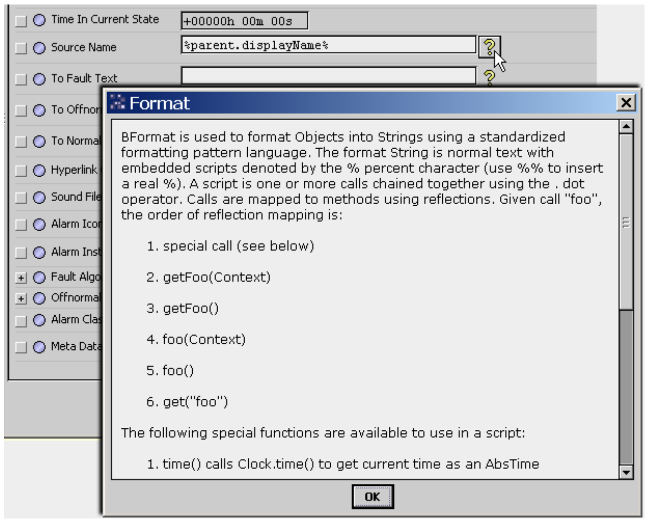
BFormat (Baja Format) Property Usage

In NiagaraAX, many properties that allow text entry use the Baja data type "BFormat," which is a class that enables localization (foreign language support), among other things.

Note: *Localization is not addressed in this document, which describes only English language applications.*

In Workbench property sheets or dialogs, such properties are conspicuous by a yellow "question" button, which produces a popup Format help window, with the Bajadoc description for BFormat (Figure 1-1).

Figure 1-1 *Format property help popup in Workbench*



Time In Current State +00000h 00m 00s

Source Name %parent.displayName%

BFormat is used to format Objects into Strings using a standardized formatting pattern language. The format String is normal text with embedded scripts denoted by the % percent character (use %% to insert a real %). A script is one or more calls chained together using the . dot operator. Calls are mapped to methods using reflections. Given call "foo", the order of reflection mapping is:

1. special call (see below)
2. getFoo(Context)
3. getFoo()
4. foo(Context)
5. foo()
6. get("foo")

The following special functions are available to use in a script:

1. time() calls Clock.time() to get current time as an AbsTime

This help popup may be useful if you are a NiagaraAX developer or someone already familiar with the class structures used in NiagaraAX, but may otherwise leave you confused. Also, there is no "context awareness"— the same help is shown for any property typed as BFormat. Obviously, there are many different ways you can use the "scripting" calls (inside "%" characters, with "." dot operator) to format text in these properties.

This document provides a few usage examples, but is not a reference. The main sections are:

- "Default values" on page 1-1
- "Example scenarios" on page 1-2
- "Errors: When scripts go bad" on page 1-8
- "Document change log" on page 1-8

Default values

As copied from palettes or originated from "manager" views, some components already have default values in certain BFormat-type properties (while others may default as empty). A few examples of those components with default values are listed in Table 1-1.

Table 1-1 *Default values for a few properties using Format data type*

Component	Property	Default Value	Notes
Alarm extension for points, e.g. OutOfRangeExt, etc.	sourceName	%parent.displayName%	Suitable “as is” in many cases, for example where all parent points are uniquely named.
History extension for points, e.g. NumericInterval, etc.	historyName	%parent.name%	
Any network component’s AlarmSourceInfo slot.	sourceName	%parent.displayName%	Often both properties are left at default values.
Any device-level component’s AlarmSourceInfo slot.	sourceName	%parent.parent.displayName% %parent.displayName%	
EmailRecipient	subject	Niagara Alarm From %alarmData.sourceName%	Much additional alarm data is used in the “body” slot.

Note that in the property value, you can use *multiples* of scripted variables along with “static text,” as done in defaults for a device’s AlarmSourceInfo “sourceName” property (a static space character separates the “`%parent.parent.displayName%`” from “`%parent.displayName%`”. The subject property of the EmailRecipient contains static text "`Niagara Alarm From `" ahead of the variable.

Why bother with it?

