

Information and/or specifications published here are current as of the date of publication of this document. Tridium, Inc. reserves the right to change or modify specifications without prior notice. The latest product specifications can be found by contacting our corporate headquarters, Richmond, Virginia. Products or features contained herein are covered by one or more U.S. or foreign patents. 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. Complete Confidentiality, Trademark, Copyright and Patent notifications can be found at: <http://www.tridium.com/galleries/SignUp/Confidentiality.pdf>. © 2012 Tridium, Inc.

JACE, Niagara Framework, Niagara AX Framework and the Sedona Framework are trademarks of Tridium, Inc.

NiagaraAX Synthetic Modules

Starting in AX-3.7, NiagaraAX support was added for synthetic modules and types. This new feature allows you to create and modify memory-resident modules at run-time. Additionally, synthetic type definitions are more flexible, allowing for run-time modifications not normally permitted for Niagara types. Also, a range of new features are now possible that build upon the flexibility provided by on-demand generation of Niagara types. For additional information on Niagara types/agents, refer to the “Object Model” section of the *Niagara Developer Guide*.


The following sections provide more details:

- “About synthetic module characteristics” on page 1
 - “About synthetic modules (sjar files)” on page 1
 - “About modules and types synthesized over the Fox connection” on page 2
- “Registering a synthetic type as an agent on another type” on page 3
- “About creating new synthetic modules” on page 3
 - “About the Synthetic Module File View” on page 4
- “Creating a new synthetic module” on page 5
- “About creating synthetic modules with the Build tool” on page 8
 - “About synthetic module directory structure” on page 8
 - “About the build.xml file” on page 9
 - “About the module-include.xml file” on page 9
- “Document change log” on page 11

About synthetic module characteristics

About synthetic modules (sjar files)

SJAR files are the mechanism for distributing synthetic modules. SJAR stands for “Synthetic Java ARchive.” An SJAR file (.sjar) is a compressed package whose components can be viewed with WinZip or other archive viewing tool. With synthetic modules and types, there is no source code, and no compiling. Rather than being loaded from precompiled module files, the Java byte code for these modules is generated in memory at run-time. This means that you no longer need to know Java or be an experienced software developer to create modules and types.

Note: Synthetic modules can be created and edited from anywhere in the file system, however they must reside in the `modules` folder of the Niagara installation to be used by the station or Workbench. Also, Workbench uses the synthetic module icon () to distinguish synthetic modules from non-synthetic ones.

After creating a synthetic module, you have a new .sjar file at the designated location under the SysHome node that is loaded in memory but it is basically empty. You must edit the module to customize it and make it usable. As with standard (.jar) modules, whenever a synthetic module is modified the station or Workbench must be restarted in order for those changes to take effect.

All synthetic (.sjar) modules:

- are composed of a single Synthetic Java ARchive (SJAR) file compliant with PKZIP compression.
- contain an XML manifest, which describes types that are compiled by the NRE on demand, at run-time.
- do not contain any compiled Java byte code.
- allow for run-time modification of type definitions.

About modules and types synthesized over the Fox connection

Starting in AX-3.7, the synthetic types framework feature allows you to browse remote stations containing modules and types not present on your local Workbench. As a result, you have the capability to manage much more diverse systems.

Note: *This synthesis occurs automatically when you connect to a remote station containing modules and types not on the local Workbench installation.*

For example, you may find that the Adr Service is installed on a remote station, however, the adr module is not installed on the local Workbench installation. Previously, in AX-3.6 and earlier, attempting to access a station from a Workbench installation without the same set of modules would result in exceptions and the inability to browse the remote station tree. Now, in AX-3.7, Workbench can query the station for module and type definitions when unknown types are encountered. The information is then used to synthesize the modules and types, minimally representing those objects in Workbench, and permitting you to access the remote station and navigate the station tree.

Modules and types synthesized over the Fox connection are bound to the current connection and are cleared from memory when the connection is closed. The synthesized modules and types are not added to the Workbench registry, they are only available while connected to the remote station.

