|  |
| --- |
| Jacobs Technology Inc. |
| Test SLATE Plugin Template Programmer Reference |
| In Support of Test SLATE 12.0 |

|  |
| --- |
| https://jtnet.jacobs.com/jtnet/market/products/logos/images/JacobsLogo_Blue_000.jpgJacobs Technology Inc.  8/23/2013 |



Programmer Reference

Table of Contents

[1 Getting Started 1](#_Toc365442497)

[1.1 Generate a new Plugin using the Test SLATE Template 1](#_Toc365442498)

[1.2 Plugin Template Folder Structure 2](#_Toc365442499)

[2 Visual Studio Solution 5](#_Toc365442500)

[2.1 Architecture 5](#_Toc365442501)

[2.2 Visual Studio Solution Setup 5](#_Toc365442502)

[2.3 Example Content Included in the Solution 7](#_Toc365442503)

[2.3.1 Linking the PlugIn to Test SLATE 7](#_Toc365442504)

[2.3.2 Creating a new SubSystem on the System Explorer 7](#_Toc365442505)

[2.3.3 Editing the Source 11](#_Toc365442506)

[2.3.4 Creating an Editor at the Configuration Level 12](#_Toc365442507)

[2.3.5 Creating a Main Menu Item 13](#_Toc365442508)

[2.4 Test SLATE Interfaces 13](#_Toc365442509)

[2.4.1 Menu Item Interfaces 13](#_Toc365442510)

[2.4.2 Help System Interface 13](#_Toc365442511)

[2.4.3 Message Log Interface 13](#_Toc365442512)

[2.5 Test SLATE Permissions 14](#_Toc365442513)

[2.6 Command Delivery 14](#_Toc365442514)

[2.7 Calling LabVIEW from .NET 14](#_Toc365442515)

[2.8 Classes 15](#_Toc365442516)

[2.8.1 Interop\Database\ExtensionDocManager.cs 15](#_Toc365442517)

[2.8.2 Interop\LabVIEW\VIWrapper.cs 15](#_Toc365442518)

[2.8.3 Interop\TestSLATE\Plugin.cs 15](#_Toc365442519)

[2.8.4 Interop\TestSLATE\Node.cs 16](#_Toc365442520)

[2.8.5 Models\Balances 16](#_Toc365442521)

[2.8.6 Models\IModel.cs 17](#_Toc365442522)

[2.8.7 Presenters\GridPresenter.cs 17](#_Toc365442523)

[2.8.8 UI\IView.cs 18](#_Toc365442524)

[2.8.9 UI\IReportable.cs 18](#_Toc365442525)

[2.8.10 UI\IExportable.cs 18](#_Toc365442526)

[2.8.11 UI\IPrintable.cs 18](#_Toc365442527)

[2.8.12 UI\Editor.cs 18](#_Toc365442528)

[2.8.13 UnitTests\BalanceTests.cs 19](#_Toc365442529)

[2.9 Debugging 20](#_Toc365442530)

[3 LabVIEW Project 22](#_Toc365442531)

[3.1 Architecture 22](#_Toc365442532)

[3.2 LabVIEW Project Setup 22](#_Toc365442533)

[3.3 Example Content Included in the Project 24](#_Toc365442534)

[3.3.1 Main VI 24](#_Toc365442535)

[3.3.2 Loading and Reloading 25](#_Toc365442536)

[3.3.3 Initialization 26](#_Toc365442537)

[3.3.4 Data Collection and Control 26](#_Toc365442538)

[3.3.5 Special Cases: Acquire, Calibration and Diagnostics 29](#_Toc365442539)

[3.4 Test SLATE Interfaces 29](#_Toc365442540)

[3.4.1 Message Log Interface 29](#_Toc365442541)

[3.4.2 Current Value Table Interface 31](#_Toc365442542)

[3.4.3 Data Storage Interface 32](#_Toc365442543)

[3.4.4 Test SLATE Time Tags 33](#_Toc365442544)

[3.4.5 Channel Status 35](#_Toc365442545)

[3.4.6 Reflective Memory 36](#_Toc365442546)

[3.4.7 RunSet 36](#_Toc365442547)

[3.4.8 Database Access 37](#_Toc365442548)

[3.4.9 Commands 37](#_Toc365442549)

[3.5 LabVIEW Classes 39](#_Toc365442550)

[3.5.1 Source 39](#_Toc365442551)

[3.5.2 Tag 40](#_Toc365442552)

[4 Building and Installing Plugins 42](#_Toc365442553)

[4.1 Manual Builds 42](#_Toc365442554)

[4.2 Help 42](#_Toc365442555)

[4.3 Automated Builds 43](#_Toc365442556)

[4.3.1 Before your first build on a Computer 44](#_Toc365442557)

[4.3.2 Executing the Source Build Scripts 44](#_Toc365442558)

[4.4 Inclusion of Plugins to Test SLATE 45](#_Toc365442559)

# Getting Started

This reference covers adding a plugin into Test SLATE, configuring it to add to the System Explorer as either a source and/or in the configuration. It covers adding items to the Main Menu, permissions, and command delivery. It also provides an example of calling a LabVIEW editor and a .NET editor.

## Generate a new Plugin using the Test SLATE Template

1. Open the **Create Test SLATE Source Project** window.

Option: Windows Start

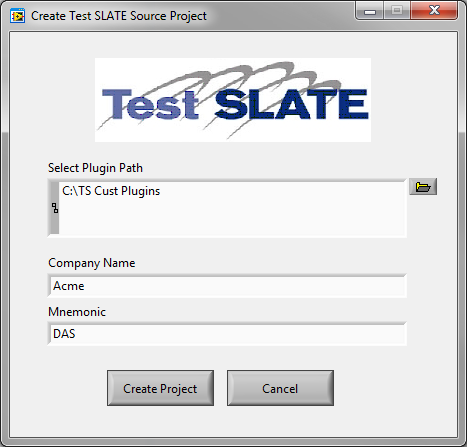
* 1. From the Windows **Start** item on the Taskbar, navigate to the **Test SLATE** folder within **All Programs**.
  2. Select **Create Test SLATE Plugin**.

Option: LabVIEW

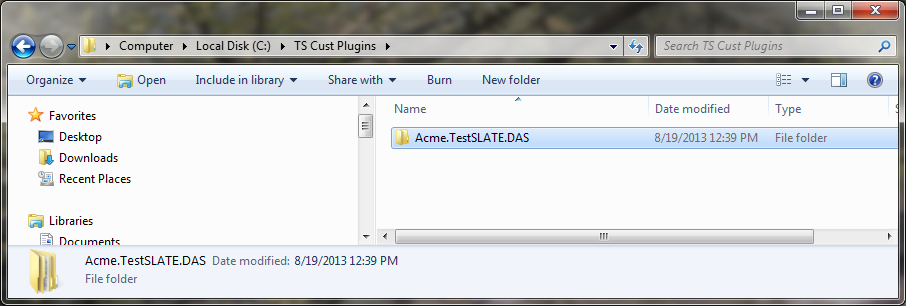
* 1. Run **LabVIEW 2013** and select **Create Project**.
  2. From the **Create Project menu**, select **Test SLATE Source** from the list of items.
  3. Click **Finish**.

1. Enter the requested information in the **Create Test SLATE Source Project** window.

| Input | Description |
| --- | --- |
| Select Plugin Path | Select the path for the new Plugin Template Folder. The new Source will be placed within this folder and named according to the CompanyName.TestSLATE.Mnemonic naming convention. |
| Company Name | Enter the Company Name or Text to appear in the naming of the new Source. |
| Mnemonic | Enter the Mnemonic or Text to appear in the naming of the new Source. |



1. Click **Create Project**. A new set of code files will be created within the selected Plugin Path. Note the folder name generated according to the naming convention in the figure below. From this point forward, this naming convention resultant name will be referred to as **MyPluginName**.



1. If the Option: LabVIEW was selected:
   1. The LabVIEW project will be renamed to **MyPlugInName.lvproj**.
   2. The LabVIEW Build Specification will be modified to create **MyPlugInName.llb**.
   3. The new **MyPlugInName.lvproj** will automatically open for editing.

## Plugin Template Folder Structure

Inside the Template folder there are a set of standard subdirectories:

| Folder | Path | Description |
| --- | --- | --- |
| V1.0 |  | By default, the template creates the first version of the plugin. Jacobs recommends that large modifications or support for different Test SLATE versions be separated into different version number directories. |
| doc | V1.0\doc | Documentation for the plugin: the designs, requirements and test plans. |
| help | V1.0\help | Help documentation to include in Test SLATE help for the plugin. |
| lib | V1.0\lib | 3rd party reference dlls that the plugin requires.  Note: As a general rule, when adding new references (dlls) to your Visual Studio Project, leave the copy local to “True”. |
| src | V1.0\src | Source code for .NET, LabVIEW or other languages. This folder contains the Visual Studio solution and LabVIEW project files. |
| Interop | V1.0\src\Interop | Contains the files needed to interact with Test SLATE, LabVIEW, Databases, etc.   * Database – Code to interact with the Test SLATE Extension Docs or to manage plugin database tables. * LabVIEW – VI Wrappers and other code to interact with LabVIEW VIs. * TestSLATE – Test SLATE Plugin, Node and Menu files. These are the files that give the plugin the capability to hook into Test SLATE and provide the points to interface with the Test SLATE UI. |
| Lv | V1.0\src\lv | LabVIEW VIs to support the Customer.TestSLATE.lvproj. |
| Models | V1.0\src\Models | Contains the data models. The model can be a representation of a database table, a file, some other object, etc. The model will also consist of its business rules and validation classes and associated interfaces. |
| Presenters | V1.0\src\Presenters | Contains the presenters that serve as the presenter/controllers for the models and their data and providing that data to the views. Typically you will have one Presenter for each type of View. |
| UI | V1.0\src\UI | Contains the views (editors) and controls and other objects that make up the user interfaces that get displayed to the user of the plugin. |
| Unit Tests | V1.0\src\Unit Tests | Contains the code for the automated unit tests (using nUnit). |

# Visual Studio Solution

## Architecture

The main feel of the architecture of the plugin is the Model-View-Presenter design pattern. This pattern formed the basis for most of the decisions in regards to how the code works in this template. Key aspects of this design pattern are that the View does not interact with the Model except in some cases of data binding, but that most interactions with the model are passed through a Presenter.