By using BFormat variables (scripting), sometimes you can enable replication of applications where desired results happen with a minimal of custom edits to property values, i.e. reduced engineering time, and consistent output results. Or, you may have specific text formatting needs. See the next section, “Example scenarios”.

Example scenarios

The following three scenarios show different “non-default” edits of BFormat-type properties.

- Alarm extension scenario
- History extension scenario
- Px widget scenarios

Alarm extension scenario

In this example scenario, you have a driver network with a single “device” application replicated many times in multiple devices, for example, a VAV application for 60 zones using 60 identical devices, each with identically-named proxy points, but under a uniquely-named device component. For simplicity in this example, devices are named simply “VAV1”, “VAV2”, …to “VAV60”.

For several proxy points in each zone, you need to establish alarming, adding an alarm extension. You could manually rename these points, or type “unique values” in BFormat type properties under each alarm extension, as well as related properties (in some cases) under its “offNormalAlgorithm” slot. This would ensure that alarm records (viewed in the Alarm Console) would show unique “Source” values for any alarm, without having to decipher by station ord (for example) which “RoomTemp” point was in alarm, for example to isolate by zone.

Or (more programatically) before replicating this VAV application, you could edit several Format-type properties of the alarm extension from defaults, such that generated alarms will contain more useful source data.

An example “solution” for this application, showing part of the station’s config structure, including the non-default values entered for Format-type properties under one alarm extension in one of the identically named points, is shown in Figure 1-2.

Figure 1-2 *Example config structure / alarm extension property values using edited Format-type data*

```
Drivers
└─LonNetwork
    ├──VAV1 (Lon Device object)
    │   ├─-Points (Point DeviceExt)
    │   │   ├─-Inputs (LonPointFolder)
    │   │   │   ├──RoomTemp (Lon proxy point)
    │   │   │   │   ├──proxyExt (LonProxyExt)
    │   │   │   │   └──OutOfRangeExt (alarm extension)
    │   │   │   │       ├──sourceName = %parent.parent.parent.parent.displayName% %parent.displayName%
    │   │   │   │       ├──toNormalText = %alarmData.sourceName% is OK now.
    │   │   │   │       └──Offnormal Algorithm
    │   │   │   │           ├─-highLimitText = %alarmData.sourceName% is too HOT!
    │   │   │   │           └─-lowLimitText = %alarmData.sourceName% is too COLD!

    │   │   │   ├──(etc.)

    ├──VAV2
    │   ├─-Points
    │   │   ├─-Inputs
    │   │   │   ├──RoomTemp (Lon proxy point)
  (etc)
    │   │   │   ├──(etc.)
  .
```

Note that the sourceName of the alarm extension has been changed to use two variables: "`%parent.parent.parent.parent.displayName%`" (plus) "`%parent.displayName%`", separated by a space. Given the tree structure in use, now the alarm record will show 4 (parent) levels up for the first part of the source, that is the device (e.g. "VAV1"), then the proxy point name as the second part of the source, for example "RoomTemp". So, in the Alarm Console the alarm source shows as "`VAV1 RoomTemp`".

Note: *Alternatively, there is also a "folder-level independent" method to retrieve the device name for any proxy point, instead of the "parent.parent" method. See the next example "History extension scenario".*

Now, here is the "tricky" part. All the "alarm text" type properties are relative to the *alarm record* component generated by an alarm, and *not* to the alarm extension responsible for generating the alarm. So, you cannot use the "parent.displayName" scheme in the alarm text properties, at least with any useful results. Alarm text properties include "toNormalText" and "toOffNormalText" in the alarm extension, and (if an OutOfRangeExt) in the "Offnormal Algorithm" properties "highLimitText" and "lowLimitText" (note if using the latter, these override any entry in the "toOffNormalText" property of the alarm extension parent).

But because each alarm extension's "sourceName" will now be unique (using the technique above), you can reference it within "alarm text type" properties, along with any desired static text. Except here, the sourceName is an "alarmData" *field*, from the alarm record.

