ASCOM SettingsProvider

Tim Long

This paper describes the method of use and the reasons for the ASCOM.SettingsProvider class, introduced in ASCOM Platform 6.0. By allowing ASCOM driver developers to leverage the Application Settings support built into Visual Studio and the .NET Framework, this class significantly simplifies the handling of driver settings, almost completely eliminating the need to write settings related code. Creating settings therefore becomes a design-time activity, increasing developer productivity and reducing the risk of bugs.

TiGra Astronomy

GTi Suite, Ty Menter
Navigation Park, Abercynon
Wales, UK

+44 (29) 2126 0011

6 July 2011

ASCOM Settings Provider

What is the ASCOM SettingsProvider

Visual Studio provides a convenient and efficient way of designing, managing and storing application settings, including the ability to automatically bind them to Windows Forms controls. Using this technique, you can completely implement all of an application's settings logic in as few as one line of code (a Save command) because Visual Studio generates most of the code for you. This frees up the developer from any concerns of loading and saving settings, displaying and editing them and their storage location and mechanism.

By default, Visual Studio stores the settings as XML files on disk. Settings can be per-machine (apply to all users), or per user. This doesn't quite match the ASCOM approach, which saves driver settings in the ASCOM Profile Store (which happens to be in the Windows Registry in Platform 6.0) and makes all settings global (i.e. per machine and not per user). The default implementation also associates settings with the application executable, which for an ASCOM in-process driver proves to be counter-intuitive and inconvenient.

Luckily, like most .NET subsystems, the Microsoft developers provide integration points for extending or replacing the default implementations. The Application Settings subsystem allows custom providers to be written that alter the default behaviours. The ASCOM *SettingsProvider* class is such an extension; it changes the behaviour of application settings so that they are stored in the ASCOM Profile Store and are always per-machine[1]. It also provides some behaviour that makes working with the ASCOM Profile Store easier at design time.

Why do I need it?

You don't have to use the SettingsProvider class at all if you don't want to. You can handle your driver's settings any way you like. You can handle all of the serialization, type conversion, validation, binding to dialog box controls and file I/O yourself, if you like. Clearly, that's a lot of work and the ASCOM Profile class offers significant savings in time and effort, plus 'future proofing' from changes in storage mechanisms, file permissions and so on. Most ASCOM developers, therefore, have historically chosen to use the ASCOM.Utilities.Profile class.

```
private void btnOK_Click(object sender, EventArgs e)
{
    //save textboxes
    _focuser.MaxStep = Convert.ToInt32(txtMaxStepPosition.Text);
    _focuser.StepSize = Convert.ToDouble(txtStepSize.Text);
    _focuser.MaxIncrement = Convert.ToInt32(txtMaxIncrement.Text);
    _focuser.Temperature = Convert.ToDouble(txtCurrentTemperature.Text);
    _focuser.TempMax = Convert.ToDouble(txtMaximumTemperature.Text);
    _focuser.TempMin = Convert.ToDouble(txtMinimumTemperature.Text);
    _focuser.TempPeriod = Convert.ToDouble(txtUpdatePeriod.Text);
    _focuser.TempSteps = Convert.ToInt32(txtStepsPerDegree.Text);

    //save checkboxes
    _focuser.TempProbe = chkHasTempProbe.Checked;
    _focuser.TempCompAvailable = chkHasTempComp.Checked;
    _focuser.CanStepSize = chkCanChangeStepSize.Checked;
    _focuser.CanHalt = chkCanHalt.Checked;
    _focuser.Synchronous = chkIsSynchronous.Checked;

    //save radio button
    _focuser.Absolute = radAbsoluteFocuser.Checked;

    _focuser.SaveProfileSettings();
    Hide();
}
```

Figure 1 Getting User Edits from the SetupDialog

Using ASCOM.Utilities.Profile has some advantages, but still requires the developer to write quite a bit of code and deal with each

[1] A future version of the ASCOM.SettingsProvider class may implement per-user settings.

setting property individually. For example, Figure 1 shows the handler for the OK button Click event taken from the Focuser simulator provided with the ASCOM Platform. This code is responsible for gathering edited values from the controls on the SetupDialog. The edited values are converted from strings and the converted values are stored into properties of another class. That other class then still has to persist the settings to storage, this is shown in Figure 2. The values are converted back to strings prior to writing to the ASCOM Profile store. A similar amount of code is required (not shown) to load the settings from the Profile Store and to populate the SetupDialog controls prior to display and editing.

```
public void SaveProfileSettings()
{
    if (Temperature > TempMax) Temperature = TempMax;
    if (Temperature < TempMin) Temperature = TempMin;
    if (_position > MaxStep) _position = MaxStep;

    //ascom items
    Profile.WriteValue(sCsDriverId, "Absolute", Absolute.ToString());
    Profile.WriteValue(sCsDriverId, "MaxIncrement", MaxIncrement.ToString());
    Profile.WriteValue(sCsDriverId, "MaxStep", MaxStep.ToString());
    Profile.WriteValue(sCsDriverId, "Position", _position.ToString());
    Profile.WriteValue(sCsDriverId, "StepSize", StepSize.ToString());
    Profile.WriteValue(sCsDriverId, "TempComp", TempComp.ToString());
    Profile.WriteValue(sCsDriverId, "TempCompAvailable", TempCompAvailable.ToString());
    Profile.WriteValue(sCsDriverId, "Temperature", Temperature.ToString());
    //extended focuser items
    Profile.WriteValue(sCsDriverId, "CanHalt", CanHalt.ToString());
    Profile.WriteValue(sCsDriverId, "CanStepSize", CanStepSize.ToString());
    Profile.WriteValue(sCsDriverId, "Synchronous", Synchronous.ToString());
    Profile.WriteValue(sCsDriverId, "TempMax", TempMax.ToString());
    Profile.WriteValue(sCsDriverId, "TempMin", TempMin.ToString());
    Profile.WriteValue(sCsDriverId, "TempPeriod", TempPeriod.ToString());
    Profile.WriteValue(sCsDriverId, "TempProbe", TempProbe.ToString());
    Profile.WriteValue(sCsDriverId, "TempSteps", TempSteps.ToString());
}
```

