SLED Plugin Guide

Please note that the specifications contained in this document are preliminary and subject to change without prior notice.

This document provides a guide to writing plugins for SLED. For information about installing and configuring SLED, see *Getting Started with SLED and Lua*. For information about using SLED, see the *SLED User's Guide*.

Version	Revision Date	Author(s)	Comments
3.2.0	2-Oct-09	Risa Galant	Initial Version.
3.3.0	16-Feb-10	Risa Galant	Updates for 3.3.0 version, copyright, filename
5.0.0	Apr-14	Gary Staas	Updates for 5.0.0. Update format. Update file paths. Update UI figures. Update API references. Update sample plugin code.
5.1.0	Oct-14	Gary Staas	Update for 5.1.0. Minor clarifications.
5.1.2	Feb-15	Gary Staas	Open source version.

Table of Contents

1 Introduction

SLED is a powerful, yet intuitive, IDE for editing and debugging Lua and other scripts. Based on the TNT Authoring Tools Framework, it delivers an array of features normally associated with a full-fledged development environment such as Microsoft Visual Studio.

The default scripting language for use with SLED is Lua. You can develop plug-ins to SLED that allow you to use SLED as the editing and debugging tool for the scripting language of your choice. This guide provides guidelines for developing language plug-ins for SLED.

About this Guide

This Guide is not a tutorial: it provides guidance to experienced developers who would like to use SLED as the editing and debugging tool for the scripting language of their choice. You should be well-versed in your scripting language, know your requirements, and have a firm knowledge of ATF. Your knowledge of ATF should include:

- How to get user-created windows to appear.
- How to add custom toolbars, commands, controls, menus, and so on.

ATF is an open source project, and you can get more information at its home page.

There are many interfaces in SLED that you can use, so what you do depends on what you would like to do in SLED with your preferred scripting language. For example, you could choose to have your plugin:

- Highlight syntax.
- Check syntax.
- Display information such as variable lists and the call stack.
- Any other custom language-specific tasks.

This guide does not detail how to implement specific tasks: it only provides guidance and the requirements for hooking your custom scripting-language plugin into SLED.

There are two aspects to writing plugins for the SLED and Lua package:

- The runtime side: LibSledDebugger is a library that allows SLED to communicate with the target machine, serving as the runtime interpreter that enables scripts to execute.
- The SLED (tool) side: IDE for editing and run-time debugging of scripts.

In general, the scripting language needs a way to get information out, and to know where it is in the execution: for example, has it hit a breakpoint?

You can use any of the exposed interfaces to develop your SLED plugin. Information about the exposed interfaces is available in the *SLED Reference*, a CHM-format reference.

What This Guide Contains

In addition to this introduction, this guide includes the following chapters:

- LibSledDebugger Runtime-side Plugin Library: Recommendations, methods and events, and starting points.
- SLED-Side Plugins: Recommendations and starting points.
- Sample Plug-in Implementation: Descriptions of a simple plugin implementation.

Additional Resources

Refer to the following resources for more information about the SLED and Lua projects:

- The "Installing and Configuring SLED" chapter in *Getting Started with SLED and Lua*
- Script Language Editor and Debugger home page

2 LibSledDebugger Runtime-side Plugin Library

LibSledDebugger is the runtime-side plugin library that allows SLED to communicate with the target machine, serving as the runtime interpreter that enables scripts to execute. SLED runtime plugins are written in C/C++.

SledDebugger Class

`SledDebugger` is the class for the SLED debugger object that allows debugging scripts during run time. To debug an application with SLED, the application must create a `SledDebugger` instance when it runs. The SLED tool communicates with `SledDebugger` instances. The LibSledDebugger library handles `SledDebugger` instance creation.

SledDebuggerPlugin Class

The runtime-side plug-in has a C++ API consisting of an abstract base class, `SledDebuggerPlugin`, and several methods that provide debug event notification hooks. All language plugins must inherit from the `SledDebuggerPlugin` base class and must implement the event notification methods to enable using SLED as your scripting language editor and debugger.

The LibSledDebugger plugin API is defined in the file `plugin.h`, located in the `components\sce_sled\src\sleddebugger` folder of your SLED installation. Other useful header files are `sleddebugger.h`, `scmp.h`, `buffer.h`, `params.h`, and `utilities.h`.

