

Technical Reference



Tektronix Logic Analyzer Family TLA Plug-In Design Guide Volume 3

071-1353-00

This document applies to TLA System Software
Version 4.3 and above.

www.tektronix.com

Copyright © Tektronix, Inc. All rights reserved. Licensed software products are owned by Tektronix or its suppliers and are protected by United States copyright laws and international treaty provisions.

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, or subparagraphs (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, as applicable.

Tektronix products are covered by U.S. and foreign patents, issued and pending. Information in this publication supercedes that in all previously published material. Specifications and price change privileges reserved.

Tektronix, Inc., P.O. Box 500, Beaverton, OR 97077

TEKTRONIX and TEK are registered trademarks of Tektronix, Inc.

TLA Plug-In Design Guide

1	Introduction	2
2	Implementing Plug-Ins	4
2.1	Software Implementation	4
2.2	Plug-In User Interfaces	6
2.3	Generic or "Tool" Plug-Ins	8
2.4	Data Window Plug-Ins	8
2.5	Data Source Plug-Ins.....	10

1 Introduction

Plug-Ins Defined

Plug-ins are managed .NET classes that implement the IPlugIn interface or an interface derived from it. The IPlugIn interface is defined by the TLA application software. Since plug-ins must implement specific interfaces defined by the application, they have a subset of known properties, methods, and events. The application uses these known members to initialize and interact with plug-in instances. In turn, plug-in instances extend the functionality of the TLA application.

As managed types, plug-ins implementations are contained within assembly DLLs. A plug-in class becomes "plugged-in" when the assembly containing it is placed into the TLA700\System\PlugIns directory of the application installation.

Development Overview

Plug-ins are developed by creating managed classes that implement specific .NET interfaces. Before a plug-in can be fully implemented, the following items are needed by the developer:

- A .NET enabled compiler, Visual Studio .NET for example.
- A copy of TlaNetInterfaces.dll.

All plug-in developers must use .NET-enabled compilers, which can create assemblies with code that executes under the Common Language Runtime. The NET interface specifications for the TLA application have been written with the expectation that plug-ins will be developed most often with Microsoft's Visual Basic, C#, or Managed C++.

In order to create a plug-in class, the developer must have a copy of TlaNetInterfaces.dll, which is a DLL present in every V4.3 TLA software installation. It is an assembly that contains metadata for all plug-in interfaces and TPI.NET types. The file holds the compiled definitions of all .NET types that are currently described in "DotNet Interfaces UIS.doc." Both the TLA application and all plug-in assemblies are compiled using references to the metadata in this assembly.

TlaNetInterfaces.dll serves a purpose similar to that of a header file. It holds the definitions of types that will be used by multiple projects, including projects that are compiled separately. Since plug-in code must be programmed and compiled using metadata from this assembly, TlaNetInterfaces.dll will be provided to all plug-in developers.

Version 4.3 of the TLA application will recognize three kinds of plug-ins: Generic plug-ins, data window plug-ins, and data source plug-ins. These plug-ins implement IPlugIn, IDataWindowPlugIn, and IDataSourcePlugIn respectively; and a particular plug-in distinguishes itself to the TLA by the interfaces that it implements. The application determines which interfaces a plug-in implements by examining its metadata, specifically by looking at the Type object associated with the plug-in class. To qualify as a plug-in, a class must implement IPlugIn. Generic plug-ins directly implement this base interface and no derived interfaces. Data window and data source plug-ins implement specialized interfaces derived from IPlugIn.

Plug-in instances are expected to need relatively detailed information about System setup and module data. This information is supplied by the TPI.NET programmatic interface, which exposes major elements of a TLA as a hierarchy of managed objects. When plug-ins are initialized, they are given a reference to an ITlaPlugInSupport object, which is at the very top of the TPI.NET hierarchy. This means that all plug-in objects are also in-process TPI.NET clients, and plug-in developers have access to all facilities of TPI.NET without needing to write any special code to connect to the programmatic interface.