The **View** in regards to our template is the Editor (typically a TSTabbedDocument) but the View could also be a UserControl, WPF Form, etc. The View’s purpose in life is to interact with the User. It responds to the User’s input, key-bound shortcuts, mouse clicks, etc., and forwards the results of those events along with the necessary data to the Presenter.

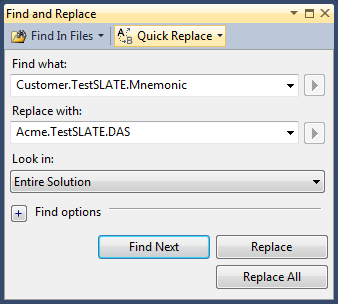
The **Presenter** is closely tied to the View. In fact, it is probably necessary to have a Presenter class for each specific type of View (Control or TSTabbedDocument) in your plugin. The Presenter responds to the events from the view and either passes the information directly to the model or creates a Command that does something to the model (depending on your implementation choice). The Presenter notifies the View of changes to the Model’s state.

The **Model** is more than just the class that we use to bind to as our data source. The Model is also all things associated with the data access layer, domain layer, business logic, validation, databases, files, and extension docs. Updates to the model are either performed in response to Commands by the Model itself or the Presenter. The Model then notifies the Presenter of changes. In cases of a data-bound model, the View would be updated through the data binding. Business Logic rules and Validation should occur in the Model and be separate from the User Interface.

This does lead to some duplicated code across the three layers, but it also allows for completely automated unit testing of both the Presenter and the Model. In addition, it allows for changing the View in the future with minimal disruption since the new View will only need to re-link its events to the Presenter.

## Visual Studio Solution Setup

1. Rename the Visual Studio **v1.0\src\Customer.TestSLATE.Mnemonic.sln** solution file to **MyPluginName.sln**.
2. Open the **MyPlugInName** solution file with Visual Studio.
3. Rename the **Customer.TestSLATE.Mnemonic** project to **MyPlugInName**.
4. Save the solution.
5. Right-click on the **MyPlugInName** project and view **Properties**.
6. Select the **Application** Tab.
   1. Change the **Assembly name** to **MyPlugInName**.
   2. Change the **Default namespace** to **MyPlugInName**.
   3. Leave **Target framework** set to “.NET Framework 4” if your plugin code will only be interacting with .NET or calling LabVIEW. If you expect LabVIEW to call your C# code, set **Target framework** to “.NET Framework 3”.
   4. Click **Assembly Information** and change the **Title** to **MyPlugInName**. Note: If your code will interact with LabVIEW, do not modify the **Assembly Version** or **File Version** from 1.0.0.0.
7. Select the **Build** Tab.
   1. If needed, change the **XML documentation file** name to match **MyPlugInName**.
   2. Jacobs recommends leaving the **Warning level** at “4”and **Treat warnings as errors** on “All”. Projects should build without any errors or warnings.
8. Select the **Resources** Tab.
   1. The **Resources** Tab shows the contents of the **Resources.resx** file for the project. This file contains strings, icons and other resources than can be referenced by name in other classes. Several strings are already present in this file. They can be modified and removed as needed. Modify the strings so that they are relevant for your project and remove any string resources that are not relevant. Look through this file closely when customizing the Template for your needs.
9. Select the **Signing** Tab.
   1. Create a new **Strong name key file** as **MyPlugInName.snk**. Do not protect the key file with a password.
   2. Delete the **Customer.TestSLATE.Mnemonic.snk** key file.
10. Save all changes.
11. Using the Find and Replace tool in Visual Studio, replace all instances of **Customer.TestSLATE.Mnemonic** with **MyPlugInName**.



NOTE: As a general rule, when adding new references (dlls) to your project, leave the copy local to “True”.

## Example Content Included in the Solution

### Linking the PlugIn to Test SLATE

When the Test SLATE application starts, it searches the \*.dll files within the Test SLATE installation folder for PlugIns. Plugins are required to identify themselves using the IPlugin Interface. This interface requires the following methods.

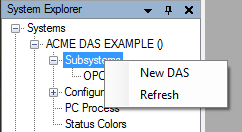
| Method | Description |
| --- | --- |
| Name | The full name of the plugin (MyPlugIn). |
| Description | A description of the plugin (e.g. "This plugin provides the Quux9000 driver for Test SLATE.”). |
| Version | The version of the plugin. (e.g. "1.2.0.9001"). |
| Path | The part of the system the plugin plugs into. (e.g. “SystemExplorer/System”) |
| Compatible With | The Core versions which this plugin is compatible with.  The CompatibleWith method must return a string of Core versions which this plugin is compatible with. You can use "\*" to indicate a range of versions. \* is assumed for any parts of the version number which are omitted, that is, "12.\*" is equivalent to "12", and "11.0.0.\*" is equivalent to "11.0.0". Prepend a "-" to indicate an exception, "+" to indicate an inclusion. e.g. "11.0.\*, -11.0.23, -12.\*, +12.1.5". (This would be interpreted as "Compatible with all 11.0 versions, except 11.0.23, incompatible with all 12.\* versions, except it will work with 12.1.5.") |

The template includes each of these methods (\Interop\TestSLATE\SubsystemPlugin.cs) and provides the appropriate values based on your entries in the Resources.resx file.

Once Test SLATE validates the Plugin information, the Plugin is allowed to create context menus at various locations within the Test SLATE interface: System Explorer\Configurations, System Explorer\SubSystems, and MainMenu locations.

### Creating a new SubSystem on the System Explorer

Most Test SLATE Sources require that the user add the Source to a Test SLATE System and then configure the Source. This template includes example code to perform these functions. From the Test SLATE main window, the user can expand their System and then right-click on the Subsystems item to choose the Select DAS option (Called “DAS” here because our example Mnemonic is DAS). This option has a "root" node implementation of the IPlugin interface, with a path of “SystemExplorer/Sources".



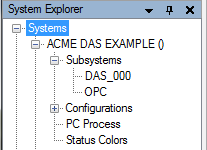
Once selected, the Plugin will add a new DAS\_000 Source to the System Explorer tree. It also adds the appropriate entries for the DAS into the database using the OnClick method. Your Source must be created first (\Interop\TestSLATE\SubsystemPlugin.cs):

| Source Field | Value |
| --- | --- |
| CellID | myData.CellID (myData is a Cells object that is set up for you) |
| SourceID | -1 (will be set to next available when saved) |
| Mnemonic | Mnemonic |
| SourceName | For the example, this is "MNEMONIC \_#" where # starts with 1 and increases for each source added. You may modify this to match your SourceName requirements. |
| EUDType | * “2” if the Source will provide data that has already been converted to Engineering Units * “1” if the Source will provide data that will allow users to add a Test SLATE Sensor in order to convert to Engineering Units. * “0” if the Source will provide data that requires linear corrections (i.e. gain) as well as Test SLATE Sensors in order to convert to Engineering Units. |
| SourceDescr | A description of your source. |
| DataBufferSampleSize | 100 (may need to be higher for high channel count or high data rate Sources) |
| RequiredSource | True if your Source should be loaded regardless of tag-channel assignment. If False, the Source will not be included in the Run Definition unless Tags are defined using the Source’s Channels. |

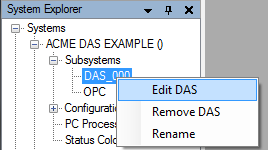
Save your Source so the SourceID field gets updated with the next available value. Then you're ready to create your Command Delivery Process entry:

| CmdDelProcesses Field | Value |
| --- | --- |
| CellID | myData.CellID (myData is a Cells object that is set up for you) |
| SourceID | The SourceID generated in the Sources table. |
| ProcessName | Your Mnemonic |
| CommandDeliveryID | Your CommandDeliveryID (see the Command Delivery section below) |
| Port | 28880 |
| PCName | "MOC" |
| WritesToRFM | False |
| HighRFMStart | Not currently used. |
| HighRFMEnd | Not currently used. |
| ExecCode | "SOURCE" |
| ExecName | "MOC" |
| Start | True if your source accepts the "START" command |
| Init | True if your source accepts the "INIT" command |
| Stop | True if your source accepts the "STOP" command (Idle mode) |
| ReloadDB | True if your source accepts the "ReloadDB" command |
| Acq | True if your source accepts the "ACQ" command |
| Cal | True if your source accepts the "CAL" command |
| Diag | True if your source accepts the "DIAG" command |
| TVKill | True |
| PerformanceFunctionsVI | The path to your Performance Functions.vi, relative to the Test SLATE install path in the registry. |
| ListenForAll | Not currently used. |
| Mnemonic | Your Mnemonic |

Once complete, the System Explorer contains a new item: DAS\_000.



The template adds multiple context-menu options to the new DAS node (Interop\TestSLATE\SubsystemNode.cs):



When the Edit DAS option is selected, the user interface launches the DAS Editor. In the example, the editor includes support for Analog Input, Analog Output, Digital Input, and Digital Output Channels. You may elect to use a similar Editor or create any other Editor you choose.

**Launching a .NET Editor**

public void OnEditDASSourceClick(object sender, EventArgs e)

{

DASEditor editor = **new** DASEditor(cellEnv, facilityEnv, src.CellID,

src.SourceID);

editor.Manager = Jacobs.TestSLATE.UX.Core.UXMap.DockManager;

editor.TabViewSavingEvent += new

TestSLATE.UX.Core.OnSaving(editor\_TabViewSavingEvent);

editor.Open();

editor.Activate();

}

The editor in this case is implemented as a TestSLATETabbedDocument, which the Core UI knows how to handle. The template has simplified this using the OpenEditor() method (Interop\TestSLATE\SubsystemNode.cs).

