Addon Framework Guide

Everbridge Control Center

Contents

About Control Center Addons

Using Control Center Addon Framework, you can develop your own addons to expand existing Control Center functionality.

The framework is divided into two distinct feature sets. The first allows you to register new components within Control Center and is the top-level integration provided to Addons. The second is a runtime interface framework that exposes Control Center functionality to loaded Addon components within the Windows Client environment.

An Addon package can define the following components.

Component	Description
Addon Object	Define new custom Control Center object types, for example, **Type Permissions** object.
Addon Property Extender	Add properties to existing Control Center objects, for example, Windows Client object's **Dashboard** property.
Addon User Interface	A control to display when and addon object is displayed in a tile layout, for example, display a **Dashboard** object.
Client Action	Add executable actions on the Windows Client, that can be used from commissioned buttons on the Main Menu graphical user interface (GUI) control, for example, **Client Template Picker**.
Client Managers	This is an addon instance that runs for the lifetime of the client process, for example, **Theme Client Manager** for managing theming set by the Windows Client Theme object.
Data Source	Add dashboard data sources to provide data in the Dashboard object, for example, **RSS Feed Data Source**.
Data Source Designer	A designer for a data source to set parameters controlling the data returned, for example, RSS Feed Designer.
GUI Plugin	Create custom Graphical User Interface controls to be used in the Control Center GUI designer from the plugin controls toolbox section, for example, **Chrome Browser** control.
Registration Manager	An addon instance that runs once after a user has successfully logged in and the client has loaded for the first time. The Registration Manager is for registering and configuring the client environment as a one-time event, for example, **Icon Set** and **Tooltip** Registration Managers.
User Interface Designer	A control to display when an addon object is double clicked in **System Configuration**. The control is used to set the contents of

	the object by its serializable settings. The control can be an editor version of an Addon User Interface, for example, **Dashboard** object designer, or a standalone control to set object contents, for example, **Type Permissions** object.
Video Extension	Add video functionality mostly through the timebar, for example, **Video Bookmarks** and **Video Loop Extensions**.

Control Center Addon Package

A Control Center addon package is a folder that contains the resources and a manifest file, that describe their capabilities, including one or more libraries/resources which define the addons in that folder.

CAUTION: You must ensure that all addon packages have a unique folder name to make sure addons do not override each other when installing.

A typical addon comprises:

- A manifest file called `AddonManifest.xml` which provides the XML that defines the addon. This points at a DLL in the same folder in which to find the addons, identifies the class for each addon, and provides additional information such as addon object Display Name and Icon.
- The DLL file where the addons are defined.
- Image files for addon object icons referenced in the manifest file.
- Additional DLL files or other references the addon DLL requires. This could include web components or third-party libraries but does not include the addon framework libraries required to develop the addon as these are loaded by the application. You must not include any associated driver DLL as these are already referenced from the installed driver packages.

Control Center Addon Manifest

An Addon Package is a folder containing a file called `AddonManifest.xml`. The folder may be anywhere under the AddonPackages folder, except inside another package.

The basic manifest XML structure:

```
<Manifest Name='ExamplePackage'>
        <Dependencies>.</Dependencies>
        <Assembly Path='ExampleAssembly.dll'>
            <Type Name='ExampleAssembly.ExampleObjectClass'>
                <Property Name='IPSC.AddonIpscObject'/>
                <Property Name='IPSC.Icon' Value='example.png'/>
                <Property Name='IPSC.DisplayName' Value='Example Addon'/>
            </Type>
        </Assembly>
    </Manifest>
```

The root element is Manifest, which can contain Assembly elements, which can contain Type elements, which can contain Property elements. The Dependencies element is optional and specifies a path to search for referenced assemblies.

Property names have significance as they are used by Control Center to find addons of certain kinds, for example, `IPSC.AddonIpscObject` identifies Addon Control Center Objects.

