice the full scale drive time

Examples Refer to [Figure 2-29](#) for this example. An actuator having a 100 second full scale drive time and initialized values of the actuator and virtual position are 0% (synchronized at fully lowered). Should the input signal increase to 40% the “Raise” output turns on for 40% of the full scale drive time (40 sec). If a subsequent input is decreased to 15% the “Lower” output is active for 25% of full scale drive time (25 sec) moving the actuator to 15% open.

Figure 2-29 Example using the RaiseLower component



kitControl-SequenceBinary

■ The SequenceBinary component provides sequenced *weighted* “staging” control of from 2 to 10 BooleanWritables based upon the status numeric In value (0–100). An adjustable delay time is also provided. It can be used to support applications that need to sequence 2 to 10 loads or stages in a binary sequence. Binary sequencing provides an analog to binary converter function that selects the outputs whose total load rating relates directly to the control need. For each successive output, the output rating is twice the previous output.

A similar object is the [SequenceLinear](#), which uses a rotating method (vs. weighted) for sequencing. SequenceBinary is available in the **HVAC** folder of the kitControl palette.

[Table 2-9](#) illustrates how, by controlling 3 loads, eight unique levels of control can be achieved:

Table 2-9 Example SequenceBinary component

Control Signal (In) %	OutC (4kw load size)	OutB (2kw load size)	OutA (1kw load size)	Stage Hysteresis
100	On	On	On	14.3
85.7	On	On	Off	14.3
71.4	On	Off	On	14.3
57.1	On	Off	Off	14.3
42.9	Off	On	On	14.3
28.6	Off	On	Off	14.3
14.3	Off	Off	On	14.3
0	Off	Off	Off	14.3

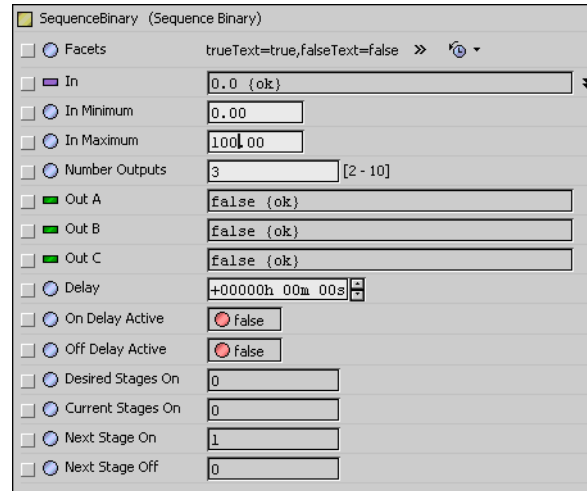
Setup of the [SequenceBinary](#) object involves the following properties:

- **Facets**
Used to set the active and inactive text to be used for the Out properties.
- **In**
Input property that is used to determine the number of stages that should currently be On.
- **In Minimum**
Value of the input that produces all outputs off.
- **In Maximum**
Value of the input that produces all outputs on.
- **Number Outputs**
This object can be configured to support 2 to 10 outputs or stages.
- **OutA - OutJ**
These are status boolean values that can be used to control 2 to 10 loads. The number of outputs used is defined by the Number Outputs property.
- **Delay**
Defines the amount of time, in seconds, that must pass between changes in outputs. The default time is 0 seconds.
- **On Delay Active**
Boolean read-only property that indicates that the on delay timer is active.
- **Off Delay Active**
Boolean read-only property that indicates that the off delay timer is active.
- **Desired Stages On**
Read-only property that indicates the calculated number of stages that should be on based on the In property.

- **Current Stages On**
Read-only property that indicates the current number of stages that are currently on. Normally the Current Stages On and the Desired Stages On will be the same. They will be different when going through a transition and the delay timer is active.
- **Next Stage On**
Read-only property that indicates the next stage that will be turned on if needed.
- **Next Stage Off**
Read-only property that indicates the next stage that will be turned off if needed.

An example of a [SequenceBinary](#) property sheet is shown in [Figure 2-30](#).

Figure 2-30 Example SequenceBinary property sheet



See also [Alphabetical list of kitControl components](#)

kitControl-SequenceLinear

SequenceLinear provides sequenced *rotating* “staging” control of from 2 to 10 BooleanWritables based upon the status numeric In value (0–100). An adjustable delay time is also provided. A similar object is the [SequenceBinary](#), which uses a weighted method (vs. rotating) for sequencing.