All plugins must have a `unit16_t pluginId` that is unique among all existing plugins. The identifier 0 is reserved for LibSledDebugger; the identifier 1 is used by LibSledLuaPlugin.

Event Notification Methods

Each language plugin must implement the following methods declared in `plugin.h` so that event notification can occur:

- `clientConnected()`: Called by the SledDebugger when a SLED client connects.
- `clientDisconnected()`: Called by the SledDebugger when a SLED client disconnects.
- `clientMessage()`: Called by the SledDebugger when a message sent by SLED is dispatched to the language plugin.
- `clientDebugModeChanged()`: Called by the SledDebugger to let plugins know when the debug mode has changed. The debug modes are listed in an

enumeration in `params.h` and correspond to the debug toolbar in SLED, which contains items like "start", "step into", "step over", "step out", and "stop".

- `clientBreakpointBegin()`: Called by the SledDebugger when a breakpoint is hit. `ClientBreakpointBegin()` passes a structure, `BreakpointParams`, that describes which language plugin hit the breakpoint. See BreakpointParams Structure for more information.
- `clientBreakpointEnd()`: Called by the SledDebugger to resume execution from the breakpoint.

BreakpointParams Structure

When a language plugin needs to hit a breakpoint, it creates a `BreakpointParams` structure and calls `debuggerBreakpointReached()`. The `BreakpointParams` structure is defined in `params.h` and contains several properties:

- `pluginId`: Unique identifier of the language plugin that is hitting the breakpoint. For example, if LibSledLuaPlugin is hitting the breakpoint, `pluginId` is set to 1 because the unique identifier of LibSledLuaPlugin is 1. `pluginId` enables all other runtime-side plugins and SLED to know which plugin is hitting a breakpoint.
- `lineNumber`: Refers to the line number that the breakpoint is on.
- `relFilePath`: Relative path (from the asset directory) of the script file that the breakpoint is in.

The `lineNumber` and `relFilePath` fields allow the SLED-side language plugin to know which file to open and line number to jump to when the breakpoint is hit.

Plugin Requirements

Note the following plugin requirements:

- The scripting language must be able to get information out and know where it is in its script execution. For example, Lua provides a C API for getting information out, as well as debug hook functionality to enable notifying the user when a function is being called, when a line of code is being executed, when a function is returning, and so on.
- The language plug-in must test whether a breakpoint has been hit and store all the information that it needs to enable it to react appropriately to the breakpoint.
- When the language plugin determines that it has hit a breakpoint, it should call LibSledDebugger's `debuggerBreakpointReached()` method to start communication with SLED. All other script execution is halted either until SLED indicates to resume execution, or SLED disconnects.
- When the plugin calls `debuggerBreakpointReached()`, LibSledDebugger notifies all current language plugins that a breakpoint has

been reached and provides the ID of the plugin that hit the breakpoint. This is how other language plugins know that their execution is being halted in case they need to react. For example, LibSledLuaPlugin keeps some timers for the profiler. The timers need to be stopped if any language plugin hits a breakpoint, lest they be off by however long they were stopped on the breakpoint.

3 SLED-Side Plugins

SLED plugins can be language plugins to support a particular language in SLED, network plugins that allow connecting SLED to devices using a particular communications protocol, or plugins that perform some other task.

All SLED-side plugins reside in the `bin\sce_sled\SLED.[VS Version]\Plugins` folder of your SLED installation, where `[VS Version]` is the version of Visual Studio used to build the plugin. You must place your SLED-side plugin in that folder. The `Plugins` folder contains SLED network plugins, such as `Sled.Net.Deci3.dll` and `Sled.Net.Tcp.dll`, as well as a SLED language plugin, `Sled.Lua.dll`, that handles all aspects of Lua debugging in SLED.

SLED plugins are Managed Extensibility Framework (MEF) components written in C# or C++/CLI. There's a dynamic plugin finder service inside SLED that looks for DLLs in the SLED `Plugins` directory that have special SLED attributes on them. SLED tries to add any classes containing these attributes to the MEF type catalog, where MEF can then take over doing the instantiating and initializing. For more information on how ATF uses MEF, see MEF with ATF in the ATF Programmer's Guide.