**Launching a LabVIEW Editor**

public void OnEditNIDAQmxClick(object sender, EventArgs e)

{

Jacobs.TestVIEW.LabVIEWInterop.VIWrappers.NidaqmxConfigurationVi myEditor = new

Jacobs.TestVIEW.LabVIEWInterop.VIWrappers.NidaqmxConfigurationVi();

myEditor.Cellid = myData.CellID;

myEditor.Sourceid = myData.SourceID;

myEditor.Run();

myEditor.FrontPanelVisible(true);

myEditor.CenterFrontPanel();

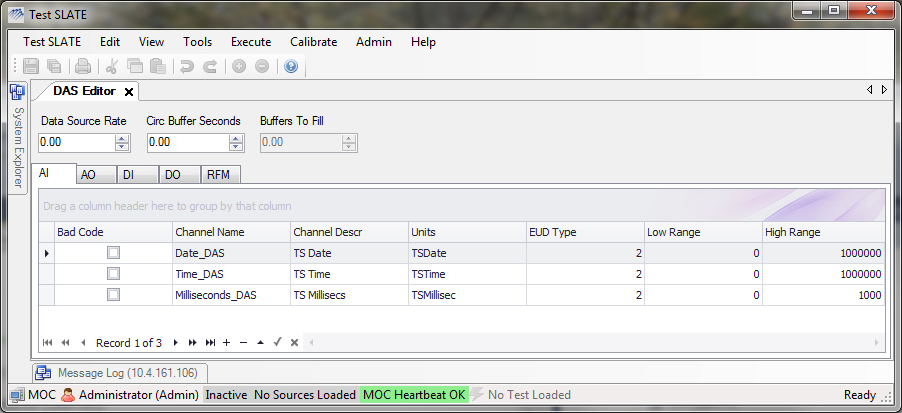
}

The editor in this case is implemented as a VI that runs on top of the Test SLATE main window as a dialog window.

Non-source plugins can provide nodes underneath configurations in Test SLATE. These nodes are implementations of the IPlugin interface whose path is “SystemExplorer/Configuration”. They launch editors just like the source nodes.

### Editing the Source

In the previous section, a Source was added to the System Explorer and to the Test SLATE database. The “Edit” option was used to launch a .NET editor for the user to provide specific information on the Source. The template also includes an example editor interface that allows users to add and remove Analog Input, Analog Output, Digital Input, and Digital Output Channels to the Source.



Many Test SLATE Sources include three time reporting Tags by default. The template adds these Tags automatically for the user when the Source is added to Test SLATE (See CreateDefaultChannels() in Interop\TestSLATE\SubSystemPlugin.cs).

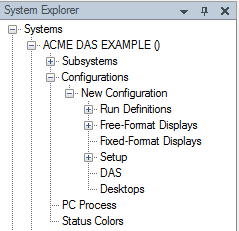
Sources sometimes need to add additional information to the database to handle specific configuration items not included in the Test SLATE database schema. These items can be included in the ExtensionDoc table. An example of this type of information is included in the RFM tab on the example editor (See CreateRFMExtensionDocs() in Interop\TestSLATE\SubsystemPlugin.cs).

When the “Edit” option is selected, the template gets the previously stored data for the Source from the database and then launches the Presenters for the user (Launch UI() in Interop\TestSLATE\SubsystemNode.cs).

As shown in the Figure above, the Channels are viewed and modified by the user via a DevExpress XtraGrid. See the Models, Presenters, and UI items for the editor to learn more about the example functionality provided in the template.

### Creating an Editor at the Configuration Level

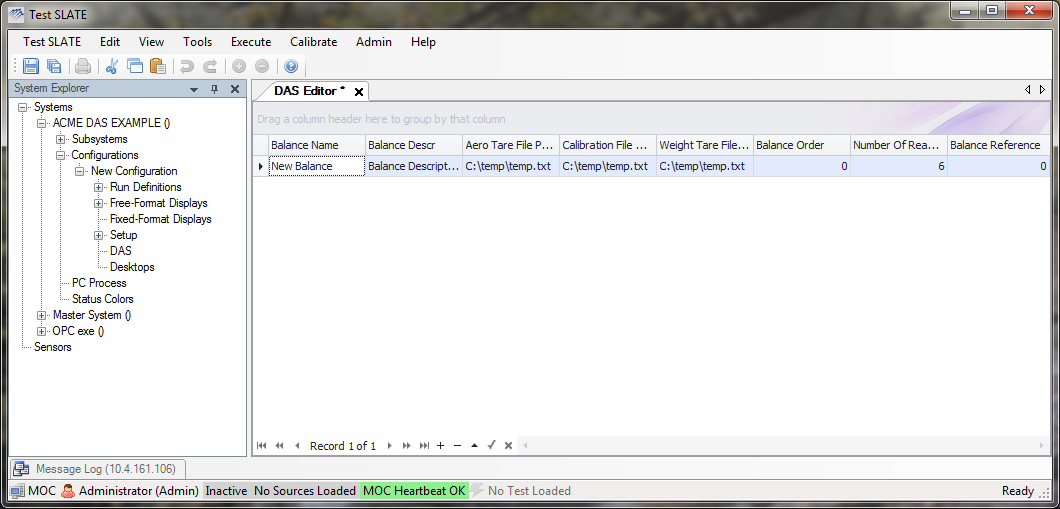
The template includes an example editor that has been added to the Configuration level of the System Explorer also (called “DAS” here because our example Mnemonic is DAS). This option has a "root" node implementation of the IPlugin interface, with a path of “SystemExplorer/Configuration".



Items at levels other than the SubSystem Node only allow context-menu options of “Edit” and “Remove” unless the item is under the Free-Format Displays Node where “Rename” and “Show” are allowed (Interop\TestSLATE\ConfigNode.cs).

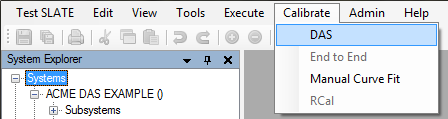
When this Node is added to the menu, it may also need to create a Source or system information within the Test SLATE Database. See Section 2.3.2.

In the example, the DAS Edit option launches a simple DAS Editor that allows users to modify Calculation type channels with ExtensionDoc stored settings (UI/Editor.cs)



### Creating a Main Menu Item

Sources may also add items to the Test SLATE Main Menu as discussed in Section 2.4.1. In the example, a new Main Menu option is added under the Calibrate Menu item (Interop\TestSLATE\MainMenuPlugin.cs).



## Test SLATE Interfaces

### Menu Item Interfaces

The following menus are available for extension: Test SLATE (our equivalent of the File menu), Edit, View, Tools, Data Analysis, Execute, Calibration, Help, and Help->Plugins.

You should inherit from the MenuPlugIn and set your parentMenu field to one of the Jacobs.TestSLATE.UX.Core.Properties.Resources.uxMenu\* items. Your menu items' OnClick handler will look like the editor ones above.

### Help System Interface

Your user help documentation must be available through the Test SLATE Help menu. Implement the IHelpMenuPlugin interface. The menu item you provide must launch your help when clicked.

### Message Log Interface

The Test SLATE Message Log is only available when the MOC is up and running. APIs are available through .NET, LabVIEW, and C/C++.

.NET API

Use the IMessageLog interface in the Jacobs.TestSLATE.Messaging assembly. Obtain a reference by calling MessageLogFactory.GetMessageLog().

LabVIEW API

The Send Message vi is the lowest level interface, but you probably want to build a proper error/warning/info message. See the Error Code Instructions for details.

## Test SLATE Permissions

Most plugins make use of Test SLATE's user/role permission system. If you are adding new permissions to the system, your plugin must add them the first time you run as Test SLATE Admin (role: 1). All new permissions must be documented. When you add your permissions to the SecurityPermissions table, you must also add them to Role 1 in the RolePermissions table. This ensures that Admin always has all permissions.

## Command Delivery

If you have custom message codes, you must document them and add them to the database when first run as Test SLATE Admin. Sources must add themselves to the Mnemonics table and to the CommandDelivery tables.

It is possible that you'll need to send a custom command from the MOC to your Data Manager. In this case, you can create custom commands in the Commands table, with unique IDs greater than 9999. You can use the CommandHandler plugin to send commands from your custom .NET UI.

## Calling LabVIEW from .NET

If you need to call a LabVIEW VI from .NET code, you can create a VI Wrapper. Inherit from WrapperBase (found in the Jacobs.TestSLATE.LabVIEWInterop.VIWrappers assembly) and implement a constructor that sets the path and calls GetReference():

**public** AiCommonArrayVi()

{

viName = "AI Common Array.vi";

viRelativeSrcDir = @"Shared\SYSCOM";

viPluginName = @"Test SLATE Core.llb";

GetReference();

}

public Single[] AiCounts

{

get { return (Single[])(GetControlValue(@"AI Counts")); }

set { SetControlValue(@"AI Counts", value); }

}

Use GetControlValue and SetControlValue to get and set control or indicator values. Use Run() to run the VI asynchronously, and Call() to wait for the VI to finish running.

## Classes

### Interop\Database\ExtensionDocManager.cs

This class contains two methods for working with ExtensionDocs. One to Load your Model from an ExtensionDoc and one to serialize your Model to XML and Save as an ExtensionDoc into the database.

To modify this class: Replace the instances of the Balance class with the one from the Model you need to save or load as an ExtensionDoc. Modify the parameter list if needed to pass in something other than the Configuration for the Key. Pass in the ExtensionDoc that the Node is controller of.

### Interop\LabVIEW\VIWrapper.cs

This class is the template for wrapping LabVIEW VIs for interacting with .NET. Several examples of different data types are provided.

To modify this class: Replace the strings in the Resources.resx file with the appropriate ones for the VI you’re wrapping. If you’re wrapping more than one VI, add additional entries to the Resources file.

Add getter / setters for each of the Controls and Indicators present on the Front Panel of the VI. The spellings must match exactly between the VI Wrapper and the Front Panel.

### Interop\TestSLATE\Plugin.cs

This class is the essential plugin component that registers the plugin with Test SLATE. This class is responsible for doing the following:

* Creating the initial Root Node
* Creating the “ADD” Context Menu Item
* OnAddClick, both should (if necessary):
  + Create Mnemonics
  + Create Command Delivery Entries
  + Create Permissions
  + Create Error Codes
  + Create Source (or System Source)
  + Add Child Nodes
  + Add Main Menu Entries

To modify these classes: Replace appropriate code with code necessary for proper configuration of your source or system source creation of any channels, permissions, error codes, etc. Mnemonics and other strings will be location in the Resources.resx file and can be modified there.

