■ ECE/CS 4984: Wireless Networking and Mobile Systems ■

Laboratory 5 Pre-lab and In-class Exercise

Part I – Objectives and Lab Materials

Objective:

- Introduce the concept of service discovery and delivery
- Case study of Universal Plug and Play (UPnP)

Hardware to be used in this lab assignment:

- Dell notebook with 802.11b card (with a fully charged battery)

Software to be used in this lab assignment:

- Microsoft Visual Studio .NET 2003
- Intel UPnP SDK and tools

Overview:

Service discovery is the process of searching and advertising services on a network. A service may be an information source, a control source offering some function or a number of other things that haven't been thought of yet. The purpose of service discovery is to make the process of finding and using services simpler. For example, suppose you have your iPAQ and are wandering around the library trying to find a book. If the library has a map service available, you could startup your service browser, find the map service and figure out where you in and where the book is that you are trying to find. This is a trivial example from the pervasive computing world. Services may be much more complex or much simpler than this depending on the application.

This lab introduces service discovery and some of the service discovery protocols. As you read the papers in the next section, think about how each would perform in different types of networks. Think about which protocol would perform best in which type of network. Also, pay attention to the similarities and differences between them. Service discovery is a single basic idea and each protocol provides a slightly different approach to solving the problem.

Part II – Pre-lab Assignment

This portion of the assignment should be completed *prior* to the in-class lab session.

Reading Assignment:

- "A Comparison of Service Discovery Protocols and Implementation of the Service Location Protocol", C. Bettstetter, C. Renner, 2000. This is a fairly old paper on S.D., but it is one of the staples on the subject. Reading section 4 is optional.

  http://citeseer.nj.nec.com/334042.html

- "Protocols for Service Discovery in Dynamic and Mobile Networks", C. Lee, S. Helal, 2002. This paper is a little more recent and talks a bit about the Bluetooth SD protocol in addition to the

ones mentioned in the previous paper.
http://www.harris.cise.ufl.edu/projects/publications/servicediscovery.pdf

- Read the article by S. Helal, "Standards for service discovery and delivery", *IEEE Pervasive Computing*, 2002, pp. 95-100. Note that this article is available from IEEE Xplore which can be accessed at the Virginia Tech library's web site.
- "UPnP Device Architecture 1.0", 2003. This paper is published by the UPnP Forum and describes the architecture of UPnP. Read the introductions to each section.
http://www.upnp.org/resources/documents/CleanUPnPDA101-20031202s.pdf

Tasks:

- Download the 2 Intel UPnP packages from Blackboard. Create 2 directories in your lab 5 directory ('tools' and 'auth tools') and put each package in the corresponding directory. Each package is a self-extracting file. Run them both and direct each to put its contents in the directory.
- The rest of this lab will involving getting acquainted with the tools from Intel. These tools are very useful in developing, testing and deploying UPnP devices and control points. Browse through the files in the directories and get a feel for where things are.

Read the following files:

- 'tools' directory:
    - Readme.htm
    - ToolsHelp.chm (skim this one)
- 'auth tools' directory:
    - Readme.htm
    - IntelAuthoringToolsHelp.chm

Part III – In-class lab assignment

Part A: Working with UPnP Devices

The in-class section of this lab will involve using some of the more useful (for us) tools. Using these tools should also give you a better understanding of how UPnP works.

For this lab you will use an example application from the Intel tools as an UPnP device. Using the sniffer and universal control point applications, you will control the device and monitor the UPnP traffic. There is no report required for this portion of the lab. The purpose is to make you familiar with the tools to do the project.

Procedure:

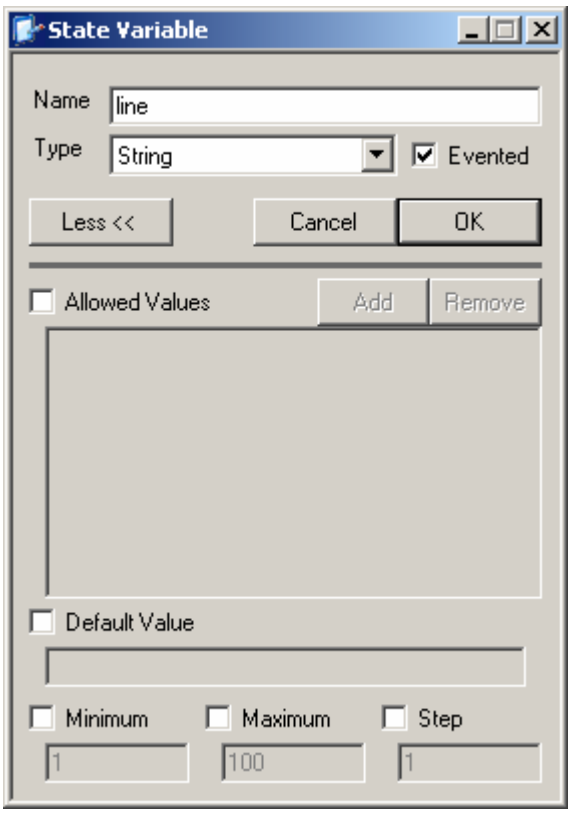
- Insert the 802.11b card into the notebook, set it to ad hoc mode and assign it an address of 192.168.<group number + 100>.1 and a 24 bit netmask (255.255.255.0). The UPnP applications need an address to work with, just like the Pocket PC emulator.
- In the 'tools' directory, start 'Device Sniffer' and 'Device Spy' applications. The 'Device Sniffer' is basically an application that listens for traffic on the UPnP multicast address (239.255.255.250). The 'Device Spy' is essentially a universal control point application and will act as our control point for this lab.