There are 3 plugin attributes: ATF based, SLED language plugin based, and SLED network plugin based. All plugins have to have the ATF attribute. The network and language attributes are optional.

Mark as ATF-Based Plugin

All SLED plugins must use the `ATFPluginAttribute` in their `AssemblyInfo.cs` file, as follows:

```
// Mark assembly as an ATF Plugin
[assembly: Sce.Atf.AtfPluginAttribute]
```

The suffix "Attribute" is actually redundant, so you can also use this:

```
// Mark assembly as an ATF Plugin
[assembly: Sce.Atf.AtfPlugin]
```

Plugins marked with the `ATFPluginAttribute` and placed in the `Plugins` folder are seen by SLED the next time it is run.

SLED Language Plugins

All SLED-side language plugins must be MEF components. The following are additional options for SLED-side language plugins:

- Must be marked as an ATF-based plugin in their respective `AssemblyInfo.cs` file.
- Must be marked as a SLED language plugin in their respective `AssemblyInfo.cs` file.
- Must implement the `ISledLanguagePlugin` interface.

Note:
Not all SLED plugins have to be SLED language plugins: it is also fine to have and use ATF-based plugins in SLED.

Mark as SLED-Side Language Plugin

In addition to the requirement that they must be ATF-based plugins, all SLED-side language plugins must be marked as SLED language plugins. Add `SledLanguagePluginAttribute` to the plugin's `AssemblyInfo.cs` file, as follows:

```
// Mark assembly as a SLED Language Plugin
[assembly: Sce.Sled.Shared.Plugin.SledLanguagePluginAttribute]
```

or

```
// Mark assembly as a SLED Language Plugin
[assembly: Sce.Sled.Shared.Plugin.SledLanguagePlugin]
```

Implement ISledLanguagePlugin Interface

SLED language plugins must implement the `ISledLanguagePlugin` interface. The `ISledLanguagePlugin` interface contains the following members:

```
/// <summary>
/// Base interface for adding a language to SLED</summary>
public interface ISledLanguagePlugin
{
    /// <summary>
    /// Name of the language
    /// </summary>
    string LanguageName
    {
        get;
    }

    /// <summary>
    /// File extensions this language supports
    /// </summary>
    string[] LanguageExtensions
    {
```

```
        get;
    }

    /// <summary>
    /// Description of the language
    /// </summary>
    string LanguageDescription
    {
        get;
    }

    /// <summary>
    /// Language Id for network messages
    /// </summary>
    UInt16 LanguageId
    {
        get;
    }
}
```

SLED Network Plugins

Mark as SLED-Side Network Plugin

In addition to the requirement that they must be ATF-based plugins, all SLED-side network plugins must be marked as SLED network plugins. Add `SledNetworkPlugin` to the plugin's `AssemblyInfo.cs` file, as follows:

```
// Mark assembly as a SLED Network Plugin
[assembly: Sce.Sled.Shared.Plugin.SledNetworkPluginAttribute]
```

or

```
// Mark assembly as a SLED Language Plugin
[assembly: Sce.Sled.Shared.Plugin.SledLanguagePlugin]
```