The SequenceLinear component can be used to support applications that need to sequence 2 to 10 loads or stages in a linear or rotating sequence. With linear sequencing the first stage on will be the last stage off. With rotating sequencing the first stage on will be the first stage off. The In property, which is a StatusNumeric, is used to control the number of stages that should be on. The input range is defined by the InMinimum and InMaximum properties. SequenceLinear is available in the [HVAC](#) folder of the kitControl palette.

On and Off setpoints are calculated for each stage by the following [Table 2-10](#) formulas (this assumes there are 5 outputs defined):

Table 2-10 SequenceLinear On / Off calculation formulas

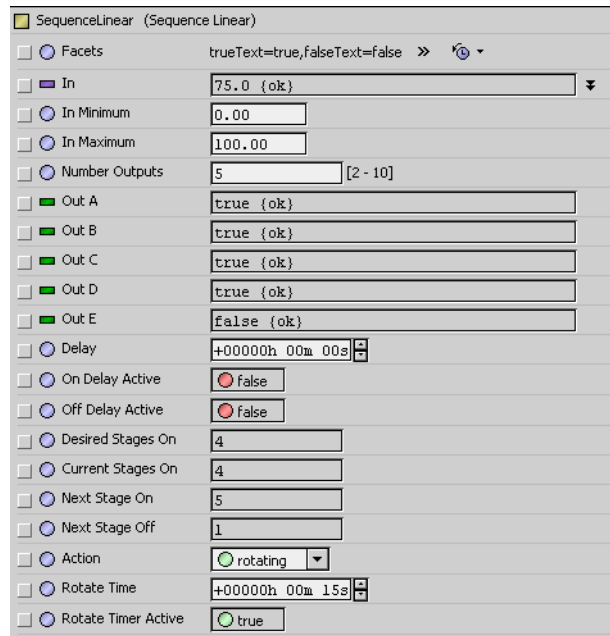
	Linear	Rotating
range = InMaximum - InMinimum	100 = 100 - 0	100 = 100 - 0
delta = range / NumberOutputs	20 = 100 / 5	20 = 100 / 5
OnSetpointA = 1 * delta	20	20
OnSetpointB = 2 * delta	40	40
OnSetpointC = 3 * delta	60	60
OnSetpointD = 4 * delta	80	80
OnSetpointE = 5 * delta	100	100
OffSetpointA = 0 * delta, 4 * delta	0	80
OffSetpointB = 1 * delta, 3 * delta	20	60
OffSetpointC = 2 * delta, 2 * delta	40	40
OffSetpointD = 3 * delta, 1 * delta	60	20
OffSetpointE = 4 * delta, 0 * delta	80	0

Setup of the [SequenceLinear](#) object involves configuring the following properties:

- **Facets**
Used to set the active and inactive text to be used for the Out properties.
- **In**
Input property that is used to determine the number of stages that should currently be On.
- **In Minimum**
Value of the input that produces all outputs off.
- **In Humidity**
Value of the input that produces all outputs on.
- **Number Outputs**
This object can be configured to support 2 to 10 outputs or stages.
- **OutA - OutJ**
These are status boolean values that can be used to control 2 to 10 loads. The number of outputs used is defined by the Number Outputs property.
- **Delay**
Defines the amount of time the must pass between changes in outputs.
- **On Delay Active**
Boolean read-only property that indicates that the on delay timer is active.
- **Off Delay Active**
Boolean read-only property that indicates that the off delay timer is active.
- **Desired Stages On**
Read-only property that indicates the calculated number of stages that should be on based on the In property.
- **Current Stages On**
Read-only property that indicates the current number of stages that are currently on. Normally the Current Stages On and the Desired Stages On will be the same. They will be different when going through a transition and the delay timer is active.
- **Next Stage On**
Read-only property that indicates the next stage that will be turned on if needed. This is primarily used when the Action is selected to be Rotating.
- **Next Stage Off**
Read-only property that indicates the next stage that will be turned off if needed. This is primarily used when the Action is selected to be Rotating.
- **Action**
This configuration property selects between Linear and Rotating action. With Linear action, Out A (Stage 1) will always be the first stage to turn on and the last stage to turn off. With Rotating action, the first stage to turn on will increment to the next stage each time the current stages on goes to 0.
- **Rotate Time**
This configuration property specifies the amount of time that the outputs will remain in a fixed configuration before the outputs are shifted to the next configuration.
- **Rotate Timer Active**
Read-only property that indicates that the rotate timer is active.