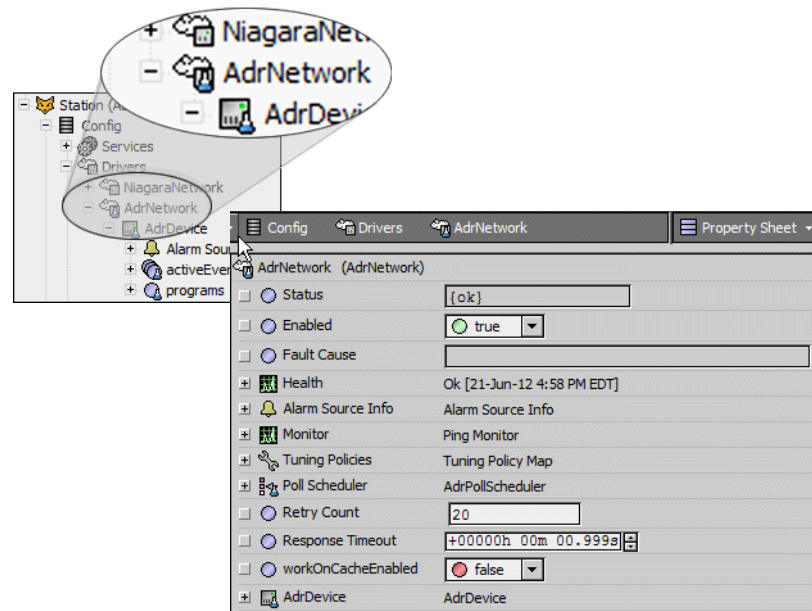
When types are synthesized, only the slots or enumeration entries are recreated in Workbench. The only accessible logic is that which is inherited from super types that exist within modules in the Workbench installation.

Components NOT accessible in synthesized types:

- **Logic**
Logic, such as Methods, compiled into the type that is being synthesized, will not be transmitted.
- **Views**
Classes, such as Device or Network Managers, will not be recreated in the synthesized module.
Note: *PxViews present in the station database WILL be available.*
- **Files**
Files within a module, such as images, px files, bog snippets or any other data, on the remote station (and not present in the Workbench installation) will not be available in the synthesized module.

Workbench uses special styling to distinguish synthesized modules and types from those that exist in the Workbench installation. For example, in Figure 1 below, a small “beaker” icon is combined as a badge on top of the normal network and device icons to indicate the AdrNetwork and AdrDevice are synthesized. The beaker badge is combined with component icons on the property sheet as well.

Figure 1 Workbench applies “beaker” icon to indicate synthesized modules and types




Registering a synthetic type as an agent on another type

One practical use of synthetic modules and types is to register a synthetic type as an agent on a specific Niagara type. You can bundle Px views for distribution and register those views as an agent on another type (something not possible in AX-3.6 and earlier) so that all instances of that type are displayed using the new Px view. Using the Software Manager you can easily distribute the module to your entire system. In a very short period, you can have this module's Px view distributed to all remote stations. For more details, refer to this procedure in the section, "Creating a new synthetic module" on page 5.

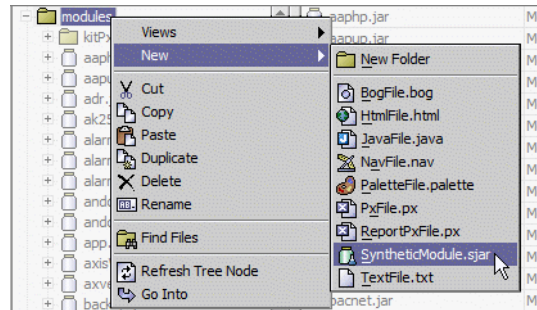
About creating new synthetic modules

Starting in AX-3.7, non-developers are able to create new synthetic modules, and customize those using the Synthetic Module File View (editor) in Workbench, as shown below in Figure 2 and Figure 3. When creating a new synthetic module, first you must navigate to a location under the **My File System > Sys Home** node. Right-click select **New > Synthetic Module.sjar** from the popup menu. At the prompt, you must enter a name for the new module (follow file naming conventions) and click Ok. The new synthetic module is created and saved in that location.

Note: Synthetic modules can be created and edited from anywhere in the file system, however they must reside in the `modules` folder of the Niagara installation to be used by the station or Workbench. Also, Workbench uses the synthetic module icon () to distinguish synthetic modules from non-synthetic ones.