### Interop\TestSLATE\Node.cs

This class represents nodes in the SystemExplorer where location is determined by a string in the Resources.resx. This class is responsible for the following:

* Creating the “EDIT” and “REMOVE” Context Menu Items.
* OnRemoveClick, both should remove their respective entries from their Sources, and Command Delivery tables.
* OnEditClick, both should initialize their Models (from the DomainLayer, ExtensionDocs, etc), Create the View & Presenter, link the three together and Open the Editor for the User.

To modify these classes: Replace appropriate code to handle the cleanup of your source or system source on Remove, and initialize your appropriate models, views and presenters on Edit.

### Models\Balances

These classes serve as a basic Model to highlight the Model-View-Presenter architecture. They expose via public properties only the items we want the user to modify. Anything else is marked protected, internal or private. Classes are flagged as [Serializable] and as [XmlRoot] so that we can serialize it to an ExtensionDoc if we need to. Public properties are marked [XmlElement] so that they too will be serialized to ExtensionDocs.

As a standard, all Models should implement IComparable<> and IEquatable<>. These two interfaces (along with an overridden HashCode()) will allow us to sort and compare instances of our class in collections.

In addition, it also overrides ToString() so that during automated unit testing or Debug.WriteLines, we can print out valuable information about the class without needing a UI to see what is in the class.

IEditableObject provides transaction support for the Model. As we are making changes to the values in the UI, they’re not committed to the Model until the user singles that they are done editing.

INotifyPropertyChanged is probably the most important interface, as this interface signals listeners that a property in our class has changed. The Validator listens for these changes, and when a property changes, it validates the property and sets an error if the property is invalid.

The Balance’s primary constructor is private and relies on a factory method to create a new Balance. The reason for this is that if an exception is thrown by a constructor it leaves a bad reference on the heap with no way to free the memory. By using a factory method, we can validate input in the method, and/or trap exceptions and allow the system to free resources should it fail.

Models in your plugin should mimic this one in functionality and implement the same interfaces as well as overriding ToString() and HashCode().

**Models\Balances.cs**

This class is a collection of Balance objects. It implements ObservableCollection in .NET 4.0 and should inherit from CollectionBase and implement IBindingList if you’re having to use .NET 3.5. This is the collection that will bind to the View, it creates Commands to manipulate itself, by adding new Balances to iteself or removing them from itself. It returns the Command back to the Presenter so the Presenter can store it in the CommandHistory for Undo/Redo capability. It also checks the contents of its collection for validity and errors. This type of collection functions similiarly to other collections based off of IList interfaces in .NET and if you plan to use a grid, you will need a collection class to hold your models and give the View something to bind to.

It also implements the IModel interface so that the Presenter has a generic interface to communicate with.

**Models\BalanceValidator.cs**

This class contains all of the validation logic for a Balance. It subscribes to a given balance’s PropertyChanged events and when those fire, it validates the property that changed and registers the error if there is one.

### Models\IModel.cs

This is a generic interface that the Balances collection implements. This interface is used by both the Presenter and the Model (in our case, the Collection). This allows us to develop both generically and substitute any classes implementing the IModel interface into the Presenter that takes an IModel, allowing for greater code reuse without having a given Presenter being tied to a specific class type.

Customize this interface with the properties and methods that the Presenter and the Model will both need to implement to make changes to the data.

### Presenters\GridPresenter.cs

This Presenter is customized to interact with a grid control. It manages the Undo/Redo history of Commands that the Model executes. It links the Model to the View and tells the View when to refresh it’s data. Any event that occurs on the UI gets passed to the Presenter for issuances to the Model and storing the Commands in history. It reports errors to the UI as well. This Presenter is written against the IModel and IView interfaces and not against specific classes making it generic enough that another Model or View supporting those interfaces could easier be interchanged with no trouble.

### UI\IView.cs

This is the interface that our View implements that provides the properties and methods necessary for the Presenter to set the View’s data and tell it to update.

Customize this interface with the properties and methods that the Presenter and the View will both need to implement to make communication easier between the two.

### UI\IReportable.cs

This is a generic interface that your View can implement if you intend for the View to write reports.

### UI\IExportable.cs

This is a generic interface that your View can implement if you intend for the View to export data to files.

### UI\IPrintable.cs

This is a generic interface that your View can implement if you intend for the View to print.

### UI\Editor.cs

This is the example View for this template. It is a very lightweight editor, containing only a single DevExpress XtraGrid, much like a lot of our editors. This view inherits from the TSTabbedDocument and implements IView, IEditable and IExportable. Along with the properties and methods required by the interfaces, a few other event handlers are present in the UI. The View handles row validation, KeyDown, NavigationButtonClicks, CellValueChanged, InvalidRow or Cell Exceptions, DragAndDrop, and DockSituationChanged events.

Events that affect only the UI are handled exclusively inside the UI, such as the DockSituationChanged event. Other events that would affect the data instead call a similar method on the Presenter (which in turn issues a Command to the Model).

The View itself does not interact with the Model at all, does not validate any input from the user. It takes the input, passes it to the Presenter (and on to the Model) and gets the results back from the Presenter which it then displays to the User.

With the UI being so lightweight, it can easily be replaced by another set of controls at any time by rewiring the events back to the Presenter. In the future, upgrading to WPF (Windows Presentation Foundation and XAML) would mean that we would only need to create a new View, rewire our events and we could have our Balance editor back in just a couple hours work. If our View were tightly coupled to our Model, it would force a full rewrite of the View which would be more time consuming.

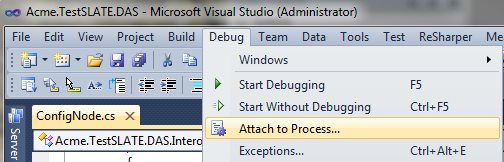
### UnitTests\BalanceTests.cs

This class has an example nUnit unit test. A quick example of the type of unit tests we can write with nUnit. Ideally, we would have unit tests for everything from the Presenter to the Model in our plugin (excepting the View). These tests would then be automated during our build process so that if some other aspect of our system changed we would detect failures early during future upgrade or if we are just tracking and fixing bugs.

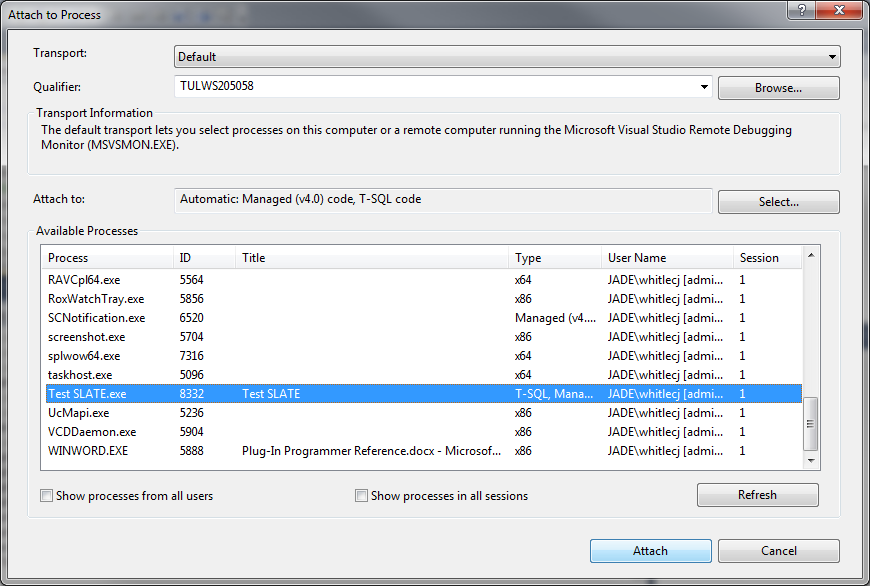
## Debugging

Sometimes it is advantageous to debug your Visual Studio projects while running Test SLATE.

1. Start the Test SLATE application.
2. Open the **MyPlugIn.sln** in Visual Studio.
3. From Visual Studio, view the **Debug** menu and select **Attach to Process**.



1. From the **Attach to Process** window, select **Test SLATE.exe** in the **Available Processes** list and click **Attach**.



1. The Visual Studio solution debugger will now attach to the Test SLATE Process. From this point, you may add breakpoints, etc into your solution for debugging purposes. Exceptions will now automatically point to the current lines of code.
2. When finished, either close the Test SLATE application or select **Detach All** from the Visual Studio **Debug** menu.

# LabVIEW Project

## Architecture

The run time portion of Source plugins are called as LabVIEW code from the main Test SLATE application. The code is expected to be located within the Test SLATE installation folder and its top level VI should be defined in the TV database at CmdDelProcesses.PerformanceFunctionsVI. The RunTime LabVIEW Code is responsible for collecting input data, sending output data to other systems, performing calculations, performing Source specific calibrations, and other functions that Test SLATE does not include. The uses of RunTime LabVIEW Code is varied, providing a powerful tool to add functionality to the core product.

The LabVIEW project template includes most of the interfaces between the Source plugin and other Test SLATE modules. It is based on a basic state machine that already handles the command and state processing required for compatibility with Test SLATE. When the template is generated, a LabVIEW project is placed in the Source folder that includes a project library, correct linkages to the Test SLATE API, and a Build Specification for the project.