Installation

A plug-in is installed by copying the assembly that contains it into the TLA700\System\PlugIns directory or one of its subdirectories. At start-up the application searches the PlugIns directory and all its subdirectories for .NET assemblies. The metadata in each assembly is examined; and if the assembly contains any classes that implement IPlugIn, those classes are added to a runtime list of installed plug-ins. Installing a plug-in class does not necessarily mean that any objects of that class will be instantiated.

Instantiation

The TLA application is responsible for instantiating plug-ins at appropriate times. These are the situations in which new plug-in instances are created:

- The plug-in class attributes indicate that an instance should be automatically created when the application starts up.
- The user requests a new instance from the application UI.
- A TPI.NET client requests a new instance by calling ITIaSystem.CreatePlugInInstance.

In the above situations, the application calls Assembly.CreateInstance to create instances of plug-in classes. Plug-in created this way are instantiated using their default constructor, so every plug-in class must implement a constructor that takes no arguments. Plug-ins are not required to take any specific action within their default constructors, but the constructor must exist in order to instantiate the object. Once a plug-in is instantiated, the application guarantees that the IPlugIn.Initialize method will be called before any other member methods are called.

A serializable plug-in that implements ISerializable can also be re-instantiated using its deserialization constructor in the following situations.

- Saved plug-ins are deserialized when the saved system that contains them is restored.
- A saved data window plug-in can be deserialized when a user selects it from the Load Data Window dialog.
- A saved data source plug-in can be deserialized when selected from the Add Data Source dialog.

Plug-In save/load is described inside section 2.1. Save and restore of data source plug-in data is described inside section 2.5.

Samples

The same CD that contains this document also contains sample code for a generic tool plug-in and a simple data window plug-in. These samples are presented in three separate languages supported by Microsoft Visual Studio .NET: Visual Basic .NET, C#, and C++ with managed extensions. Each sample has a solution file that can be opened and built to produce a working plug-in. To be used, the plug-in needs to be installed in the following directory of a TLA software installation:

C:\Program Files\TLA 700\System\PlugIns

The tool plug-in sample is contained in the following directories:

TPI.NET and PlugIn Documentation\Tool PlugIn Samples\CSharp Tool PlugIn,
TPI.NET and PlugIn Documentation\Tool PlugIn Samples\VB Tool PlugIn,
TPI.NET and PlugIn Documentation\Tool PlugIn Samples\CPP Tool PlugIn.

The data window plug-in sample is contained in the following directories:

TPI.NET and PlugIn Documentation\Data Window PlugIn Samples\CSharp Data Window PlugIn,
TPI.NET and PlugIn Documentation\Data Window PlugIn Samples\VB Data Window PlugIn,
TPI.NET and PlugIn Documentation\Data Window PlugIn Samples\CPP Data Window PlugIn.

To use these sample projects, copy the directory contents of the desired sample from the CD into a directory on the computer that contains your development environment.

Please note that all the Visual Studio projects contain a pre-release version of the TlaNetInterfaces.dll assembly. If you experience any runtime problems related to TlaNetInterfaces.dll, you might get better results by compiling the samples, and any other TPI.NET clients, with the final release version of this assembly. The release version is always installed in the following directory of your TLA application software:

C:\Program Files\TLA 700\System\TlaNetInterfaces.dll

2 Implementing Plug-Ins

This section describes how to implement IPlugIn so that the resulting class can be installed for use with the TLA application. The techniques and requirements presented in this section are applicable to developing all kinds of plug-ins. Implementation specifics for the three different kinds of plug-ins are described in later subsections.

2.1 Software Implementation

Minimum Requirements

The following list summarizes the minimum requirements for a plug-in class implementation.