NOTE: The property that has the name identifying the type of addon (for example, `IPSC.AddonIpscObject`) often does not have a value; its purpose is simply to indicate the Addon type. However, some addon types support another type (for example, `IPSC.Designer` as the User Interface Designer is the designer for a specific Addon Object). In this case, Value is the supported type.

Installing Addons

Addons must be installed manually on every client and server instance. On standard installations, a default AddonPackages folder is created in `C:\Program Files (x86)\Everbridge\Control Center\AddonPackages`. Where a custom install path has been specified, the path is `<custom install path>\Everbridge\Control Center\AddonPackages`. The addon package folder must be deployed to this location and is loaded by Control Center on startup. This requires restarting any active server or client.

Each client and server can be installed with only the local instance stopped. However, you must not configure or start using an addon until an entire site is installed.

Updating Addons

Updating addons requires replacing the contents of the installed addon package within the **AddonPackages** folder. You must do this on every client and server instance. To upgrade an addon, you must stop the local Control Center instance (client or server) to update/replace the folder contents. There are some key considerations when developing updates to addon packages.

- The addon package must have the same folder name when deployed. If you need to rename the folder, on upgrade, you must remove the previous folder.
- Ensure the referenced DLL has a File Version and that the File Version is incremented for each new version.
- Consider backwards compatibility. Some components will already exist and be commissioned. Ensuring existing configuration is compatible with the updated package is very important. Breaking changes can include (but is not limited to):
    - Adding new non-nullable properties to addon objects.
    - Removing GUI plugin control events, event variables, or properties.
    - Renaming registered components or their namespace.
- Unless all instances are stopped and updated at once, consider that some Control Center instances may not be updated when updated components are loaded.
- In federated environments, you may need to ensure that updated definitions for addon objects and plugin controls are compatible with the versions of the addon packages that may be installed at remote sites.

- For addons that include GUI Plugin Controls, when a GUI Plugin is a Control Center GUI, Control Center Windows Client generates a Control Center GUI file for the associated GUI. This caches the GUI plugin implementation at the time of the file creation and may need to be recreated on all upgraded clients after updating. The Control Center GUI file is in the folder `C:\ProgramData\Everbridge\Control Center\Windows Client\Compiled\Plugin Controls`

If the GUI designer using the GUI Plugin stopped working after an update, delete the Control Center GUI file and restart Control Center. Although, this should happen automatically on first login after updating, if the DLL file version was incremented as recommended.

Control Center Addon Framework Assemblies and Interfaces

The Control Center assemblies can be referenced in an addon to access the features exposed through the Addon Framework. There are three major assemblies:

- **CNL.IPSecurityCenter.CoreAddonContracts** declares the attributes, interfaces and related types for Control Center specific addons, providing hooks into various Control Center functionalities.
- **CNL.IPSecurityCenter.GraphicalUserInterface.Plugin.Interface** declares the attributes and related types required for GUI plugin controls.
- **CNL.IPSecurityCenter.AddonFramework.Common.UI** provides use of Telerik theming (specifically, the Telerik Visual Studio 2013 theme used across the front-end UI) through a XamlResourceHelper to merge the Telerik theme into the resources of any WPF controls.

Typically, an assembly that defines an addon only needs to reference **CNL.IPSecurityCenter.CoreAddonContracts.** It does not to reference Control Center core. Unless it is defining a GUI plugin it does not need the **GraphicalUserInterface.Plugin.Interface**.

The Addon Framework Environment Interfaces are defined in the **CNL.IPSecurityCenter.CoreAddonContracts** and provide access to certain functionality.

The following primary interfaces are provided for Control Center client:

Interface	Description
ICommonEnvironment	This interface provides basic access to the Control Center runtime environment, and allows object resolution, metadata fetching and getting other interfaces.
IUserInterfaceEnvironment	This interface is injected into a client-side addon at runtime (derives from **ICommonEnvironment**) and provides additional access to UI features.
IDesignerEnvironment	This interface is injected into the designer for a client-side addon object (derives from

	IUserInterfaceEnvironment) and provides additional access to UI features including drag-drop.
IVideoExtensionEnvironment	This interface is injected into the BeginSession method of a **Video Extension** addon at runtime (derived from **IUserInterfaceEnvironment**) and additionally allows toolbar button interactions and overlaying a UI control over the video. This interface also derives from **IVideoControl** and so it represents the video player instance, allowing interaction with the video control.