Restart

Command

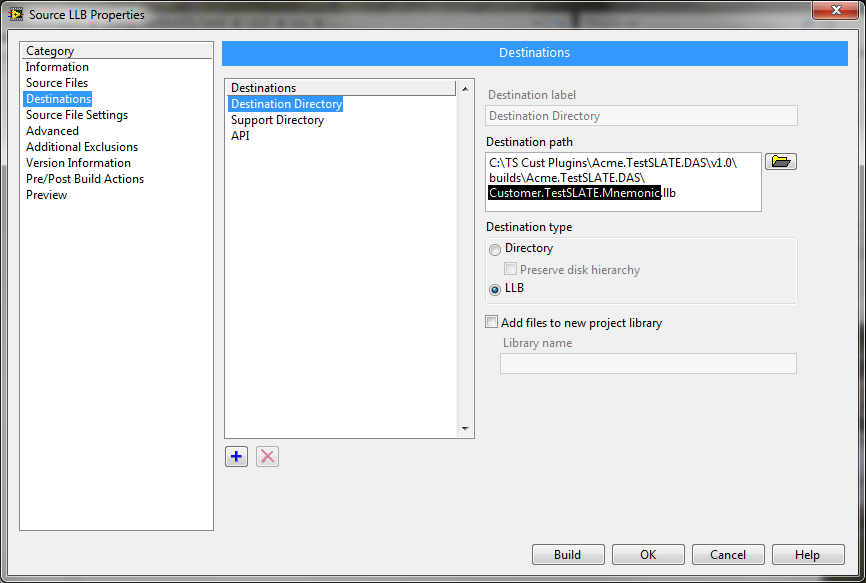
Load or Reload Command

Soft Reload

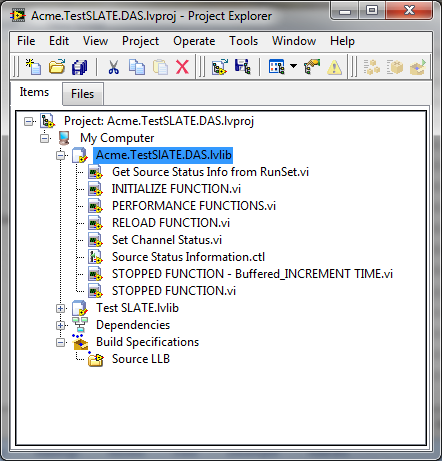
Command

## LabVIEW Project Setup

1. If the Windows Start Option was used to generate the plugin:
   1. Open the LabVIEW project at **v1.0\src\Customer.TestSLATE.Mnemonic.lvproj.**
   2. Rename the project to **MyPluginName.lvproj**.
   3. Open the Build Specification **Source LLB**.
   4. Select the **Destinations Category** from the **Source LLB Properties** Window and replace “Customer.TestSLATE.Mnemonic” with **MyPlugInName** in the **Destination Path** for the Destination Directory.
   5. Select **OK** to save and exit the **Source LLB Properties** Window.



1. From the Project Explorer, rename **Customer.TestSLATE.Mnemonic.lvlib** to **MyPlugInName.lvlib**.
2. Do NOT modify the Test SLATE.lvlib under any circumstances when editing your plugin.
3. **MyPluginName.lvlib** must contain all Source plugin code that you generate. This ensures that your code has its own namespace and that naming collisions will not occur between multiple VIs with the same name.
4. Save the LabVIEW Project.

****

## Example Content Included in the Project

### Main VI



Performance Functions.vi is the main interface to Test SLATE. You may rename the VI as long as your plugin uses the same name when adding information to the CmdDelProcesses table in the TV Database. The Connection Pane items are required for successful linkage to Test SLATE.

****

The internal code of the Performance Functions.vi is a simple state machine that has already been preconfigured for Test SLATE control. In general, do not add or delete states as your Source may not function properly with Test SLATE.

For most developers, the only modifications to the template will be to add Source-specific functionality for the Load & Reload, Initialization, and Data Collection & Control States illustrated in Figure 1 User Modifiable Source States.

Figure User Modifiable Source States

| State | Case Label | Description |
| --- | --- | --- |
| Load | RELOAD DB | Modify the RELOAD FUNCTION.vi to load any new settings that should apply to the Source or Tags. This will include loading any changes to configuration items that may have occurred in the database. |
| Initialization | INIT | Modify the INITIALIZE FUNCTION.vi to do any activities that do not need to be repeated while running the other functions. This should do as much hardware settings and pre-calculations as possible to make your Run Time functions more efficient. |
| Data Collection & Control | STOP  ACQ  DIAG  CAL | Stop performs all of the normal run-time activities for the Source. Modify the STOPPED FUNCTION.vi to handle the data inputs and outputs and Tag Status reporting. Unless your Source has specific functionality to apply for Calibrations or Diagnostics, use the same Stopped Function for all states. |

The information Test SLATE supports for all Sources is included in the Test SLATE RunSet and within many LabVIEW global spaces. This core set of information is automatically preloaded and provided within the Performance Functions.vi for your use and includes Source, Tag, and associated Channel information. For more information the items included, see Section 3.5 LabVIEW Classes.

Test SLATE also provides an interface to its RunSet that you may use to load Test SLATE information about the current test in progress (See Section 3.4.7). This information will not change while your Performance Function.vi is running so you may load RunSet information prior to entering the main state machine loop. Doing so will improve the speed of Soft Reload and Restart command processing if this information is preloaded.

### Loading and Reloading



Each time a user selects to Load or Reload a Run Definition, any currently running Sources are commanded to exit (Kill) and then restarted after all new Test SLATE settings have been reloaded from the databases. Next the RELOAD DB state will be initiated in Performance Functions.vi.

Modify the RELOAD FUNCTION.vi within the template project to handle all additional configuration settings that must be loaded from the RunSet, Database, or other files your Source may use. Much of the information needed may already be available in the Source and Tag objects supplied to RELOAD FUNCTION.vi. Access to the databases can be made via the Test SLATE domain layer (See Section 3.4.8).

### Initialization



Each time a user selects to Restart your Source or the RELOAD DB state is exited without errors, the INIT state is called. Modify the INITIALIZE FUNCTION.vi to do any pre-run activities needed. These should include making any initial communications to the device and performing any pre-calculations. This case should do as much as possible to prepare for data collection.

### Data Collection and Control



For many Sources, the response to all specific run-time states is the same. These Sources simply read and write data to devices or perform calculations when Test SLATE is Idle, in Diagnostics, or performing an End to End Calibration. The template assumes this is the case, but the placeholders are available within the template to add Source–specific functions for the ACQ, DIAG, and CAL states.

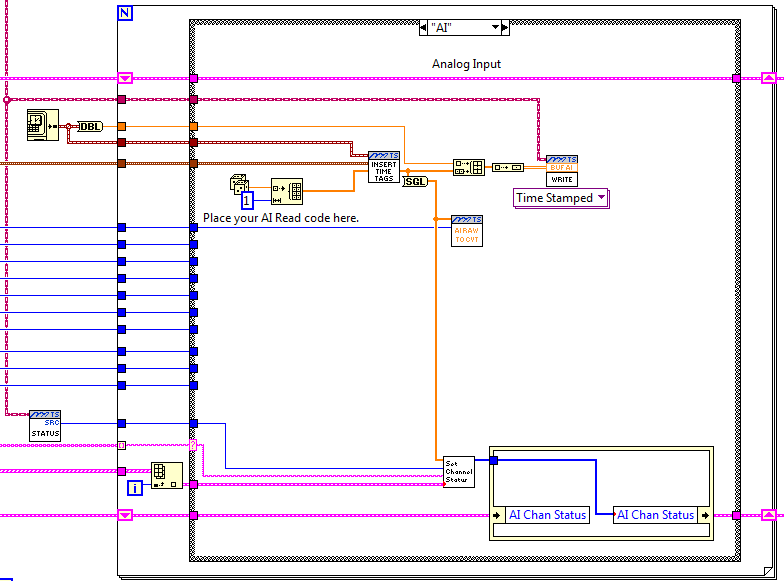
The basic functions required for the Idle (STOP) state is included in the STOPPED FUNCTION.vi. This vi includes examples of how to interface with Test SLATE to report Analog Input, Frequency Input, Calculation, Character, and Digital Input data. It also includes the interfaces necessary to determine output values for Digital Output and Analog Output Tags. Finally, the template includes an example of determining Channel Status (see next section).

The included example illustrates the coding for a Source that provides a single data value for each Tag at the Test SLATE System Rate. In this case, the Source also time stamps its own data prior to reporting to Data Storage. All data types are represented, but types that are not available for the Source may be removed by deleting the appropriate Cases.

The template code will predetermine which data types are included in the current test and will call each data type in turn within the For Loop of the STOPPED FUNCTION.vi. Data types that are not included in the current test are simply not called. For each data type that is included, each Source must provide the following basic information to Test SLATE: Current Table Value per Tag, Data Storage Buffer Data per Tag, and the Channel Status of each Tag.

**Example Input Process**

1. Get the current timestamp – In the example, the PC Clock time is used but the time source could be from a timing card or another device.
2. Read the data from the device – The example shows a single value example, but the Source should return a single value per Tag included in the current test.
3. Generate the time values for the Test SLATE Time Tags (See Section 3.4.4) and add them to the data returned from the device.
4. Send the data to the Current Value Table for display and other run time usage by Test SLATE.
5. Add the time stamp (as a double) to the data and report the buffer to Data Storage (See Section 3.4.3). Since the data includes the time stamp, the Time Stamped option is selected for Write AI Raw Data to Data Storage.vi.
6. Determine the status of each Channel (See Section 3.4.5).



**1**

**6**

**5**

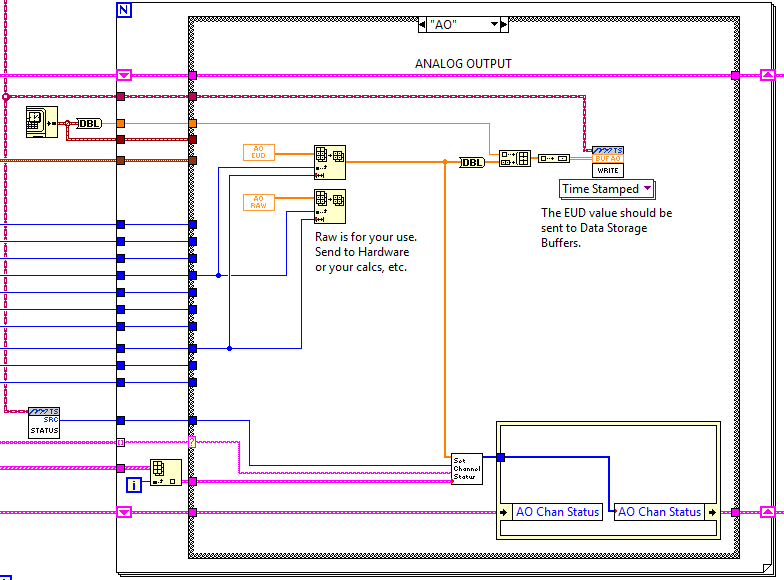
**4**

**3**

**2**

**Example Output Process**

1. Use the current timestamp – In the example, the PC Clock time is used but the time source could be from a timing card or another device. Do not generate multiple time stamps within the same source for the same samples.
2. Get the output data values requested by Test SLATE via the Current Value Table. For Analog Outputs, there are two levels of data available – RAW and EUD. RAW data should be output to the device as it has already been modified by any specified corrections from the Tag Editor. The EUD data is in user understood Engineering Units. For example, a 0-10 V controlled position source might need 5 (RAW) output to move the position 10 cm (EUD).
3. Add the time stamp (as a double) to the EUD data and report the buffer to Data Storage (See Section 3.4.3). Since the data includes the time stamp, the Time Stamped option is selected for Write AO Raw Data to Data Storage.vi.
4. Determine the status of each Channel (See Section 3.4.5).



