|  |
| --- |
| PNNL |
| LcmsNet FluidicsSDK Plugin Tutorial |
| **Christopher Walters** |
| **10/3/2014** |

|  |
| --- |
| This is a tutorial of how to create a fluidics designer enabled device plugin for LcmsNet. |

Contents

[Creating a Fluidics Enabled Plugin 1](#_Toc396741575)

[Creating a Fluidics Device Glyph 3](#_Toc396741576)

[Terms 4](#_Toc396741577)

[Example Plugin 5](#_Toc396741578)

[Example Fluidics Glyph 8](#_Toc396741579)

# Creating a Fluidics Enabled Plugin

1. Select built-in Fluidics Glyph to be used, or create a custom glyph.
2. Create plugin class and above the class declaration put a classDeviceControlAttribute.
   1. classDeviceControlAttribute must have
      1. deviceControlType use typeof(YourUserControl) here. YourUserControl should be a control derived from the IDeviceControl class.
      2. If using a custom fluidics glyph, you need typeof(MyFluidicsGlyph) where MyFluidicsGlyph inherits from the FludicsDevice class in the FluidicsSDK. Otherwise it should be null.
      3. Device name(ex: “Test Valve”)
      4. Device Category(ex: “Valves”)
      5. Example:

**[classDeviceControlAttribute(typeof(MyAdvancedControl), typeof(MyFluidicsGlyph), “”, -1, false)]**

**Public class MyDevice**

**{**

**}**

Would define that the device had an advanced control of type MyAdvancedControl, a fluidics glyph of type MyFluidicsGlyph, no data provider, and does not supply performance data

* + 1. Example:

**[classDeviceControlAttribute(null, null, “Test Valve”, “Test Devices”)]**

**Public class MyDevice**

**{**

**}**

This would define that the device has no advanced control, no fluidics glyph (unless it’s using a built-in and is implementing the built-ins interface), is named “Test Valve”, and belongs to the group “Test Devices”.

1. The class must implement the IDevice interface and if using a lcmsnet built-in glyph the interface associated with the glyph(ex: must implement the IFluidicsPump interface to use the FluidicsPump glyph)
   1. **[classDeviceControlAttribute(typeof(myDeviceAdvancedControl), null, “MyDevice”, “Valves”)]**

**Public class MyDevice:IDevice, IFluidicsPump**

**{**

**//IFluidicsPump interface implementation here.**

**}**

1. Write code for the required interfaces and the methods to operate the physical device
   1. Each method that operates the device that is to be used in an LCMethod should be marked with the classLCMethodAttribute above its declaration and the attribute requires
      1. Method name(string) to be displayed to the user
      2. Operation Time(double) max amount of time this method should take, or enumMethodOperationTime enum telling lcmsnet max time is user defined.
      3. Data provider (string), this may be an empty string name of an event that will provide data to user controls. This name allows the method editor or other system to register an event handler through the IDevice interface, without having to know what class it is. In essence, if a method accessible to the method editor needs specific data for a parameter, you would provide the name of the event which would be raised when the list of available parameters changed. For instance, if you had a method called “Load File” you might have a data provider event named “AvailableFiles” which, when fired, would return a list of type string, containing all the files available to be loaded, thus allowing the method editor to retrieve the list of files that the user could tell the device to load. It also may be used by advanced controls or glyphs to retrieve information to display.
      4. Data provider index (integer), -1 for none.
      5. Performance data available (boolean) tells lcmsnet if the device provides performance data.
      6. Example:

**[classDeviceControlAttribute(typeof(myDeviceAdvancedControl), null, “MyDevice”, “Valves”)]**

**Public class MyDevice:IDevice, IFluidicsPump**

**{**

**[classLCMethodAttribute(“Start Pump”, enumMethodOperationTime.Parameter, “”, -1, false)]**

**Public void StartPump()**

**{**

**If(!EmulationEnabled)**

**{**

**//Operate device**

**}**

**}**

**//IFluidicsPump interface implementation here.**

**}**

* 1. Methods should respond to the emulation property, if the property is true, the method should not attempt to communicate with a physical device, which may not be connected.
  2. If an error occurs in operation, the plugin may report them to lcmsnet in 3 ways, via a “false” Boolean return value, by raising a C# standard Exception, or by raising a Notification system event.
     1. Boolean return values should be used when the device fails to perform an operation as directed, but is still capable of performing its operations.
     2. Exceptions should be raised when the device the plugin represents cannot perform an operation and is not able to continue performing other operations.
     3. The notification system should be used when an operation cannot be performed, but the plugin may be able to recover the device to an operational or “safe” status via certain methods or operations which the user may be able to define in an LCMethod.
        1. To use the notification system, the plugin must define what notifications it can report in the GetErrorNotificationList and GetStatusNotificationList required methods.
        2. In the method which is to use the notification system, it must call one of the interface required events of “Error” or “StatusUpdate” with a classDeviceErrorEventArgs or classDeviceStatusEventArgs argument.