In the example shown in Figure 1-2, when RoomTemp in VAV1 has a high limit alarm, the alarm data message text will be: "`VAV1 RoomTemp is too HOT!`", and when it returns to normal the alarm data message text will be: "`VAV1 RoomTemp is OK now.`"

If desired, you could further modify the OffnormalAlgorithm high and low limit text properties to include the numerical (alarm) limit, using another alarmData field. For example, if highLimitText is set to a Format value of "`%alarmData.source% is above %alarmData.highLimit% degrees!`", and the extension's highLimit is set to 74.5, upon a high limit alarm the message text generated will be "`VAV1 RoomTemp is above 74.5 degrees!`". This technique may be useful if routing the alarm using a minimum of alarm data text, say including only the timestamp and the message text.

Note: *To see what alarmData fields are available for use in this manner, go to a station's Alarm Console and view the complete details (Alarm Record popup) for any one alarm.*

History extension scenario

Consider the same network with “replicated” device applications described previously in the “Alarm extension scenario”. In each VAV zone, you wish to have histories of several proxy points, all identically named. For example, in each zone you wish to have a numeric interval history on “RoomTemp” and another one on “Damper.” However, you must either rename the parent point(s) or rename the history extensions’ “historyName” to something unique, as duplicate history Ids are forbidden.

Or (more programatically) before replicating this VAV application, you could edit the “historyName” in all history extensions, similar to the “sourceName” in the previous example.

An example “solution” for this application, showing part of the station’s config structure, including the non-default value entered in the historyName property of two history extensions is shown in Figure 1-3.

Note: *This example uses a different technique than the previous “parent.parent” method, and may be preferable because it is “folder-level independent”, as explained ahead.*

Figure 1-3 *Example config structure / history extension property values using edited Format-type data*

```
Drivers
└─LonNetwork
    ├──VAV1 (Lon Device object)
    │   ├─Points (Point DeviceExt)
    │   │   ├─Inputs (LonPointFolder)
    │   │   │   ├──RoomTemp (Lon proxy point)
    │   │   │   │   ├──proxyExt (LonProxyExt)
    │   │   │   │   └──numericInterval (history extension)
    │   │   │   │       └──historyName = %parent.proxyExt.device.displayName%_%parent.displayName%
    │   │   │   ├──(etc.)
    │   │   │
    │   │   ├─Damper (Lon proxy point)
    │   │   │   ├──proxyExt (LonProxyExt)
    │   │   │   │   └──numericInterval (history extension)
    │   │   │   │       └──historyName = %parent.proxyExt.device.displayName%_%parent.displayName%
    │   │   │   ├──(etc.)
    │   │   │
    │   │   ├─Outputs (LonPointFolder)
    │   │   │   ├──(etc.)
```

Note that each of the two history extensions’ historyName has been changed to use two variables: “`%parent.proxyExt.device.displayName%`” (plus) “`%parent.displayName%`”, separated by an underscore. This syntax utilizes a special “`getDevice()` method” in the first variable, where the proxy point’s parent device’s name is resolved, regardless of its folder depth under the Points extension. (Note in this example, proxy point “RoomTemp” is in an “Inputs” point subfolder, while the “Damper” proxy point is in the root of the Points extension).

This differs from the “`%parent.parent.parent.displayName%`” value used in the previous example—in this particular case, that would work for “RoomTemp,” but not for “Damper”. Therefore, this method is more “fault tolerant” as a result of moving a proxy point, especially to change its hierarchy.

Given the tree structure of this network, now the resulting histories will appear as “`VAV1_RoomTemp`”, “`VAV1_Damper`”, and if replicated, “`VAV2_RoomTemp`”, “`VAV2_Damper`”, and so on.

Note: *How this works: “*`%parent.proxyExt.device.displayName%`*”*

The ‘`parent`*’ steps up one level to the Lon proxy point (say, RoomTemp). The ‘*`proxyExt`*’ is the slot name that “walks back down” the tree to a different child component, in this case the LonProxyExt. The ‘*`device`*’ calls the* `getDevice()` *method, and the ‘*`displayName`*’ calls the* `getDisplayName()` *method.*

Note also in this example, the assumption is that no other components in the station will also have a “VAV1” component with a “RoomTemp” child (that also requires a history extension). Also, an underscore was used instead of a space in this case—although spaces in object names are permitted (history being one type of object), they are “escaped” in the database using a “`%20`” string, and this can be confusing in certain scenarios.