An example of a `SequenceLinear` property sheet is shown in [Figure 2-31](#).

Figure 2-31 *SequenceLinear example property sheet*



See also [Alphabetical list of kitControl components](#)

kitControl-SetpointLoadShed

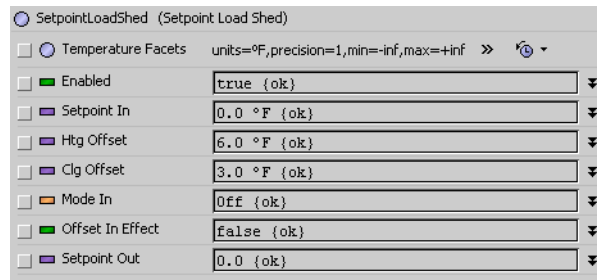
● This component provides a convenient way to implement load shedding strategies. It causes a specified setpoint to be raised or lowered by a specific amount in response to an input link, that enables the SetpointLoadShed component. Generally, this component solves the problem often found in temperature control sequences where shutting down an output directly may be complicated by other control dependencies. By changing the setpoint based on a shed request, the output is shut down under the direction of the control sequence and the interdependencies are maintained.

Setup of the `SetpointLoadShed` involves the following slots, as follows:

- **Enabled**
True value in this property causes the setpoint to be adjusted by the offset value.
- **Setpoint In**
This is the normal setpoint value that is be adjusted by the SetpointLoadShed object, if needed.
- **Htg Offset**
The setpoint is adjusted by this signed (+ or -) amount if active and in heating mode.
- **Clg Offset**
The setpoint is adjusted by this signed (+ or -) amount if active and in cooling mode
- **Mode In**
This StatusEnum type property has three possible modes: Off, Heat, Cool. The modes are selectable from the options list in the property sheet.
- **Offset In Effect**
This is an output to indicate whether or not the Setpoint Out property value has been adjusted.
- **Setpoint Out**
This property value is the adjusted setpoint value, if active, otherwise it passes through original setpoint.

A example `SetpointLoadShed` object property sheet is shown in [Figure 2-32](#).

Figure 2-32 SetpointLoadShed example property sheet



See also, Figure 2-9 for an example wiresheet view using this component.

See also [Alphabetical list of kitControl components](#)

kitControl-SetpointOffset

SetpointOffset provides setpoint control for electrical demand limiting applications, for use with the [ElectricalDemandLimit](#) and [ShedControl](#) objects. SetpointOffset is available in the [Energy](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-ShedControl

ShedControl receives inputs from a primary (network) EDL source and a local (secondary) EDL source (separate [ElectricalDemandLimit](#) objects) that specify the number of load levels that should be shed. The Secondary Shed Level is used as a backup whenever the Primary Shed Level is not available. ShedControl has StatusBoolean outputs for up to 16 contiguous levels, as specified in the Number Levels property. A Status slot provides an output message to indicate this component’s state in reference to the overall demand limiting control scheme. Execution of this component can be enabled or disabled by setting the Shed Enable property.

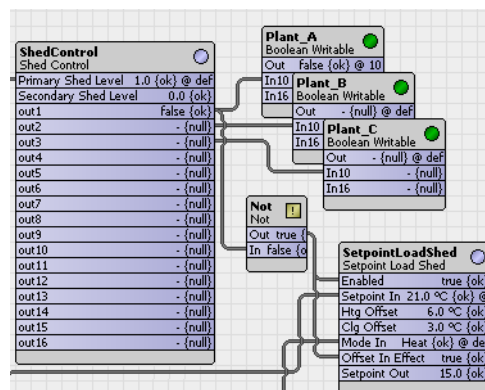
ShedControl is available in the [Energy](#) folder of the kitControl palette, along with related objects.

Following is a description of the ShedControl component properties:

- **Primary Shed Level**
This is an input that allows you to link in a Shed Level property value from an EDL component. Typically this would be a component on the network.
- **Secondary Shed Level**
This is an input that allows you to link in a Shed Level property value from a secondary (or “backup”) EDL component. Typically this would be an EDL component with a locally available connection. The Secondary Shed level is used only if the Primary Shed Level property is not available.
- **out(1-16)**
These 16 properties have binary status values that reflect the current active Shed Level. For example, a Shed Level of 3 (as indicated by the Primary Shed Level, or Secondary Shed Level when Primary is not available) sets the first three out properties (out1, out2, out3) to false. This false value may be used to turn off power by linking to an appropriate control. When a “restore” changes the Shed Level to 2, the out3 property returns to null, relinquishing control to the next (out2) priority level.

