HomeOS Programming

2/11/2011

HomeOS is an experimental operating system for the home which focuses on providing centralized control of connected devices in the home, useful programming abstractions for developers, and allows for the easy addition of new devices and application functionality to the home environment.

This document explains the software architecture of HomeOS how to write device drivers and applications. It is not a list of what is available in the software package; that would be whats-included.docx. Because it might be a little out of date, your most up-to-date source of information is the source code itself ☺

Contents

Getting Started

Prerequisites

1. Windows 7 (we haven't tested on Vista or XP but those should work too)
2. Visual Studio 2010
3. .NET Framework v4.0
4. Silverlight v4.0 (for UIs that run in a browser). http://www.silverlight.net/getstarted/

Some UI code also needs the Windows Phone 7 SDK (http://www.silverlight.net/getstarted/devices/windows-phone/) and Silverlight for Windows Phone Toolkit (http://silverlight.codeplex.com/). If you don't need phone-based access, you don't need this SDK. Simply disable the WP7 projects.

Running for the first time

1. Unzip homeos.zip. This would create a subdirectory `homeos`.
2. Open using Visual Studio the .sln file in the `homeos directory`.
    a. Ignore any warnings about source control symbols.
    b. If you don't have WP7 SDK, ignore the warnings about WP7 projects.
3. Build the solution (from the Build Menu on top or shortcut F6)
4. Start a command prompt as an Administrator and go to directory `homeos/output`
5. Run the command `./Platform.exe -c DistConfig`

The command above runs the main homeos platform and two Dummy modules.

If you want to run code from within Visual Studio (which is handy for debugging purposes), start Visual Studio as an Administrator and do two things:

1. Set Platform as the startup project
    a. Right click on Platform in Solution Explorer window
    b. Click on "Set as startup project"
2. Set the right command line arguments
    a. Right click on Platform in Solution Explorer window
    b. Click on Properties
    c. Click on Debug on the left
    d. Enter `-c DistConfig` in the text box for command line arguments
3. Click on "Start debugging" from the Debug Menu on top (shortcut F5)

The default configuration runs two instances of the Dummy module. The source of this module (in Apps/AppDummy/Dummy.cs) is a good reference for understanding how things work. This module exports a port with role "dummy" and two operations "echo" and "echosub." If it finds another port with Role "dummy" registered with the platform, it invokes that port's echo operation periodically. It also subscribes to the other port's echosub operation. It also sends out notifications to other modules that subscribe to its own echosub operation.

The other modules that are part of the package are configured to not run by default. The list of available module is documented in whats-included.docx. If you embark on writing driver for additional devices, give us a holler, and we'll see if we have something in our repository that can help as an example.

The other modules (under Apps and Drivers folders) serve as good examples of writing HomeOS code. For example, one interesting driver is Drivers/DriverAxisCamera (for web camera made by Axis) which takes in the camera IP address and user credentials as starting arguments. It exports a port with operations that correspond to controlling the camera (pan and zoom) and getting the current image. Apps/AppCamera is designed to interact with DriverAxisCamera. It provides a GUI to view the image received from DriverAxisCamera driver as well as control the camera. When it runs, it begins looking for camera ports and once one is found it starts a thread which gets a new image each second and renders it. It's interaction with DriverAxisCamera provides an example of how complex objects such as images can be passed across modules.

Software Architecture

The HomeOS software is structured like a plugin framework. As such, it has two core pieces – the host platform and the plugin modules. The platform is implemented by the (visual studio) project called the **Platform**, and each module (that is, a driver or application) is implemented as its own project.

Isolation between platform and modules and between modules is achieved using two mechanisms. The first is that each modules runs in its own **application domain** (WikiPedia, MSDN), which is a lightweight isolation mechanism provided by the .NET Framework.

The second mechanism is the **System.AddIn framework** (MSDN) which builds on top of application domains. It provides a model for developing plugin frameworks in .NET and means for expressing interfaces across modules as well as independent versioning of modules and platform. These benefits come at the cost of increased programming complexity and restrictions, i.e., programming discipline. We do not delve into the details of the System.AddIn framework (which can be learned using the link above or a Web search), but focus on how HomeOS uses this framework.

Most of the classes are defined in the **Common** project. With the System.AddIn framework, interfaces (i.e., function calls and their signatures) across the isolation boundary must be clearly specified. In HomeOS, these are defined in the **Contracts** project. The level of access available for remote objects (in different application domains) is different from that for local objects. This access is specified in the **Views** project. The **Adapters** project defines the translation for each object type -- how the view of an object should be constructed from its contract as well as how its contract should be constructed from its view. Translations can be arbitrary (e.g., versioning aware) but we have kept them simple for now. The translations of a type includes the translations of the input and output parameters of the methods of the type. If an explicit translation of parameter type is not provided, by default basic types (e.g., integers, strings) are passed by value and complex types are passed by reference using .NET Remoting. We recommend providing explicit translations unless you fully understand .NET Remoting.

As an example, for the class Port in HomeOS:

- Common/Port.cs contains the definition of the class. The full functionality of this class is only available for objects that are instantiated within the same application domain.
- Views/VPort.cs specifies how a Port appears to entities outside the creating application domain. Common/Port inherits Views/VPort.
- Contracts/IPort.cs specifies the contract of this class across the isolation boundary.
- Adapters/APort.cs defines the translations between views and contracts.

A Note about Garbage Collection

Objects that are transmitted across isolation boundaries are automatically garbage collected just like local objects. If a pointer to the object no longer exists in either a remote domain or the creating domain, the object is garbage collected. However, **garbage collection for objects that are transmitted across application domains is slow**. It can take up to a few seconds after the last use for the object to it being garbage collected.

This delay will be problematic only if you need to make very frequent calls across application domains with newly minted complex objects such that, without garbage collection, you run the risk of running out of memory. If you encounter this problem, instead of the programming pattern on the left, use the pattern on the right which updates the object instead of creating a new one each time.

```
int variable = 0;

while (true) {
     Param param = new Param(variable);
     int answer = CallAcrossDomain(param);
     variable++;
}
```

```
int variable = 0;
Param param = new Param(variable);
while (true) {
     param.value = variable;
     int answer = CallAcrossDomain(param);
     variable++;
}
```

The concern above is relevant only for complex types that are passed by reference. Types that are passed by value (e.g., basic types) do not face this issue.