- The class must be a managed .NET class, meaning it's code executes under the management of the Common Language Runtime.
- The assembly containing the class implementation must be installed in the TLA700\System\PlugIns directory or in a subdirectory.
- The class must implement IPlugIn or implement an interface derived from IPlugIn.
- The class must have public visibility.
- The class must have been compiled with a copy of TlaNetInterfaces.dll whose version number is less than or equal to the version of TLA700.exe.
- The class must have a default constructor.

The TLA application will be able to find, recognize, and instantiate a plug-in class that meets all the above requirements.

Custom Attributes Applicable to Plug-Ins

TlaNetInterfaces.dll defines two custom attributes that plug-in developers can apply to the classes they design. Note that these attributes are applied to plug-in classes and not to instances of classes. These attributes are used to describe a plug-in class to the TLA before any objects of the type are created.

PlugInIdentityAttribute is used to specify the name and icon that the TLA should use to represent a plug-in class to users. The name and icon will appear in the TLA UI in places where the user can create instances of plug-ins. The kind of plug-in determines where the name appears in the application UI. This attribute is required for any kind of plug-in to be instantiated from the UI. When PlugInIdentityAttribute is absent from a plug-in class, the plug-in will not appear in any menus, toolbars or dialog boxes.

PlugInInstantiationAttribute: Developers use this attribute to control whether the TLA automatically creates an instance at startup and whether the TLA will allow more than one instance of the plug-in to be created.

Implementing IDisposable and IValidity

IPlugIn derives from IDisposable, which is a standard interface defined by the .NET Framework. IDisposable has only one method, namely the Dispose method. When this method is called on an implementing class, it must free all its unmanaged resources, such as file handles, device contexts, or unmanaged memory. In the implementation of its own Dispose method, a plug-in must call Dispose on all the visual components it has instantiated, such as Form and Control objects.

In addition to IDisposable, plug-ins must also implement the IValidity interface defined in TlaNetInterfaces.dll. This interface has two properties and an event. The IsValid and IsGarbage properties indicate whether the object is valid for use by other objects. When either of these properties change, the implementing object must raise the IValidity.ValidityChanged event. The TLA application subscribes to the ValidityChanged event, and removes any plug-in whose IsValid property has become false. In addition the TLA calls Dispose on any object that it removes from the system.

Plug-in implementations need to coordinate IValidity with IDisposable. Whenever a plug-in is disposed, it must set IsValid to false, set IsGarbage to true, and raise the ValidityChanged event.

Multi-threading

Plug-ins are expected to be single-threaded. If a plug-in attempts to be multithreaded, it must protect against deadlock. Any call from the TLA application, including event callbacks, must complete without blocking the thread on which the call is made.

Plug-In Deletion

Plug-ins will sometimes need to delete themselves from the system. For example if a data window plug-in is closed by the user, it is reasonable for the plug-in to remove itself from the system.

The recommended way for a plug-in to delete itself is by changing its IsValid property to 'false', then raising its ValidityChanged event. The TLA application will respond to the event by removing the plug-in from the system.

Implementing Plug-In Save/Load

The V4.3 TLA application supports saving and loading the state of plug-ins to and from a .tla setup file. Supporting save/load is optional for plug-in classes. A plug-in supports being saved and loaded by both implementing the .NET Framework interface ISerializable and applying the Serializable attribute to the implementation class.

To properly implement a serializable plug-in, the class must fulfill these requirements:

- The SerializableAttribute must be applied to the plug-in class.
- The class must derive from ISerializable and implement this interface.
- The class must implement the method GetObjectData in a way that is consistent with the .NET Framework's defined semantics for that method.
- It must implement a deserialization constructor with the signature Constructor(SerializationInfo info, StreamingContext context). This constructor is in addition to the default constructor required of all plug-in classes.