Figure 2 Persisting Settings to the ASCOM Profile Store

The developer needs to write code to load and save each setting individually, perform data type conversions to and from strings (the ASCOM Profile Store always loads and saves a string), populate the controls on the setup dialog with the stored values, get the user's edits, validate them, convert back to strings and then write them into the Profile Store. That's a lot of code with potential for bugs. In the sample code shown, no attention has been given to internationalization issues, such as the use of comma as the decimal point. Type conversions in .NET always use the locale of the current thread, which in most cases comes from the Windows control panel setting. Users in many European countries would produce a settings profile containing comma as the decimal separator, but 'best practice' is to always persist data using the Invariant Culture, which uses a period/full stop as the decimal separator. This is a fiendishly common source of issues in astronomy software.

Visual Studio offers a much better solution, in the guise of Application Settings. This feature consists of settings designers, automatic code generation and base classes built into the .NET Framework that handle pretty much all of the tasks previously required of the developer. The Visual Studio settings designers take care of all of those steps for you, and generate bug-free code. Handling an application's settings is essentially reduced, in the simplest case, to one or two lines of code. Settings are declared at design time using the built-in designers; visual studio generates a Settings class containing strongly-typed settings properties that handles all of the loading, saving and type conversions. This is a compelling reason to use Visual Studio's built-in tools, but unfortunately they

```
private void ButtonOKClick(object sender, EventArgs e)
{
    Settings.Default.Save();
    Close();
}
```

Figure 3 ASCOM.SettingsProvider reduces complexity

don’t work in quite the right way to be useful for ASCOM drivers. The *ASCOM.SettingsProvider* class provides the bridge between the world of ASCOM and the powerful tools built into Visual Studio.

So, in summary, with the SettingsProvider, you can use Visual Studio’s integrated designers, which save you a fair bit of time and effort in terms of coding and debugging. The Application Settings framework handles all type conversion, persistence, internationalization issues and reduces the developer burden to essentially a single line of code, which is clearly easier to write and debug than the hundred or so lines of code required in the Focuser Simulator example. Figure 3 shows the code required to handle settings in the Digital DomeWorks driver, which uses the new SettingsProvider class. Settings are handled completely automatically and require just a single line of code to save them to the ASCOM Profile Store. ***No code at all*** is required to load the settings, give them default values or to bind them to Windows Forms controls, that is all handled at design time by Visual Studio and at run time by automatically generated code.

How To Use the SettingsProvider

Wiring up the ASCOM.SettingsProvider class requires a few simple steps. Those steps are enumerated here, in a later section there is a worked example.

Adding the SettingsProvider to an ASCOM Template Project

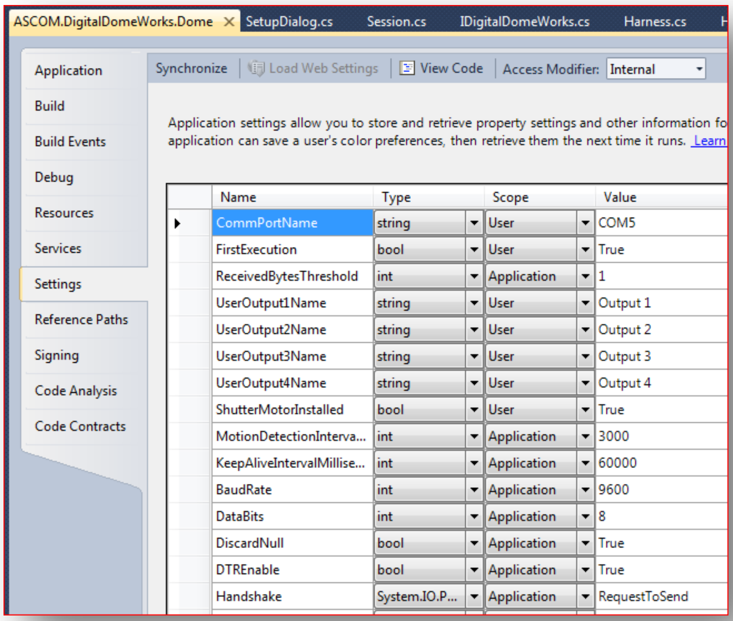
These are the steps needed to add the SettingsProvider to any of the standard ASCOM driver templates.