An alternative, for software developers who want more control or who want to manage their module build through the Workbench command line, is create to their own synthetic modules and types and then compress the files into an .sjar file format using the Build tool. For details refer to "About creating synthetic modules with the Build tool" on page 8.

Figure 2 New Synthetic Module



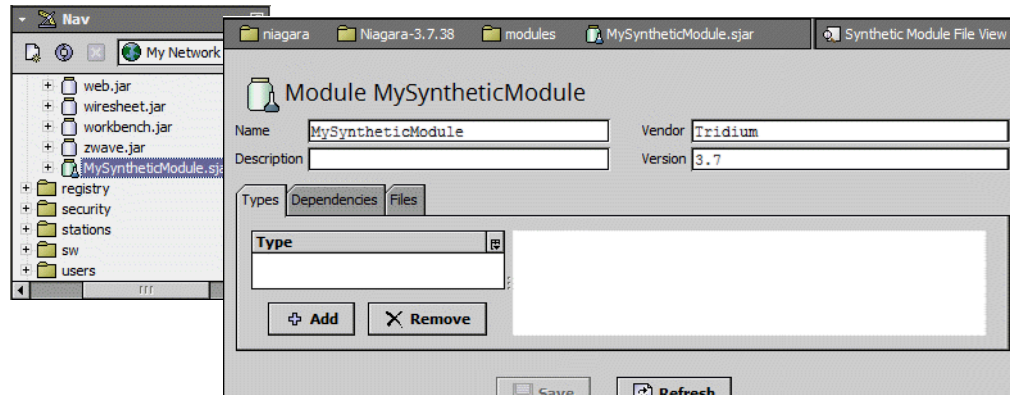
Note: The option to create a new synthetic module is NOT found under the Workbench Tools menu. The option (in AX-3.7 and later) displays under the "New" dropdown menu when you right-click in the SysHome node.

For more details on creating a new synthetic module, refer to that procedure in the section on "Creating a new synthetic module" on page 5.

As described earlier, after creating the new (empty) synthetic module you must customize it to make it usable. Double-clicking the file icon for the new synthetic module launches the Synthetic Module File View (editor) where you customize and save your changes, and restart Workbench or the station. Another method to launch the Synthetic Module File View is to right-click the file icon for the new synthetic module and select **View**.

About the Synthetic Module File View

Figure 3 Double-click new synthetic module in nav tree launches the Synthetic Module File View (editor)

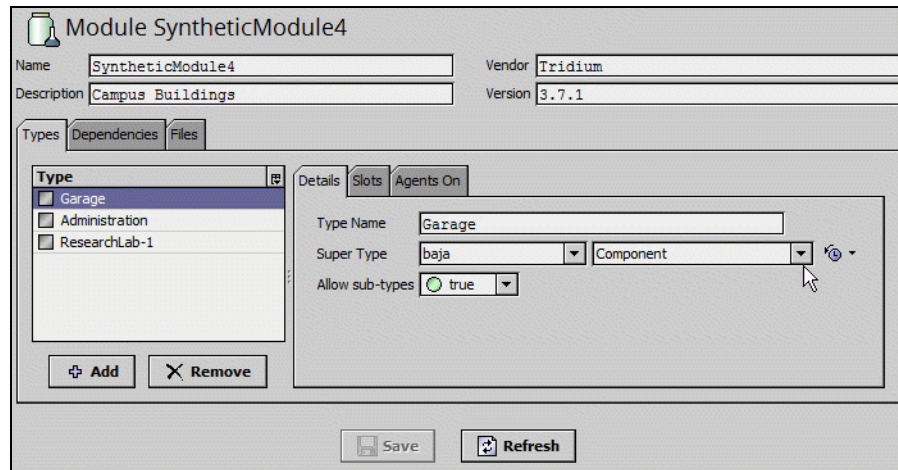


In the **Synthetic Module File View**, there are fields that allow you to manage the module attributes. These include required attributes for **Name**, **Vendor**, and **Version**. Only the **Description** attribute is not required.