**3**

**4**

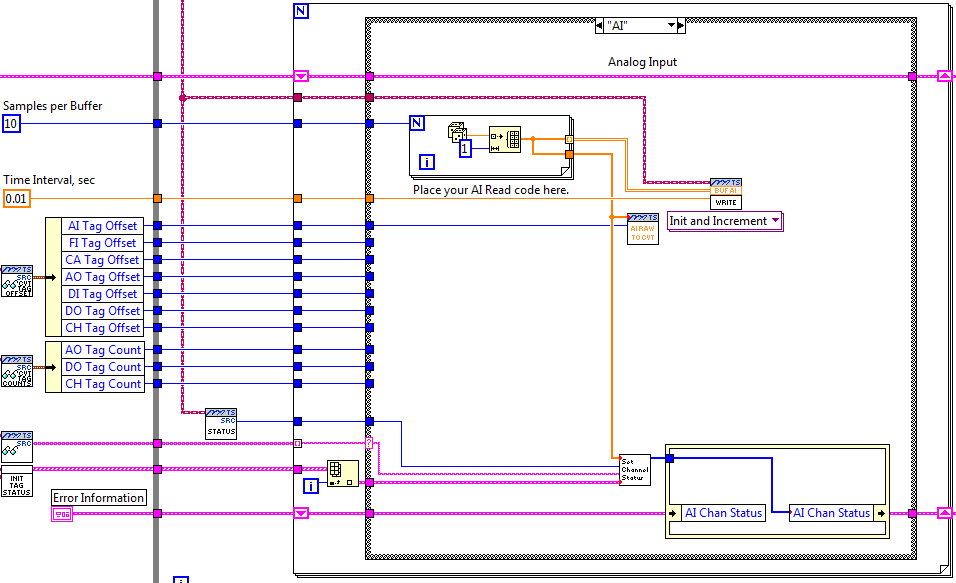
**2**

**1**

A similar example (STOPPED FUNCTION – Buffered\_INCREMENT TIME.vi) adds the complexity of a Source that provides multiple data samples for each Tag at the Test SLATE System Rate. Note that the general process is the same, only the amount of data and source of time stamps change.

**Example Input Process**

1. Read the data from the device – The example shows a single Tag example, but the Source should return data samples per Tag included in the current test. This example also includes buffering (where multiple samples are collected during each acquisition loop).
2. All of the samples per Tag should be collected for the Data Storage Buffer. However, only one sample per Tag is available in the Current Value Table. Most Sources add the most recent sample to the Current Value Table, but Sources sometimes use averages or other calculations on the buffer.
3. In this example, the Write AI Raw Data to Data Storage has “Init and Increment” selected so it will handle the time stamps for the Buffer. This example also assumes that the Source does not support Test SLATE Time Tags. Please note that if ten samples of data is sent to the Data Storage Buffer during the current acquisition loop for one data type, ten samples must be sent to the Buffer for all other included data types.
4. Send the data to the Current Value Table for display and other run time usage by Test SLATE.
5. Report the buffer to Data Storage (See Section 3.4.3).
6. Determine the status of each Channel (See Section 3.4.5).



**5**

**4**

**3**

**2**

**1**

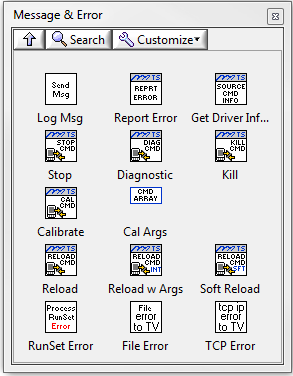
### Special Cases: Acquire, Calibration and Diagnostics

Many Sources do not require any modifications to the ACQ, CAL, and DIAG states and the example does not include any specific code to illustrate these modifications but the Case for state may be modified to match specific Source needs.

## Test SLATE Interfaces

### Message Log Interface

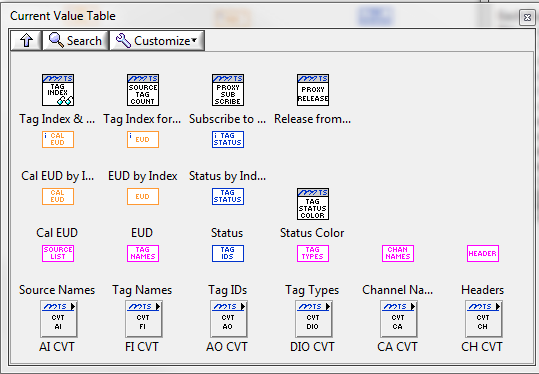
LabVIEW VIs to support Message Logging are located on the Test SLATE\Message & Error palette.



| Item | Description |
| --- | --- |
| Log Msg | Sends a message to the Message Log. The message components must be specified by the Source and are not derived from an Error input. |
| Report Error | Sends a message to the Message Log if an Error is noted. The message components are derived from an Error input. |
| RunSet Error | This VI may be called after communicating with the RunSet. If the Runset returns an error, the VI will format and send the error to the Message Log. |
| File Error | This VI may be called after performing file operations with LabVIEW. It will convert the LabVIEW error out control into a Test SLATE error cluster with appropriate Test SLATE Message Log error codes. To send an error to the Message Log, follow this VI with Report Error.vi. |
| TCP Error | This VI may be called after performing TCP operations with LabVIEW. It will convert the LabVIEW error out control into a Test SLATE error cluster with appropriate Test SLATE Message Log error codes. To send an error to the Message Log, follow this VI with Report Error.vi. |

### Current Value Table Interface

The Current Value Table (CVT) contains a single sample of data for each Tag within the current test. The CVT does not buffer data and does not control write access so that the last write to the CVT is always the CVT value. The CVT Interface consists of two levels: Full Tag List and Type Tag Lists.



**Full Tag List**

| Item | Description |
| --- | --- |
| Tag Index and Type Info | Using a Tag Name, this VI determines the CVT Index into all Full Tag List VIs, the Tag Type, and the Tag Type Index into the Type Tag Lists. |
| Source Tag Types and Count | Given a Source Name, this VI returns the Source’s Tag Types included for the current test, the offset of the Source into each Tag Type List, and the number of Tags per Tag Type. |
| Subscribe to Proxy | Adds the list of Tag Names to obtain information from the Data Proxy for when the VI is located on an AOC without Reflective Memory capabilities. |
| Release from Proxy | Removes the list of Tag Names to obtain from the Data Proxy. |
| Cal EUD | Returns the calibrated Engineering Unit Value. This value is after any Sensors have been applied, but no unit conversion from Cal to Display EUD. When by Index, use the CVT Index of a Tag to return the single Tag Value. Otherwise, all Tags are returned. |
| EUD | Returns the display Engineering Unit Value. When by Index, use the CVT Index of a Tag to return the single Tag Value. Otherwise, all Tags are returned. |
| Status | Returns the current Tag Status value. When by Index, use the CVT Index of a Tag to return the single Tag Value. Otherwise, all Tags are returned. |
| Status Color | Returns the color to display on user interface screens for the selected Tag to indicate current status conditions. |
| Source Names | Returns a list of Sources included in the current test. |
| Tag Names | Returns the list of all Tag Names in the current test. The names are in the same order as all other Full Tag List items. |
| Tag IDs | Returns the list of all Tag IDs in the current test. The Tag IDs are in the same order as all other Full Tag List items. |
| Tag Types | Returns the list of all Tag Types (AI, DI, etc) in the current test. The Tag Types are in the same order as all other Full Tag List items. |
| Channel Names | Returns the list of all Channel Names in the current test. The names are in the same order as all other Full Tag List items. |
| Headers | Returns the list of Headings and associated Values defined during Test Initialization for the current test. |

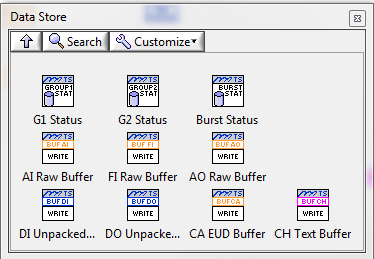
**Type Tag Lists**

A separate sub palette exists for each Data Type within the CVT. These palettes contain more detail for each Tag in the CVT for the data type. Examples of further information include Count values, Intermediate values, Conversion Information, and On/Off Text. In addition, the VIs to send data to the CVT per type is included in each of the respective sub palettes.



### Data Storage Interface

The Data Storage palette includes VIs to access information about Data Storage and to send buffer data.



| Item | Description | |
| --- | --- | --- |
| Status | Returns information about the respective Data Storage file including active state and time remaining. | |
| Buffer | Provides the method to send data to the Data Storage buffers per data type. All submits to the buffers must have the same # of samples per data type per acquisition loop. Between submits, the # of samples may vary. | |
| Correct Implementation  Loop 1:  - AI 10 Samples  - DI 10 Samples  Loop 2:  - AI 15 Samples  - DI 15 Samples | Incorrect Implementation  Loop 1:  - AI 10 Samples  - DI 2 Samples  Loop 2:  - AI 15 Samples  - DI 2 Samples |

### Test SLATE Time Tags

Test SLATE Time Tags may be created as part of the Analog Input (AI) channels for the Source. The name of the channels must be Date\_*SourceName*, Time\_*SourceName*, and Milliseconds\_*SourceName*, where *SourceName* is the name of the Source. For example, if we are adding the channels to the OPC Source, the Channel Names will be Date\_OPC, Time\_OPC and Milliseconds\_OPC. These channels must be defined as EUD Type 2 so that no other conversion is done to them.

These Channel/Tag Units will be used by display processes to make the determination that the values will need to be decoded before being displayed. The units for the Date Tag is “TSDate”, the units for the Time Tag is “TSTime”, and the units for the Milliseconds Tag is “TSMilliseconds”.