1. Add a reference to ASCOM.SettingsProvider
2. Create a Settings partial class and decorate it with the SettingsProvider attribute and a DeviceId attribute.
3. Use Visual Studio to design your settings.
4. Access your strongly-typed settings in code using the Properties.Settings.Default object.

Designing your Settings

Visual Studio provides a design-time editor for application settings. For a C# project, there are two ways to access the project’s settings.

1. Right click on the project, and select Properties… then click on the Settings tab.

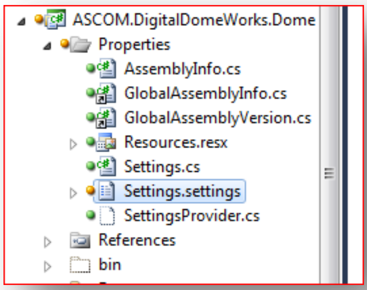


Application settings allow you to store and retrieve property settings and other information fo
application can save a user's color preferences, then retrieve them the next time it runs.

Name	Type	Scope	Value
CommPortName	string	User	COM5
FirstExecution	bool	User	True
ReceivedBytesThreshold	int	Application	1
UserOutput1Name	string	User	Output 1
UserOutput2Name	string	User	Output 2
UserOutput3Name	string	User	Output 3
UserOutput4Name	string	User	Output 4
ShutterMotorInstalled	bool	User	True
MotionDetectionInterva...	int	Application	3000
KeepAliveIntervalMillise...	int	Application	60000
BaudRate	int	Application	9600
DataBits	int	Application	8
DiscardNull	bool	Application	True
DTREnable	bool	Application	True
Handshake	System.IO.P...	Application	RequestToSend

Figure 4 Visual Studio 2010 Settings Designer

2. Expand the Properties folder and double click the Settings.settings file.



Note: in the current implementation, the ASCOM.SettingsProvider class makes all settings per-machine, that is, they behave as if their scope has been set to 'Application'. In the example above, it will be noticed that some settings have been set to 'User' scope. This has no effect and was done for documentation purposes. Settings in User scope are those expected to be edited within the SetupDialog; those in Application scope are not expected to be edited, but could be changed using the ASCOM Profile Explorer if necessary. This provides an interesting opportunity to export configuration details that might have been hard-coded otherwise. For example, in the settings shown above, all the of the serial port configuration parameters have been defined as settings along with certain timeout values. The ability to change such items can prove useful in dealing with unforeseen problems that may occur in the field.

What Visual Studio Generates

Visual Studio generates a partial class called <ProjectNamespace>.Properties.Settings. This class has a Default property, which contains the default instance of the class. It also has a property corresponding to each of your settings. Figure 5 shows the settings class used by the Digital DomeWorks driver and the base class provided by the .NET Framework.

