

NOTE: This article and all content are provided on an "as is" basis without any warranties of any kind, whether express or implied, including, but not limited to the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. In no event shall Wonderware North be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of information contained in this article.

Introduction

If you have dealt with a Wonderware System Consultant when first developing a System Platform implementation, you may be familiar with the Base Template Library. The library is a set of objects, developed by the North American System Consultants group, that allows the developer to set up automatic addressing with a few simple UDA settings. However, upon first opening the Base Template Library (or BTL), the architecture can sometimes look very daunting to the unfamiliar developer. The purpose of this TechTip is to show how simple the BTL is to configure and set up.

The Template Structure

The first thing you will notice about the BTL after import is that there are a great deal of templates added to the template toolbox. You'll notice that for each base template in the galaxy, there are both a set of \$m templates and a set of \$a templates. This system is supposed to simplify the development process, so if there are so many templates, how is this accomplished? Well, to start off, the developer should never touch the \$m templates. This set of templates contains the inherent code and attributes of the BTL (which we'll cover in more detail later in the article) and is not meant to be modified by the user. If Wonderware makes modifications to the Base Template Library, they will make changes to the \$m templates which will allow the user to import the changes without affecting any development the end user has done. Thus, you can hide these templates in the toolbox and never even worry about them.

The \$a templates, however, are yours to modify, add to, or derive off of. Since the BTL scripting is locked in parent, you cannot modify the base code; thus you can add or modify without risk of breaking the functionality of the BTL. Since you will be using the \$a set as your "base", feel free to rename them with your company or project name. Additionally, you may also want to move and hide the original base templates, so that all you would see is the \$a set.

The Functionality

Why was the BTL created? If you have attended any Wonderware training classes on System Platform, you know that one of the main features of the software is the ability to automatically assign addresses to the individual attributes of your objects, removing the need for the developer to enter each individual I/O address via scripting. In the training, the trainees write a script to do this. To remove the developer's need to write this scripting for every template, the BTL gives the developer a set of configurable attributes. These attributes interact with built-in scripting that automatically assigns the I/O based on the settings the developer configures.

TechTip: Debunking the Base Template Library: Part I

The BTL, on the surface, looks very daunting. If you were to open the script that does the binding, you would see a great deal of code that makes little to no sense. Additionally, it looks as though there are over 20 different UDAs (User Defined Attributes) that you need to configure. This is supposed to be simple, why are there so many things to configure? Do not get discouraged; in reality developers only need to concern themselves with 3 to 5 different attributes out of that laundry list (the remainder are intermediary attributes used by the scripting). And since the code is locked, you will not have to worry about modifying it.

To understand how the BTL works, let's take a look at how an I/O reference is structured:

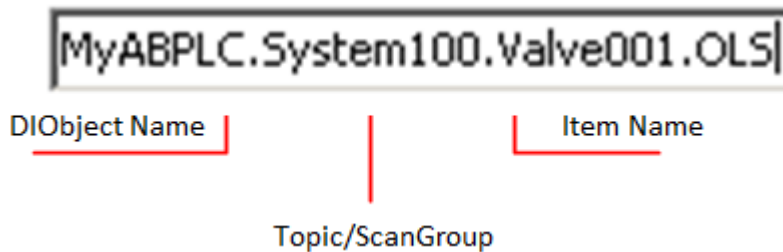


Figure 1: An example I/O reference

Notice that an I/O reference can be divided into three sections; the DIObject Name, the Topic/Scan group, and the Item Name. The BTL allows you to configure 3 to 5 UDAs which will automatically build these three sections of a reference for every item that has either an InputSource, OutputDest, or both.

The DIO.BindLevel Attribute

Developing a successful BTL application starts at the DIObject level. All of the base DI templates (\$aDDESuiteLinkClient, \$aOPCCClient, \$aInTouchProxy) have two UDAs named DIO.BindLevel and DIO.BindLevelEnum. These two are linked; the number assigned to DIO.BindLevel tells the BTL which value to use in the DIO.BindLevelEnum. If you look at the DIO.BindLevelEnum attribute, you will notice it is a string array:

Index	Value
1	DisableThisFunction
2	MyPlatform
3	MyEngine
4	MyArea
5	Use DIO.AppObjectName

Figure 2: The DIO.BindLevelEnum

This string array determines which relative reference the DIObject is going to give its instance name to. In each of the System templates (\$aArea, \$aAppEngine, \$aWinPlatform) you will notice that there is a UDA called DIO.Name. DIO.Name is the lynchpin of the BTL - this is where the name of the DIObject your templates will want to access will be stored. When developing your BTL system, a good portion of your decision making will revolve around what to set this attribute to. The attribute defaults to 4 (MyArea), so for purposes of this document, let us leave the default.