The following secondary interfaces can be cast from **IUserInterfaceEnvironment**. These interfaces are injected into client-side addons at runtime.

Interface	Description
IAlarmTypesEnvironment	Provides minimal alarm information – alarm count for a given location.
IUserInterfaceEnvironmentControls	Serves as a factory for creating video or scene controls.
IUserInterfaceEnvironmentVideoExport	Provides access to submit a **Video Export** job or show the **Video Export Wizard**.
IUserInterfaceEnvironmentDragDrop	Provides a way to interpret drag and drop events of Control Center objects from inside Control Center, for Windows Forms and WPF drag/drop events.
IUserInterfaceEnvironmentAlarms	Provides more detailed access to the alarm stack visible to the user.
IUserInterfaceEnvironmentTracks	Provides access to a listener for user-associated changes to tracks.
IUserInterfaceEnvironmentIcons	Provides a way to obtain the binary data for an icon in Control Center from the currently configured icon set.
IUserInterfaceEnvironmentIdentity	Provides access to the identity of the currently logged on user and tenant ID of the local site.
IUserInterfaceEnvironmentDragDropSource	Allows a UI to initiate a drag and drop operation of multiple Control Center objects from Windows Forms or WPF.
IUserInterfaceConfigurationReference	Provides the ID and type of the object being displayed in the current context.
IUserInterfaceEnvironmentEvents	Allows creation of an event listener.
IUserInterfaceEnvironmentDisplayWindows	Provides access to display window settings of client.

IUserInterfaceEnvironmentEnterpriseSettings	Provides an entry point to get and set enterprise settings.
IUserInterfaceEnvironmentClientTileSource	Provides an entry point to get and set the selected objects in the product.
IContrainerRegistar	Allows registration of interfaces in **Unity**, as well as registering event control providers that allow custom rendering of an event's properties in an additional tab in the Event Viewer.
IDataSourceClient	Provides access to groups of DataSources from remote clients.
IUserInterfaceTileLayouts	Allows managing the displaying of Tile Layouts on the client.
IUserInterfaceEnvironmentSearchTypeDialog	Provides access to show the Types selection dialog, and get device and object information.
IUserInterfaceDropZone	Used to map commands executed against Drop Zone
IUserInterfaceEnvironmentToastAlerts	Provides the ability to display a Toast Alert in the client.

The following secondary interfaces can be cast from the **ICommonEnvironment**.

Interface	Description
ICommonEnvironmentExtendedProperties	Provides access to the list of installed **Extended Property** definitions for a particular type.
ISearchEnvironment	Provides the ability to search for configured objects in Control Center, in code, via the popup **Search** dialog, or using the Inline Search User Control.
ICommonEnvironmentDevices	Provides access to a remote proxy of a device via the device contract interface.
ICommonEnvironmentCustomStorage	Provides access to the custom storage capability in Control Center.
ICommonEnvironmentLocation	Provides the ability to get the parent locations of a Control Center object.
ICommonEnvironmentServiceState	Provides the ability to get the connection status of a site, and the status of a Control Center object.
ICommonEnvironmentVideoBookmarks	Provides access to the database of video bookmarks maintained inside Control Center.

ICommonEnvironmentLicensing	Provides access to the effective combined licensed capabilities for the installation.
ICommonEnvironmentNotifications	Provides events about changes being made to Control Center object configuration.
ICommonEnvironmentEvents	Provides the ability to query time-bar bookmark events on a device over a timeframe.
ICommonEnvironmentDataAccessLayer	Provides the ability to read and update addon objects and save custom property changes.
ISecurityEnvironment	Provides the capability to query security related functionality.
IDataAcessEnvironment	Provides the ability to get the folder IDs of built-in folders.
ICommonEnvironmentVariables	Provides the ability to replace the Environment Variables found in a path with their configured values in Control Center.