Px widget scenarios

When engineering Px widget properties, especially for “BoundLabel” types with a binding to a component, there is an important property using BFormat type:

`BoundLabel:`**`Text`**` (ObjectToString)`: Determines the content of the displayed text.

By default, when you drag a component onto a Px page (for example a BooleanWritable point), the “**`Make Widget`**” wizard associates a BoundLabel’s **`Text`** with (and makes a binding to) the *ord* of that component, with the default text value of: `%.%`

A simple edit is to add “static” text in front of this default Text value, as shown done in Figure 1-4.

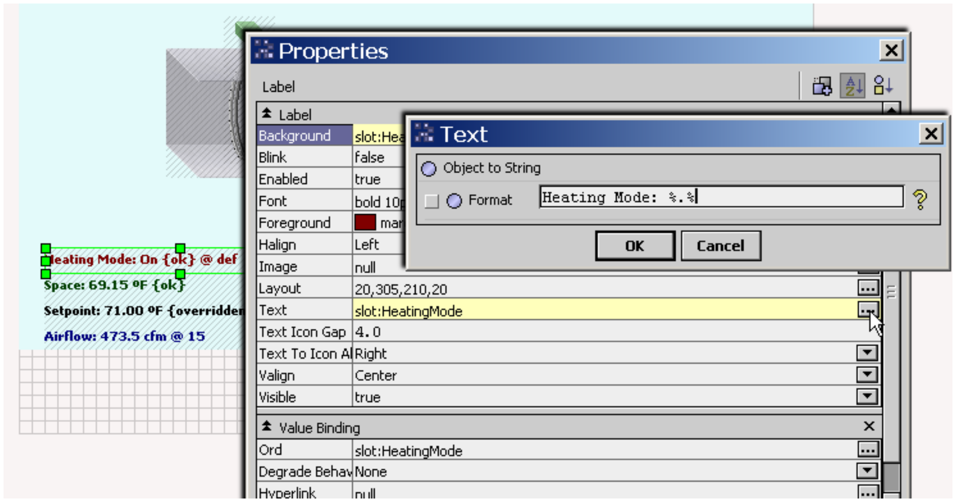


Figure 1-4 *Example adding static text in Text property of a BoundLabel*

Properties

Label

Label

Background	slot:Hea
Blink	false
Enabled	true
Font	bold 10
Foreground	🟥 mar
Halign	Left
Image	null
Layout	20,305,210,20
Text	slot:HeatingMode
Text Icon Gap	4.0
Text To Icon Al	Right
Valign	Center
Visible	true

Value Binding

Ord	slot:HeatingMode
Degrade Behav	None
Hyperlink	null

Text

Object to String

Format `Heating Mode: %.%`

OK Cancel

Heating Mode: On {ok} @ def
Space: 69.15 ºF {ok}
Setpoint: 71.00 ºF {overridden
Airflow: 473.5 cfm @ 15

See the following related subtopics:

- Default BoundLabel Text results
- Editing BoundLabel Text for points
- Advanced BoundLabel Text editing
- Weather Service example

Default BoundLabel Text results

For any point (or any component with an “Out” property), default text from the binding is identical to the “out” value displayed in that component’s property sheet. This includes any facets, plus additional information as follows:

- If bound to a *writable* point, there are 3 pieces of data in the text, namely:
  `<value> <status> @priorityLevel>`
  for example:
  “`On {ok} @16`” (a BooleanWritable) or “`20% {ok} @12`” (a NumericWritable)
- If bound to a read-only point, the default Text provides 2 pieces of data in the text, that is:
  `<value> <status>`
  for example:
  “`Clean {ok}`” (a BooleanPoint) or “`72.3 °F {ok}`” (a NumericPoint)
- If bound to a “non-point” component (that is there is no “Out” property), you must bind to a particular *slot* of that component, in order to display text other than its component “type”.
  For example, if you drag a DegreeDays component onto a Px page, the default text displayed is “Degree Days”. However, if you change the binding’s ord to “`<objectName>/clgDegDays`”, the text displays its calculated cooling degree-days value (and status), for example: “`5.0 {ok}`”

Editing BoundLabel Text for points

If desired, you can edit the Text property in any BoundLabel widget to include additional static text, and/or modify (or limit) the real-time data in the text. Table 1-2 provides a few example Formatting variables and results for writable points.