This is likely all you need to know about the software architecture of HomeOS unless you plan to extend it in serious ways rather than only writing drivers and applications. If you intend to do that and cannot figure out a way forward, please contact us and we'll be happy to help.

Programming Abstractions

The programming model for HomeOS is **service-oriented**: all functionality provided by drivers and applications is provided via **Ports** which export one or more services in the form of **Roles.** Each Role has a list of operations which can be invoked by applications. Role for a dimmer switch might have an operation called "setdimmer" which takes in an integer between 0 and 99 that represents the desired value for the dimmer.

Operations can also return values, so the same lightswitch may have an operation called “getdimmer” which returned an integer that corresponds to the current dimmer value. Further, some operations can be subscribed to allowing for later notifications concerning the operation. For instance, subscribing to the “getdimmer” operation might provide a callback whenever the dimmer’s value changed.

Architecturally, HomeOS makes little distinction between drivers and applications. Both are referred to as **Modules**. Usually, driver modules tend to communicate directly with devices and offer their services to other modules. Application modules tend to use the services of drivers. But a given module can both export its own services and use those of others. As mentioned above, HomeOS isolates modules from each other using application domains and the System.AddIn framework.

Input and output parameters of operations are of type `ParamType`. We define a special class so we can have one translator (contract/views/adapters) for operation parameters rather than defining one per possible type. `ParamType` currently has provisions for exchanging basic types such as integers and strings as well complex types such as ranges. The class has the following members:

- Maintype: denotes the main type of the object being represented using ParamType. This can be one of integer, range, image, sound, text, etc.
- Subtype: a string that provides more detail about the object being represented and it is relative to the main type. E.g., for the maintype of range, a subtype of “0 99” specifies the end points of the range.
- Value: captures of the actual object.
- Name (optional): a string that captures a friendly name for this parameter (e.g., “dimmer”).

Complex types can be passed using this framework. For instance, we pass images as (maintype=image; subtype=”bitmap”; value=byte[]). You may need to extend this class if you need to pass something that we currently do not have provisions for.

Writing Applications

Generally, writing an application is done in 2 steps:

1. **Discovering Interesting Ports:** There are two ways to discover ports in HomeOS. The first is using the `GetAllPortsFromPlatform()` function which will return a list of all currently active registered ports. The second is the `PortRegistered()` function which modules must override and is called every time a new port is registered in HomeOS. To establish whether you are interested in a given port, each port describes its functionality in terms of *Roles* which can be enumerated using `port.GetInfo().GetRoles()`. Roles are uniquely identified by their names, and each role has a list of operations that it supports. Operations are characterized by their name, the list of arguments that they receive, the list of return values, and whether they can be subscribed. The list of arguments and return values must belong to `ParamType` class.
2. **Building Application Logic:** Usually this is the simplest part of writing an application and just involves appropriately coordinating calls to the various relevant Operations. In particular, there are two primary ways to call an `Operation`: Invocation and Subscription.

a. **Invocation:** This is done using the `Invoke()` function of the port and passing into the name of the role, name of operation, the input parameters, and a capability showing permission to call the operation.
b. **Subscription:** This is similar to Invocation, but uses the `Subscribe()` function rather than returning once immediately. The values will be returned later via the `AsynReturn()` function that subscribing modules must implement. The semantics of when these notifications occur is left up to the driver, but typically it is fired periodically or whenever the return values would have changed.

Example Application Bits

To find interesting ports, when the application starts, it can do the following.

```
1    IList<View.VPort> allPortsList = GetAllPortsFromPlatform();
2    foreach (View.VPort port in allPortsList)
3          PortRegistered(port);
```

PortRegistered() is also called when new ports are registered with the platform. An implementation of it for an application that was looking for all switches in the home could be:

```
1    public override void PortRegistered(View.VPort port)
2    {
3          if (Role.ContainsRole(port, "roleswitch"))
4                switchPorts.Add(port);
5    }
```

Role.Contains() is a helper utility that iterates over all roles offered by `port` and checks if any of them match `roleswitch`.

We present below three examples of operation calls on ports. The first calls an operation with no return values in order to turn a light on by setting its dimmer value to 99.

```
1.   IList<View.VParamType> args = new List<View.VParamType>();
2.   args.Add(new ParamType(ParamType.SimpleType.range, "0 99", 99, “level”));
3.   switchPort.Invoke(“roleswitch”, “setdimmer”, args,
                       ControlPort, switchPortCapability, ControlPortCapability);
```

This example assumes that switchPort exports a role named `roleswitch with` an operation called `setdimmer` that takes one parameter.

The second example instead calls an operation with no parameters to discover what media a player is currently playing and how far into the media it is. Here rather than setting the parameters, we must parse the return values.

```
1.   IList<View.VParamType> retVals = DmrPort.Invoke(“roledmr”, “getstatus”, new List<View.VParamType>(),
                                                    ControlPort, DmrPortCapability, null);
2.   if (retVals != null && retVals.Count == 2)
3.   {
4.     String uri = (string)retVals[0].Value();
5.     String time = (string)retVals[1].Value();
6.   }
```

The third example shows a subscription and the notification handler `AsyncReturn()`.

```
1.   switchPort.Subscribe(“roleswitch”, “getdimmer”,
                   this.ControlPort, this.switchPortCapability, this.ControlPortCapability);
```

```
1.  public override void AsyncReturn(string roleName, string opName, IList<View.VParamType> retVals,
                                     View.VPort senderPort)
2.   {
3.       if (roleName.Equals("roleswitch") && opName.Equals("getdimmer"))
4.       {
5.          byte newDimmerValue = (byte)retVals()[0].Value();
6.        }
7.   }
```

Writing Drivers

Generally, writing a driver is done in 5 steps:

1. **Instantiating Roles:** A role can be instantiated using `role = new Role("lightswitch")`. Operations are instantiated by calling a constructor with `operation = new Operations(name, paramTypes, retvalTypes, canSub)`, where `paramTypes` and `retvalTypes` are, respectively, lists of the types of parameters and return values, and `canSub` denotes whether the operation can be subscribed to. Operations must be added to roles by calling `role.AddOperation(operation)`. The task of instantiating a role can be embedded in a class that corresponds to the role. See the various class defined in Role.cs.
2. **Instantiating Ports:** A port in instantiated for each service that the driver wants to offer. The driver should first call `portInfo = GetPortInfoFromPlatform(name)`, where `name` is unique across all ports of the module (not across the entire system), and then call `InitPort(portInfo)`.
3. **Binding Ports to Roles:** This is done by calling `BindRoles(port,roleList, OnOperationInvoke)`, where `OnOperationInvoke` is a function of type `public IList<View.VParamType> OnOperationInvoke(string roleName, String opName, IList<View.VParamType> args)`. This function is called when any of the operations are invoked by an application. The names of the role and operation as well as the argument are passed to this function. A separate handler for each operation can also be defined. See the code for the implementation of BindRoles() for details on how this can be done.
4. **Registering the Ports:** Registration tells HomeOS that this port is now open for business. It is accomplished using `RegisterPortWithPlatform(port)`.
5. **Implementing functions for handling operation invocations:** Here the custom logic of handing operation invocation resides.

Example Driver Bits

The examples should clarify how drivers respond to operation invocations. Assume that we are writing a driver with a port that exports a role "dummryrole" with two Operations, "echo" and "echosub". To initialize the module, we can do the following:

```
1.  // ..... initialize the list of roles we are going to export
2.  List<View.VRole> listRole = new List<View.VRole>();
3.  RoleDummy dummyRole = new RoleDummy();
4.  listRole.Add(dummyRole);
5.
6.  //.................instantiate the port
7.  View.VPortInfo portInfo = GetPortInfoFromPlatform("port");
8.  dummyPort = InitPort(portInfo);
9.
10. //..... bind the port to roles and delegates
11. BindRoles(dummyPort, listRole, OnOperationInvoke);
12.
```

```
13.  //.................register the port after the binding is complete
14.  RegisterPortWithPlatform(dummyPort);
```

The constructor for class RoleDummy instantiates the role as:

```
1.  // …. The first operation
2.  List<View.VParamType> args = new List<View.VParamType>();
3.  args.Add(new ParamType(ParamType.SimpleType.integer, "", 0));

4.  List<View.VParamType> retVals = new List<View.VParamType>();
5.  retVals.Add(new ParamType(ParamType.SimpleType.integer, "", 0));

6.  AddOperation(new Operation(OpEcho, args, retVals));
7.
8.  // …. The second operation
9.  List<View.VParamType> retVals = new List<View.VParamType>();
10. retVals.Add(new ParamType(ParamType.SimpleType.integer, "", 0));
11.
12. AddOperation(new Operation(OpEchoSub, null, retVals, true));
```

This operation handler could be implemented as follows:

```
1.  public List<View.VParamType> OnOperationInvoke(String roleName, String opName, IList<View.VParamType> args)
2.  {
3.          If (!roleName.Equals("dummyrole"))
4.                Throw new exception("Invalid role");
5.
6.          switch (opName.ToLower())
7.            {
8.                //OpEcho: receives an int and returns the value after multiplying by -1
9.               case RoleDummy.OpEcho:
10.                  int arg0 = (int) args[0].Value();
11.                  List<View.VParamType> retVals = new List<View.VParamType>();
12.                  retVals.Add(new ParamType((ParamType.SimpleType.integer, "", -1 * arg0));

13.                   return retVals;
14.
15.              //OpEchoSub: does nothing at all
16.              case OpEchoSub:
17.                   return new List <View. VParamType >();

18.               default:
19.                    Throw new exception("Invalid operation: " + opName);
20.           }
21. }
```

Let's further assume that OpEchoSub is a subscribable function that returns an internal `counter`. When the driver wants to notify its subscribers, it can do so by doing something like the following

```
1.  IList<View.VParamType> retVals = new List<View.VParamType>();
2.  retVals.Add(new ParamType(ParamType.SimpleType.integer, "", counter));
3.  dummyPort.Notify(RoleDummy.RoleName, RoleDummy.OpEchoSubName, retVals);
```

Other Utilities

This section describes some other utilities that module writers might find useful.

Logger

The recommended way to generate log messages in HomeOS is using the Logger class, which can redirect messages to either the stdout or a file. When a module is instantiated, HomeOS passes to it a pointer to its log object (see ModuleBase.cs). Modules can either use this log object for their own messages or instantiate their own. With the first option, messages from the module will appear at the

same place as those of the platform. This option is the default and modules can start logging by simply calling logger.Log() (the logger object resides in ModuleBase from which all modules inherit).

SafeThread

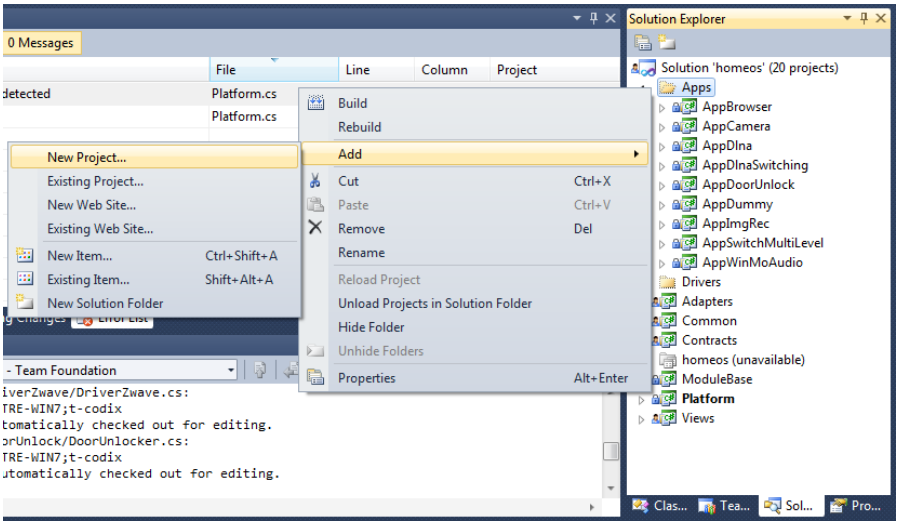
The description above focuses mostly on the communication between modules. In HomeOS modules are isolated from each other such that if one of them crashes, it doesn't impact others or the platform itself. Care has been taken in our design to retain this property but one exception where it fails is if a thread spawned within a module throws an exception that is not caught.