Accessing Addon Framework Interfaces

You must use the `GetEnvironmentInterface` method exposed on **ICommonEnvironment** to access the secondary Addon Framework interfaces.

```
var alarmTypeEnv =
userInterfaceEnvironment.GetEnvironmentInterface<IAlarmTypesEnvi
ronment>();
```

However, some interfaces are not accessible through the `GetEnvironmentInterface` method, in which case they can be obtained by casting the environment object to its secondary Addon Framework interface.

```
var alarmTypeEnvironment = userInterfaceEnvironment as
IAlarmTypesEnvironment;
```

Versioning

To ensure that old plugins continue to be compatible with new releases of Control Center, the interfaces in **CoreAddonContracts** are permanently frozen once they appear in a public release of Control Center. New functionality is only ever added by defining separate new interfaces.

For convenience, some extension methods are provided so you do not have to perform casts and so on.

Control Center Addon Component Types

Control Center Objects

You can define new custom Control Center object types. Custom Control Center object types:

- Can hold serialized settings.
- (*optionally*) be edited in a designer (see Control Center User Interface Designer).
- (*optionally*) be displayable in Tile Layouts (see Control Center Addon User Interface). Control Center objects handle specific areas of configuration. For example, device, GUI, response plan, trigger, scene and so on.

As standard, a Control Center object:

- has a label, description, any other properties per object type (appear in property grid).
- can be exported/imported as a kind-of XML.
- can be searched for - see Using the Search service.
- has the ability to be published and federated, if specified.
- has tracked dependencies on other Control Center objects (used by export and publishing).
- has access to control list.
- can be enabled/disabled
- can be put into an Alert State.

Defining a Control Center Addon Object

Manifest Properties

In the Addon Manifest, these properties can be defined:

- `IPSC.AddonIpscObject` - value is ignored, simply indicates addon type.
- `IPSC.Icon` - value is the path (relative to the package folder) to a 16x16 image used in the UI.
- `IPSC.DisplayName` - value is the name to display for the type (currently not currently support localization).

For example:

```
<Type Name='ExampleAssembly.ExampleObjectClass'>
      <Property Name='IPSC.AddonIpscObject'/>
     <Property Name='IPSC.Icon' Value='example.png'/>
     <Property Name='IPSC.DisplayName' Value='Example Addon Object'/>
 </Type>
```

Implementation Details

To create an Addon Object the Type Name must refer to a class, that implements **IAddonIpscObject** (it can usually be derived from the helper class AddonIpscObject).

A Control Center object can have methods, properties and events, though this is entirely optional. Historically, Control Center uses Visual Basic. This code requires your Control Center object type to inherit `Pacific.Core.BaseObject`. To bridge between Control Center and Addons, there is a class called `AddonBaseObject`.

The `AddonBaseObject` class maps between the serialization mechanisms (.NET binary in the Control Center code and XML in Addons). The `AddonBaseObject` class also maps the attributes that can be put on methods and properties. It essentially is a wrapper around an addon that implements IAddonIpscObject.

Example 1:

```
public class Dashboard : AddonIpscObject<DashboardSettings> { }
```

(The `AddonIpscObject` helper class used here takes a type parameter: a POCO that holds the settings of the addon, and which is assumed to be serializable by Data Contracts.)

Define the Control Center object properties in the Addon Manifest as follows:

Example 2:

```
public class SiteReference : IAddonIpscObject { }
```

Notes:

- This object is called `SiteReference`. In Control Center user interface its called Location Reference. In other words, in its `IPSC.DisplayName` in the manifest.
- It uses custom XML serialization to deal with a backward compatibility problem, so it directly implements `AddonIpscObject` to be in complete control.