Table 1-2 *Edit options for writable points and Text in BoundLabel*

Text (BFormat) value	**Description**	**Example text if out is "On {ok} @ 16"**	**Example with static text, resulting display**
%out.value%	Value only (with facets).	On	`AHU is %out.value%.`
			AHU is On.
%out.status%	Status including priority level, if writable point.	{ok} @ 16	`Status of AHU is %out.status%.`
			Status of AHU is {ok} @ 16.
%activeLevel%	Number only (1-16, def) for priority level, writable points only.	16	`AHU is %out.value% at level %activeLevel%.`
			AHU is On at level 16.
%status.flagsToString%	String value(s) for status flags set, without braces. If non: "ok".	ok	`AHU status is %status.flagsToString%.`
			AHU status is ok.

Advanced BoundLabel Text editing

The "Object to String" scripting is quite flexible when working with BoundLabel widgets, where you are limited only by your understanding of Baja (see online Bajadoc in NiagaraAX Help). For the non-developer, these few simple rules may help:

1. The BoundLabel widget must actually be *bound* to an object (using ord)—in other words, you cannot simply drag a BoundLabel from the kitPx palette onto the Px page, edit the Text property, and get results. However, the Text value may be totally *unrelated* to the bound object, if needed.

   For example, you can bind to any object and enter a "system type" call, for example: `%time().toDateString%` to produce text like "01-Nov-08".

2. Relative to the bound object, you can use the "parent" technique to "walk up" the component tree for text for a slot (or name), for example "`%parent.parent.name%`" for the name of the parent two levels up.

   An example use of parent technique could be a BoundLabel bound to a DiscreteTotalizerExt under a BooleanPoint, where you wish to display the (parent) point's name and the number of times it has changed state, since its last reset. This could be done using this Text entry:
   `%parent.displayName% had %changeOfStateCount% COS since last reset.`
   Where the displayed text might look like: "ChWPump2 had 14 COS since last reset."

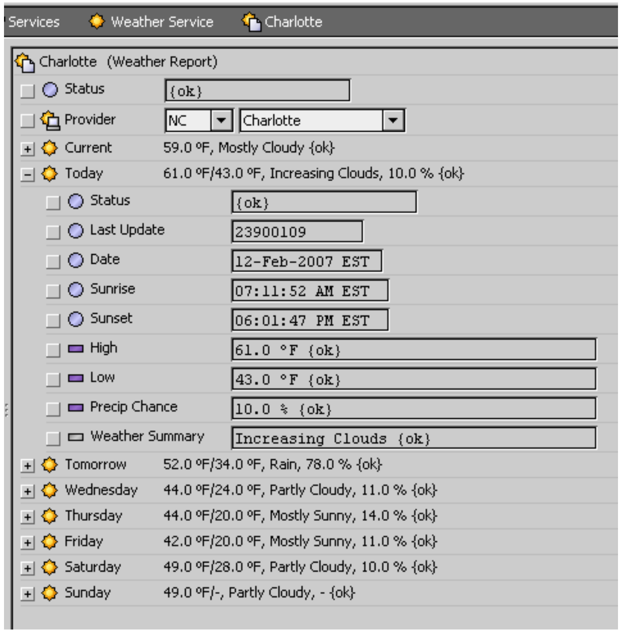
3. In addition (contrary to what was stated in previous revisions of this document), relative to the bound object, you can also "walk down" the tree in a parallel path, using the *slot name* (versus name or displayName).

   An example of this "walk down" method (via slot name) is in the previous history extension (historyName) example, along with the "parent" technique. See "History extension scenario" on page 1-4.

Weather Service example

The WeatherService can provide many pieces of information including current conditions and forecasts. Figure 1-5 shows the property sheet for a weather report (one locale under the weather service), with examples.