- In the ‘tools\Micro Tools’ directory, start the ‘Micro Light (Windows)’ application. This is an example UPnP device application written by Intel. It will act as our UPnP device. You may control the dimmer and whether the ‘light’ is on or off from the application itself; see the ‘File’ menu.
- When you start the Micro Light application your sniffer should collect a lot of traffic. The device spy should also gain a new section.
- In the device spy, right click on the ‘Intel MicroLight’ and choose ‘Expand all devices’. Browse the 2 services offered by the Micro Light. Clicking on the state variables will give you the information at that current time. Clicking any of the service methods will give you information about that method. Double-clicking a method will allow you to invoke that method. Right clicking on the service name will give you the opportunity to subscribe to that service.
- Using the control point (the Device Spy) invoke some of the methods of the MicroLight service. Toggle the power and change the dimmer from the control point. Watch the messages in the sniffer as you manipulate the device. Clicking any individual message in the sniffer will give you the full message in the lower part of the window. Observe the different message types and what information is carried in them.
- Using the control point, subscribe to the events of both services. This will split the right part of the window into 2 parts. The lower part contains information involving subscribed events. Using the MicroLight application, change the power and dimmer settings. Pay attention to the event subscription window.

Part B: Designing UPnP Devices

Now you will be working on creating UPnP device stacks. There are 3 steps. The first is to create the service description file. The service description file contains all the information about a service, including the actions and state variables available. Using the service description, you will be able to create the UPnP stack for the device. The stack is essentially all the parts of UPnP that you really don’t want to have to write. It handles registering the service, advertising the service, dealing with variable subscriptions and doing the processing of the service requests. The final part is to implement the actions of the service.

- Create a ‘tutorial_device’ directory in your lab 5 directory.
- Creating the service descriptions.
    a. In the ‘tools’ directory, start the ‘Service Author’ application. Service author is another application that makes your life easier. It generates the XML description of the services you wish to offer.
    b. You will be creating 3 actions for you UPnP device. Right clicking in either area will give you the option to add actions or state variables. Add 4 state variables:
        i. ‘line’: string, eventing: on;
        ii. ‘x’, ‘y’, ‘sum’: integer 32; eventing off

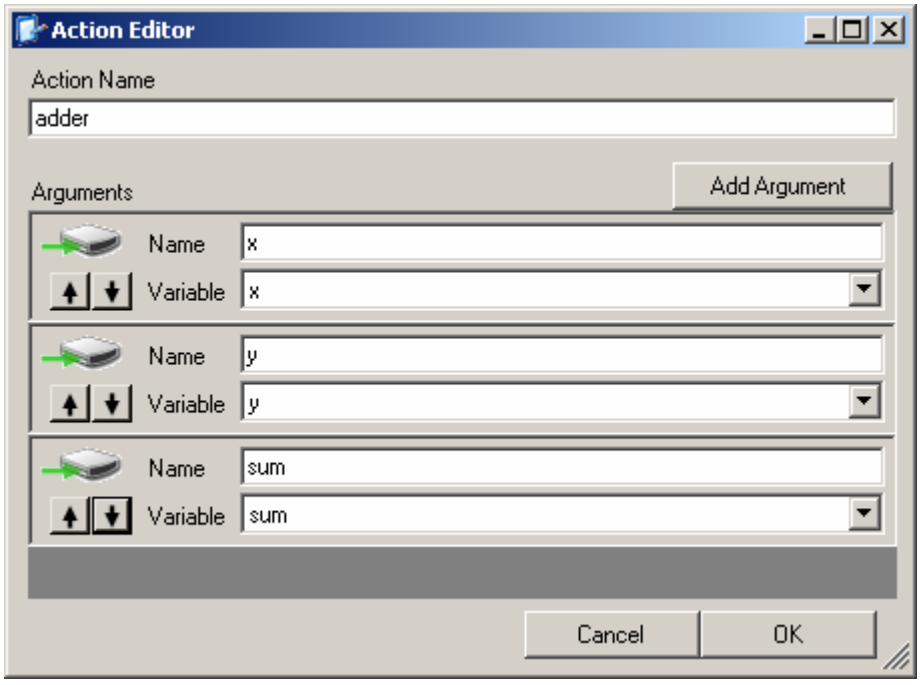


c. Create 3 actions

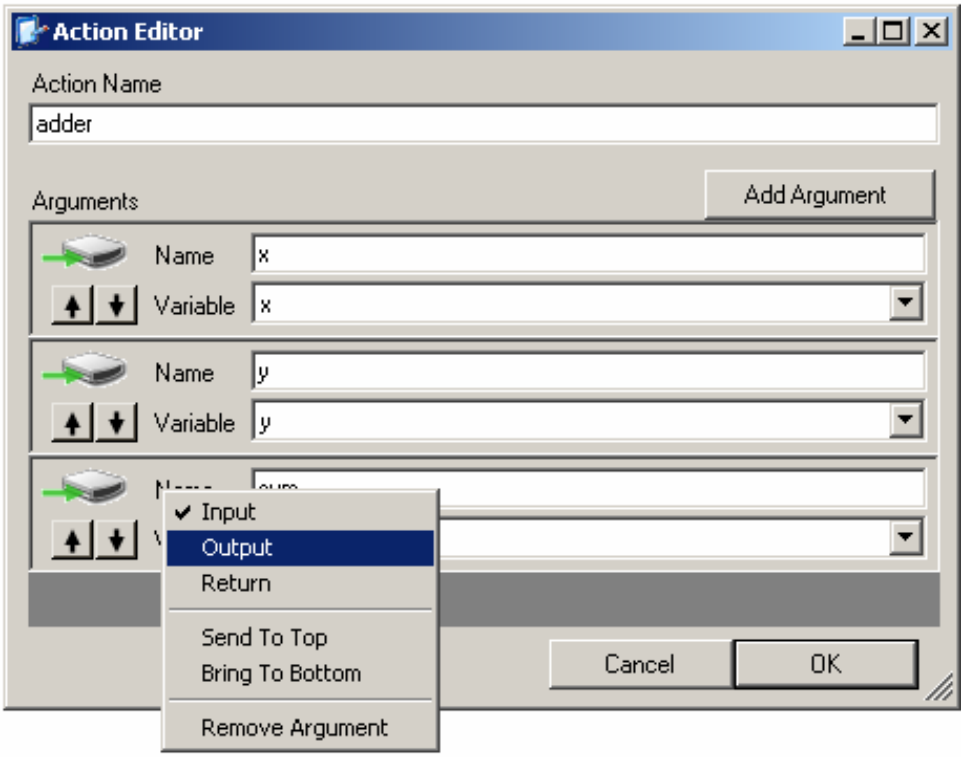
i. ‘getLine’: 1 arg: {‘line’,‘line’} // {‘Name’, ‘Variable’}