1. Any properties that you wish to be persisted between program restarts should be marked with the classPersistenceAttribute above the declaration. This attribute tells LcmsNet to add the value of the property to the HardwareConfig.ini file that is saved upon shutdown. It also tells LcmsNet to load this value into the property from the HardwareConfig.ini file at program start up. classPersistenceAttribute requires The device itself should know how to turn the desired attribute into a string upon get request, and how to convert it back into a useful attribute again from its string representation upon set request:
   * 1. Property name(string)
        1. Example

**[classPersistenceAttribute(“Name”)]**

**Public string Name**

**{**

**Get;**

**Set;**

**}**

1. Implement an Advanced User Control. This should be a WinForms User Control object that inherits from IDeviceControl. This control allows the user full control over the physical device.

Optional (but recommended) step:

1. Implement a fluidics device glyph
   1. If you are going to use a built-in fluidics device you will need to make sure the plugin implements the fluidics devices’ associated interface. For instance, if you wanted to use the fluidics pump glyph, you would need to implement the IFluidicsPump interface in the plugin.
   2. If you intend to use a custom glyph, you will need to first create the glyph, as covered in [Creating a Fluidics Device Glyph](#_Creating_a_Fluidics). Once created you will need to add the type of the glyph to the classDeviceControlAttribute so that LcmsNet and the FluidicsSDK know what to use.

# Creating a Fluidics Device Glyph

All glyphs for use in the FluidicsSDK must inherit from the abstract FluidicsDevice base class. This class provides basic functionality that may be overridden in order to display the device to the user. The following steps are required to implement a fluidics device glyph.

1. Inherit from the FluidicsDevice abstract class.
   1. Implement all abstract methods.
      1. Some of them may end up doing nothing, for instance not all devices will have states, so the “ActivateState” method may be implemented to simply ignore any calls.
   2. Override any virtual methods that are not suitable to the device. Some likely candidates are DrawControls(), and GenerateState().
2. Define all primitives that represent the device.
   1. Primitives currently available are FluidicsCircle, FluidicsRectangle, FluidicsTriangle, and FluidicsLine.
      1. add these via the AddPrimitive() method of the FluidicsDevice base class. You can also add the Rectangles and Circles via AddRectangle and AddCircle methods.
         1. First create an object of one of the concrete GraphicsPrimitives, FluidicsRectangle, FluidicsCircle, FluidicsTriangle, or FluidicsLine.
         2. Use AddPrimitive(object) to add it to the list of primtives to be drawn as part of this glyph, UNLESS it is supposed to cause an action to occur when clicked.
         3. If the glyph should cause an action to occur us AddPrimitive(Object, ActionDelegate), where ActionDelegate is a parameterless void method.
            1. The default implementation of DrawControls is for 2 ActionPrimitives. If you are going to have fewer, or more, review the DrawControls method in the FluidicsDevice base class and override it to fit your requirements.
      2. New primitives can be defined by inheriting from the GraphicsPrimitive abstract class in the FluidicsSDK.
3. Add any needed ports to the device using the AddPort method, it is overloaded so you can either add a default-constructed port at a specific location (via providing a System.Drawing.Point), or construct your own Port object and add that.
4. The device will need to know how to generate its own states, so review the GenerateState() method and override it if the default implementation is insufficient.
5. The device will need to know how to move itself. If the review the implementation of MoveBy() in the FluidicsDevice and override if necessary.
6. Set ports that are sources of fluid into the system as sources by setting the Source property of that port to true. Do the same for ports that are Sinks for fluids to exit the system by setting that ports Sink property to true.

# Terms

**IDevice**

The basis of this plugin architecture is the **IDevice** interface.  There are a number of things the **IDevice** interface requires each plugin extend.  Some of these are required by the dynamic notification service that is used to recover from system failures.  Most of this is done through publishing lists of internal named events.

**IDeviceControl**

The basis of the winforms advanced control system. All advanced controls for plugins must be derived from this class.