Additionally, there are tabs for **Types**, **Dependencies**, and **Files** which are described here:

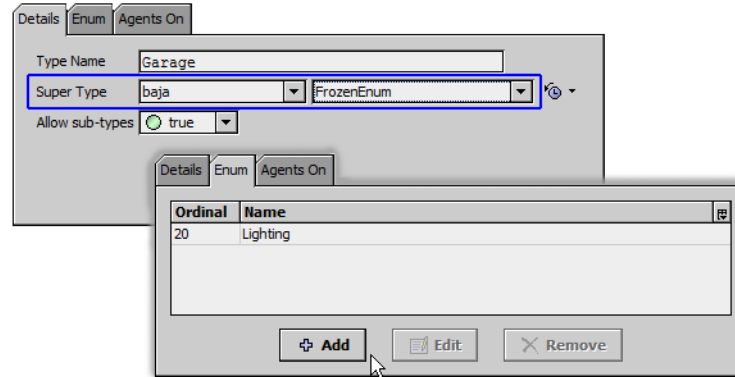
- **Types tab**
Manages types stored within the module. On the left side of the editor is the list of current types. Buttons at the bottom allow you to create new types or remove types from the module. Selecting a type on the left side loads the information in the Details, Slots and Agents On tabs on the right side, as shown in [Figure 4](#):

Figure 4 Synthetic module with added types, selected type populates the Details tab



- **Details tab**
 - **Type Name** is the name of the Niagara type. The name does not need to start with “B”.
 - **Super Type** is the Niagara type which this type extends.
Note: Selecting a `baja:FrozenEnum` for the super type will replace the Slots tab with the Enum tab (see Enum tab below), as shown in [Figure 5](#).
 - **Allow sub-types** value can be set to **false** to make this a “final” type, so that this type cannot be a super type for other types.
- **Slots tab**
Displays a slot-sheet view of the slots for the current type. Use this to create, edit or remove frozen slots from the type definition. Each slot has a defined name, type, default value, and optional flags and facets.
- **Enum tab**
Manages enumeration elements. Displays when editing a type that extends from `BFrozenEnum`. This tab is present only when editing these `BFrozenEnum` sub-types. Each element has an integer ordinal and a string tag.

Figure 5 When SuperType selection on Details tab is changed to baja:FrozenEnum, the Enum tab replaces the Slots tab



- **Agents On tab**
Use this to manage and define relationships between types. For example, declare a synthetic type as an agent on another type. Refer to the procedure, “[To register a synthetic type as an agent on another type](#)” on page 6, for more details.
- **Dependencies tab**
Indicates which other modules must be installed for a station using this synthetic module to operate correctly. The external module is referenced by name and version. Dependencies are automatically added to the synthetic module, and assume the currently available module version, however, the version can be changed through the interface. Use the **Edit Version** button to change the version number.
Note: Two situations can result in a new dependency requirement: either a type that extends a type from another module, or a slot whose value is a type from another module.
- **Files tab**
Displays the additional resource file contents of the module and allows you to make changes using file manager features. Use the buttons to **Add** additional resource files to the module (images, px files, bog snippets or any other data), create a **New Folder**, and **Remove** files or folders.

Creating a new synthetic module


Starting in AX-3.7, the synthetic types framework feature enables non-developers to easily create and customize synthetic modules and types. A new (empty) synthetic module must be customized to make it usable. Of course, any changes you make must be saved and a restart of Workbench or the station is required.

The following procedures are described in this section:

- [To create a new synthetic \(.sjar\) module](#)
- [To register a synthetic type as an agent on another type](#)

To create a new synthetic (.sjar) module

You can create your own custom synthetic modules without having to write Java source code and compile it. The synthetic module is immediately available for customization without restarting Workbench or the station. Use the Synthetic Module File View to customize the module and save your changes. Resource files (images, px files and other sorts of data) may be included in the module. After customizing the module a restart is required.

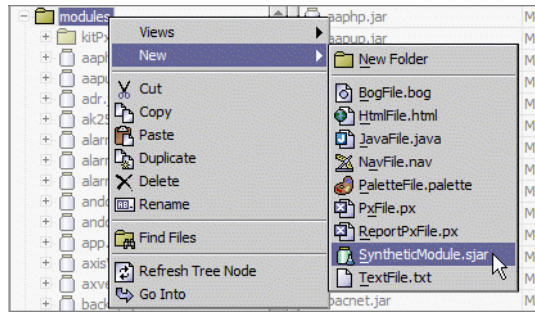
Note: Synthetic modules can be created and edited from anywhere in the file system, however they must reside in the `modules` folder of the Niagara installation to be used by the station or Workbench. Also, Workbench uses the synthetic module icon () to distinguish synthetic modules from non-synthetic ones.