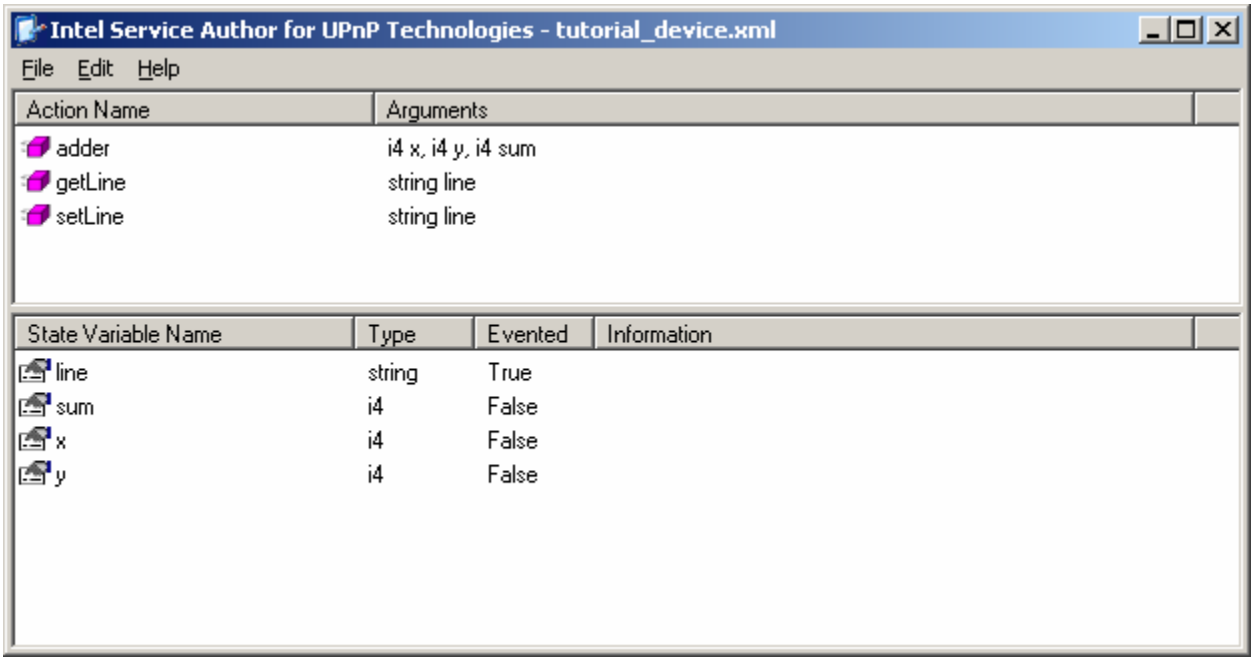
ii. ‘setLine’: 1 arg {‘line’, ‘line’}

iii. ‘adder’: 3 args {‘x’, ‘x’} {‘y’, ‘y’} {‘sum’, ‘sum’}



Changing the direction of arguments: Right click on the green arrow to change the direction of the arguments. Leave ‘x’ and ‘y’ as inputs, but change ‘sum’ to be an output state variable. This means that ‘sum’ will not accept any input and a value can be assigned to it in the action. This value will be sent back to the calling method. Likewise, change the direction of the ‘getLine’ to output.