- It also has a property:

```
[Export(Editable = true, Editor =
typeof(LocationTypeEditor))]
    public Guid TargetLocation
    {
        get { return _settings.TargetLocation; }
        set
        {
            _settings.TargetLocation = value;
            _settings.SiteHostName = string.Empty;
        }
    }
```

  This makes TargetLocation appear in Control Center's property grid so a user can directly edit it.

NOTE: Note the specifying of a custom Editor type.

Control Center Property Extenders

Extended properties provide the ability to add properties to Control Center objects. A user supplies the value and the addon supplies the definition. They are a useful way of associating new addon functionality to existing objects.

Supported Object Types

These types of Control Center Object can have extended properties.

- Asset Group
- Device
- Location
- Windows Client

Defining a Control Center Property Extender

In the Addon Manifest, the IPSC.AddonIpscObjectPropertyExtender property can be defined. In the IPSC.AddonIpscObjectPropertyExtender property, value is ignored. It simply indicates addon type. For example:

```
<Type Name='ExampleAssembly.ExamplePropertyExtenderClass'>
      <Property Name='IPSC.AddonIpscObjectAddonIpscObjectPropertyExtender'/>
 </Type>
```

Implementation Details

To create an Addon User Interface the Type Name must refer to a class that implements IPropertyExtender.

The **IPropertyExtender** interface is trivial: it specifies the type name of the object it applies to (the ObjectTypes class contains suitable string constants such as Location), and a collection of **ExtendedProperty** objects that are the extra properties that should be defined on that type.

Control Center Addon User Interface

An Addon User Interface is a control to display when an addon object is displayed in a **Tile Layout**. This allows visualization of the information stored in the object.

Manifest Properties

In the Addon Manifest, the IPSC.UserInterface property can be defined. Its value is the **FullName** of the addon object that this is a user interface for.

For example,

```
<Type Name='ExampleAssembly.ExampleUserInterfaceClass'>
    <Property Name='IPSC.UserInterface'
Value='ExampleAssembly.ExampleObjectClass'/>
 </Type>
```

Implementation Details

To create an Addon User Interface, the Type Name must refer to a class that implements IAddonIpscObjectUserInterface<T> where T is the main type of the addon (on which this property is defined). The class must also be a WPF User Control, so the interface will be implemented on the xaml.cs of the control.

User Environment Injection

The method on the interface BeginSession(TObject systemObject, IUserInterfaceEnvironmentenv) injects the User Interface Environment into the Addon User Interface to allow interaction with the Control Center client.

Control Center Client Action

Client Actions are executable actions on the Windows Client, that can be used from commissioned buttons on the Main Menu Graphical User Interface (GUI) control.

Manifest Properties

In the Addon Manifest, these properties can be defined:

- `IPSC.ClientAction` - Value is ignored, simply indicates addon type.
- `IPSC.DisplayName` - Value is the name to display in the button configuration screen to select the client action the button performs (does not currently support localization).

For example,

```
<Type Name='ExampleAssembly.ExampleClientActionClass'>
    <Property Name='IPSC.ClientAction'/>
    <Property Name='IPSC.DisplayName' Value= 'Example Client Action'/>
 </Type>
```

Implementation Details

To create a Client Action the Type Name must refer to a class that implements `IClientAction`.

User Environment Injection

The method on the interface `Execute(IUserInterfaceEnvironment userInterfaceEnvironment)` injects the User Interface Environment into the Client Action to allow interaction with the Control Center client.

Control Center Client Managers

Client Managers run for the lifetime of the client process and allow long-running management of information on the client machine.

Manifest Properties

In the Addon Manifest, the `IPSC.ClientManager` property can be defined. Its value is ignored, it simply indicates addon type.

For example,

```
<Type Name='ExampleAssembly.ExampleClientManagerClass'>
    <Property Name='IPSC.ClientManager'/>
 </Type>
```

Implementation Details

To create a Client Manager the Type Name must refer to a class that implements IClientManager.

User Environment Injection

The method on the interface `Initialise(IUserInterfaceEnvironment userInterfaceEnvironment, bool isNewClient)` injects the Client Manager into the Addon User Interface to allow interaction with the Control Center client.