Figure 2-33 shows an example wiresheet view of a ShedControl component in use.

Figure 2-33 Wiresheet view of Shed Control component




Note the following about the example wiresheet view:

- The Primary Shed Level (linked from an EDL component) is “1” (no Secondary Shed Level is used).
- out1, 2, and 3 are linked to boolean controls that are set to turn off power to Plant_A, B, and C, respectively, if required.
- The current Shed Level of “1” sets out1 to `false` and sets the “Plant_A” boolean control status to False. This should turn off power to any power consuming devices that are linked to this object.
- The out1 value is also linked to the SetpointLoadShed component, which adjusts a setpoint to reduce the electrical demand.
- If more shed levels are needed, out2 and out3 will be set to `false`. If shed levels are no longer needed (a “restoration” is invoked) then out3, out2, and out1 are restored (to `null`), in that order, as allowed by the current Shed Level.


See also [Alphabetical list of kitControl components](#)

kitControl-Sine

 Sine performs the operation `out = sin(inA)`. The Sine is in the [Math](#) folder of the kitControl palette.


See also [Alphabetical list of kitControl components](#)

kitControl-SineWave

 SineWave generates a sine wave as a StatusNumeric out. The SineWave is available in the [Util](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-SlidingWindowDemandCalc

 This component is available in the [Energy](#) folder of the kitControl palette. It simulates a demand meter (see, “[About demand meters](#)”) and calculates the sliding window demand for 5, 15, and 30-minute demand intervals based on an accumulative pulse counter input. It also calculates the total kWh since last reset and the hourly and daily kWh values.

The hourly and daily kWh values can then be logged using a standard history extension setup to execute on-trigger only. The SlidingWindowDemandCalc object fires the hourly and daily triggers to align the kWh data correctly to actual clock values. The pulse input into the object is assumed to be accumulative (not delta pulses) that roll over after reaching the 16 bit limit (65535).

The SlidingWindowDemandCalc object can be configured to reset all the calculated accumulative values on a preset interval such as at “noon on the first Sunday, every month”.

For more information about demand meters, see “[About demand meters](#)”.

See also [Alphabetical list of kitControl components](#)

About demand meters A demand meter measures peak demand using electromechanical components, and provides a pulsed contact output in proportion to the rate of electrical consumption over a interval such as 15 or 30 minutes. The utility meter will record the highest average interval rate (kW), which will be billed as the “Demand Charge.” At the end of the billing cycle, peak demand will be read and the demand will be reset to zero for the start of a new billing cycle. The total consumption (kWh) is also totaled by the meter to determine the “Usage Charge.”

The [SlidingWindowDemandCalc](#) object can simulate the demand meter by calculating the average value over an interval. On a normal update frequency, the kW data from the oldest sample is replaced by the kW data from the most recent sample. This constant updating of the kW information every scan is called a “Sliding Window Demand Value.”

The highest “Sliding Window” demand reading may be higher than the utility demand since the calculation updates average demand every 2 seconds and the utility meter may be resetting on a fixed or discrete 15 or 30 minute interval.

Setup of the [SlidingWindowDemandCalc](#) object involves the following properties:

- **Consumption Facets**

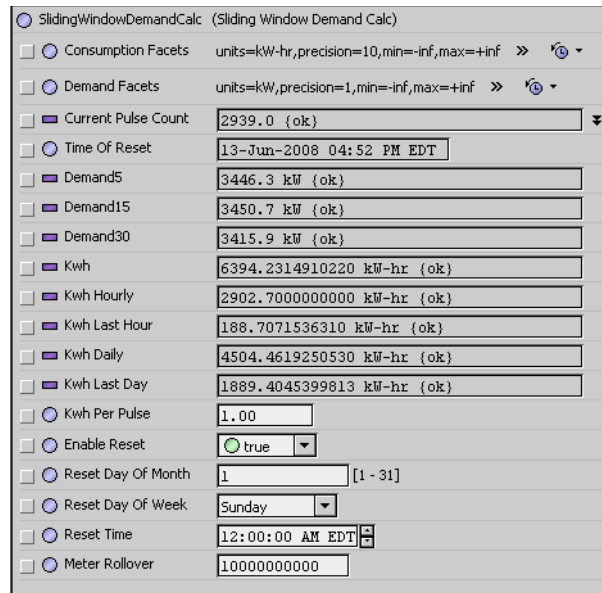
This property allows you to set the facets for the consumption output property:

- **units**
Select the desired units from the drop-down option list. Default units are `energy` and `kilowatt hour (kW-hr)`.
- **precision**
Type in an integer to set the precision level for your data (how many decimal places to display).
- **min**
Type in a value for the smallest allowable value of the output property (default is `-inf`)

- **max**
Type in a value for the largest allowable value of the output property (default is +inf)
- **Demand Facets**
This property allows you to set the facets for the demand output property:
 - **units**
Select the desired units from the drop-down option list. Default units are `power` and `kilo-watts (kW)`
 - **precision**
Type in an integer to set the precision level for your data (how many decimal places to display).
 - **min**
Type in a value for the smallest allowable value of the output property (default is -inf)
 - **max**
Type in a value for the largest allowable value of the output property (default is +inf)
- **Current Pulse Count**
Displays data from a link to a pulse counter input object indicating the running total of pulses.
- **Time of Reset**
Displays the date and time of the last reset.
- **Demand 5**
Displays the demand (kW) for a five minute window.
- **Demand 15**
Displays the demand (kW) for a fifteen minute window.
- **Demand 30**
Displays the demand for a 30 minute period.
- **Kwh**
Displays the running kWh (consumption) value since the last reset.
- **Kwh Hourly**
Displays the running value since the last hourly reset.
- **Kwh Last Hour**
Displays the kWh (consumption) value for the last hour.
- **Kwh Daily**
Displays the kWh (consumption) value since the last daily reset.
- **Kwh Last Day**
Displays the kWh (consumption) value for the last day.
- **Kwh Per Pulse**
This field allows you to set the value per pulse. It is usually noted on the meter or provided by the power company. It is how much energy each pulse represents.
- **Enable Reset**
A true value in this field allows recurring automatic resets to happen at a frequency based on the following properties.
- **Reset Day of Month**
Allows you to set the day of month for recurring automatic reset (if enabled) to occur.
- **Reset Day of Week**
Allows you to set the day of the week for recurring automatic reset (if enabled) to occur.
- **Reset Time**
Allows you to set the time of day for recurring automatic reset (if enabled) to occur.
- **Meter Rollover**
Specifies the maximum value the meter provides before it rolls over to zero (0). The default value is 65535, the data type is a long (up to a very large number, 9223372036854775807).

An example [SlidingWindowDemandCalc](#) property sheet is shown in [Figure 2-34](#):

Figure 2-34 Example SlidingWindowDemandCalc property sheet



kitControl-SquareRoot

- SquareRoot performs the operation $out = \sqrt{inA}$ (square root of inA). The SquareRoot is available in the **Math** folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-StatusBooleanToBoolean

- StatusBooleanToBoolean converts a StatusBoolean value to Boolean. See “[Status value to simple value](#)” on page 1-5. StatusBooleanToBoolean is available in the **Conversion** folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-StatusDemux

- StatusDemux provides a method to check for individual status flags of the In-linked object, and sets corresponding (demuxed) StatusBoolean out slots active (true) if that status is found. StatusDemux is available in the **Util** folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-StatusEnumToEnum

- StatusEnumToEnum converts a StatusEnum value to Enum. See “[Status value to simple value](#)” on page 1-5. StatusEnumToEnum is available in the **Conversion** folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-StatusEnumToInt

- StatusEnumToInt converts a StatusEnum value to an Integer. See “[Status value to simple value](#)” on page 1-5. StatusEnumToEnum is available in the **Conversion** folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-StatusEnumToStatusBoolean

- StatusEnumToStatusBoolean converts a StatusEnum value to a StatusBoolean value. See “[Status value to status value](#)” on page 1-7. It is available in the **Conversion** folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-StatusEnumToStatusNumeric

- StatusEnumToEnum converts a StatusEnum value to a StatusNumeric value. See “[Status value to status value](#)” on page 1-7. It is available in the **Conversion** folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-StatusNumericToDouble