Test SLATE tags by design are floating point numbers. It is impossible to fit date and time (with time defined to the milliseconds) into two floating point numbers without losing resolution. Therefore, the date is stored in Date\_*SourceName,* time in Time\_*SourceName* (hours/minutes/seconds), and the milliseconds are stored in Milliseconds\_*SourceName*. By keeping the data to the size of the mantissa, Test SLATE guarantees that the number will be represented exactly, therefore not losing any accuracy.

The following figures are the encoded representations for each of the Test SLATE Time Tags:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Sign | Exponent | | | | | | | | Mantissa | | | | | | | | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 09 | 08 | 07 | 06 | 05 | 04 | 03 | 02 | 01 | 00 |
| Not Used | | | | | | | | | | | Month | | | | Day | | | | | Year | | | | | | | | | | | |

Figure 2 Date Encoded into a Float (Date\_*Source*)

To encode date into this format using C# the following formula can be used:

Date\_*Source* = ((float) ((month << 17) + (day << 12) + year)));

To decode date items from this format using C# the following formulae can be used:

UInt32 Month = (((UInt32)Convert.ChangeType(Date\_*Source*, typeof(UInt32))) & 0xFFFE0000) >> 17);

UInt32 Day = (((UInt32)Convert.ChangeType(Date\_*Source*, typeof(UInt32))) & 0x0001F000) >> 12);

UInt32 Year = ((UInt32)Convert.ChangeType(Date\_*Source*, typeof(UInt32))) & 0x00000FFF);

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Sign | Exponent | | | | | | | | Mantissa | | | | | | | | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 09 | 08 | 07 | 06 | 05 | 04 | 03 | 02 | 01 | 00 |
| Not Used | | | | | | | | | | | | | | | Hour | | | | | Minute | | | | | | Second | | | | | |

Figure 3 Time Encoded into a Float (Time\_*Source*)

To encode time into this format using C#, the following formula can be used:

Time\_*Source* = ((float)((Hour << 12) + (Minute << 6) + Second));

To decode time items from this format using C#, the following formulae can be used:

|  |
| --- |
| UInt32 Hour = (((UInt32)Convert.ChangeType(Time\_*Source*, typeof(UInt32))) &  0x0001F000) >> 12; |
| UInt32 Minute = (((UInt32)Convert.ChangeType(Time\_*Source*, typeof(UInt32))) &  0x00000FC0) >> 6; |
| UInt 32 Seconds = (((UInt32)Convert.ChangeType(Time\_*Source*, typeof(UInt32))) &  0x0000003F); |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Sign | Exponent | | | | | | | | Mantissa | | | | | | | | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 09 | 08 | 07 | 06 | 05 | 04 | 03 | 02 | 01 | 00 |
| Not Used | | | | | | | | | | | | | | | | | | | | Milliseconds | | | | | | | | | | | |

Figure Milliseconds Encoded into a Float (Milliseconds\_*Source*)

To store milliseconds into this float just assign the milliseconds value into it (e.g. Milliseconds\_*Source* = (float) milliseconds)

To extract milliseconds from this tag just assign the float value to your variable (e.g. UInt32 milliseconds = (UInt32)Milliseconds\_*Source*;).

### Channel Status

Test SLATE provides Health Monitoring functionality that includes the Channel Status for all Tags in a current test. The Health Monitor (HM) Source provides the reporting and user interface functionality for the status information, but each Source is responsible for providing status for certain events and conditions to the Health Monitor as described in the Table below.

|  |  |  |
| --- | --- | --- |
| Status | Description | Status Bit |
| Saturated | The Tag has reached the maximum or minimum reading possible for the Channel. Possible Source checks for Saturation include:   * Comparing maximum/minimum expected Raw Data values to hardware specified values. * Monitoring hardware status information for vendor-supplied saturation status * Comparing user-specified Channel High Range and Low Range values to current Raw Data.   The actual check used may be different for each Source and should be documented. | 0 |
| Conversion Error | The Raw Data provided by the Source for the Tag is suspect due to:   * An error converting the data provided by hardware into Raw Data for Test SLATE * An error generating a calculation for a Tag * An error indication from a hardware or interface providing Raw Data to Test SLATE.   The actual check used may be different for each Source and should be documented. | 1 |
| In Calibration | The Tag or its Source is currently performing a calibration. This status signifies that the data may not be actual signal data, but calibration feedback values. | 2 |
| In Diagnostic | The Tag or its Source is currently in a diagnostics mode. This status signifies that the data may not be actual signal data, but diagnostic information. | 3 |
| Bad Coded | Users may select to Bad Code any Channel. The status is intended to denote a Channel that is either suspect or invalid. The Bad Code information should be returned to the Source via the RunSet during testing. | 4 |
| Inactive | The Channel is not updating or is in an error condition. This is usually set when the Source is Inactive or the Channel’s current value is “Not a Number” (NaN). | 5 |

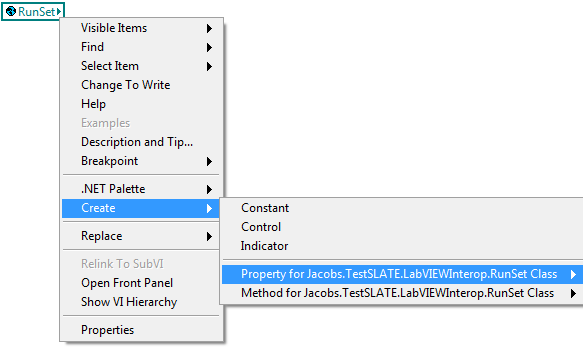
At each Source update interval, the current status of each Tag/Channel should be updated to reflect whether any of the conditions above are true. Each Tag has a separate U16 numeric assigned to it in the status arrays. When a condition is true, the bit corresponding to the Status Bit in the table above should be set to a 1. Multiple status values may be true at the same time.

### Reflective Memory

This section is TBD.

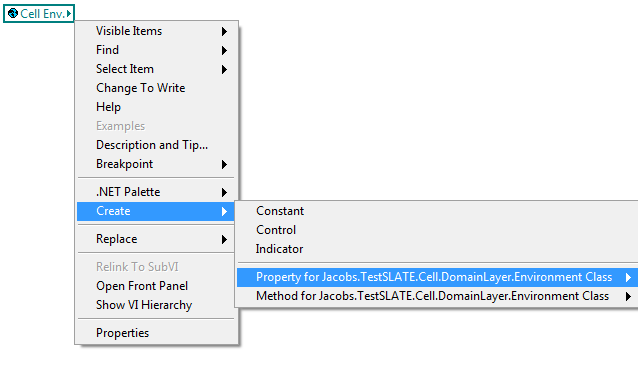
### RunSet

The interface to the Test SLATE Runset when using LabVIEW is via the Runset Global located in the Test SLATE Application palette. This global provides the .NET connection to the current Test SLATE RunSet. The RunSet provides the read-only information for the System, Configuration, and Sensors used in the current test.



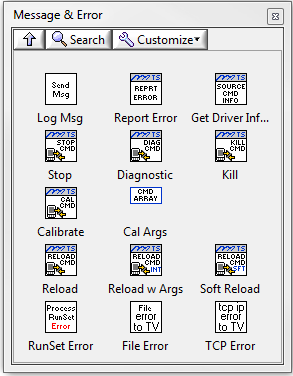
### Database Access

The interface to the Test SLATE Databases when using LabVIEW is via the Environment Global located in the Test SLATE Application palette. This global provides the .NET connection to the current Test SLATE Domain Layer for read and write access to the TV and Facility databases.



### Commands

LabVIEW VIs to support Source Commands are located on the Test SLATE\Message & Error palette. These Commands are normally not necessary unless the Source must perform special Calibration and/or Diagnostic functions. In that case, the Calibrate and Soft Reload Commands are often required.

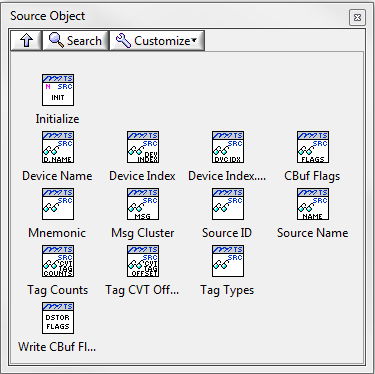


| Item | Description |
| --- | --- |
| Get Driver Info | * Returns the connection information for the requested Source Name, including the IP Address. |
| Stop | * Sends a Stop Command to the requested Source Name. * Used to exit Acquire, Diagnostics, or Calibration and return to Idle Mode. |
| Diagnostic | * Sends a Diagnostic Command to the requested Source Name. * Used to Exit idle mode and collect data for Diagnostics. * When in this mode, the source is reserved. It will not accept commands for acquire, calibrations, etc. The Stop command is used to exit this mode. |
| Kill | * Sends a Kill Command to the requested Source Name. * Used to stop collecting data and enter inactive mode. * Only accepted when in idle mode. |
| Calibrate | * Sends a Calibrate Command to the requested Source Name. * Used to Exit idle mode and collect data for calibration. * When in this mode, the source is reserved. It will not accept commands for acquire, diagnostics, etc. The Stop command is used to exit this mode. * The Calibrate command usually contains Integer Arguments which specify the type of calibration. These must be set using Cal Args prior to calling the Calibrate Command. |
| Cal Args | * Argument = 0, Perform End to End calibration. In this calibration type, the source should output data as it does in idle mode. * Argument = 1, Special 1. This calibration mode is generic and can be used for calibrations specific to your source. * Argument = 2, Special 2. This calibration mode is generic and can be used for calibrations specific to your source. * Argument = 3, Special 3. This calibration mode is generic and can be used for calibrations specific to your source. * Argument = 4, Waiting for Type calibration. This is the first mode that your source will be set to upon entering calibration mode. In this calibration type, the source should output data as it does normally in idle mode. * Arguments 0 to 3 are only accepted when the source is in the Waiting for Type calibration mode. |
| Reload | * Sends a Reload DB Command to the requested Source Name. * Used to send the Source to the INIT state. |
| Reload w Args | * Sends a Reload DB Command to the requested Source Name. * Used to send the Source to the RELOAD DB state. * This command also includes integer arguments that can be passed to the Source to modify the Reload settings. The first argument is used to specify whether a Soft Reload is requested (value =1 for Soft Reload, otherwise set to 0). |
| Soft Reload | * Sends a Soft Reload Command to the requested Source Names. In addition to the Sources receiving the Soft Reloads, all other Test SLATE components will be notified that new conversion information may be available. |

## LabVIEW Classes

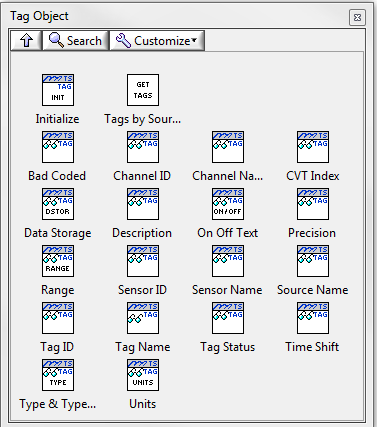
### Source

The LabVIEW Source Class provides information about a particular Source including its Name, starting CVT Index, Data Types, Device Index, and Mnemonic.