From the plug-in perspective, a save is simply a call made on its implementation of GetObjectData(). The plug-in is expected to place its state into a serialization stream by calling SerializationInfo.AddValue(). The SerializationInfo object is provided as one of the GetObjectData() arguments. Note that acquisition data for a data source plug-in should not be

saved when `GetObjectData()` is called. Data from data source plug-ins are saved using a separate mechanism as described in section 2.5 on data source plug-ins.

A plug-in is restored by instantiation of the class through its deserialization constructor. The plug-in is expected to deserialize its model state using the `SerializationInfo` object provided as a deserialization constructor argument. A data source plug-in should not attempt to de-serialize acquisition data at this step – only setup data. Acquisition data are restored using the `ISaveData` interface, which is only applicable to data source plug-ins.

2.2 Plug-In User Interfaces

The TLA application provides mechanisms for plug-ins to integrate their user interfaces into the user interface of the application. By using these mechanisms and by adhering to TLA application conventions, a plug-in can present a UI that has appearance and behavior consistent with built-in window in the system.

Getting Forms to Appear in The Main Application Frame

Many plug-in developers will want to create windows that are children of the main application MDI frame window. This is complicated by the fact that V4.3 of the application is not a pure .NET application, and the main window is not a Form object. To integrate Form windows into the main window, TPI.NET provides the method `ITlaPlugInSupport.TopLevelFormToChild`. This method takes a Form object and makes it visually appear like other MDI child windows in the TLA application UI. It does this by embedding the form object inside a standard windows MDI child frame.

When Form objects are placed inside the main application window by calling `ITlaPlugInSupport.TopLevelFormToChild()`, certain Form properties are used to determine the appearance and behavior of the Form and the frame window that surrounds it. The following table describes the properties used and their affects on the user interface:

Form.Text	Determines the title bar text of the frame that surrounds the form.
Form.Icon	Determines the icon used in the frame that surrounds the form.
Form.Menu	Determines the menu items that are added to the TLA menu bar. All menus in the Form will be inserted into the application menu between the File and System menus. Note that the .NET menu merging is not used and the 'MergedMenu' property is ignored.
Form.ControlBox	Determines whether or not the window can be closed by the user. If this is set to true, then the 'X' button, the 'Close' system menu command and the 'Delete Window' menu commands are all enabled. If the value of this property is false, then those UI elements are all disabled. Note that Forms that become children of the main application frame always have enabled minimize and maximize buttons.
Form.Size	The value of this property determines the initial dimensions of the client area of the frame window that surround the Form. After a plug-in has been made a child of the main frame, changing the size of the form will also change the size of the frame around it. Likewise, when the user changes the size of the frame, the embedded Form will also be sized to match.
Form.WIndowState	If the Form is a <code>TlaForm</code> , then this value determines whether the initial state of the child window is Maximized, Minimized, or Normal. If the Form is not a <code>TlaForm</code> , then <code>TopLevelFormToChild</code> sets this to Normal; and the initial state of the child window is Normal.

Form.TopLevel	TopLevelFormToChild sets this to false.
Form.ShowInTaskbar	TopLevelFormToChild sets this to false.
Form.FormBorderStyle	This is set to false. A new frame in the style of a data window or module setup window frame is created and the Form window is embedded within the new frame.
Form.MdiParent	This property is set to null. Although the Form will appear like an MDI child. It will not have a .NET MDI parent.

The TlaForm Class

To tighten the integration of the plug-in-created Form objects into the main window, the TlaNetInterfaces.dll implements a TlaForm class that directly inherits from Form. Plug-ins should derive their Form-based windows from TlaForm instead of Form. TlaForm overrides of some Form methods, allowing it to behave correctly as MDI child of the main application window even though the main window is not a .NET Form.

TlaForm overrides the following Form methods:

```
Activate()
BringToFront()
Hide()
SendToBack();
Show();
```

TlaForm overrides the following Form property:

```
WindowState
```

Unless a Form object is a TlaForm, a plug-in should not use the members listed above after performing the ITlaPlugInSupport.TopLevelFormToChild() operation. Using these members on a non-TlaForm object will have undesirable affects on the UI. Deriving from TlaForm allows the overridden members to work as expected on TlaForm objects that are children of the main frame. If a plug-in implements a TlaForm-derived class and overrides any of these methods, the derived implementation should call the base TlaForm method to ensure the best UI integration.