To create a new synthetic module using Workbench, follow these steps:

- Step 1 In the nav side bar, expand **My File System > Sys Home** nodes.
- Step 2 Right-click on the `modules` node in the tree and select **New > Synthetic Module.sjar** from the popup menu to create a synthetic module, as shown in [Figure 6](#). Enter a name for the module, such as “**MySyntheticModule**” and click **Ok**. In keeping with Niagara file naming conventions, the .sjar module name should not include spaces or symbols.

Note: The new synthetic module option is NOT found under the Workbench Tools menu. This option, new in AX-3.7, displays in the dropdown menu when you right-click in the SysHome node.

Figure 6 New synthetic module



After creating a new (empty) synthetic module, you must launch the Synthetic Module File View to customize the new module and make it usable. The next procedure provides an example.

To register a synthetic type as an agent on another type

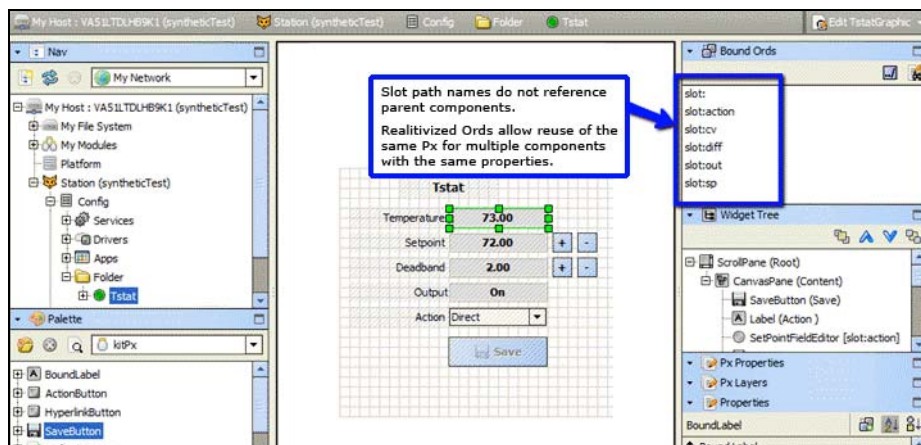
This procedure describes how to modify a new synthetic module, create a new type and register that type as an agent on all components of a particular Niagara type.

For example, if you add a Tstat component (from kitControl palette in the /hvac subfolder) to a station and create a new PxView assigned to that Tstat component. Making sure you select the option to “relativize” the ords so that there are no references to parent components in the slot paths, will set the stage to assign this PxView to any Tstat component in any station. Using standard methods you can build the PxView with some of the properties from that component. Next you can create a synthetic module with the intention of assigning the PxView as an agent on all Tstat components. To accomplish, this follow the steps below.

This procedure assumes you have already done the following:

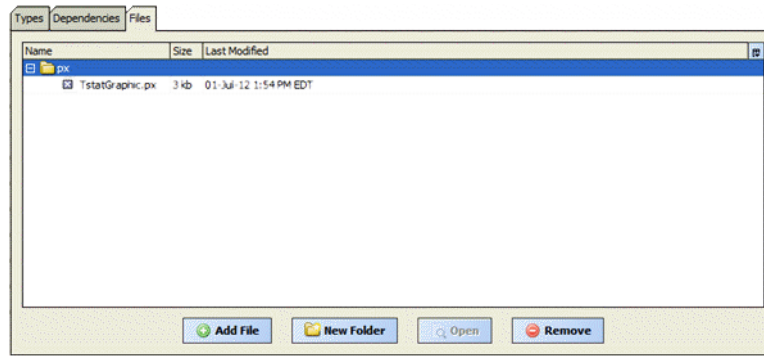
- Created a new (empty) synthetic module, such as the one created in the previous procedure.
- Added a Tstat component (from kitControl /hvac subfolder) to a station, created a PxView (similar to the one shown in Figure 7) assigned to that Tstat component, and relativized the bound ords, as shown in Fig7.