Control Center Data Source

A DataSource provides data to Control Center. It executes inside a service (currently the Alarm Types Service). Its purpose is to publish constantly updating tables of data.

Currently DataSources are used by Dashboard Widgets, though as a concept they are not dependent on dashboards.

Manifest Properties

In the Addon Manifest, these properties can be defined:

- `IPSC.DataSource` - Value is ignored, simply indicates addon type.
- `IPSC.DisplayName` - Value is the name to display in the data source selection dialog in Dashboards or other users of the data source (does not currently support localization).
- (*optional*) `IPSC.HideInDashboard` - Value should be true to hide the data source from being selected in the Dashboard data source selection dialog.

For example,

```
<Type Name='ExampleAssembly.ExampleDataSourceClass'>
    <Property Name='IPSC.DataSource'/>
    <Property Name='IPSC.DisplayName' Value= 'Example Data Source'/>
    <Property Name='IPSC.HideInDashboard' Value= 'true'/>
 </Type>
```

Implementation Details

To create a Data Source, the Type Name must refer to a class that inherits from `DataSource<TSettings>` where *`TSettings`* is a class of settings to configure the Data Source.

Data Source Environment Injection

The method on the class `Start(IDataSourceEnvironment environment)` injects the Data Source Environment into the Data Source to allows very specific Data-related operations.

Control Center Data Source Designers

A Data Source Designer is the designe for a data source to set parameters controlling the data returned.

Manifest Properties

In the Addon Manifest, the `IPSC.DataSourceDesigner` can be defined. Its value is the **FullName** of the Data Source that this is a designer for.

For example,

```
<Type Name='ExampleAssembly.ExampleDataSourceDesignerClass'>
    <Property Name='IPSC.DataSourceDesigner'
              Value='ExampleAssembly.ExampleDataSourceClass'/>
 </Type>
```

Implementation Details

To create a Data Source Designer, the Type Name must refer to a class that implements `IDataSourceDesigner<TSettings>`. The class must also be a WPF User Control, so the interface will be implemented on the `xaml.cs` of the control. TSettings should be the same type as the settings for the Data Source this Designer relates to.

Control Center GUI Plugin

GUI Plugins allow the creation of custom Graphical User Interface controls to be used in the Control Center GUI designer. The Graphical User Interface controls are available as custom controls in the GUI Editor Toolbox from the Plug-in Controls tab.

A Control Center plugin for the GUI designer is essentially a .Net user control with attributes to identify that they should be loaded as plugins, as well as the functions and properties that should be available to the end-user.

Manifest Properties

In the Addon Manifest, the `IPSC.GuiPlugin` property can be defined. Its value is ignored. It simply indicates addon type.

For example,

```
<Type Name='ExampleAssembly.ExampleGuiPluginClass'>
    <Property Name='IPSC.GuiPlugin'/>
 </Type>
```

Implementation Details

To create a GUI Plugin, the Type Name must refer to a class which has the ToolboxName and ToolboxDescription attributes mentioned. The class must also be a WindowsForms UserControl. The class can optionally inherit PluginControl – this provides drag-drop functionality of Control Center objects.

The plugins can be used as custom controls in the GUI Editor Toolbox (Plug-in Controls tab).

The original code for defining GUIs used its own internal base class for controls. So that plugins could be isolated from this, a bridging system was created to wrap plugins in an auto-generated class that inherits the internal base class. This results in the cached compiled GUI and plugin DLLs that appear in ProgramData. This can make it difficult to debug or extend further.

When writing a plugin:

- A Plugin class must have ToolboxName, ToolboxDescription attributes implemented:

  ```
  [ToolboxName('File Picker Control'),
   ToolboxDescription('A plugin File Picker control')]
   public partial class FilePickerControl : PluginControl
   {
   }
  ```

- A GUI Plugin must have a Reference to the DLL:

  `CompiledAssemblies\CNL.IPSecurityCenter.GraphicalUserInterface.Plugin.Interface.dll`

User Environment Injection

This is a facility available to plugins to communicate with the Windows Client environment.