Deriving from TlaForm also allows the 'Help on Window' menu item to be enabled. TlaForm has two properties that determine the help to be shown when the 'Help on Window' command is given. TlaForm.HelpFile is a string property that contains the full path of the Windows Help (.hlp) file that is associated with the window. TlaForm.HelpID is an integer property that indicates the topic within the help file that should be displayed. If HelpFile is not null or empty, and if HelpID is a positive value, the 'Help on Window' menu item will be enabled for the child window. When the window containing the Form is active, the 'Help on Window' command will open the HelpID topic in the HelpFile.

All Tektronix plug-ins that place Form objects in the main window, need to derive those objects from TlaForm and set the HelpFile and HelpID properties. This will make the Form child window consistent with the built in data and setup windows, all of which support the 'Help on Window' command. In most cases Tektronix plug-ins will specify a help topic from the standard TLA700.hlp file.

Message Boxes

Message boxes produced by the TLA application usually have the title 'TLA'. Message boxes produced by plug-ins should use the name of the plug-in as the title in order to differentiate the source of the message. If the plug-in has a PlugInIdentityAttribute, the Name property should be used as the title. If a plug-in does not have this attribute, then it should consistently use a name that helps identity which plug-in is responsible for the message.

2.3 Generic or "Tool" Plug-Ins

Generic plug-ins are managed classes that implement `IPlugIn` and do not implement any derived interfaces such as `IDataWindowPlugIn`. When a generic plug-in is decorated with the `PlugInIdentity` attribute, the plug-in class is revealed to users through the UI.

V4.3 of the TLA application allows generic plug-ins to be instantiated from a Tools menu and a Tools toolbar. The `PlugInIdentityAttribute` affects how, and if, a generic plug-in class is presented to the user in the TLA user interface. If a generic plug-in class does not have the attribute, the TLA user interface will provide no means for users to instantiate it. If the class is marked with `PlugInIdentity` attribute, then an entry for the plug-in is made in the Tools menu and Tools toolbar using the `Name` property of the attribute. Clicking the menu item (or toolbar button) causes the TLA to instantiate the plug-in class.

`PlugInIdentityAttribute` can also be used to group related plug-ins. This is done by setting the `GroupName` parameter in the attribute declaration. The example below declares a user-friendly name and a group name for a plug-in class in C#.

```
[PlugInIdentity("Run Monitor", GroupName="Run Analysis Tools")]  
public class RunMonitorPlugIn : IPlugIn { ... }
```

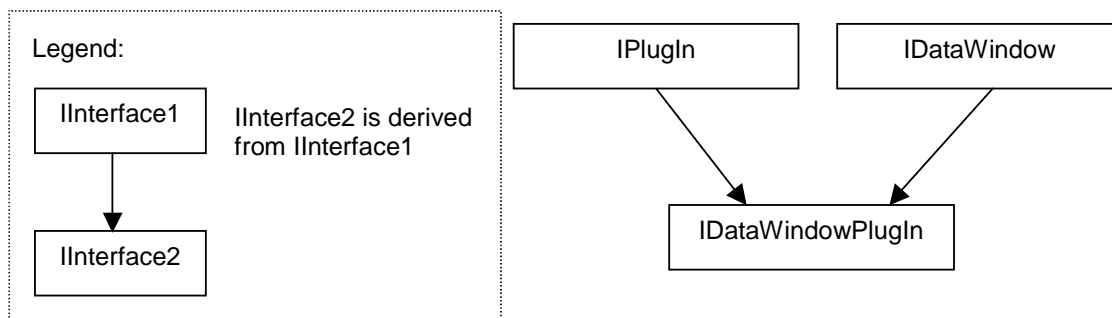
For generic plug-ins, the `GroupName` becomes a pop-up menu item in the Tools menu. All plug-ins that share a `GroupName` will be placed in the pop-up menu that appears when the `GroupName` menu item is selected.