Figure 7 Station with new PxView assigned to Tstat component



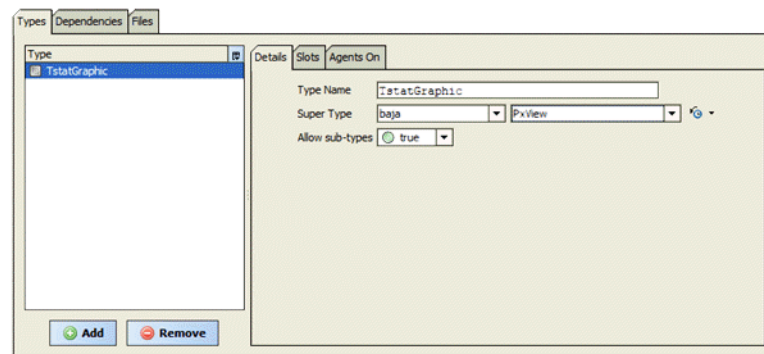
- Step 1 Expand **My File System** > **Sys Home** > **modules** folder to locate the new synthetic module created earlier and double-click to open it. The module opens in the **Synthetic Module File View** (editor). For more details on using the synthetic module editor, refer to the section, “[About the Synthetic Module File View](#)” on page 4
- Step 2 On the **Files** tab, add a new folder named “px”, then select the px folder and add the Px file created earlier, as shown in Figure 8. Including your Px file in the module guarantees file resources are available for the module to function properly when copied to another station.

Figure 8 On Files tab, add new folder named "px", then select it and add the Px file created earlier



Step 3 On the **Types** tab, add a new type named TstatGraphic, and on the type **Details** tab, select **Super Type, baja:PxView** from the drop down menu, as shown in [Figure 9](#)

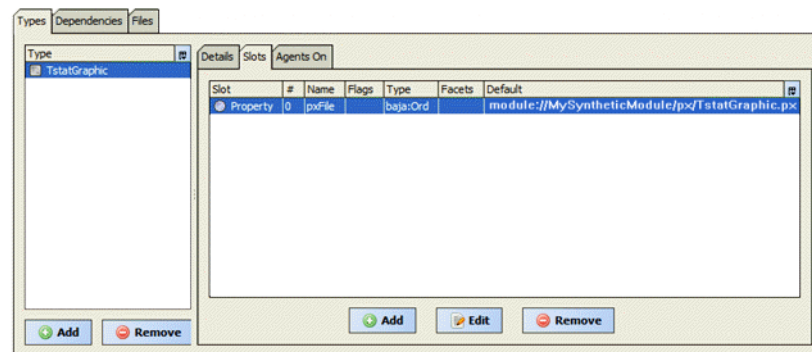
Figure 9 Add new Type with Super Type, baja:PxView



Step 4 On the **Slots** tab, add a new slot named "pxFile" and select **Type, baja:Ord**, and configure the default value to reference the TstatGraphic.px file (added to the synthetic module in step 2), and click **OK**. The slots tab will display the entry, as shown in [Figure 10](#).

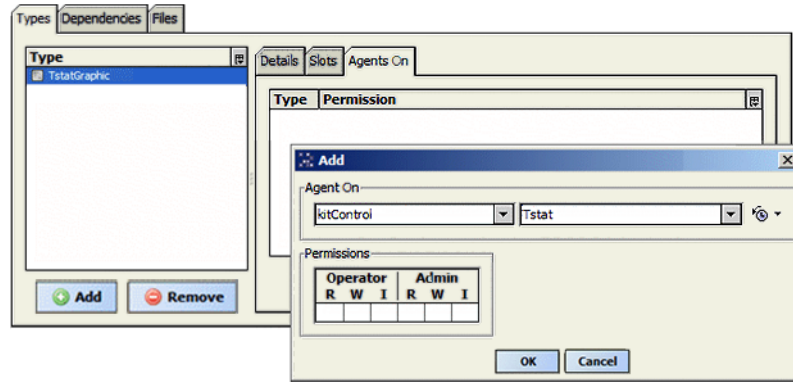
Note: The slot name must be pxFile because we are overriding the pxFile property from BPxView.

Figure 10 On Slots tab, add a slot for the Px file



Step 5 On the **Agents On** tab, click the **Add** button. In the dialog box, select **kitControl:Tstat** from the dropdown menu, and click **OK**, as shown in [Figure 11](#).