- StatusNumericToDouble converts a StatusNumeric value to a Double value. See “[Status value to simple value](#)” on page 1-5. It is available in the **Conversion** folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-StatusNumericToFloat

- StatusNumericToFloat converts a StatusNumeric value to a Float value. See “[Status value to simple value](#)” on page 1-5. It is available in the [Conversion](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-StatusNumericToInt

- StatusNumericToInt converts a StatusNumeric value to an Int (integer). See “[Status value to simple value](#)” on page 1-5. It is available in the [Conversion](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-StatusNumericToStatusEnum

- StatusNumericToStatusEnum converts a StatusNumeric value to a StatusEnum value. See “[Status value to status value](#)” on page 1-7. It is available in the [Conversion](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-StatusNumericToStatusString

- StatusNumericToStatusString converts a StatusNumeric value to a StatusString value. It is available in the [Conversion](#) folder of the kitControl palette.

Object properties specify the number of integer (whole number) digits and decimal digits to use in the output string, with a default value of 6 each. Unused digits are padded with a zero. For example, with an input numeric value of 72.8, the default output string would be “000072.800000”.

See “[Status value to status value](#)” on page 1-7 for other status-to-status conversion types. See “[About String components](#)” on page 1-13 for related string operations.

See also [Alphabetical list of kitControl components](#)

kitControl-StatusStringToStatusNumeric

- StatusStringToStatusNumeric converts a StatusString value to an StatusNumeric value. It is available in the [Conversion](#) folder of the kitControl palette.

The input string must contain only numeral characters, and optionally one period (“.”) for decimal notation. Valid input strings examples are “145678” or “34.81”. Leading and/or trailing space characters are allowed and ignored. Any other input string characters (e.g. alpha, punctuation) result in a fault status.

See “[Status value to status value](#)” on page 1-7 for other status-to-status conversion types. See “[About String components](#)” on page 1-13 for related string operations.

See also [Alphabetical list of kitControl components](#)

kitControl-StringConcat

- StringConcat simply concatenates (joins) up to 4 StatusString values present on inputs InA, InB, InC, and InD and outputs the concatenated string. String output order is A + B + C + D.

The StringConcat is available in the [String](#) folder of the kitControl palette. See “[About String components](#)” on page 1-13.

See also [Alphabetical list of kitControl components](#)

kitControl-StringConst

- Provides constant StatusString value, with available action to Set. It is available in the [Constants](#) folder of the kitControl palette. See “[About Constant components](#)” on page 1-4.

See also [Alphabetical list of kitControl components](#)

kitControl-StringIndexOf

- StringIndexOf compares two StatusString inputs (InA, InB), looking for an exact match of string InB within string InA. The object outputs a StatusBoolean result (true, false), as well as “begin” and “after” index character positions. Three properties are integer types: FromIndex, BeginIndex, and AfterIndex. The object works as follows:

inputs: InA, InB, FromIndex

outputs: Out, BeginIndex, AfterIndex

The object tests an exact match of string InB within string InA starting at character position FromIndex within string InA.

- If a match *does* occur:
Out is set to true. BeginIndex is set to the character position within string InA where the match starts, and AfterIndex is set to the character position just after the match ends.
- If a match *does not* occur:
Out is set to false. BeginIndex and AfterIndex are both set to -1.

The StringIndexOf is available in the [String](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-StringLatch

- StringLatch provides a latch for a StatusString input, and is available in the [Latches](#) folder of the kitControl palette. See “[About Latch components](#)” on page 1-8.

See also [Alphabetical list of kitControl components](#)

kitControl-StringLen

- StringLen simply outputs the total number of non-null characters in the In StatusString value. Out value is StatusNumeric from 0 to n . The StringLen is available in the [String](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-StringSelect

- StringSelect is a string select. The StringSelect is available in the [String](#) folder of the kitControl palette. See “[About Select components](#)” on page 1-13 for an overview.

See also [Alphabetical list of kitControl components](#)

kitControl-StringSubstring

- StringSubstring outputs a portion of the “In” slot StatusString value, as specified by integer properties Begin Index and End Index. By default, Begin Index=0 and End Index=-1, which means the entire In string is passed.

The StringSubstring is available in the [String](#) folder of the kitControl palette. See “[About String components](#)” on page 1-13.