We thus recommend that new threads be spawned using the SafeThread class in the package. This class encapsulates the thread's activities in a try catch block. It can be spawned in a manned similar to c# threads:

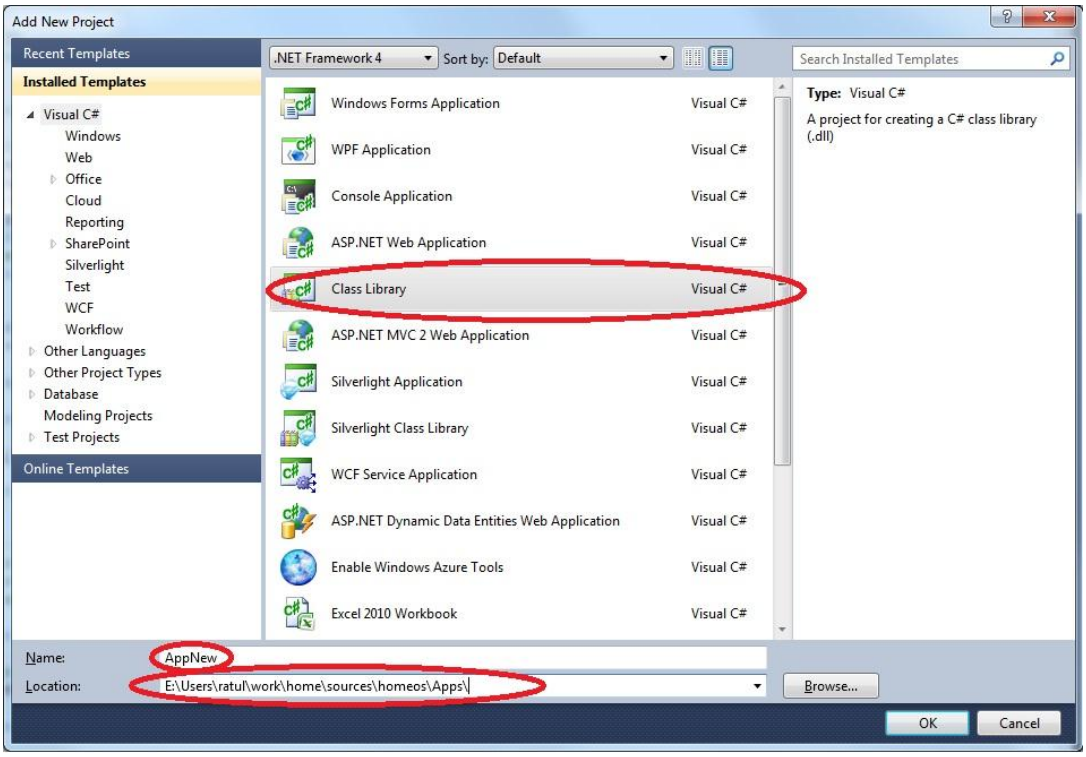
```
1.  SafeThread safeThread = new SafeThread(delegate() {Console.WriteLine("thread spawned");},
2.                                         "SafeThreadExample",
3.                                         logger);
4.  safeThread.Start();
```

Creating a new HomeOS Module in Visual Studio 2010

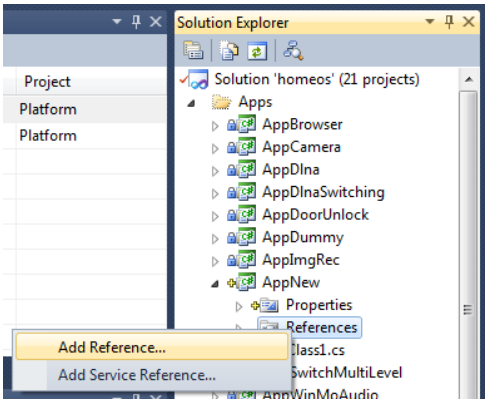
1. Through solution explorer, right click on the Apps folder and go to Add -> New Project



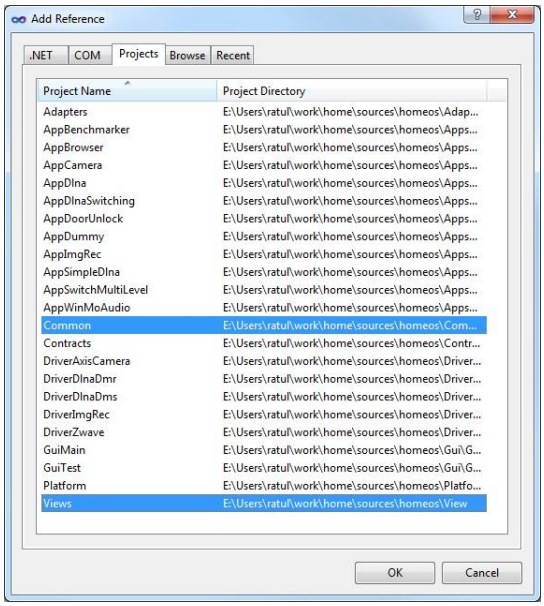
2. In the Add New Project dialog box,
    a. The project template should be Visual C# class library
    b. Change the Name of the project to what you want. We'll use AppNew as an example.
    c. Change the Location to the Apps or Drivers subdirectory under homeos