Figure 11 Add type as an agent on Niagara type kitControl:Tstat

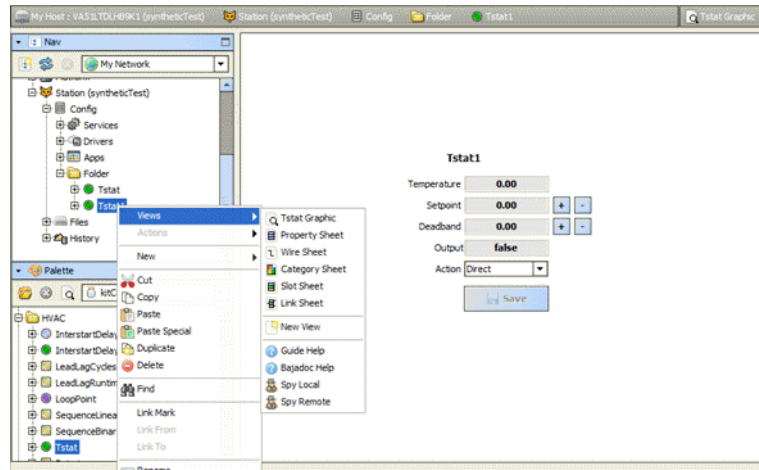


Step 6 In the synthetic module editor, click the **Save** button to save the synthetic module.

Step 7 Restart Workbench and either restart your local station or install the synthetic module to the remote JACE station that you are using and restart the JACE.

The end result of registering this synthetic type as an agent on the Niagara type, `kitControl:Tstat`, is that any Tstat component from kitControl will now have the PxView assigned as the default view, as shown below in Figure 12.

Figure 12 End result is new default view for any Tstat component from kitControl



About creating synthetic modules with the Build tool

Software developers who prefer to manage their module build through the Workbench command line, can create their own synthetic modules and types and then compress the files into an .sjar file format using the Build tool.

About synthetic module directory structure

Each synthetic (.sjar) module is managed in its own directory structure. Synthetic modules have a folder and file structure similar to that of standard modules. The .sjar module's contents must include a build.xml file and a module-include.xml file, and cannot contain any Java source code. Figure 13 shows an example of an .sjar directory structure, including a number of other file resources in addition to the required build.xml and module-include.xml files.

Figure 13 Synthetic module directory structure

```
\build.xml
\module-include.xml
\module.lexicon
\module.palette
\src
\src\px
\src\rc
\src\px\ComponentView.px
\src\rc\cloudy32.png
\src\rc\nightClear32.png
\src\rc\nightMostlyCloudy32.png
\src\rc\rain32.png
\src\rc\sunny32.png
```

Once the .sjar contents are ready, the build command can be used just as with standard modules to compress the files into an .sjar module within the modules directory. For information on using the Build tool, refer to that section in the “Niagara Developer Guide.”

About the build.xml file

The build.xml is almost identical to what you would find in a standard module. It contains module meta information, a list of external module dependencies, it defines the packages the module contains, and specifies other file resources to be included within the module. The one key difference is the inclusion of the "synthetic" attribute on the <module/> tag, which instructs the build tool to assemble the files as an .sjar module instead of the standard .jar module.

Note: The "synthetic" attribute on the <module/> tag instructs the build tool to assemble the files as an .sjar module.

Figure 14 Sample build.xml file

```
<module
  name = "synthMod"
  bajaVersion = "0"
  vendor = "Tridium"
  description = "Synthetic Module"
  preferredSymbol = "syn"
  synthetic = "true"
<!-- Dependencies -->
  <dependency name="baja" vendor="Tridium" vendorVersion="3.7"/>
  <dependency name="history" vendor="Tridium" vendorVersion="3.7"/>
<!-- Packages -->
  <package name="com.tridium.synth" />
<!-- Resources -->
  <resources name="px/*.px"/>
  <resources name="rc/*.png" install="ui"/>
</module>
```

About the module-include.xml file

The module-include.xml file for a .sjar synthetic module contains the full definition of each type, used by the framework to create on-demand java bytecode classes. For this reason the syntax extends upon the standard module-include elements to include the additional details required to generate the class. Each Niagara type is represented by a <type> element, which has three required attributes, and two optional attributes.