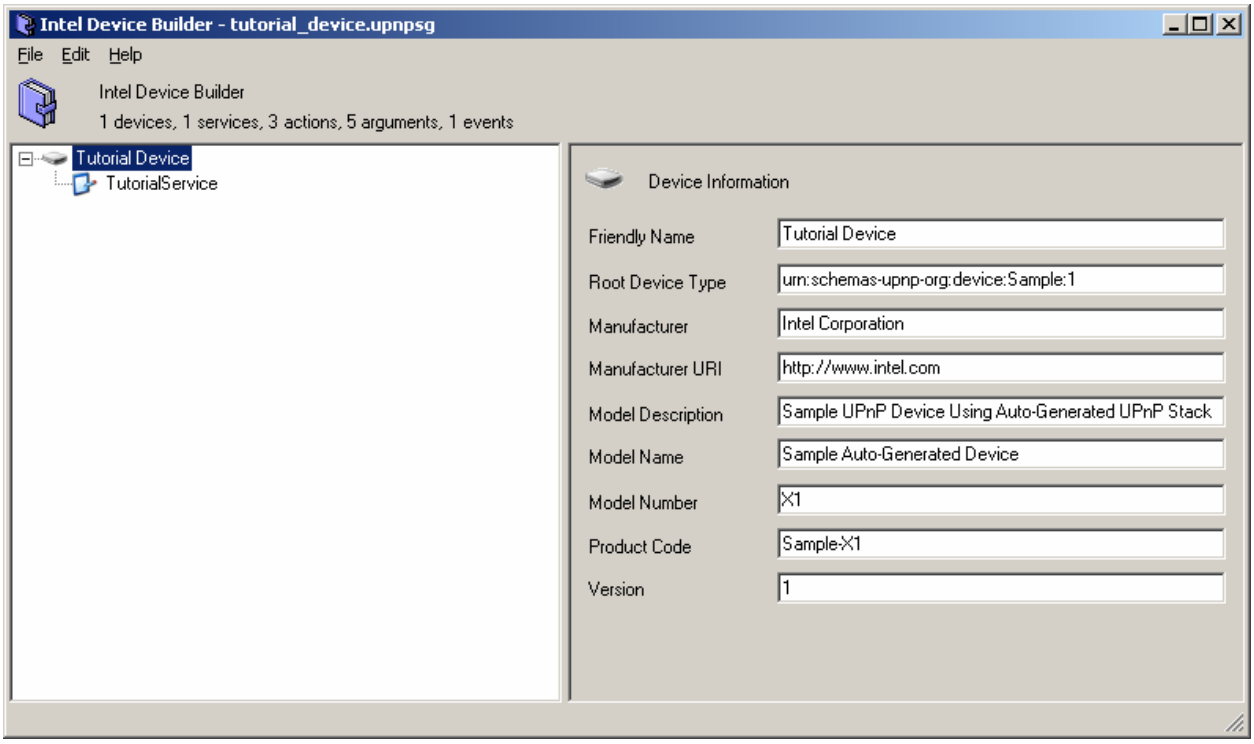


Final ‘Service Author’ example

d. Save the file as ‘tutorial_device.xml’ in the ‘tutorial_device’ directory.

e. Open the file you just created in VS. Identify the different actions and state variables you defined using the service author.

- Creating the device stack
  a. In the ‘auth tools’ directory, start the ‘Device Builder’ application. Writing the service descriptions by hand would be tedious, but not altogether _that_ painful. The service author is a tool that helps make our life easier. Device builder, on the other hand makes it possible to do this project. Given the service description, it generates the UPnP device stack code. This is not a task to be taken lightly. The code that will not see for this lab is mostly a generic UPnP device stack that has the proper registration and invocation calls already done. This makes it much easier to write UPnP devices applications, because all you have to worry about is the code that is _different_ from the other UPnP devices, the services.
  b. Right click on the area underneath ‘Root Device’ and choose ‘Add Service from File…’. Select the service file you just created. Now, a few descriptions need to be changed.
  c. Click on the ‘Root Device’ and in the right hand pane, change the ‘Friendly Name’ to ‘Tutorial Device’.

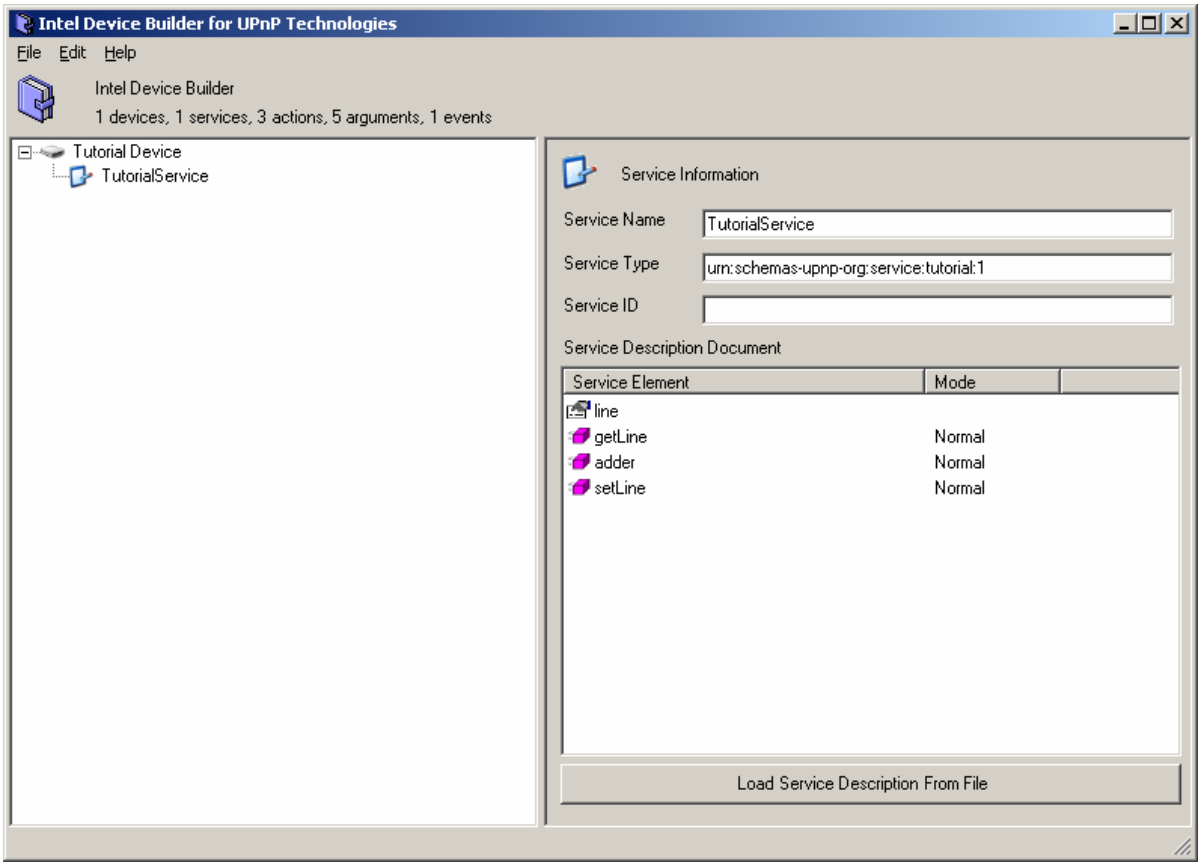


d. Click on the ‘ImportedService’ tag and change the following:

   i. ‘Service Name’ to ‘TutorialService’

   ii. ‘Service Type’ to ‘urn:schemas-upnp-org:service:tutorial:1’

e. The service type is very important because it is the parameter that will be searched for by an UPnP control point. Make sure it matches the above, removing the single quotes of course.



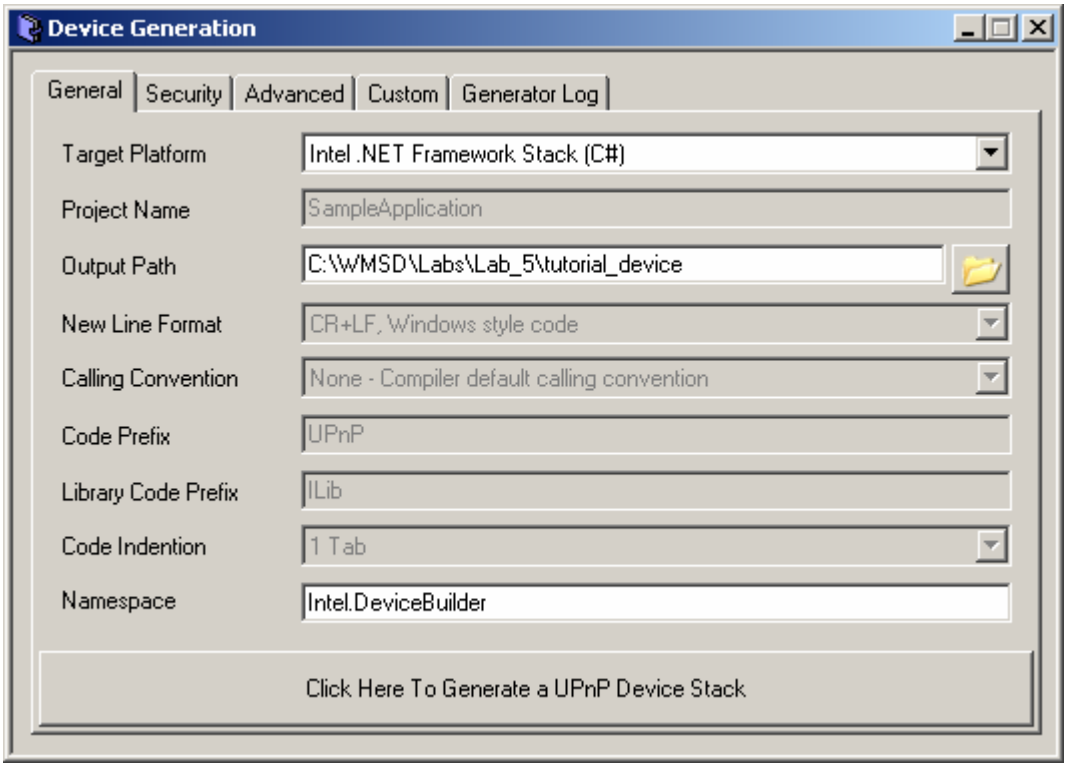
f. Save the file as 'tutorial_device.upnpsg'. This will be used in creating an UPnP control point that would use this service.

g. For the final step, we need to generate the device stack. Under the 'File' menu choose, 'Export Device Stack…'; a real shocker there! In the 'Device Generation' window choose:

   i. 'Target Platform' to be 'Intel .NET Framework Stack (C#)'

   ii. 'Output Path' to be your tutorial_device directory

   The rest can stay the same. Click the big button at the bottom and *poof* your tutorial_device directory should contain a new C# project that is your device. You may close Device Builder, its job is done.



- The Coding: Now that the device stack has been built, the code for the actions needs to be implemented. Open the tutorial_device project (the .csproj file). You will be questioned if you want to convert the project to the new format, click 'Yes'. Device Builder builds projects for the VS .NET 2002 platform and apparently there is some difference between VS.NET 2002 and 2003 projects.
- Modifying SampleDevice.cs: Open 'SampleDevice.cs' and scroll down to the middle of the file to the 'SampleDevice' constructor. There are 3 lines that look something like this:

```
TutorialService.External_adder = new
                    Intel.DeviceBuilder.TutorialService.Delegate_adder(TutorialService_adder);
TutorialService.External_getLine = new
                    Intel.DeviceBuilder.TutorialService.Delegate_getLine(TutorialService_getLine);
TutorialService.External_setLine = new
                    Intel.DeviceBuilder.TutorialService.Delegate_setLine(TutorialService_setLine);
```

Change them to this:

```
TutorialService.External_adder = new
                    Intel.DeviceBuilder.TutorialService.Delegate_adder(TutorialService.adder);
TutorialService.External_getLine = new
                    Intel.DeviceBuilder.TutorialService.Delegate_getLine(TutorialService.getLine);
TutorialService.External_setLine = new
                    Intel.DeviceBuilder.TutorialService.Delegate_setLine(TutorialService.setLine);
```

(The change was in the parameter passed. The '_' was changed to '.'.)