TechTip: Debunking the Base Template Library: Part I

At this point, when you create the instance and place it under an instance of \$aArea and deploy, the scripting in the DIOObject will place the name of the object into the DIO.Name UDA of the \$aArea instance. Let us say, for example, your model view looked like the figure below:

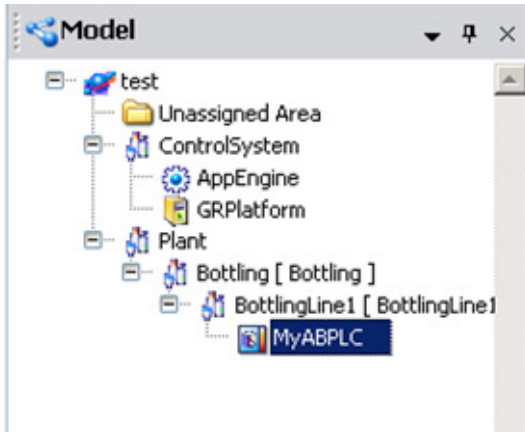


Figure 3: Example Model View

In this configuration, the MyABPLC DIOObject will place its name, "MyABPLC" into the DIO.Name attribute of BottlingLine1.

Now that you have the name of the DIOObject stored in the area, you need to configure the Application Objects to get their DIOObject name from the same place. If you look in the Application Objects (i.e. \$aUserDefined), you will notice that there is also a DIO.BindLevel and a DIO.BindLevelEnum. However, also note that there are more options in the Enum in an Application Object:

Index	Value
1	DisableThisFunction
2	MyPlatform
3	MyEngine
4	MyArea
5	MyContainer
6	Me
7	Use DIO.Name InitialValue
8	Reserved For Future Use
9	Reserved For Future Use

Figure 4: DIO.BindLevelEnum of Application Objects

Most of the time, you will want to match this level to the level used in your DIOObject (there are a few cases where you may want to use options 5-7, or even 1. This will be discussed in Part II of this note, which will be released next newsletter). What this will do is tell the object, when it is deployed, to get the DIO.Name from the level specified and place it in its own DIOObject.Name to use in the automatic binding. So, in following example,

Valve001 will grab the value of DIO.Name from the BottlingLine1 area (which, from the previous step, is "MyABPLC"), and place it into its own DIO.Name.

The DIO.ScanGroup Attribute

The DIO.BindLevel and DIO.Name takes care of getting the first part of our IO reference, the DIOObject Name. Next, the system needs to get the topic or scan group. This will tell the object which PLC it will be getting its data from. This is configured using the DIO.ScanGroup and DIO.ScanGroupIndex attributes. The DIO.ScanGroupIndex attribute ties to the list of scan groups/topics in the source DI Object. For example, if the MyABPLC topic list looked like this:

Available topics:

Topic
System100
System200

Associated attributes for System100:

Attribute

Figure 5: List of topics in sample DIOObject

Setting DIO.ScanGroupIndex to a value of 1 would return "System100" to DIO.ScanGroup, setting it to a value of 2 would return "System200" to DIO.ScanGroup.

Additionally, you can set DIO.ScanGroup to 0, which tells the object that either you will directly specify the DIO.ScanGroup literally, or you will be extending DIO.ScanGroup to another attribute in the Extensions Tab. This can be useful when you do not want to have to worry about determining the ScanGroupIndex for every instance of your template, and different strategies for this will be covered in Part II of this article next newsletter.

The DIO.ItemNameOption, DIO.ItemPrefix, and DIO.ItemSeparator attributes

Now that the DIOObject name and the Scan Group/Topic have been determined for the object, the last thing to be determined is the item name in the PLC. This is configured through the DIO.ItemNameOption and DIO.ItemNameOptionEnum attributes in the Application Objects. If you take a look at the DIO.ItemNameOptionEnum, you see the following options:

Index	Value
1	Scripted
2	AttributeName
3	Tagname.AttributeName
4	HierarchicalName.AttributeName
5	ContainedName.AttributeName
6	Area.Tagname.AttrName
7	Area.HierarchicalName.AttrName
8	Area.ContainedName.AttrName
9	AreaContainedName.Tagname.A
10	AreaContainedName.HierName.A
11	AreaContainedName.ContainNa
12	AreaHierName.Tagname.AttrNa
13	AreaHierName.HierName.AttrNa
14	AreaHierName.ContainName.Attr
15	Reserved Future Use