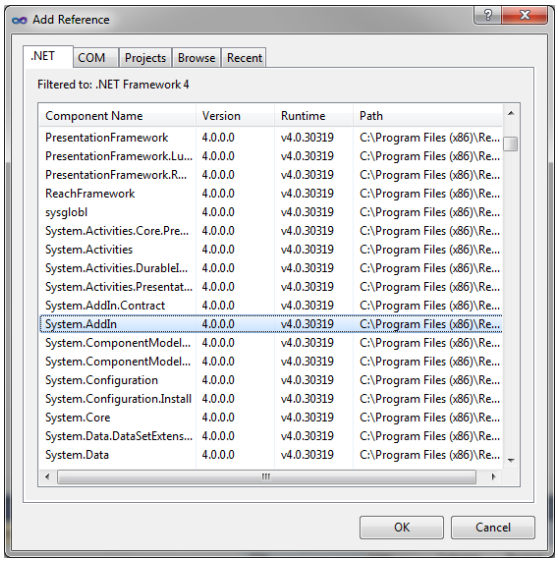
3. Expand the new project and right-click on the references to add the following new references.



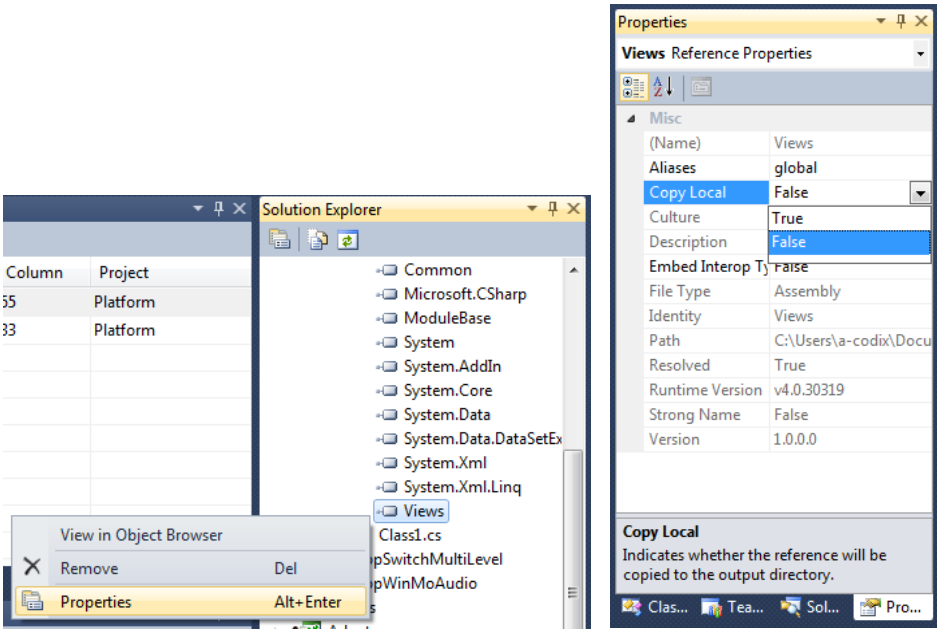
a. Project references: Views, Common



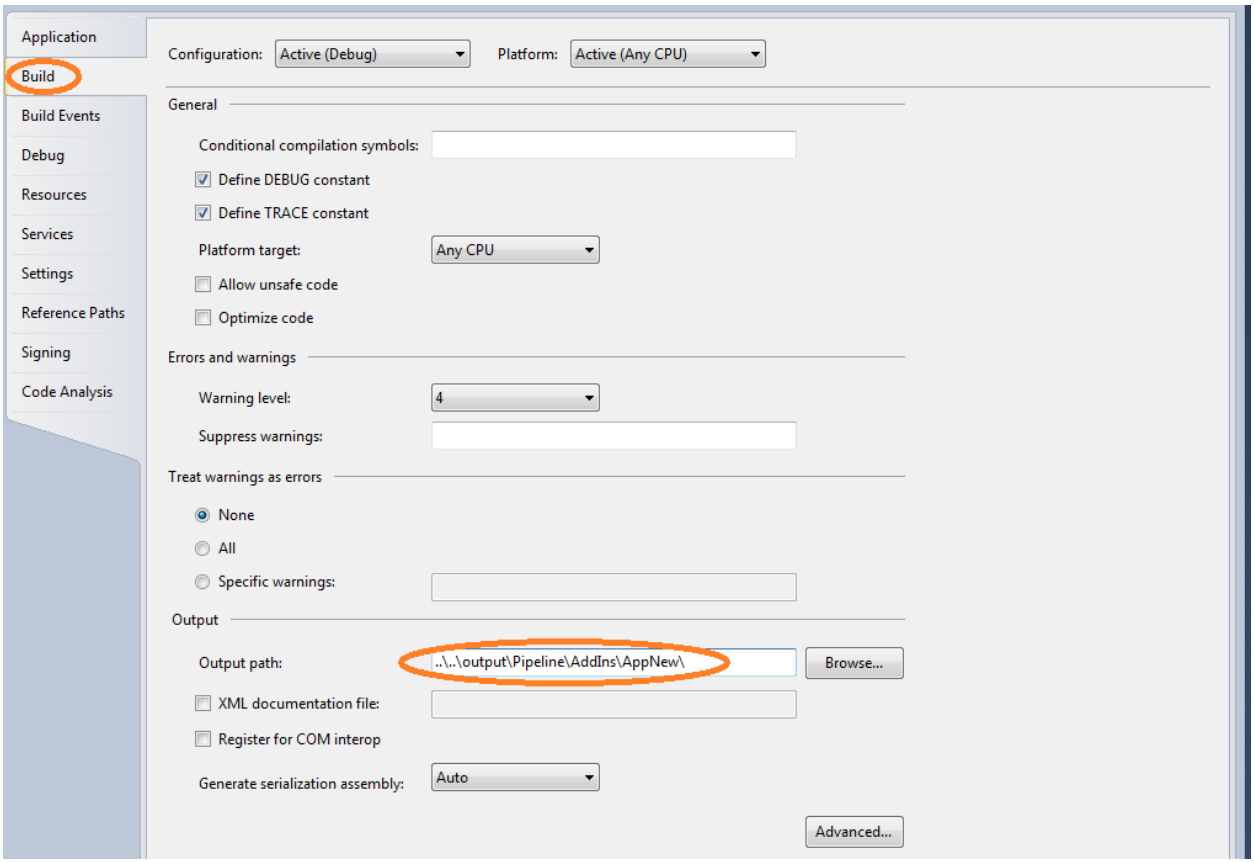
b. .NET references: System.AddIn



4. Expand the references item and for the reference Views, right click on it, go to properties and change the property Copy Local to False.