### Tag

The LabVIEW Tag Class provides information about a particular Tag including Tag Name, CVT Index, Data Type,Type Index, Channel, Sensor, Units, and Status.



# Building and Installing Plugins

## Manual Builds

Builds may be performed manually using the build utilities in the Visual Studio Solution and the LabVIEW project.

## Help

Most users welcome help and information about each of their Sources. You can add your own help to the Test SLATE Help by adding the files to the help folder in the Test SLATE Installation folder. Use the **v1.0\help\Help Document Example.doc** to generate your help file and convert it to PDF prior to performing an Automated Build or placing the file into Test SLATE Help.

You can add a link to your new help document so that it can be viewed when Test SLATE Help is invoked from the Test SLATE Help Menu. You must have a PDF editor (e.g. Adobe Acrobat Standard) to prepare your help document for Test SLATE Help. The example below was done with Acrobat X Standard.

To add your help document to Test SLATE Help:

1. Convert your MS Word help document to PDF (e.g. convert “MyPluginName Help.docx” to “MyPluginName Help.pdf”).
2. Move your PDF file (“MyPluginName Help.pdf”) to this folder:

C:\Program Files (x86)\Jacobs\Test SLATE\help\Customer Specific Help\

1. Using your PDF Editor, open the PDF file and then show Bookmarks.
2. Add a bookmark, move it to the top of the Bookmark list, and rename it to “<BACK> to Specific Help”.
3. Add a bookmark link to call the Test SLATE Customer Specific Help:
   1. Right-click on the new bookmark, and select Properties…
   2. On the Bookmark Properties window, go to the Actions tab.
   3. In the Actions pane, select any Actions that are listed. Press the Delete button to remove the action(s).
   4. In the Add an Action pane, select “Open a file” from the dropdown. Click the Add… button. Navigate to C:\Program Files (x86)\Jacobs\Test SLATE\help\Customer Specific Help\ , select the file Test SLATE CUSTOMER SPECIFIC HELP.pdf, then press the Open button.
   5. When prompted to Specify Open Preference, select Existing window, then press the OK button.
   6. Click the OK button on Bookmark Properties.
4. Set the PDF File Properties from the menu by selecting File->Properties…
   1. In the Document Properties window, go to the Initial View tab.
   2. In the Layout and Magnification pane Navigation tab selection, select Bookmarks Panel and Page. Click the OK button.
5. Save your PDF file.
6. Next, you must add a link from the Test SLATE Customer Specific File to your new PDF file.
   1. Using your PDF Editor, open C:\Program Files (x86)\Jacobs\Test SLATE\help\Customer Specific Help\Test SLATE CUSTOMER SPECIFIC HELP.pdf
   2. Add a bookmark, move it to the location you want in your Bookmark list, and rename it for your new plugin (e.g., “MyPlugInName”).
   3. Right-click on the new bookmark, and select Properties…
   4. On the Bookmark Properties window, go to the Actions tab.
   5. In the Actions pane, select any Actions that are listed. Press the Delete button to remove the action(s).
   6. In the Add an Action pane, select “Open a file” from the dropdown. Click the Add… button. Navigate to C:\Program Files (x86)\Jacobs\Test SLATE\help\Customer Specific Help\, select the PDF file you created as your plugin help document (e.g. MyPluginName Help.pdf), then press the Open button.
   7. When prompted to Specify Open Preference, select Existing window, then press the OK button.
   8. Click the OK button on Bookmark Properties.
7. Save the file Test SLATE CUSTOMER SPECIFIC HELP.pdf.

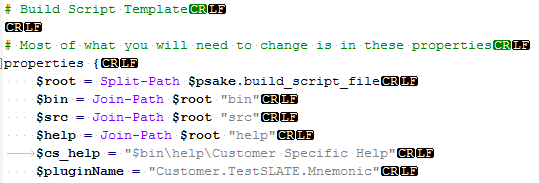
You can test your links by opening Test SLATE Help using Acrobat.

## Automated Builds

The template includes build files designed to run within Powershell. These files may be modified to meet your specific source needs but the major components are included in the templates supplied.

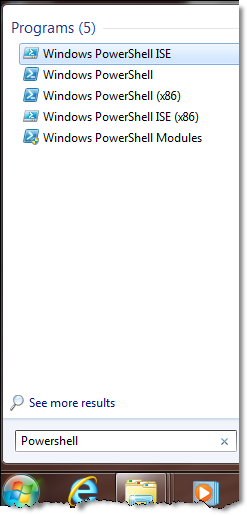
To set the correct Plugin name within the automated build file:

1. Open **v1.0\default.ps1** in a text editor.
2. Locate the property **$pluginName** and modify it to **MyPlugInName**.
3. Save the file.

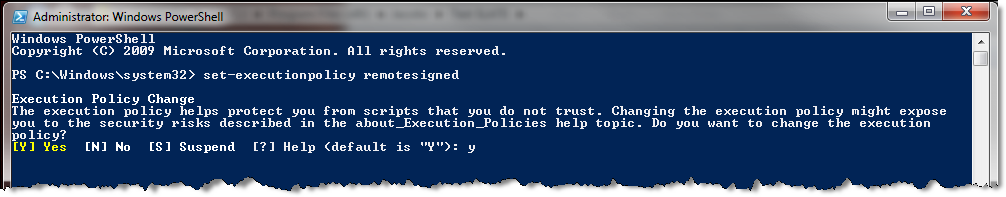


### Before your first build on a Computer

1. Left-click the **Start Button** and type <**powershell>**. This should provide a list of PowerShell related items.

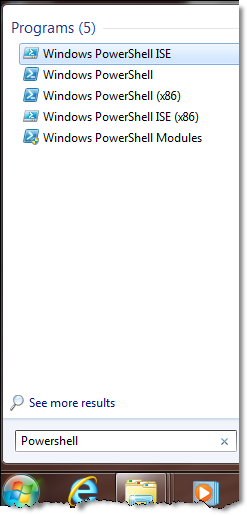


1. **Right-click** on the "**Windows PowerShell**" and select "**Run as Administrator**".
2. At the Prompt type "**set-executionpolicy remotesigned**" and press **Enter.**
3. When prompted, answer **“Y”.**

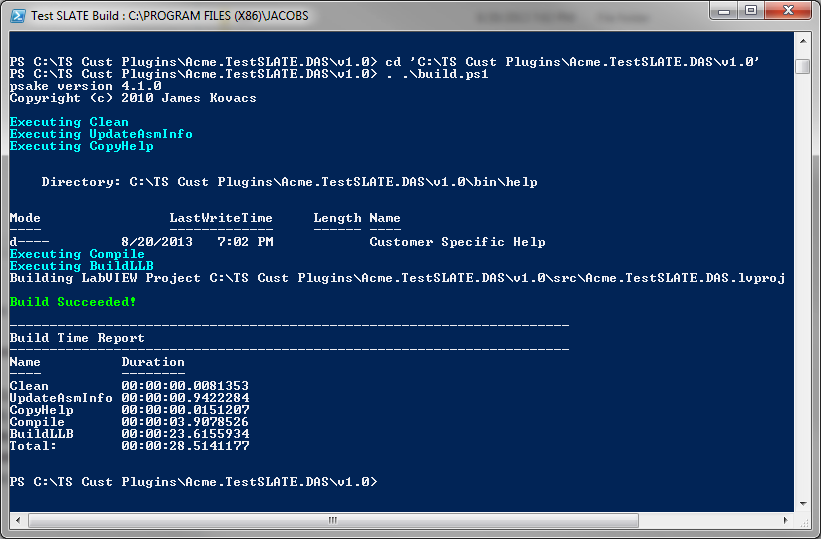


### Executing the Source Build Scripts

1. Left-click the **Start Button** and type <**powershell>**. This should provide a list of PowerShell related items.



1. Navigate to the **v1.0** folder.
2. At the prompt type "**. .\build.ps1**" (period space period backslash build.ps1) and press **Enter.**

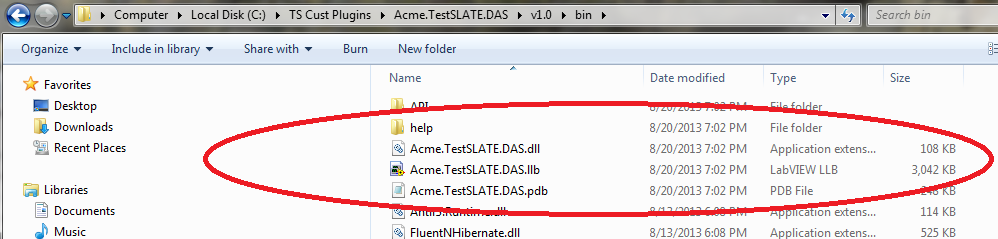


## Inclusion of Plugins to Test SLATE

After the successful build of the plugin, the associated files must be copied into the working Test SLATE directory to enable them.

The number and types of files required are specific to each plugin, but normally two files (a \*.dll and \*.llb) must be copied. In addition, source specific help files should be copied. The example below illustrates the Acme.TestSLATE.DAS used in this example.

1. Navigate to the **v1.0\bin** folder.
2. Copy the following files:
   1. **Acme.TestSLATE.DAS.dll**
   2. **Acme.TestSLATE.DAS.llb**
   3. **Help\**



1. Paste the files onto each Test SLATE Computer (MOC, DTM, and AOCs) at the following path:

C:\Program Files (x86)\Jacobs\Test SLATE