2.4 Data Window Plug-Ins

Plug-ins that behave like data windows implement the `IDataWindowPlugIn` interface. This interface derives from the same `IPlugIn` interface that all plug-ins must implement. `IDataWindowPlugIn` has additional members that allow it to be integrated into the system much like a Listing, Histogram, and other built in kinds of data windows.

Required Interfaces

Due to interface inheritance, developers who implement `IDataWindowPlugIn` must actually implement several interfaces in order to achieve proper behavior; and they can optionally implement more. The diagram below shows the minimum interface hierarchy for data window plug-ins.



`PlugInIdentityAttribute` and Instantiation from the TLA UI

Users will be able to instantiate data window plug-ins from the New Data Window dialog box. Classes that implement `IDataWindowPlugIn` can control how they are presented in the dialog by using the `PlugInIdentityAttribute`, which specifies the name, icon, and group associated with the plug-in class. All data windows that can be instantiated by the TLA will be listed in the dialog. The Name of data window plug-ins that have an associated `GroupName` will be shown as children of a parent node in a tree control. The `GroupName` will be a root parent, and the `Name` will be a leaf node. The absence of `PlugInIdentityAttribute` prevents the plug-in from participating in the dialog.

A user commands the TLA to instantiate a data window plug in by selecting it in the New Data Window dialog, then pressing OK. When a plug-in is instantiated, the TLA passes the plug-in `Initialize()` method a Boolean argument to indicate instantiation from the UI. When implementing the `Initialize` method of a data window plug-in, it is important to check this argument.

If a data window plug-in is instantiated from the UI, it can present the interactive user with a wizard that helps the user set up the initial settings and data sources displayed in the window. However, if the TLA indicates that the class was not instantiated from the UI (meaning it was instantiated programmatically), the plug-in must not display any form of modal dialog. Doing so will cause the application to hang while waiting for user input.

System Window User Control

By implementing `IDataWindowPlugIn`, a class gains the ability to place a control in the System window. The plug-in is responsible for creating a `UserControl`-derived object, and the TLA application is responsible for showing the control in an appropriate place in the System window.

The System window control is implemented as a .NET control of type `System.Windows.Forms.UserControl`. Developers using C# or VB should note that they can use a graphical designer tool to create the visual look of this control. At runtime, the TLA application obtains a reference to the instantiated control through the property `IDataWindowPlugIn.SystemWindowControl`. The plug-in implementation must set this property within the `Initialize()` method. If `SystemWindowControl` is null after the TLA calls `Initialize()`, then no icon for the data window will appear in the System window.