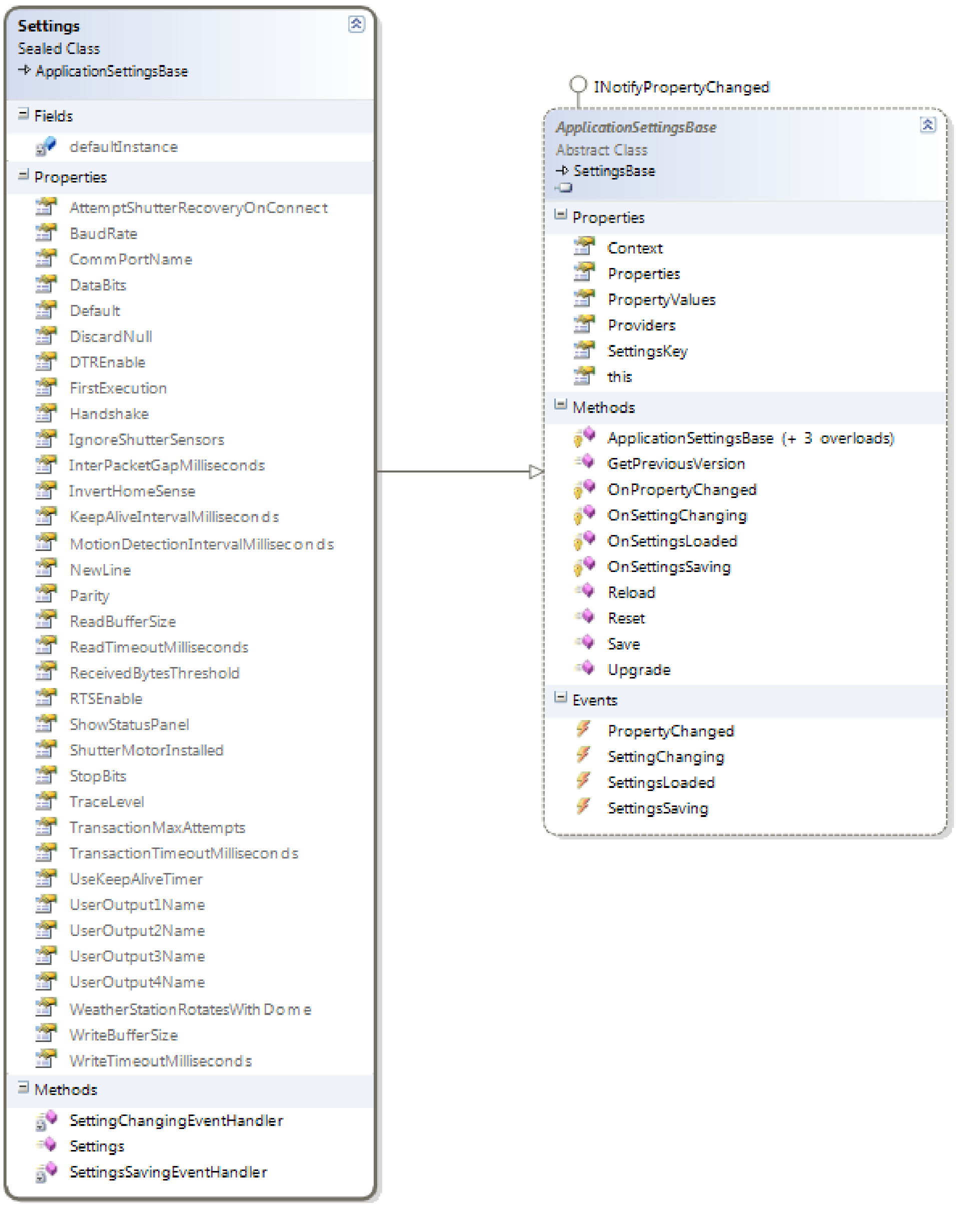
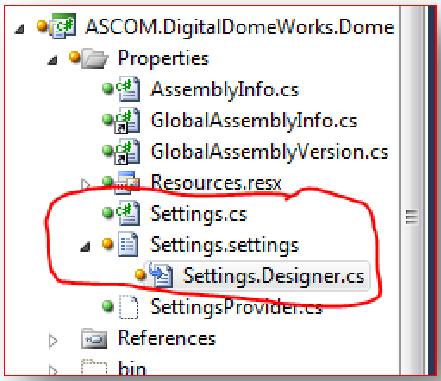
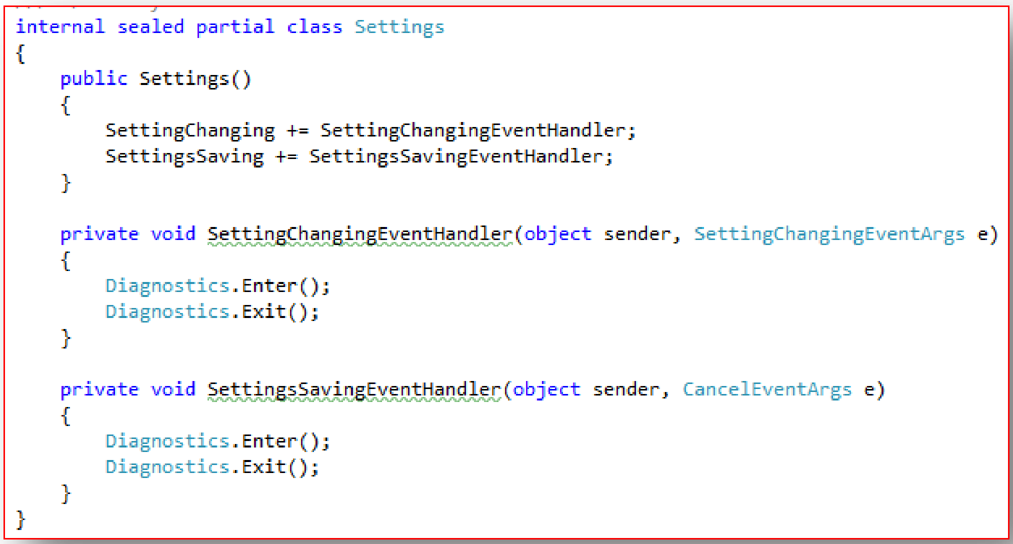


Figure 5 Strongly-typed Settings class from the Digital DomeWorks driver (generated by Visual Studio) and its parent ApplicationSettingsBase class, contained in the .NET Framework

The *Settings* partial class is implemented across two files, both contained within the project's Properties folder.