NOTE: It is not compatible or tested with GUIs running in Hosted mode.

The plugin must link to this assembly:

`CNL.IPSecurityCenter.CoreAddonContracts.dll`

In addition to the usual:

`CNL.IPSecurityCenter.GraphicalUserInterface.Plugin.Interface.dll`

This then allows it to declare a magic property:

```
public IUserInterfaceEnvironment UserInterfaceEnvironment { get; set; }
```

The Contol Center client looks for a property with that name and type, and if it finds one then it sets the value to an object that implements IUserInterfaceEnvironment.

CAUTION: This injection happens after construction, so you cannot use the environment object inside your constructor.

Control Center Registration Manager

A Registration manager is an addon instance that runs once after a user has successfully logged in, and the client has loaded for the first time. The Registration Manager is for registering and configuring the client environment as a one-time event.

Manifest Properties

In the Addon Manifest, the `IPSC.RegistrationManager` can be defined. Its value is ignored, it simply indicates addon type.

For example,

```
<Type Name='ExampleAssembly.ExampleRegistrationManagerClass'>
    <Property Name='IPSC.RegistrationManager'/>
 </Type>
```

Implementation Details

To create a Registration Manager, the Type Name must refer to a class that implements `IRegistrationManager`.

User Environment Injection

The method on the interface `Initialise(IUserInterfaceEnvironment userInterfaceEnvironment)` injects the User Interface Environment into the Registration Manager to allow interaction with the Control Center client.

Control Center User Interface Designer

A User Interface Designer is a control to display when an addon object is double clicked in System Explorer. This is a control to display when an addon object is double clicked in System Configuration. The control is used to set the contents of the object by its serializable settings.

Manifest Properties

In the Addon Manifest, the `IPSC.Designer` property an be defined. Its value is the **FullName** of the Addon Object that this is a designer for.

For example,

```
<Type Name='ExampleAssembly.ExampleUserInterfaceDesignerClass'>
    <Property Name='IPSC.Designer'
Value='ExampleAssembly.ExampleObjectClass'/>
 </Type>
```

Implementation Details

To create a User Interface Designer, the Type Name must refer to a class that implements `IAddonIpscObjectDesigner<in TObject>` which is the main type of the Addon Object (on which this property is defined). The class must also be a WPF User Control, so the interface will be implemented on the `xaml.cs` of the control.

Designer Environment Injection

The method on the interface `BeginSession(TObject systemObject, IDesignerEnvironment env)` injects the Designer Environment into the User Interface Designer to allow interaction with the Control Center with additional designer-specific functionality.

Control Center Video Extensions

A video extension is an add-on that enhances the user interface of a video playback tile control in the Windows client. It can do this by adding buttons to the toolbar, by replacing the video (probably temporarily) with any other UI, and by adding icons and bookmarks to the time bar.

Manifest Properties

In the Addon Manifest, the `IPSC.VideoExtension` property can be defined. Its value is ignored. It simply indicates addon type.

For example,

```
<Type Name='ExampleAssembly.ExampleVideoExtensionClass'>
    <Property Name='IPSC.VideoExtension'/>
 </Type>
```

Implementation Details

To create a Video Extension the Type Name must refer to a class that implements `IVideoExtension`.

Compared with the similar `BeginSession` methods in `IAddonIpscObjectUserInterface` and `IAddonIpscObjectDesigner`, in this case, it returns bool because it is quite normal/expected for an extension to refuse to work with some cameras. For example, you could query the device for a specific interface that it must implement.

The environment is of type `IVideoExtensionEnvironment`. It inherits `IVideoControl` so it allows the extension to control playback.

Video Extension Environment Injection

The method on the interface `BeginSession(Reference videoDevice, IVideoExtensionEnvironment environment, List<string> eventsList)` injects the Video Extension Environment into the Video Extension to allow video interactions.