5. In the project Properties (obtained by right clicking on the project AppNew), under Build, change the Output path to "..\..\output\Pipeline\AddIns\AppNew\" (replace AppNew accordingly)



6. You probably want to right click and rename Class1.cs in your project to something more useful, here we rename it to AppNew.cs same as the project and say yes when it asks if you also want to rename the class contained in the file.

7. Your main class should:

   a. Inherit Common.ModuleBase

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

namespace AppNew
{
    public class AppNew : Common.ModuleBase
    {
    }
}
```

   b. Have the attribute [System.AddIn.Addin("AppNew")] (replace AppNew accordingly)

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

namespace AppNew
{
    [System.AddIn.AddIn("AppNew")]
    public class AppNew : Common.ModuleBase
    {
    }
}
```

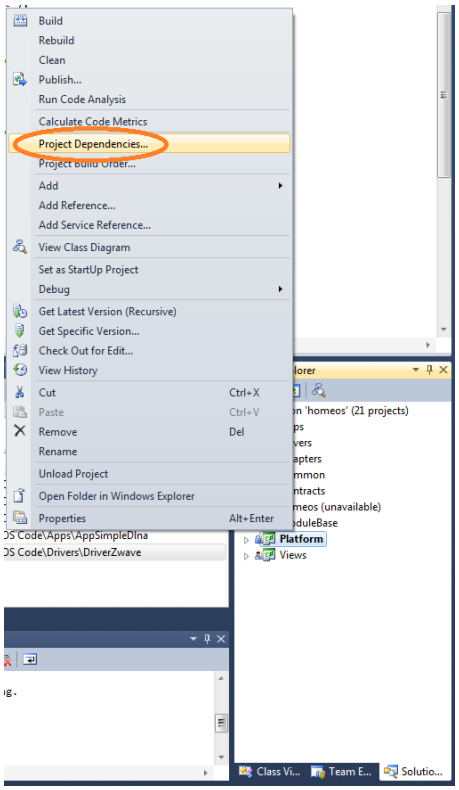
   c. Implement the functions marked abstract in ModuleBase

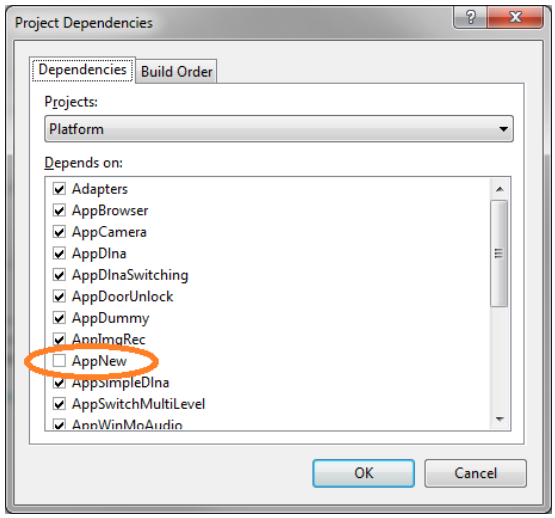
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AppNew
{
    [System.AddIn.AddIn("AppNew")]
    public class AppNew : Common.ModuleBase
    {
        public override void Start()
        {
            //initialize your module here

            // make sure that the control never falls out of this function.
            // o/w, your module will be unloaded then
            // so, if your module doesn't do anything active but reacts to events from other places
            // use "System.Threading.Thread.Sleep(System.Threading.Timeout.Infinite)"  as the last
            // line of this function
        }

        public override void Stop()
        {
            // take any cleanup actions here
        }

        public override void PortRegistered(View.VPort port)
        {
            //called when a port is registered with the platform
        }