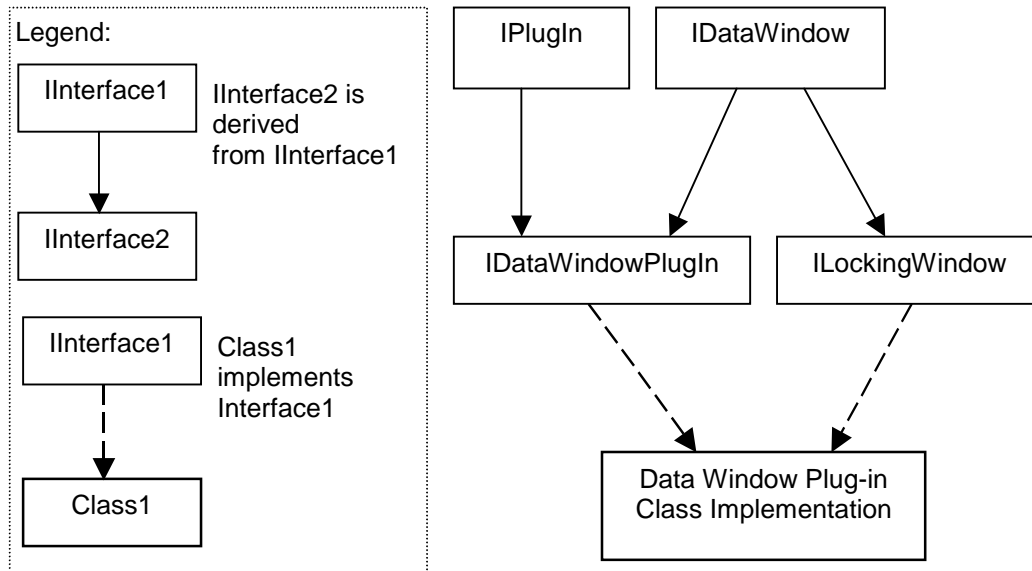
Obtaining Data

TPI.NET defines a set of interfaces that work together to form a standardized representation of module data. Data sources that provide this standard implement `IStandardDataSource`. TPI.NET represents non-plug-in modules with objects that implement the standard. All `ILAModule`, `IDSOModule`, and `IExternalOscilloscopeModule` objects implement this interface. Data window plug-ins that use standard data sources obtain data by obtaining `IAcquisitionData` objects from `IStandardDataSource` objects. `IAcquisitionData` objects, in turn, can create `IRecordAccess` objects that provide access to acquisition data.

Since data source plug-ins are not required to implement `IStandardDataSource`, the representation of data provided by plug-ins can vary.

Implementing a Locking Data Window Plug-In

To achieve more specialized behavior, data window plug-ins can implement any interface that derives from `IDataWindow`. An important example is the `ILockingWindow` interface. Data window plug-ins that implement `ILockingWindow` can participate in the Lock Window dialog box, which time-locks the cursors and data displays of a set of data windows. The following diagram shows that data window plug-ins that can be locked to other windows inherit from both `IDataWindowPlugIn` and `ILockingWindow`. The following diagram illustrates the relationships among the implemented interfaces.



Note that, due to multiple inheritance of interfaces, the plug-in implemented in the above diagram appears to inherit IDataWindow twice. This kind of interface inheritance is allowed in .NET classes, and the derived class implements a multiply inherited interface only once.

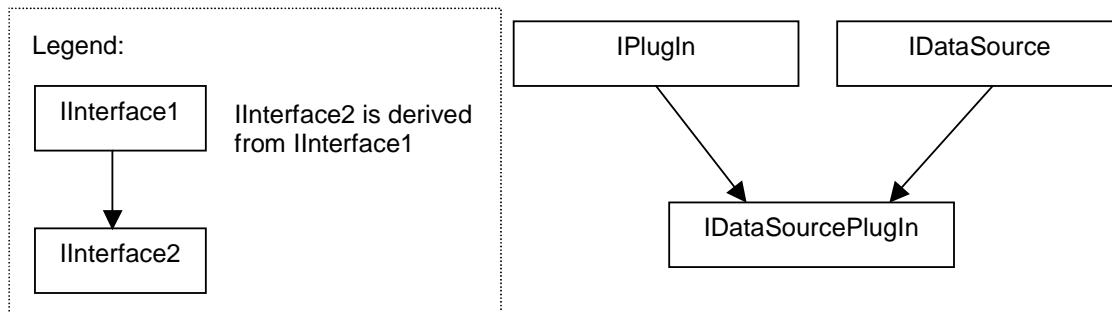
The ILockingWindow interface uses timestamp values to define the positions of cursors and the display center. In order to achieve proper locking, the data window must be able to move cursors based on timestamp values. This also implies that the data sources being displayed have meaningful sample timestamps or that the data window plug-in can manufacture meaningful timestamps from the underlying data.