Figure 6: *DIO.ItemNameOptionEnum listing*

These are the different options that you can use to set the Item Name for each attribute. Notice that most contain the Attribute Name. This means that the script will actually use the Attribute's name as part of the Item Naming Convention, so the name of the attribute either needs to match the tag in the PLC, or needs to match the alias given in the DIOObject (more on this later in the article). The DIO.ItemNameOption defaults to 4, which is the HierarchicalName.AttributeName. Looking at the example, it would make more sense at this point to use option 3, which is the tagname (or Object instance name).AttributeName, as we are simply looking for Valve1.OLS.

In the event that your naming convention is not separated by periods (.), you can also utilize the DIO.ItemSeparator. For example, if the PLC tag was Valve1_OLS as opposed to Valve1.OLS, change the DIO.ItemSeparator to an underscore (_). This will change the separator that is used in the ItemNameOption in the script.

Furthermore, in the event that you would like to prefix the Item Name, you can use the DIO.ItemPrefix option. By default this is blank. For example, if it were necessary to put "BL1_" in front of every tag coming from BottlingLine1, "BL1_" could be placed into the DIO.ItemPrefix.

Runtime Behavior

When the object is deployed, the script first takes the DIO.Name and the DIO.ScanGroup and concatenates them, placing them into an intermediary attribute called DIO.RefHeader. When troubleshooting your settings, DIO.RefHeader will show you the concatenation. If there is a problem with the configuration this will commonly show "NONE", which is an indication that either the DIO.BindLevel or the DIO.ScanGroup is improperly configured.

TechTip: Debunking the Base Template Library: Part I

From this point, the script will parse through each attribute and look for attributes that have an InputSource or an OutputDest (the sub-attributes that allow an attribute to be tied to I/O). When it finds one of these attributes, it then builds the address string based on the DIO.RefHeader, DIO.ItemOption (and if configured differently, the DIO.ItemSeparator and DIO.ItemPrefix). Thus, if I create a Field Attribute called OLS and deploy the Application Object, I get the following result in the OLS.InputSource in Object Viewer:

AttributeReference	Value
Valve001.OLS.Input.InputSource	MyABPLC.System100.Valve001.OLS

Figure 7: Value of Valve001.OLS.Input.InputSource

There! With just a few simple setup steps, the galaxy is set up to auto-reference its I/O based on the needs of the project. Granted, this is a simple galaxy with a simple architecture, but this framework can be used to suit the needs of any size galaxy or architecture

Frequently Asked Questions

When I create an \$a<template>, I get a warning on my instance to start off. When I drop it underneath an engine, it goes away. What is this error?

Remember, the templates in the BTL have scripting and attributes already built into them at the template level that propagate down to the instance level. By right clicking on the instance opening properties, and tabbing to the Errors/Warnings tab, you will most likely see that the errors tie to either MyEngine, MyArea, or even MyPlatform, depending on what type of object it is. These errors disappear when the object is assigned to the proper system objects in the deployment view.

I also see a lot of warnings telling me that the validator could not resolve reference "---.---" for my I/O. I thought I didn't need to assign anything!

The validator is built to tell you when you have not assigned a reference (or assigned an improper reference) to an instance. However, in this case, the references are being assigned at runtime, so there needs to be a way to tell the validator to ignore these unassigned references. If, in the template, you instead assign "---" to the I/O source rather than "---.---", it tells the validator that the reference is not assigned, but to ignore it. This will allow you to see warnings that may affect your objects as opposed to having to check every time whether your object just has "---.---" warnings.

I configured the BTL and it doesn't assign I/O references: Help!

The main thing about the BTL is that it will not assign references if the DIO.RefHeader is either NONE or blank. This oftentimes happens when either the DIO.BindLevel or the ScanGroup is not valid. Checking your configuration and making sure the DIO.BindLevel is pointing to a place that contains a valid string is imperative to making the BTL work, and is the most common solution to the BTL not assigning I/O references.

Conclusion



TechTip: Debunking the Base Template Library: Part I

See? The Base Template Library may LOOK daunting, but in reality it just takes a small understanding of the individual pieces of the equation. By following these steps, a developer can use the library to rapidly get his project up and running. If you do not have the Base Template Library and would like it, feel free to email us at support@wonderwarenorth.com to obtain a copy of the latest release. In the next Informer Newsletter, the topic will be different strategies and scenarios for implementing the Base Template Library in your System Platform environment.