**Persistence**

Persistence is done to a flat file (INI) so no object hierarchies are preserved.  No custom parsing is required.  Regardless, each plugin for this application requires the persistence properties (like below) to be decorated.

       [classPersistenceAttribute("TotalMonitoringMinutes")]

        public int TotalMonitoringMinutesDataToKeep

        {

            get;

            set;

        }

# Example Plugin

Suppose we had a device whose only job was to send an electrical pulse when told to, called a trigger. First we would define the plugin itself:

using TriggerLib; // the driver library for our device if needed.

using LcmsNetDataClasses.Devices;

using LcmsNetDataClasses.Method;

[Serializable]

[classDeviceControlAttribute(typeof(TriggerAdvancedControl), typeof(CustomTriggerFluidicsGlyph), “Trigger”, “Electric Triggers”)] // The TriggerAdvancedControl listed here would have to be defined as well, but that’s just a WinForms IDeviceControl derived object, so I’m not going to go over it here.

public class Trigger:IDevice

{

private TriggerLib.Trigger device;

public Trigger()

{

//construct the plugin

device = new TriggerLib.Trigger(args);

}

[classLCMethodAttribute(“TriggerOn”, 1.0, false, -1, false)]

public void TriggerOn()

{

If(!EmulationEnabled)

{

try

{

device.Trigger(args);

}

catch(Exception ex)

{

//recover from error, report failure via rethrow or notification system

}

}

}

[classLCMethodAttribute(“TriggerOff”, 1.0, false, -1, false)]

public void TriggerOff()

{

If(!EmulationEnabled)

{

try

{

device.Trigger(args);

}

catch(Exception ex)

{

// do something with the exception

}

}

}

[classPersistenceAttribute(“Name”)]

public string Name

{

get;

set;

}

#Region IDevice required methods

Public bool Initialize(ref string errorMessage)

{

//initialize hardware here

}

Public bool Shutdown()

{

//shutdown hardware here

}

Public void RegisterDataProvider(string key, DelegateDeviceHasData remoteMethod)

{

Switch(key.ToUpper())

{

case “DataProvider1”:

DataProvider1 += remoteMethod;

Break;

case “DataProvider2”:

DataProvider2 += remoteMethod;

Break;

}

}

Public void UnregisterDataProvider(string key, DelegateDeviceHasData remoteMethod)

{

Switch(key.ToUpper())

{

case “DataProvider1”:

DataProvider1 -= remoteMethod;

Break;

case “DataProvider2”:

DataProvider2 -= remoteMethod;

Break;

}

}

Public List<string> GetStatusNotifications()

{

return new List<string>();

}

Public List<string> GetErrorNotifications()

{

List<string> errors = new List<string>();

errors.Add(CONST\_ERROR\_STRING1);

errors.Add(“Error1”);

return errors;

}

Public FinchComponentData GetData()

{

FinchComponentData myData = new FinchComponentData();

//modify myData as needed

return myData;

}

Public void WritePerformanceData(string directoryPath, string methodName, object[] parameters)

{

//no performance data, so ignore this call

}

#endregion

#region Events

Public event EventHandler<classDeviceStatusEventArgs> StatusUpdate();

Public event EventHandler<classDeviceErrorEventArgs> Error;

#endregion

}

# Example Fluidics Glyph

using FluidicsSDK.Base;

using FluidicsSDK.Graphics;

using Trigger; //our device plugin library, if the fluidics device glyph isn’t defined in the same namespace/file

public class TriggerGlyph:FluidicsDevice

{

private Trigger device;

public TriggerGlyph()

{

Point p = new Point(0,0);

Size s = new Size(100,100);

FluidicsRectangle r = new FluidicsRectangle(p, s, Color.Black, new Brushes.SolidBrush(Color.White));

base.AddPrimitive(r);

/\* no ports are necessary here, but hey, whatever, lets add one for an example\*/

base.AddPort(new Point(5,5));

/\* or construct one first\*/

Port p1 = new Port(new Point(5,5), this);

base.AddPort(p1);

}

public void ActivateState(int state)

{

// no states to this device to display. Ignore it.

}

protected override void SetDevice(IDevice idevice)

{

device = idevice as Trigger;

}

protected override void ClearDevice(IDevice idevice)

{

If(device == idevice)

{

device = null;

}

}

public override string StateString()

{

return “”;

}

// I could override any other methods I needed to in here as well, but there’s not much reason for this example.

}