        public override void PortDeregistered(View.VPort port)
        {
            //called when a port is un-registered with the platform
        }
    }
}
```

8. Lastly, there are few things that may make your life easier once you start building and running your application.

    a. You probably want to add a "using Common;" line at the top of your application so you can make use of the most common classes in HomeOS without needing to add a `Common.` in front of them every time.

    b. Also, you will likely want to add a project dependency from the Platform project to your application so that it is automatically compiled before Platform is run each time. Right click on the Platform project and go to Project Dependencies… and then check your application.





9. At this point, your code should compile even if it doesn't do anything interesting (yet).

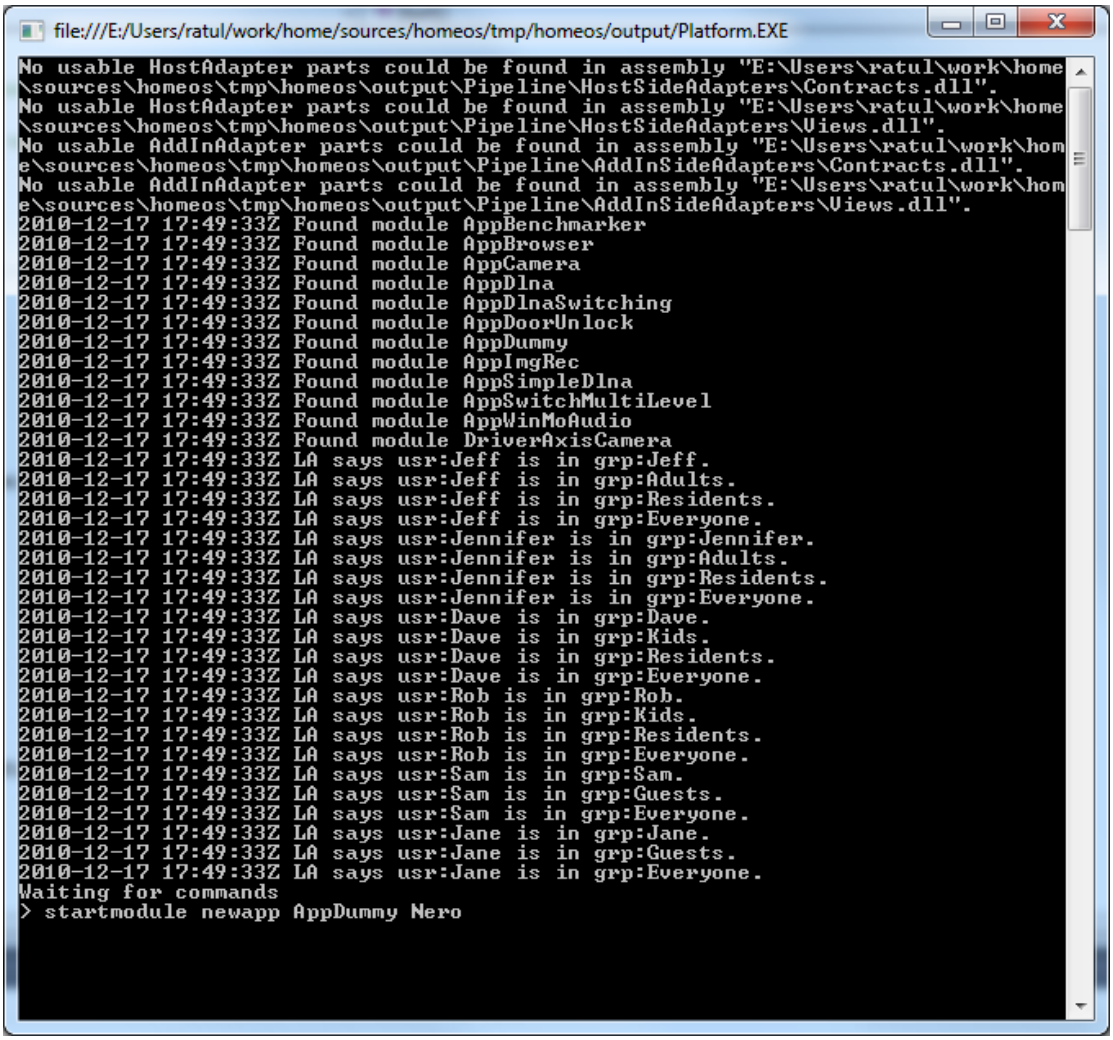
Running Your HomeOS Modules

There are three ways to run your module now that you've created it.

The first is to just go to Debug -> Start Debugging which will launch the platform and after some debug output, you will be given a command line. One of the available commands lets you start new module, you could launch your application by typing "startmodule FriendlyName AppNew AppArgs". The general command is:

```
startmodule <friendly-name> <app-name> <app-args>
```

where an argument is required, but it need not be meaningful. Your application can access the argument string via `this.moduleInfo().ArgStr()`. FriendlyName should be unique among all running modules.



The second way is to modify Platform.cs to run your module at launch. To do this, open Platform.cs in project Platform and navigate to the `Start()` function. Call the StartModule() function with appropriate parameters to instruct the platform load and run your module.

```
1.   public void Start()
2.   {
3.       AppDomain.CurrentDomain.UnhandledException += new UnhandledExceptionEventHandler(HandleUncaughtExceptions);
4.
5.       InitAutoStartModules();
6.
7.       StartModule(new ModuleInfo("friendly name is newnewapp", "app name is AppNew",
8.                                  "AppNew", null, false, “no arguments here”));
9.
10.        if (startWithGui)
11.            StartGui();
```

```
12.
13.         ReadCommandsFromConsole();
14.  }
```

The third way is to modify the configuration files. The relevant file for modules is Config/DistConfig/Modules.xml. It currently looks like:

```
<Modules>
  <Module FriendlyName="DummyHero" AppName="AppDummy by dumbo" BinaryName="AppDummy" AutoStart="1">
    <Args Count="1" val1="Hero"/>
    <RoleList>
      <Role Name="dummy"/>
    </RoleList>
  </Module>