2.5 Data Source Plug-Ins

Data source plug-in are specialized classes that behave like other data sources in the system, providing data that can be consumed by data windows and TPI.NET clients. These classes implement the IDataSourcePlugIn interface.

Required Interfaces

Due to interface inheritance, developers who implement IDataWindowPlugIn must actually implement several interfaces in order to achieve proper behavior. The diagram below shows the minimum interface hierarchy for data source plug-ins.



PlugInIdentityAttribute and Instantiation from the TLA UI

Data source components can be instantiated from the Add Data Source dialog. Classes that implement `IDataSourcePlugIn` can control how they are presented in the dialog by using the `PlugInIdentityAttribute`, which specifies the name, icon, and group associated with the plug-in class. All data sources that can be instantiated by the TLA will be listed in the dialog. The Name of data source plug-ins that have an associated `GroupName` will be shown as children of a parent node in a tree control. The `GroupName` will be a root parent, and the `Name` will be a leaf node. The absence of `PlugInIdentityAttribute` prevents the plug-in from participating in the dialog.

System Window Icon

By implementing `IDataSourcePlugIn`, a class gains the ability to place a control in the System window. The plug-in is responsible for creating a `UserControl`-derived object, and the TLA application is responsible for showing the control in an appropriate place in the System window.

The System window control is implemented as a .NET control of type `System.Windows.Forms.UserControl`. Developers using C# or VB can use a graphical designer tool to create the visual look of this control. At runtime, the TLA application obtains a reference to the instantiated control through the property `IDataSourcePlugIn.SystemWindowControl`. The plug-in implementation must set this property within its `Initialize()` method. If `SystemWindowControl` is null after the TLA calls `Initialize()`, then no icon for the data window will appear in the System window.

Save/Load of Data

As with all plug-ins classes, the TLA application can save the state of a data source plug-in if it implements `ISerializable`. The `ISerializable` interface should not be used to save data though. `TlaNetInterfaces.dll` defines an `ISaveData` interface for saving the data of data source plug-ins. If a data source plug-in implements `ISaveData`, it must also implement `ISerializable`; otherwise the `ISaveData` interface is ignored.

The application calls the plug-in `ISaveData.CanSaveData` to determine if the plug-in has acquisition data to save. It should return true if data exists to be saved and false to skip saving data. Before data is saved, the application calls `ISaveData.PrepareToSaveData()`. This method tells the plug-in to reset its internal state so it can begin the process of saving data. Subsequent calls to `ISaveData.SaveData()` are made from the application. This call passes a `Stream` object as a parameter, and the plug-in writes its data into this `Stream`. The data source plug-in is expected to save a “reasonable” amount of data on each call to enable periodic checking for a user abort. Each call should complete in no more than a few seconds. The plug-in returns true to indicate that there is more data to be saved and that it needs to be called again. The plug-in returns false when all data has been saved. The same `Stream` object is passed into each call of `SaveData()` for output of acquisition data. The `Stream` object does not support seeking and the plug-in should not keep a reference to this object between calls to `SaveData()` or after.

The plug-in is free to save any form of data to the `Stream`. It may or may not utilize serialization. From the application point of view, this is just binary data understood by the plug-in.

When the application restores a data source plug-in as a reference memory, it sets the `ISaveData::SavedDataStream` property to a `Stream` based instance created by the application. The property is set before the `IPlugIn.Initialize` is called. The `Stream` object supports seek and read, but not write. It overrides the seek related aspects of `Stream` to confine the plug-in to the region of the TLA file that has its acquisition data. To the plug-in it appears that the acquisition data is located at the beginning of the file and contains nothing else.

Once the `SavedDataStream` property has been set with a reference to the `Stream` object, it is up to the plug-in to interpret the data. If the plug-in supports compatibility with Listing and Waveform windows, it should use the `SaveStream` internally to provide data via the `IRecordAccess` interface.