- Settings.cs contains the constructor, which you may edit. This is a great place to put diagnostic code. The Settings class exports two events that you may subscribe to, as shown in this example from Digital DomeWorks:



```
internal sealed partial class Settings
{
    public Settings()
    {
        SettingChanging += SettingChangingEventHandler;
        SettingsSaving += SettingsSavingEventHandler;
    }

    private void SettingChangingEventHandler(object sender, SettingChangingEventArgs e)
    {
        Diagnostics.Enter();
        Diagnostics.Exit();
    }

    private void SettingsSavingEventHandler(object sender, CancelEventArgs e)
    {
        Diagnostics.Enter();
        Diagnostics.Exit();
    }
}
```

- *Settings.settings* is a metadata file that holds a copy of the settings you create in the designer. It is edited using the Visual Studio settings editor, which will open automatically when the file is double-clicked.
- *Settings.Designer.cs* is a code-behind file, nested under Settings.settings – this code is auto-generated by Visual Studio and may be regenerated at any time, therefore it ***should not be edited***.
- *App.config* – you may or may not get this file depending on the order in which your project's settings were created and whether your project uses any other classes that require it. The default SettingsProvider uses this file to store settings, but once the ASCOM.SettingsProvider class is wired up correctly, it will no longer be used for storing settings. If you have this file, it's probably best to leave it alone.

Using Your Settings in Code

At runtime, using settings is simplicity itself. If you defined a setting called '*CommPortName*', then access it in your code as *Properties.Settings.Default.CommPortName*.

Settings are strongly typed. In Figure 4 the *ShutterMotorInstalled* item has a type of *bool*, so in code it can be directly set to *true* or *false*.

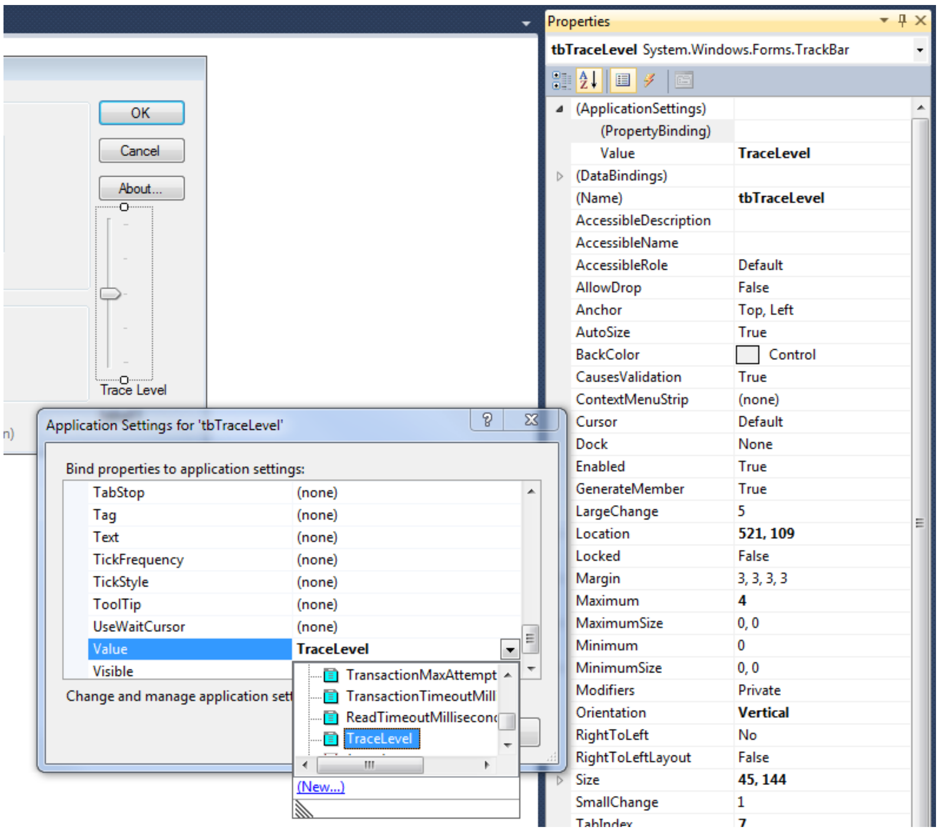
Settings can be saved, reloaded and set to defaults by using (respectively)

- Properties.Settings.Default.Save()
- Properties.Settings.Default.Reload()
- Properties.Settings.Default.Reset()

In most situations, you'll only need to use the Save() method in your SetupDialog's event handler for your OK button and the Reload() method in your form's Cancel button.

Binding to Windows Forms Controls

Another powerful feature of Visual Studio enables you to easily bind your settings to controls on your SetupDialog. This is done in the Properties pane for the control. For example, Digital DomeWorks provides a *TrackBar* control that is used to set the diagnostic trace level:



tbTraceLevel System.Windows.Forms.TrackBar

(ApplicationSettings)	
(PropertyBinding)	
Value	**TraceLevel**
(DataBindings)	
(Name)	**tbTraceLevel**
AccessibleDescription	
AccessibleName	
AccessibleRole	Default
AllowDrop	False
Anchor	Top, Left
AutoSize	True
BackColor	☐ Control
CausesValidation	True
ContextMenuStrip	(none)
Cursor	Default
Dock	None
Enabled	True
GenerateMember	True
LargeChange	5
Location	**521, 109**
Locked	False
Margin	3, 3, 3, 3
Maximum	**4**
MaximumSize	0, 0
Minimum	0
MinimumSize	0, 0
Modifiers	Private
Orientation	**Vertical**
RightToLeft	No
RightToLeftLayout	False
Size	**45, 144**
SmallChange	1
TabIndex	7

Application Settings for 'tbTraceLevel'

Bind properties to application settings:

TabStop	(none)
Tag	(none)
Text	(none)
TickFrequency	(none)
TickStyle	(none)
ToolTip	(none)
UseWaitCursor	(none)
Value	**TraceLevel**
Visible	

Change and manage application sett

TransactionMaxAttempt
TransactionTimeoutMill
ReadTimeoutMilliseconc
TraceLevel
(New...)