- Reason: The code that is generated is generated in such as way so that you can add your action implementations into the SampleDevice.cs file. If you scroll down you'll see the method calls that match the first 3 lines. The problem is that you can not reference anything inside the TutorialService object from here. This means you do not have access to your state variables. I do not understand why Intel did it this way. If you change it to the 2nd set of lines, you write the action method code in the action method declarations which are _inside_ of the TutorialService class. It would make sense to me that you would want to have the actions, which are part of the service, defined _inside_ the service. If anyone has a better understanding or a reason they did this, please let me or the TAs know. Anyway, the changes will work.

- Open the TutorialService.cs file. Take a look at what is exposed in this file. The actions method definitions are there, but, notice, the state variables are not there. The state variables are defined in a region that is 'hidden' using the #region directive. Scroll to the top and click the '+' on the left beside the line 'Autogenerated code'… Scroll down through this code. Close it back up after you have taken a look at it.

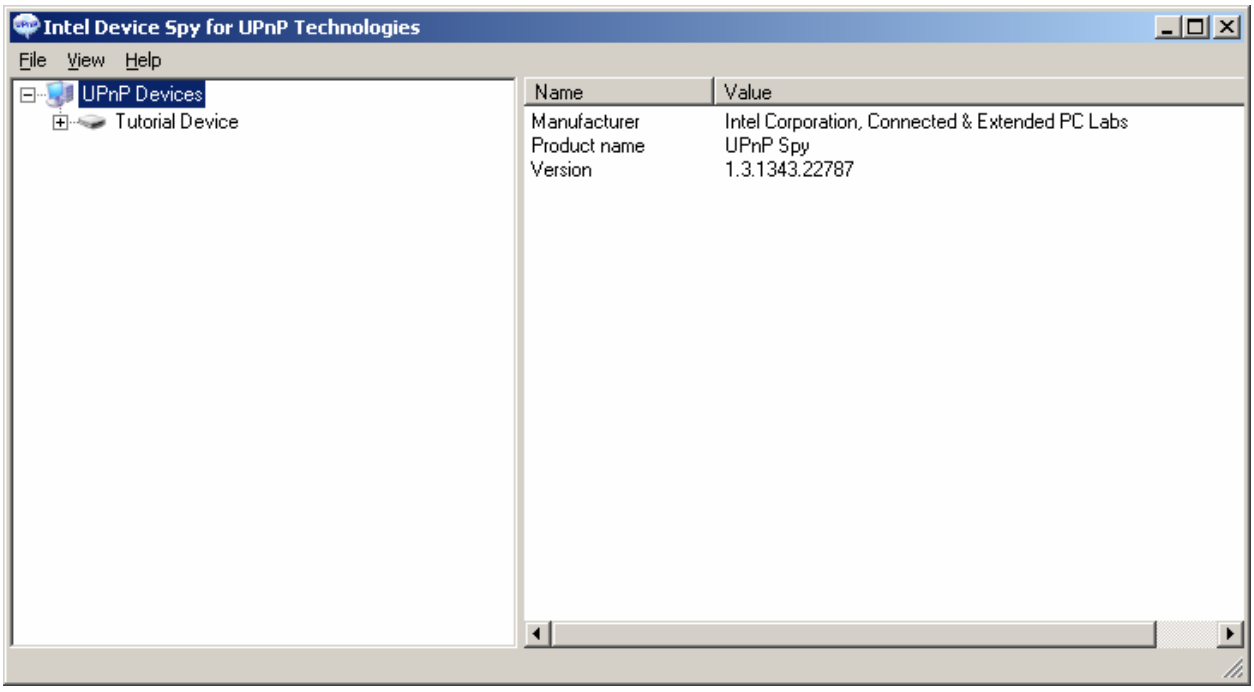
- Add the needed code to the action methods. The 'out' identifier means that variable is passed by reference and any changes to it will change to the object itself, therefore the calling method's value will change as well. Also, comment out the 'throw' line; that exception is no longer needed. Your methods should look like this:

```
public void adder(System.Int32 x, System.Int32 y, out System.Int32 sum)
{
    sum = x + y ;
    //ToDo: Add Your implementation here, and remove exception
}
/// <summary>
/// Action: getLine
/// </summary>
/// <param name="line">Associated State Variable: line</param>

public void getLine(out System.String line)
{
    line = this.Evented_line ;
    //ToDo: Add Your implementation here, and remove exception
}
/// <summary>
/// Action: setLine
/// </summary>
/// <param name="line">Associated State Variable: line</param>

public void setLine(System.String line)
{
    this.Evented_line = line ;
    //ToDo: Add Your implementation here, and remove exception
}
```