- **class**
A required string value containing the full type name, which is made up of the package name and the Class name.
- **name**
A required string value containing the name of the Niagara type, which is the Class name without the preceding "B".
- **extends**
A required string value containing the full type name of the superclass, made up of the superclass's package name and the Class name.
- **final**
An optional boolean value (default is "true", or "false") which defines whether the class created can be extended by other classes. If the final attribute is "true" the class created can not be the superclass of other types.

- **abstract**
An optional boolean value (default is "true", or "false") which defines whether the class created can not be instantiated and must be extended by other classes. If the abstract attribute is "true" the class can not be added to a station, and must be the superclass of other types.

About Complex Types

Types which extend from BComplex can define their frozen slots using the <slots> element. Currently, only "property" slots are supported, through use of the <property> element. Figure 15 shows an example of creating a new BOfficeUser type which extends BUser. This new type is final, so it cannot be further extended, and adds two new BInteger properties named "fingers" and "toes".

Figure 15 Complex types example

```
<type name="OfficeUser" class="com.tridium.synth.BOfficeUser"
extends="javax.baja.user.BUser" final="true">
  <slots>
    <property name="fingers" flags="0" type="baja:Integer" value="10" facets=""/>
  >
    <property name="toes" flags="0" type="baja:Integer" value="10" facets=""/>
  </slots>
</type>
```

The <property> element has three required attributes, and two optional attributes.

- **name**
A required string value containing the name of the property slot in its parent.
- **type**
A required string value containing the type spec.
- **value**
A required string value containing a representation of the default value for the property slot. For properties containing a BSimple type, this is the string encoding of their value. For properties whose value is a BComplex, this is the BOG encoding of their value.
- **flags**
An optional integer value (default is "0") containing the default flags for this property slot in its parent.
- **facets**
An optional string value (default is BFacets.DEFAULT encoded as "") contains the string encoding of the slots BFacets.

The <action> element has one additional optional attribute.

- **return**
An optional string value containing the type spec of the action's return value, or null, made up of the superclass's package name and the class name.

Types can be registered as Agents on other types. This allows the registry to manage relationships between the current class, and the classes this type is an agent on. Proper use of agents requires that the type also implement the BIAgent interface, using the <interfaces> and <interface> elements, as shown in Figure 16, which are specific to synthetic types.

The example shown in Figure 16 creates a new ComponentView type which extends PxView. The type is final, so it can not be the superclass of other types. A single "pxFile" property is defined, which overrides a property of the same name already defined on the PxView superclass. This new type then implements BIAgent, and is registered as an agent on BComponent. This example type registers the specified "ComponentView.px" file as a view on all BIService instances within the station.

Figure 16 ComponentView type Example

```
<type name="ComponentView" class="com.tridium.synth.BComponentView"
extends="javax.baja.agent.BPxView" final="true">
  <slots>
    <property name="pxFile" flags="0" type="baja:Ord" value="module://synthMod/
px/ComponentView.px" facets=""/>
  </slots>
  <interfaces>
    <interface class="javax.baja.agent.BIAgent"/>
  </interfaces>
  <agent>
    <on type="baja:IService"/>
  </agent>
</type>
```

About Frozen Enum Types

The ComponentView type can be used to create types that extend BFrozenEnum for defining enumerations as synthetic types. These types must directly extend from BFrozenEnum and be declared final. The definition for enumerations is similar to what is used for types extending BComplex, except for the use of the <entries> and <entry> elements. Each <entry> element has two required attributes for defining the enumeration value:

- **name**
A required string value containing the name of the enumeration value.
- **ordinal**
A required integer value containing the numeric representation of the enumeration value.

The type definition shown below, in [Figure 17](#), creates a new FloorType enumeration with three possible values: carpet, tile, and cement.

Figure 17 FrozenEnum type Example

```
<type name="FloorType" class="com.tridium.synth.BFloorType"
extends="javax.baja.sys.BFrozenEnum" final="true">
  <entries>
    <entry name="carpet" ordinal="0"/>
    <entry name="tile" ordinal="1"/>
    <entry name="cement" ordinal="2"/>
  </entries>
</type>
```

Document change log

Changes/additions to this *NiagaraAX Synthetic Modules* Engineering Notes document are listed below.

- Initial publication: July 17, 2012