Figure 6 Binding Settings to Windows Forms Controls

In Figure 6, the Value property of the TrackBar control named tbTraceLevel is being bound to the TraceLevel setting. This is done by clicking in the (PropertyBinding) box beneath (ApplicationSettings) in the property pane then clicking the browse button. The dialog shows all of the control's properties that may be bound, in this case we need the Value property (another commonly used property is Text for combo boxes and text input fields).

Hint: This technique makes it extremely easy to save the size and position of windows and dialog boxes. Just bind settings properties to your form's Size and Position properties.

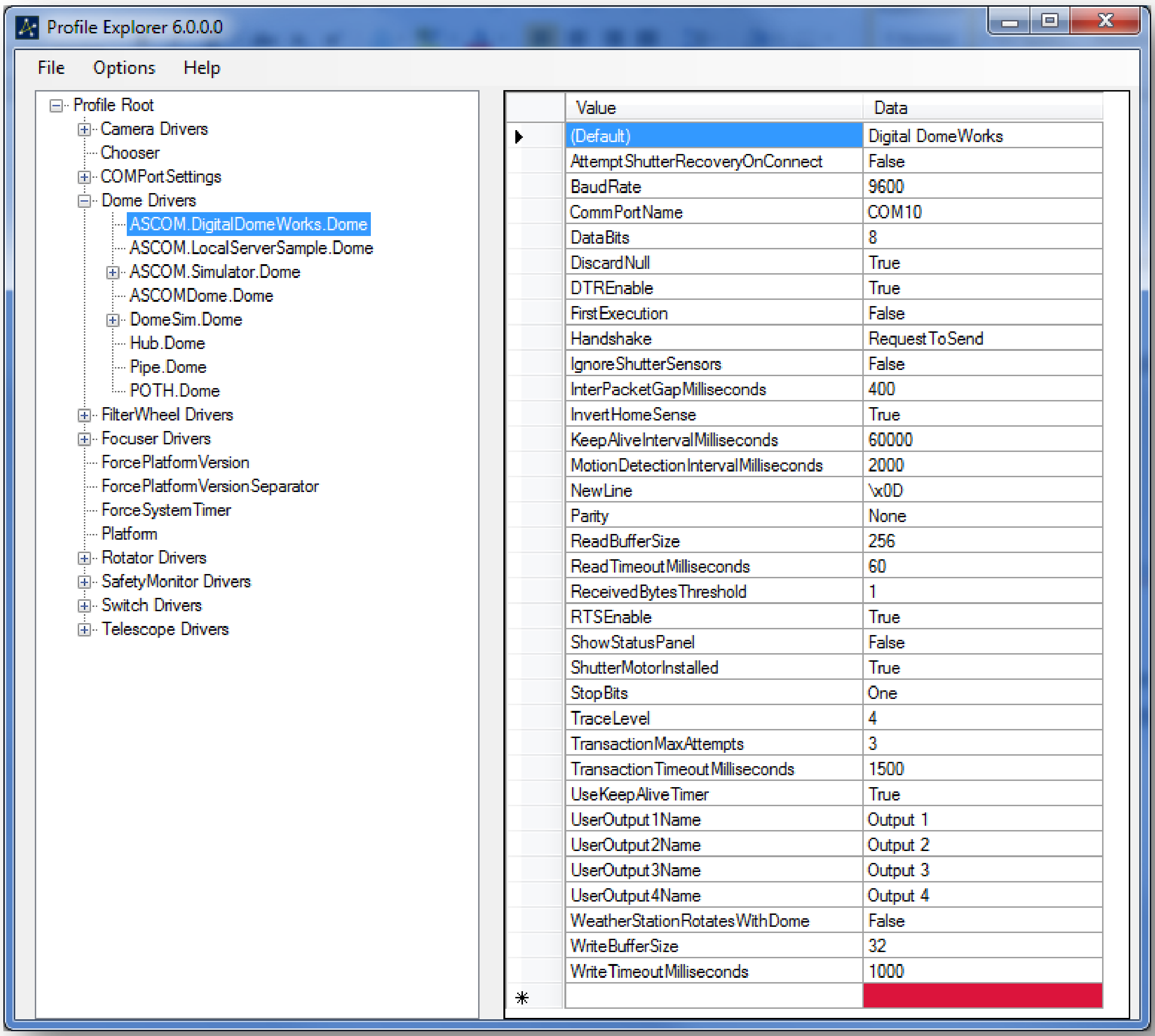
Saving Your Settings

In the *Click* event handler of your form's OK button, call *Properties.Settings.Default.Save()*.

In the *Click* event handler of your form's Cancel button, call *Properties.Settings.Default.Reload()*.