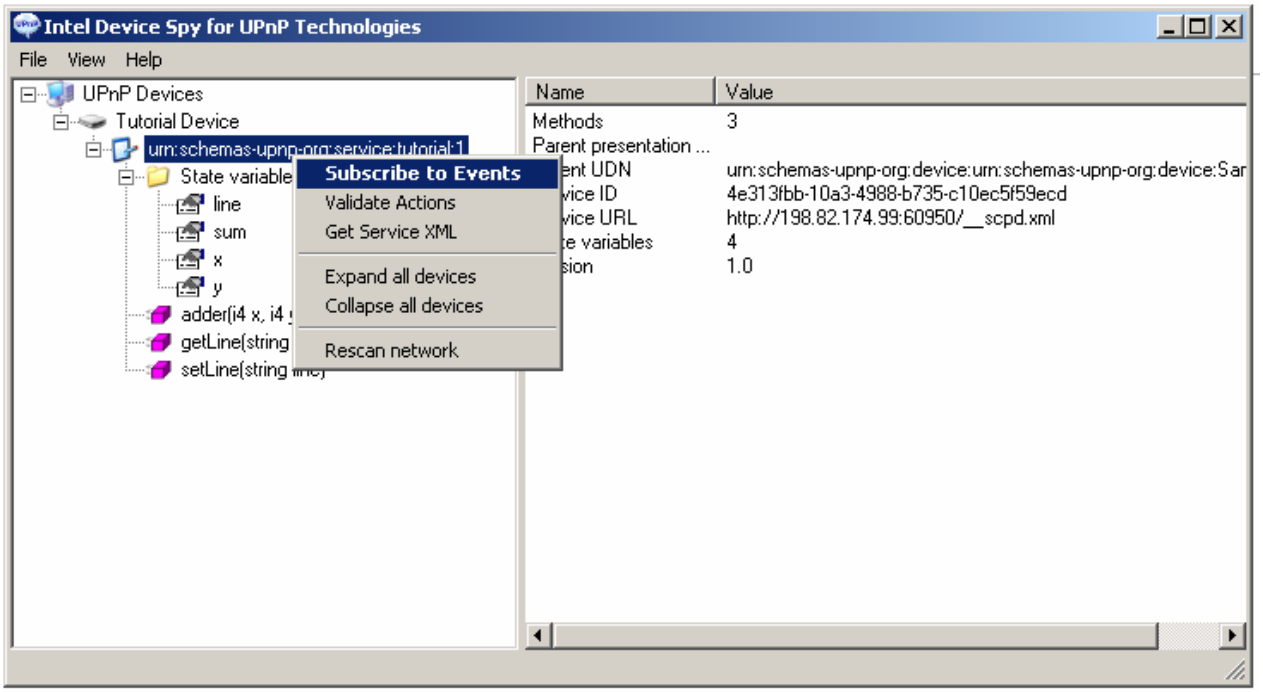
  Why is the incoming string assigned to 'Evented_line' when we defined our state variable to be simply, 'line'. There isn't a simple way to watch a variable and tell when it changes, which is needed since control points can subscribe to the variables event. In order to watch a variable the stack uses the C# 'properties' ability. Properties allow an almost wrapper like functionality. The Evented_line property takes care of the eventing part of UPnP. 'sum', 'x', and 'y' can not be subscribed to, so we do not need to worry about eventing and can address them directly (sort of).

- Build and run the project. Use the 'Device Spy' from earlier to make sure the actions work correctly.

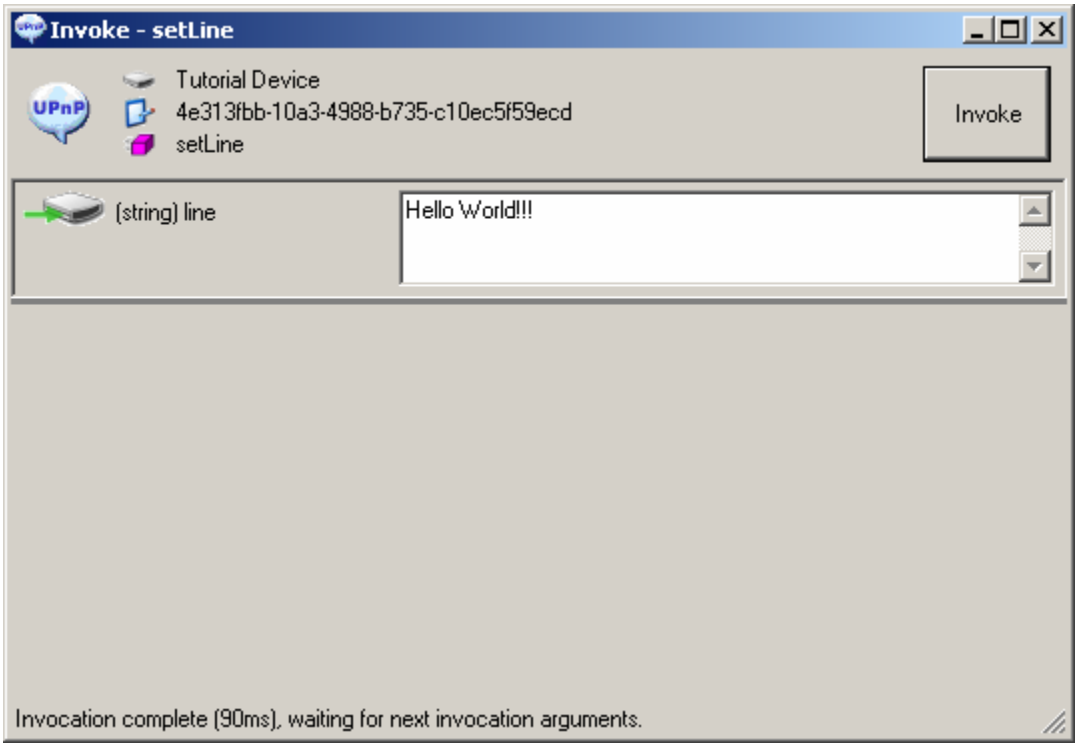
  On the 'Device Spy', you should see your device.