  <Module FriendlyName="DummyZero" AppName="AppDummy by dumbo" BinaryName="AppDummy" AutoStart="1">
    <Args Count="1" val1="Zero"/>
    <RoleList>
      <Role Name="dummy"/>
    </RoleList>
  </Module>
</Modules>
```

You can add your new modules entries here. If the AutoStart flag is set to 1, your module will start when the Platform launches, through `InitAutoStartModules()` in Platform.cs.

If you modify these .xml files from within VS, be sure to first compile (F6) and then run (F5). It is the first step that copies the modified files to the output directory. Skipping it will lead to running the old version of the files.

If Your Module Doesn't Run

A common error is that the build output directory for your project is not the same directory that HomeOS looks for modules in because the project's location wasn't properly specified in step 2. Most of the time, this can be fixed by repeating step 5, but using "..\output\Pipeline\AddIns\AppNew\" instead of "..\..\output\Pipeline\AddIns\AppNew\" for the build output directory.

If you see token "appnew" not found error despite the binaries being in the right place, you probably forgot Step 4 (not copying View.dll)

Remote Application UIs

Local application UIs (that is, running on the same machine as HomeOS) can be created easily using WPF (Windows Presentation Framework). See examples in several application projects that (e.g., `AppCamera\CameraWindow.xaml`). However, local UIs are insufficient if you need to interact with your application remotely (e.g., from a smartphone or from a browser on another machine).

Our current model for programming remote application UIs is for applications for expose a WCF (Windows Communication Foundation) service endpoint. Then UIs that run in a browser or on a phone are written to consume these WCF services. Instructions below assume that you already know WCF and Silverlight and WP7 programming if you are going to write phone-based UIs. If you don't, we recommend that you get to know those first.

Here is what you need to (assuming Silverlight and Windows Phone SDKs are installed):

1. Export a self-hosted WCF service from the main application logic. The endpoint of the service should be Globals.InfoServiceAddress/<app-friendly-name>. Globals.InfoServiceAddress is currently defined as http://localhost:51430/. <app-friendly-name> is the friendly name given to this application when starting it.

2. For the Silverlight-based application:
    a. Start a new project of type Silverlight application.
    b. Call it AppNewSlGui.
    c. Change the output directory of the project to be the same as where the main application binaries go. (Step 5 above.)
    d. Write the Silverlight application against the WCF service exported by the main application logic. Since we are self-hosting, you'll have to generate service stubs manually using SlSvcUtil.exe (normally at `c:\Program Files\Microsoft SDKs\Silverlight\v4.0\Tools\SlSvcUtil.exe`).
    e. To access this UI, point your browser first at http://localhost:51430/uihtml?<friendly-name> to see if you can access it on the same machine. Then, try accessing from a remote machine by replacing the localhost with the IP address or name of your machine. (Make sure that the firewall is configured to allow connections to this port.) Essentially, the UI you wrote is dynamically downloaded to the browser. The logic for this is contained in InfoService.cs in the project Platform.

3. For the WP7 application
    a. Start a new project of type Windows Phone application.
    b. There are no requirements on what you call this application or where the binaries go, but it helps to be consistent. So name the project AppNewWp7 and co-locate the binaries with other AppNew binaries.
    c. Write this application against the WCF service exported by the main application logic. Use the WP version of SlSvcUtil.exe (normally at `c:\Program Files\Microsoft SDKs\Windows Phone\v7.0\Tools\SlSvcUtil.exe`) to generate the service stub.
    d. The URL that your WP7 application should use is http://<machine-name-or-ip>:51430/<app-friendly-name>

AppCamera and AppDoorNotifier offer examples of how this is done. The latter is the simpler of the two cases. The former actually exports two services, one for normal clients and another for clients that are capable of receiving callback.

Programming Reference

Classes

Most of the core classes in HomeOS have two versions: one is the version which a module uses to access its own resources locally and the other is the version which modules use to access resources received

from another module. For example a Capability is passed to other modules as a View.VCapability and when a module is browsing another module's offered Operations it is looking through View.VOperations in the View.VPortInfo of a View.VPort.

The classes you are likely to run into are:

- **Module:** The basic unit of isolation in HomeOS. Modules are Applications and Drivers, two modules can only communicate or share data via Opeartions on Ports.
- **ModuleInfo:** The static configuration information about a module including its name, working directory, the path to its binary and the arguments it was started with.
- **Port:** Communication endpoints. Ports describe their functionality at a high level with Roles and at a technical level with Operations.
- **PortInfo:** The static configuration information about a Port including its Roles, Location, Operations and name.
- **Role:** A high-level, human-readable description of some functionality provided by a port, such as lightswitch, display, or camera.
- **Capability:** An object granting permission to ports. They are granted by the Platform by providing appropriate credentials and have explicit expiration times. All Operation invocations and subscriptions require a capability.
- **User:** Represents a username and password that can be presented to the Platform to request a capability.
- **Location:** An optional property of ports which describes where whatever functionality the port is offering is.
- **Operation:** The object representing either the description of an operation, a call to that operation or a return from that operation. Parameters and return values are of type ParamType.
- **ParamType:** This represents type which can be passed to an invocation and back. It consists of a simple type, a subtype, an optional name and an optional value. While the value is an Object, only certain objects can be cleanly passed through the isolation boundary without serialization and deserialization: all primitives, strings, and all of the classes mentioned above.

Functions You Must Implement in a Module

All modules are required to override 5 functions from ModuleBase before they will compile and interact with HomeOS correctly:

- **Start:** The function that will be called when the module is run. This is the main thread of execution and you should not let the thread terminate or the module may be garbage collected.
- **Stop:** The Platform will call this function if the module needs to be stopped for any reason. It is the module's only chance to clean up an state it needs to.
- **PortRegistered:** Called each time a new port is registered with the Platform. Useful for listening to find ports you might be interested in.

- **PortDeregistered:** Called each time a port is deregistered for any reason. Useful for tracking when a port you were using might be offline.
- **AsyncReturn (optional):** Called whenever an Operation you are subscribed to pushes out a notification.

ModuleBase Class

The ModuleBase class provides a series of useful functions for interaction with the Platform and carrying out tasks. They are briefly described here:

- **GetPortInfoFromPlatform:** This issues a new PortInfo object to the module that can then be used to instantiate a port.
- **InitPort:** This takes a PortInfo object, creates a port and a capability for access that port and returns the port back. It does not register the port with HomeOS.
- **RegisterPortWithPlatform:** This tells HomeOS that the port exists and is ready to be exposed to other modules.
- **GetAllPortsFromPlatform:** Returns a list of all currently active ports in HomeOS.
- **GetCapbility:** Used to request a capability to call Operations on a given port.
- **Finished:** Informs HomeOS that the module has finished its task and can now be cleaned up.
- **IsMyPort:** Returns true if the given port belongs to this module and false otherwise.
- **BindRoles:** Binds the port to the list of roles

Also it provides two properties which are useful mostly for subscribing to Operations

- **ControlPort:** the default port created for this module, it offers no operations but can receive callbacks from notifications
- **ControlPortCapbility:** a capability which can be handed to other modules so that they can respond with notification callbacks

Glossary

Role: a string describing a high-level function that a port provides

Operation: a named action which a port exposes taking a certain number of typed inputs and returning a certain number of typed outputs

Capability: an object that grant permission to access a certain port

Location: each port has a location field in its PortInfo data structure, this enables applications to make decisions based on where devices are

Module: the unit of code in HomeOS applications and drivers are both modules

Port: a communication endpoint, ports offer operations and contain information about their location and other details via their PortInfo data structure obtained using `GetInfo()`

Platform: the kernel of HomeOS responsible for allowing communication and managing running modules