Figure 1-5 *Example Weather Report property sheet*



Services › Weather Service › Charlotte

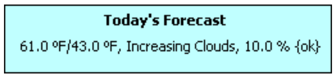
Charlotte (Weather Report)

Status	{ok}
Provider	NC / Charlotte
Current	59.0 °F, Mostly Cloudy {ok}
Today	61.0 °F/43.0 °F, Increasing Clouds, 10.0 % {ok}
Status	{ok}
Last Update	23900109
Date	12-Feb-2007 EST
Sunrise	07:11:52 AM EST
Sunset	06:01:47 PM EST
High	61.0 °F {ok}
Low	43.0 °F {ok}
Precip Chance	10.0 % {ok}
Weather Summary	Increasing Clouds {ok}
Tomorrow	52.0 °F/34.0 °F, Rain, 78.0 % {ok}
Wednesday	44.0 °F/24.0 °F, Partly Cloudy, 11.0 % {ok}
Thursday	44.0 °F/20.0 °F, Mostly Sunny, 14.0 % {ok}
Friday	42.0 °F/20.0 °F, Mostly Sunny, 11.0 % {ok}
Saturday	49.0 °F/28.0 °F, Partly Cloudy, 10.0 % {ok}
Sunday	49.0 °F/-, Partly Cloudy, - {ok}

This information may be displayed on a graphic by creating a bound label that references the weather service and the applicable property. For example in this case, if you wanted to display the forecast for today, the referenced ord would be "`station:|slot:/Services/WeatherService/Charlotte/day0`".

Note you could simply expand the WeatherService in the Nav tree to find this slot, and then drag it into the Px page, where the "**`Make Widget`**" wizard would automatically resolve to this ord. If the format Text is left at the default "`%.%`", then the text values in appear in the graphic as shown in the second line in Figure 1-6.

Figure 1-6 *Default BoundLabel Text to Weather Report's "Today"*



Today's Forecast

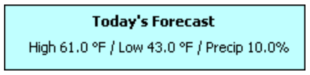
61.0 °F/43.0 °F, Increasing Clouds, 10.0 % {ok}

Each of the properties of "Today" can be referenced individually in the format Text to control whether they appear, or to provide additional text and formatting. For example, setting the Text to the following:

```
High %High.value% °F / Low %Low.value% °F / Precip %PrecipChance.value%%%
```

will result in the graphic displaying text as shown in the second line in Figure 1-7.

Figure 1-7 *Example modified BoundLabel Text to Weather Report's "Today"*



Today's Forecast

High 61.0 °F / Low 43.0 °F / Precip 10.0%

Note: *In order to display a "%" symbol, you must enter two percentage symbols, since they are used as delimiters in the format Text fields.*

Errors: When scripts go bad

Not all attempts at customizing Format-type property values may be successful—if a syntax error causes the script to fail, an "`ERROR`" or "`err:<item>`" appears in the produced text.

For example:

- If you forget a "`%`" on BoundLabel Text entry, say:
  `Fan is %out.value`
  The text displayed is: "ERROR Fan is %out.value"
- Or, a script call to a misnamed slot might fail with a displayed error similar to:
  `ChwPump2 had %err:control:DiscreteTotalizerExt:changeOfStates% COS since last reset.`

It is recommended that you test all modifications of BFormat-type properties, to make sure you get the intended results.

Document change log

Updates (changes/additions) to this *BFormat Usage - Engineering Notes* document are listed below.

- Updated: August 27, 2008
  Updated examples for both the "Alarm extension scenario" and "History extension scenario" to use "displayName" instead of "name", updating the figures used in each. Changed the history extension example to use a *different syntax* in the "historyName" format value, which is a "folder-level independent" method explained in the accompanying discussion. Related to this, corrected an erroneous statement in the section "Advanced BoundLabel Text editing" on page 1-6. (see new item 3.)
- Updated: December 10, 2007
  Applied new formatting for printed document.
- Updated: February 12, 2007
  Converted to "single source" document available both as standalone PDF and as an entry in the NiagaraAX "`docEngNotes.jar`" module for Workbench online help access.
- Draft: November 1, 2006
  Initial "Engineering Notes" type document, available in PDF only.