The Bridge to ASCOM

The end result of all this designing of settings is that your driver's area in the ASCOM Profile Store gets populated with the settings you've defined. This area can be browsed (and edited) with the ASCOM Profile Explorer application:



Value	Data
(Default)	Digital DomeWorks
AttemptShutterRecoveryOnConnect	False
BaudRate	9600
CommPortName	COM10
DataBits	8
DiscardNull	True
DTREnable	True
FirstExecution	False
Handshake	RequestToSend
IgnoreShutterSensors	False
InterPacketGapMilliseconds	400
InvertHomeSense	True
KeepAliveIntervalMilliseconds	60000
MotionDetectionIntervalMilliseconds	2000
NewLine	\x0D
Parity	None
ReadBufferSize	256
ReadTimeoutMilliseconds	60
ReceivedBytesThreshold	1
RTSEnable	True
ShowStatusPanel	False
ShutterMotorInstalled	True
StopBits	One
TraceLevel	4
TransactionMaxAttempts	3
TransactionTimeoutMilliseconds	1500
UseKeepAliveTimer	True
UserOutput1Name	Output 1
UserOutput2Name	Output 2
UserOutput3Name	Output 3
UserOutput4Name	Output 4
WeatherStationRotatesWithDome	False
WriteBufferSize	32
WriteTimeoutMilliseconds	1000

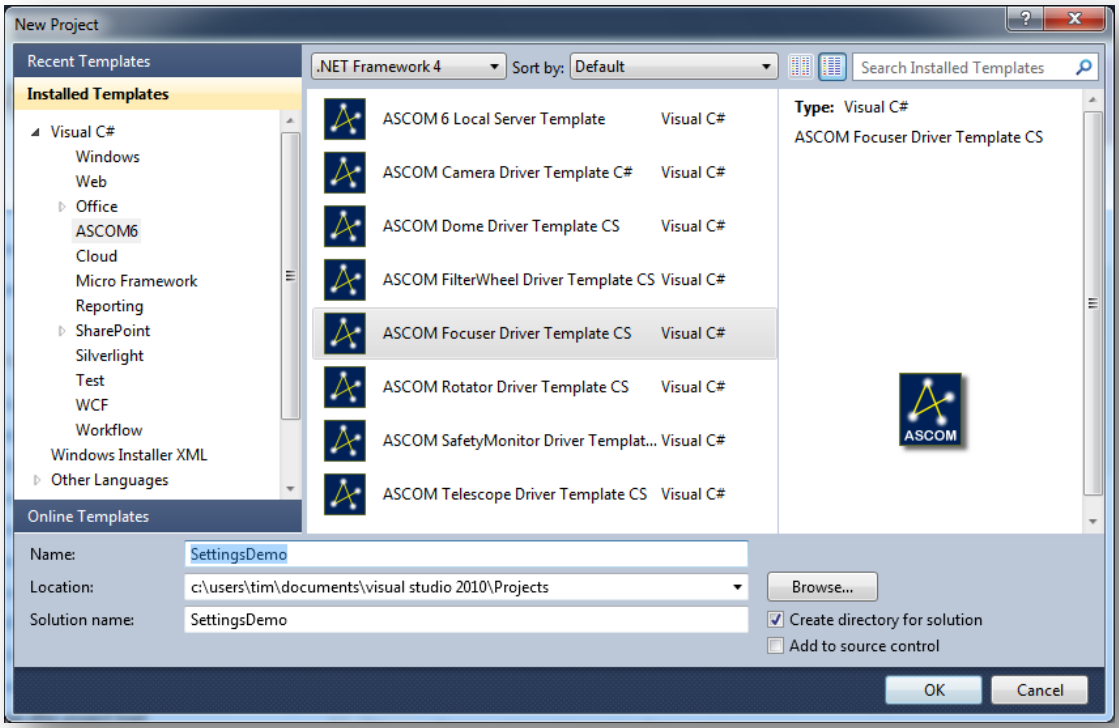
Figure 7 Settings are persisted in the ASCOM Profile Store

Worked Example

This example assumes you'll be using C# as your development language and is based on the Focuser C# template project.

Creating the SettingsDemo project

1. Create a new project, select ***ASCOM Focuser Driver CS*** from the ***ASCOM 6*** category. If you don't see the ASCOM templates, ensure that you have installed the Developer Components available from http://www.ascom-standards.org
2. Name your project ***SettingsDemo*** and click OK.