Implement ISledNetworkPlugin Interface

SLED network plugins must implement the `ISledNetworkPlugin` interface. The `ISledNetworkPlugin` interface contains the following members:

```
/// <summary>
/// Base network plugin interface for SLED
/// <remarks>Derived from IDisposable so a Dispose method is needed as well and
/// gets called by SLED after disconnecting from a target or if an unhandled
/// exception event is fired and received by SLED</remarks>
/// </summary>
public interface ISledNetworkPlugin : IDisposable,
ISledNetworkPluginPersistedSettings
{
    /// <summary>
    /// Make a connection to a Target
    /// <remarks>Upon successful connection to a target the
```

```
    /// ConnectedEvent event should be fired or if an error
    /// occurs the UnHandledExceptionEvent should be fired.</remarks>
    /// </summary>
    /// <param name="target"></param>
    void Connect(ISledTarget target);

    /// <summary>
    /// Returns whether or not the plugin is connected to a target
    /// <remarks>This method should not lock or rely on locking mechanisms as it
    /// will get called a lot as the status of GUI elements get updated to
reflect
    /// the connection state.</remarks>
    /// </summary>
    bool IsConnected
    {
        get;
    }

    /// <summary>
    /// Disconnect from the target
    /// <remarks>This method should fire the DisconnectedEvent after successful
    /// disconnection or fire the UnHandledExceptionEvent if an error occurs
    /// while trying to disconnect.</remarks>
    /// </summary>
    void Disconnect();

    /// <summary>
    /// Send data to the target
    /// </summary>
    /// <param name="buffer">data to send</param>
    /// <returns>Length of data sent or -1 for failure</returns>
    int Send(byte[] buffer);

    /// <summary>
    /// Send data to the target
    /// <remarks>Allow specification of length of buffer</remarks>
    /// </summary>
    /// <param name="buffer">data to send</param>
    /// <param name="length">length of data to send</param>
    /// <returns>Length of data sent or -1 for failure</returns>
    int Send(byte[] buffer, int length);

    /// <summary>
    /// Event to fire after connecting to a target
    /// </summary>
    event ConnectionHandler ConnectedEvent;

    /// <summary>
    /// Event to fire after disconnecting from a target
    /// </summary>
    event ConnectionHandler DisconnectedEvent;

    /// <summary>
    /// Event to fire when data has been received from the target
    /// </summary>
    event DataReadyHandler DataReadyEvent;

    /// <summary>
    /// Event to fire when an unhandled exception has occurred
    /// </summary>
    event UnHandledExceptionHandler UnHandledExceptionEvent;

    /// <summary>
```

```
    /// Name of plugin
    /// <remarks>This can be anything</remarks>
    /// </summary>
    string Name
    {
        get;
    }

    /// <summary>
    /// Protocol plugin uses
    /// <remarks>This should be a simple string like "TCP", "DECI3",
etc.</remarks>
    /// </summary>
    string Protocol
    {
        get;
    }

    /// <summary>
    /// Create a settings control based on a specific target
    /// </summary>
    /// <param name="target">Target to build settings from or null if no target
created yet</param>
    /// <returns>Settings control</returns>
    SledNetworkPluginTargetFormSettings CreateSettingsControl(ISledTarget
target);

    /// <summary>
    /// Create and setup a new target
    /// </summary>
    /// <param name="name">Name of the target</param>
    /// <param name="endPoint">IP and port of the target</param>
    /// <param name="settings">Optional settings to pass in</param>
    /// <returns>A new target or null if some kind of error happened</returns>
    ISledTarget CreateAndSetup(string name, IPEndPoint endPoint, params object[]
settings);

    /// <summary>
    /// Create and return any automatically generated targets
    /// <remarks>Some network plugins may be able to notify SLED of targets
    /// based on outside software</remarks>
    /// </summary>
    ISledTarget[] ImportedTargets { get; }

    /// <summary>
    /// Unique identifier for the network plugin
    /// </summary>
    Guid PluginGuid { get; }
}
```

4 Sample Plug-in Implementation

This chapter describes how to create a sample ATF-based plugin for SLED that is later converted to a SLED language plugin. The sample plugin, SledSampleATFPlugin, creates a menu and two commands that can be used in SLED.