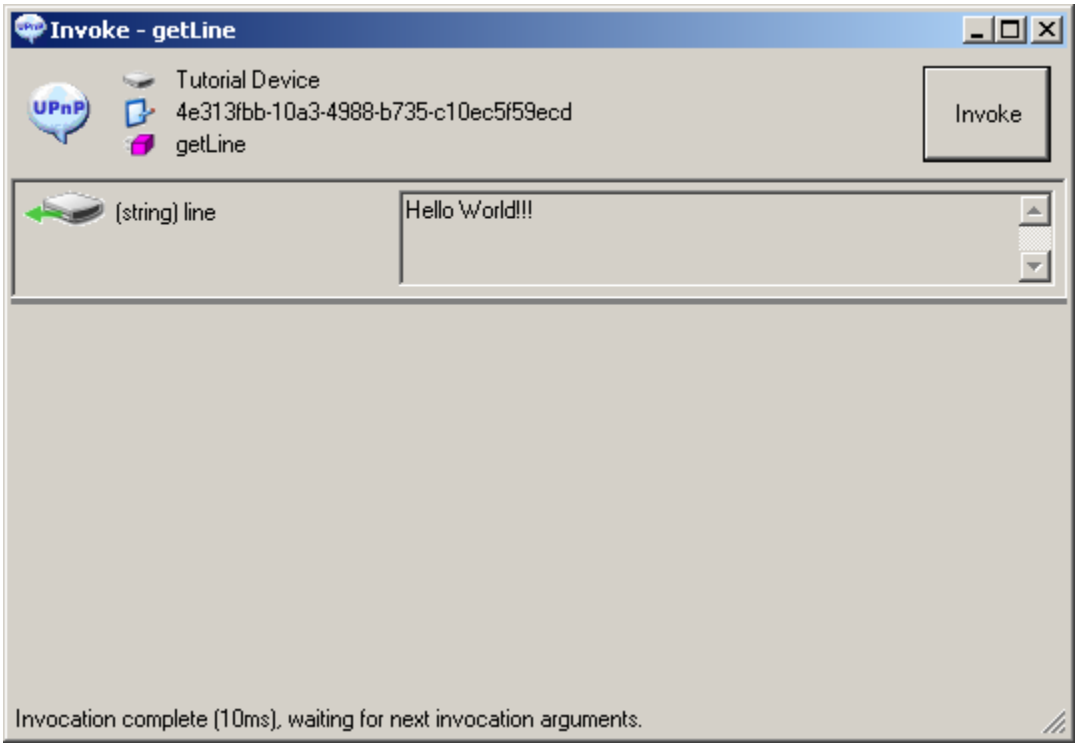


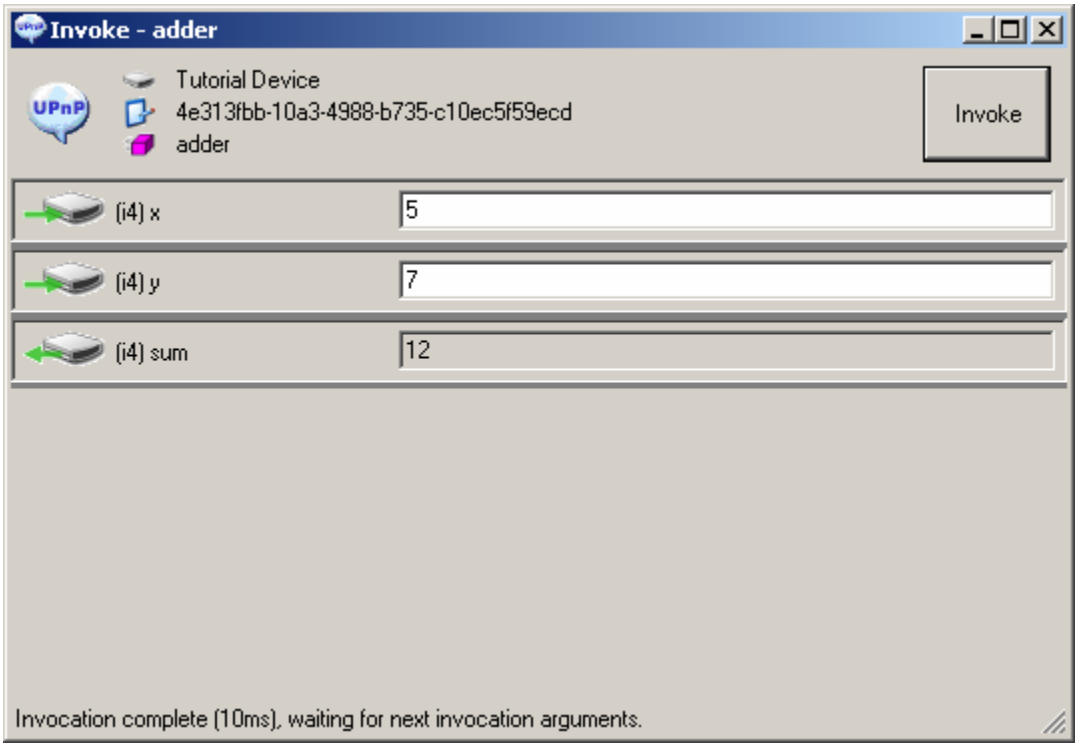
Subscribe to events of your service.



Set the string value using ‘setLine’ method and verify it with ‘getLine’ method, and check if ‘adder’ method works as well.


Invoke - setLine
Tutorial Device
4e313fbb-10a3-4988-b735-c10ec5f59ecd
setLine
Invoke
(string) line
Hello World!!!
Invocation complete (90ms), waiting for next invocation arguments.


Invoke - getLine
Tutorial Device
4e313fbb-10a3-4988-b735-c10ec5f59ecd
getLine
Invoke
(string) line
Hello World!!!
Invocation complete (10ms), waiting for next invocation arguments.


Invoke - adder
Tutorial Device
4e313fbb-10a3-4988-b735-c10ec5f59ecd
adder
Invoke
(i4) x
5
(i4) y
7
(i4) sum
12
Invocation complete (10ms), waiting for next invocation arguments.