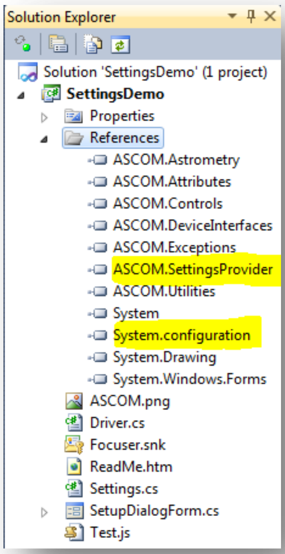
New Project
Recent Templates
Installed Templates
Visual C#
Windows
Web
Office
ASCOM6
Cloud
Micro Framework
Reporting
SharePoint
Silverlight
Test
WCF
Workflow
Windows Installer XML
Other Languages
Online Templates
.NET Framework 4
Sort by: Default
Search Installed Templates
ASCOM 6 Local Server Template Visual C#
ASCOM Camera Driver Template C# Visual C#
ASCOM Dome Driver Template CS Visual C#
ASCOM FilterWheel Driver Template CS Visual C#
ASCOM Focuser Driver Template CS Visual C#
ASCOM Rotator Driver Template CS Visual C#
ASCOM SafetyMonitor Driver Templat... Visual C#
ASCOM Telescope Driver Template CS Visual C#
ASCOM
Browse...
OK
Cancel

Type: Visual C#
ASCOM Focuser Driver Template CS

Name: SettingsDemo
Location: c:\users\tim\documents\visual studio 2010\Projects
Solution name: SettingsDemo
[✓] Create directory for solution
[ ] Add to source control

3. Visual Studio creates your template project and opens a ReadMe document. Follow the steps in this document at least to step 2 and ensure the driver builds correctly. For a real driver implementation, you'd need to follow all of the steps in this ReadMe document, but for the purposes of this worked example, we'll ignore steps 3 onward.

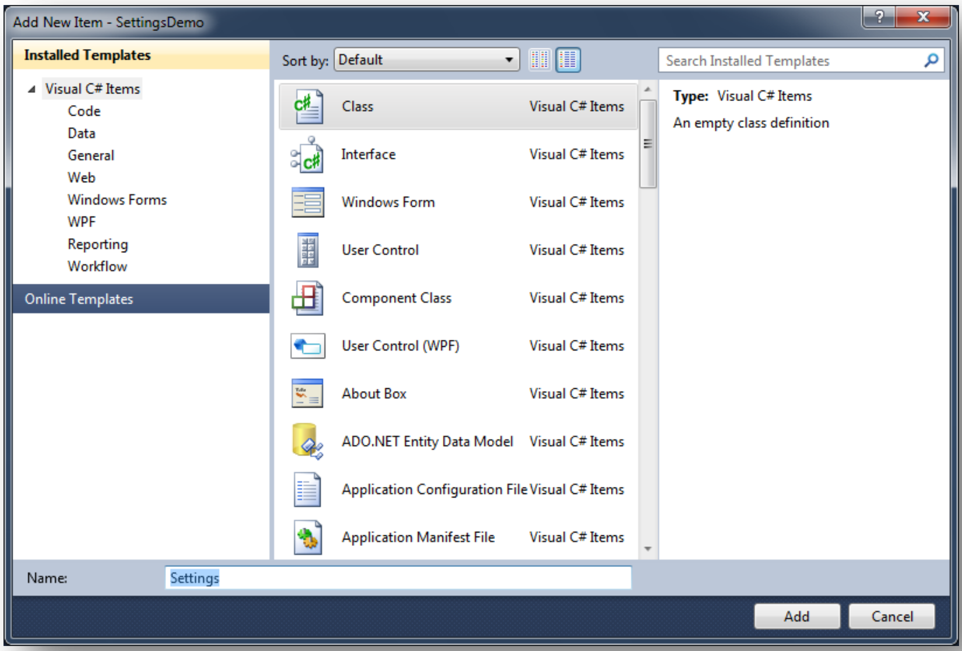
4. Ensure that your solution has references to ASCOM.SettingsProvider and System.Configuration – if not, add them from the .NET tab of the Add Reference dialog.



Wiring Up the ASCOM SettingsProvider

There are a few simple steps needed to 'wire up' the SettingsProvider class. We need add a reference to the ASCOM.SettingsProvider assembly and to decorate our settings class with a couple of attributes.

1. In the ***Project*** menu, select ***Add Class…***
2. In the Add New Item dialog, ensure ***Class*** is the selected item, and give your new class the name ***Settings***.


Add New Item - SettingsDemo
Installed Templates
Visual C# Items
Code
Data
General
Web
Windows Forms
WPF
Reporting
Workflow
Online Templates
Sort by: Default
Search Installed Templates
Class Visual C# Items
Interface Visual C# Items
Windows Form Visual C# Items
User Control Visual C# Items
Component Class Visual C# Items
User Control (WPF) Visual C# Items
About Box Visual C# Items
ADO.NET Entity Data Model Visual C# Items
Application Configuration File Visual C# Items
Application Manifest File Visual C# Items
Type: Visual C# Items
An empty class definition
Name: Settings
Add
Cancel

Click Add. In the Solution Explorer window, observe your new class named Settings.cs containing the following boiler-plate code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ASCOM.SettingsDemo
{
    class Settings
    {
    }
}
```

3. Modify the class definition and make it partial internal, thus:

```
    internal partial class Settings
```

4. Add the following two attributes to the class

```
[DeviceId("ASCOM.SettingsDemo.Focuser", DeviceName = "Demo focuser")]
[SettingsProvider(typeof(ASCOM.SettingsProvider))]
internal partial class Settings {}
```

5. That's it; there should be no code at all in this class. If there is, delete it. Build your project and check there are no errors. If there are any errors, please fix them before proceeding.