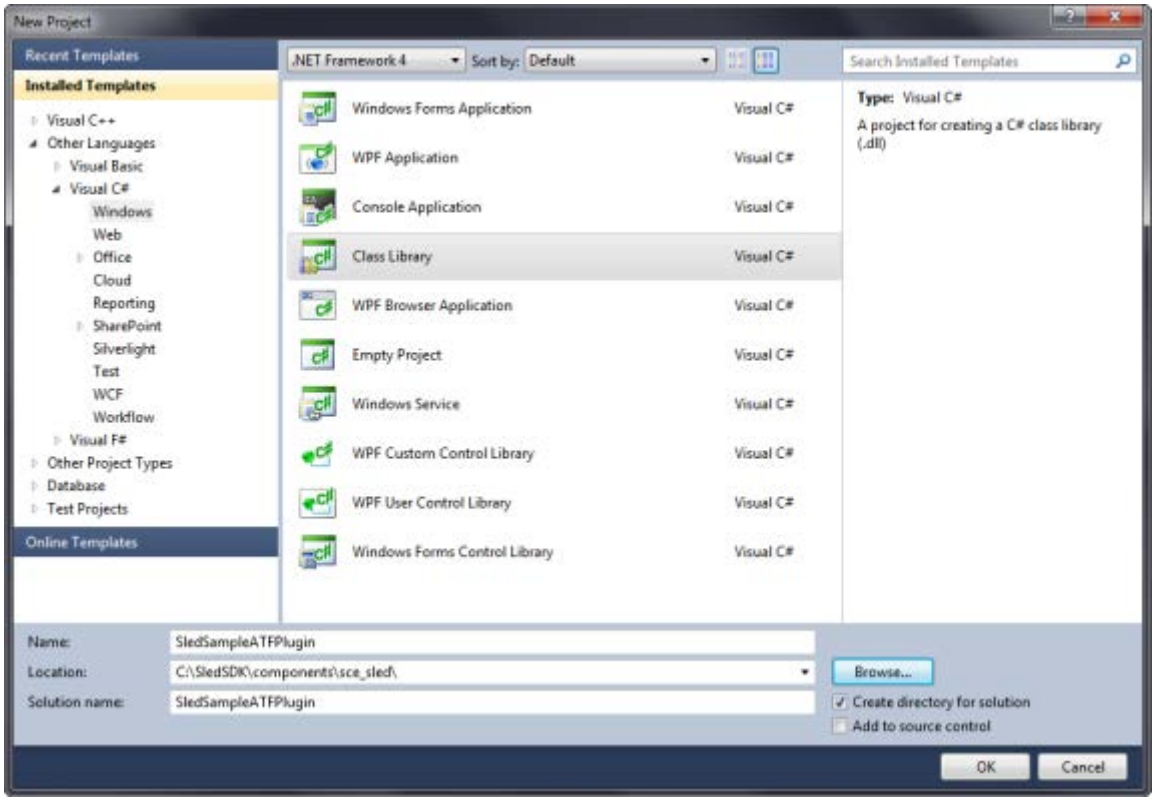
To implement this sample, you need Visual Studio 2010 and several DLLs, as noted.

Create the SledSampleATFPlugin Sample

This section describes how to create a sample SLED language plugin, SledSampleATFPlugin. The code in this section is built upon in subsequent sections.

1. Open Visual Studio 2010 and create a new C# class library project, as shown in the following figure.

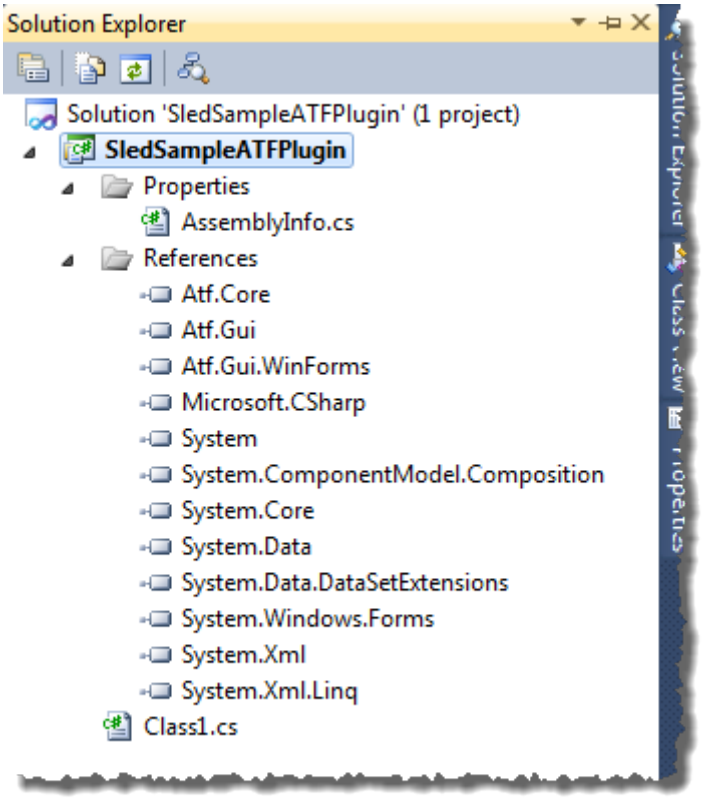
Figure 1 Create a New C# Class Library Project in Visual Studio



2. Add the following DLLs as references, as shown in the following figure:

- `Atf.Core.dll`
- `Atf.Gui.dll`
- `Atf.Gui.WinForms.dll`
- `System.ComponentModel.Composition.dll`
- `System.Windows.Forms.dll`

Figure 2 Additional References



3. Create a class that is a Managed Extensibility Framework (MEF) component. For more information on how ATF uses MEF, see MEF with ATF in the ATF Programmer's Guide.