See also [Alphabetical list of kitControl components](#)

kitControl-StringTest

- StringTest compares 2 StatusString values (InA and InB) and outputs a StatusBoolean Out (true/false) result. Test Selection property choices are:
 - aEqualsB
 - aEqualsBIgnoreCase
 - aStartsWithB
 - aEndsWithB
 - aContainsB

The StringLen is available in the [String](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-StringToStatusString

- StringToStatusString converts the string In value to a StatusString Out value. See “[Simple value to status value](#)” on page 1-7. StringToStatusString is available in the [Conversion](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-StringTrim

- StringTrim “trims whitespace” from both ends of the StatusString In value. Whitespace is typically leading/trailing space characters, but may also include control characters. The StringTrim is available in the [String](#) folder of the kitControl palette. See “[About String components](#)” on page 1-13.

See also [Alphabetical list of kitControl components](#)

kitControl-Subtract

- Subtract performs the operation $out = (inA - inB)$. If either input is Numeric.NaN, the output will be Numeric.NaN. The Subtract is available in the [Math](#) folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-Tangent

- Tangent performs the operation $\text{out} = \tan(\text{inA})$. The Tangent is available in the **Math** folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-TimeDifference

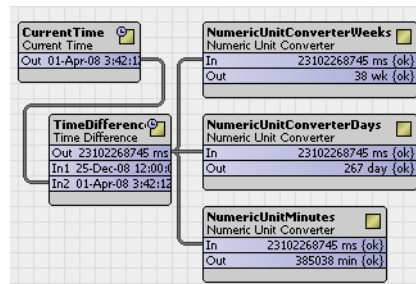
- TimeDifference has two inputs (In1 and In2), each requiring an absolute time (AbsTime) value. The TimeDifference component subtracts In2 from In1, and outputs it as a StatusNumeric, in milliseconds. You may link a **CurrentTime** object to one of the inputs for a “countdown” or “count-up” type output.

The following types of properties are used in the TimeDifference component:

- **Out**
This property displays the numeric value (in milliseconds) that represents the time difference between In1 property value and In2 property value.
- **In1**
This is the property value from which In2 is subtracted.
- **In2**
This is the property value that is subtracted from In1.

Figure 2-35, shows an example use of the TimeDifference component:

Figure 2-35 Example use of TimeDifference component



Note the following about the example wiresheet view in Figure 2-35:

- This example shows the configuration of a TimeDifference component that is used to keep a running count of the “Days until Christmas”.
- A fixed target date (December 25) is set into the In1 property.
- A CurrentTime component is used to link into In2. This date is “subtracted from” the date in In1.
- The Out value is linked to three separate conversion components that convert the time difference Out value from milliseconds into Weeks, Days, and Minutes.
- After December 25, the Out value is negative

The TimeDifference component is in the **Timer** folder of the kitControl palette.

See also [Alphabetical list of kitControl components](#)

kitControl-Tstat

- Tstat provides basic thermostatic (On/Off) control with a StatusBoolean Out property and StatusNumeric inputs for controlled variable (Cv), setpoint (Sp), and differential (Diff). An Action property allows operation as Direct or Reverse. A “Null On Inactive” property is also available. Default action is Direct (cooling). Tstat is available in the **HVAC** folder of the kitControl palette.

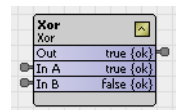
See also [Alphabetical list of kitControl components](#)

kitControl-Xor

- Xor performs a logical XOR on all valid inputs and writes the result to the out property. It is available in the **Logic** folder of the kitControl palette. [Table 2-11](#) shows the Xor object truth table when using two inputs (typical). [Table 2-12](#) shows the Xor object truth table if using all four inputs. EQUIV gate logic is accomplished by linking to a **Not** object.

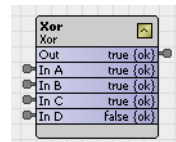
See also [Alphabetical list of kitControl components](#)

Table 2-11 Xor object truth table (2 inputs)



In A	In B	Out
false	false	false
false	true	true
true	false	true
true	true	false

Table 2-12 Xor object truth table (4 inputs)



In A	In B	In C	In D	Out
false	false	false	false	false
false	false	false	true	true
false	false	true	false	true
false	false	true	true	false
false	true	false	false	true
false	true	false	true	false
false	true	true	false	false
false	true	true	true	true
true	false	false	false	true
true	false	false	true	false
true	false	true	false	false
true	false	true	true	true
true	true	false	false	false
true	true	false	true	true
true	true	true	false	true
true	true	true	true	false