The following code illustrates creating a class, `Class1`. It implements `IInitializable` to create a new menu **Sample Atf Plugin** with two commands. It implements `ICommandClient` to perform the commands' actions.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.ComponentModel.Composition;

using Sce.Atf;
```

```
using Sce.Atf.Adaptation;
using Sce.Atf.Applications;

namespace SledSampleATFPlugin
{
    [Export(typeof(IInitializable))]
    [PartCreationPolicy(CreationPolicy.Shared)]
    public class Class1 : ICommandClient, IInitializable
    {

        [ImportingConstructor]
        public Class1(
            ICommandService commandService,
            IControlHostService controlHostService,
            IControlRegistry controlRegistry)
        {
            m_commandService = commandService;
            m_controlHostService = controlHostService;
            m_controlRegistry = controlRegistry;
        }

        enum Command
        {
            SomeCommand1,
            SomeCommand2
        }

        enum Menu
        {
            SampleAtfPlugin
        }

        enum Group
        {
            SampleAtfPlugin
        }

        /// <summary>
        /// Finishes initializing component by registering tab commands</summary>
        public virtual void Initialize()
        {
            m_commandService.RegisterMenu(Menu.SampleAtfPlugin, "Sample Atf
Plugin", "Sample Atf Plugin Description");
            m_commandService.RegisterCommand(
                            Command.SomeCommand1,
                            Menu.SampleAtfPlugin,
                            Group.SampleAtfPlugin,
                            "SomeCommand1",
                            "SomeCommand1 Description",
                            Keys.None,
                            null,
                            CommandVisibility.All,
                            this);

            m_commandService.RegisterCommand(
                Command.SomeCommand2,
                Menu.SampleAtfPlugin,
                Group.SampleAtfPlugin,
                "SomeCommand2",
                "SomeCommand2 Description",
                Keys.None,
                null,
                CommandVisibility.All,
```

```
                this);
        }

        public virtual bool CanDoCommand(object commandTag)
        {
            bool bCanDoCommand = false;

            if (commandTag is Command)
            {
                switch ((Command)commandTag)
                {
                    case Command.SomeCommand1:
                        bCanDoCommand = true;
                        break;

                    case Command.SomeCommand2:
                        bCanDoCommand = true;
                        break;
                }
            }

            return bCanDoCommand;
        }

        public virtual void DoCommand(object commandTag)
        {
            if (commandTag is Command)
            {
                switch ((Command)commandTag)
                {
                    case Command.SomeCommand1:
                        MessageBox.Show("SomeCommand 1", "Sample Atf Plugin");
                        break;

                    case Command.SomeCommand2:
                        MessageBox.Show("SomeCommand 2", "Sample Atf Plugin");
                        break;
                }
            }
        }

        void ICommandClient.UpdateCommand(object commandTag, CommandState
commandState)
        {
        }

        private readonly ICommandService m_commandService;
        private readonly IControlHostService m_controlHostService;
        private readonly IControlRegistry m_controlRegistry;
    }
}
```

4. Add the `ATFPluginAttribute` to `AssemblyInfo.cs`:

```
// Mark assembly as an ATF Plugin
[assembly: Sce.Atf.AtfPluginAttribute]
```

5. Compile the new DLL and place it in the `Plugins` folder.
6. Start SLED. You should see the new menu and be able to use the two new commands, as shown in the following figures.

Figure 3 Sample ATF Plugin Menu Item

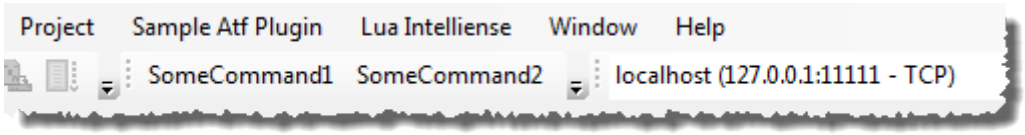
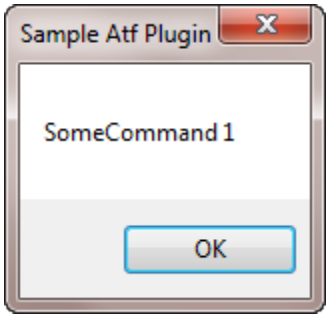
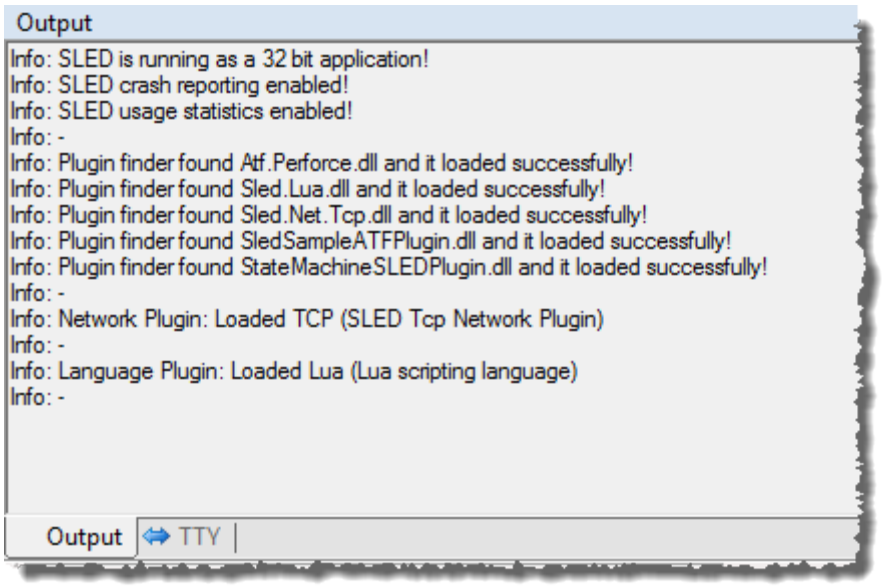


Figure 4 Sample ATF Plugin SomeCommand 1



The **Output** window also shows that the new SledSampleATFPlugin plugin was found and loaded:

Figure 5 Output Window Showing Loaded Plugins



Make the Sample ATF-based Plugin a SLED Language Plugin

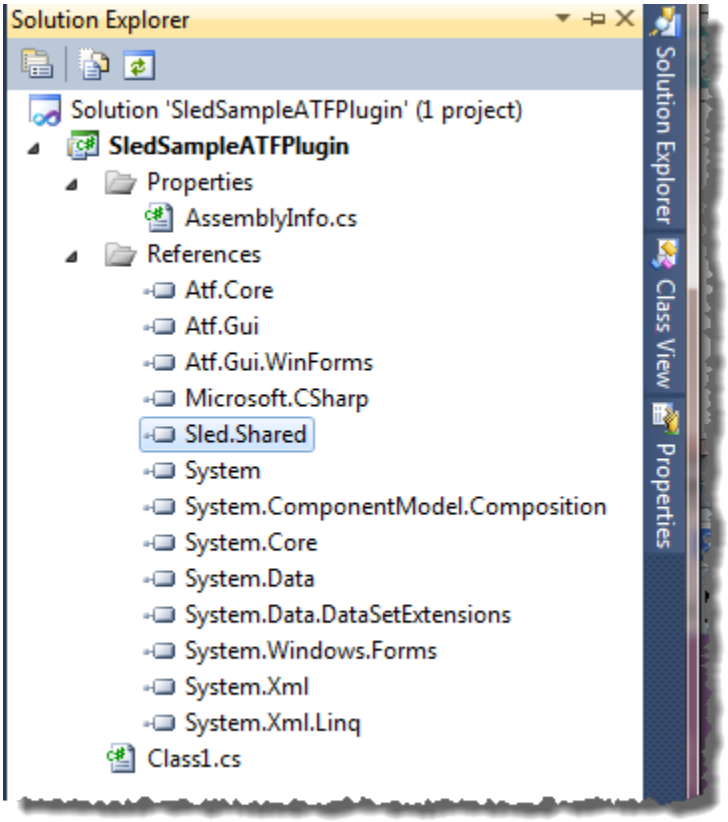
This section describes how to make your sample ATF-based plugin a SLED language plugin. The basic tasks are:

- Tag the plugin with the `SledLanguagePluginAttribute`.
- Implement the `ISledLanguagePlugin` interface in one class in the DLL. This also requires adding `Sled.Shared.dll` as a reference, because `ISledLanguagePlugin` is defined there.

To accomplish this, follow these steps:

1. In Visual Studio, open the SampleAtfPlugin project if it is not already open.
2. Add `Sled.Shared.dll` as a reference, as shown in the following figure:

Figure 6 Adding Sled.Shared.dll as a Reference to SledSampleATFPlugin



3. Mark the assembly as a SLED language plugin by adding the `SledLanguagePluginAttribute` to `AssemblyInfo.cs`, in addition to the `AtfPluginAttribute` attribute already there:

```
// Mark assembly as an ATF Plugin
[assembly: Sce.Atf.AtfPluginAttribute]
// Mark assembly as a SLED Language Plugin
[assembly: Sce.Sled.Shared.Plugin.SledLanguagePluginAttribute]
```

4. Add the `using` statement for the namespace containing the `ISledLanguagePlugin` interface:

```
using Sce.Sled.Shared.Plugin;
```

5. Add the `ISledLanguagePlugin` interface to the class declaration:

```
public class Class1 : ICommandClient, IInitializable, ISledLanguagePlugin
```

6. Add the `ISledLanguagePlugin` interface implementation to `Class1`:

```
namespace SledSampleATFPlugin
{
    [Export(typeof(IInitializable))]
    [PartCreationPolicy(CreationPolicy.Shared)]
    public class Class1 : ICommandClient, IInitializable,
ISledLanguagePlugin
    {
        ...

        #region ISledLanguagePlugin Interface

        public string LanguageName
        {
            // The language name - like "Lua", "Squirrel", etc.
            get { return "SampleAtfPlugin Language"; }
        }

        public string[] LanguageExtensions
        {
            // File extensions this language should be associated with
            get { return new string[] { ".sampleatfplugin" }; }
        }

        public string LanguageDescription
        {
            // Description of the language this plugin implements
            get { return "SampleAtfPlugin Language Description"; }
        }

        public ushort LanguageId
        {
            // A unique identifer amongst all SLED/LibSledDebugger plugins
            get { return 11; }
        }
        #endregion
    }
}
```

7. Compile the DLL and copy it to the `Plugins` folder.
8. Restart SLED. Text appears in the **Output** window showing that SLED has loaded the plugin, as shown previously.

Now that you have a new SLED language plugin, you can add support for your scripting language. SLED provides several interfaces that you can use.

Many services of interest live in the `Sce.Sled.Shared.Services` namespace. For example:

- `ISledBreakpointService`: Subscribe to this service to receive breakpoint events, that is, to know when the user adds or changes breakpoints in open documents in SLED.
- `ISledProjectService`: Use this service to see when project files are opened, closed, or renamed.
- `ISledDocumentService` services: Subscribe to these services to obtain file related events and perform document related tasks, such as opening and closing documents.
- `ISledDebugService`: Subscribe to this service to know when the plugin is connecting to or disconnecting from a target, receiving data from a target, and enable sending data to the target.

See the `SledReference.chm` document for more information about the services available in the `Scea.Sled.Shared.Services` namespace.

Use MEF to import SLED or ATF service components. For more information on using MEF, see MEF with ATF in the ATF Programmer's Guide.

For example, the following code imports components implementing the `ISledBreakpointService`, `ISledProjectService`, `ISledDcoumentService`, and `ISledDebugService` interfaces just described:

```
[Import]
private ISledProjectService m_projectService;
[Import]
private ISledDocumentService m_documentService;
[Import]
private ISledBreakpointService m_breakpointService;
[Import]
private ISledDebugService m_debugService;
```

All of these services expose events and methods that may be useful to the plugin.

`SledReference.chm` is a good guide for finding out more about these and other available services in the `Scea.Sled.Shared.Services` namespace.