Lab of Things Developer Guide

Contents

Introduction

Lab of Things is a shared infrastructure designed to help researchers develop and evaluate technologies in the home environment. Lab of Things uses the HomeOS software system which provides a PC-like abstraction for in-home hardware and simplifies the tasks of writing applications and managing sensors.

This document explains the software architecture of HomeOS, and covers useful software abstractions to help you write your own custom device drivers and applications.

Additional Documentation

Be sure to visit the download page to get other developer documentation for this release.

Glossary

Capability: An object that grants access permission for a certain port**.**

Location: The actual physical location for a device (e.g. "kitchen", "basement", etc.). This enables applications to make decisions based on location. Each port has a *location* field in its PortInfo data structure.

Module: A HomeOS app or driver.

Operation: A named action exposed by a port. Each operation takes a certain number of typed inputs and returns a certain number of typed outputs.

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

Port: A communication endpoint. Ports announce the availability of a device or service, and describe which services are available. Use `GetInfo()` to return information about a port.

Role: A service definition that describes the high-level functionality provided by a port, similar in nature to an interface. Roles are defined in `Hub\Common\Role.cs`.

Scout: A small program (.dll) that runs and detects devices on the home system, and tells the platform which driver to start when a user wants to set up a new device, and which parameters the driver should be run with.

Configure a Development Environment

Prerequisites

In order to build the HomeOS solution, you need to install the following software:

- Microsoft Visual Studio 2012 Professional (Visual Studio Express is NOT supported). When launching for the first time, be sure to select the Web Developer Tools option. For more information, see Visual Studio Settings).
- .NET Framework 4.5
- Microsoft Sync Framework 2.1 SDK.
- Windows Azure SDK (Note that this can also be installed by using the Web Platform Installer).
- Silicon Labs USB to UART driver (needed for Z-Wave dongles).

Build the HomeOS Solution

To work with the source code on your development computer, take these steps:

1. Unzip the solution files to your local computer.
2. Launch Visual Studio 2012.
3. Click File, Open, Project/Solution.
4. Go to the Hub subfolder of the HomeOS solution.
5. Open Hub.sln. This solution contains all of the platform code for drivers, apps, scouts, watchdogs, and the web dashboard. For more information see the Developer Guide.
6. Build the solution. All of the files you will need to run the platform are located in the `Hub\output\binaries\Platform` directory.

Run the Hub (batch file)

1. Open an elevated command prompt.
2. Navigate to `\Hub\output\`.
3. Run `startplatform.bat`. The console will display messages for the various services, as well as errors (if there are any). When the platform is ready, you will see the message `Waiting for commands`.
4. To get help for commands, type `help`.
5. To stop the platform, type `exit` at the prompt.

Resetting your Configuration

If you'd like to reset to the default configuration (due to strange behavior, bugs, etc.) you can run reset.bat.

1. Open an elevated command prompt.
2. Navigate to `\Hub\output\`.
3. Run `reset.bat [config that was used]`.

Open the Dashboard

- In the browser of your choice, navigate to: http://localhost:51430/guiweb/index.html.

Open an Elevated Command Prompt

1. Open Start by swiping in from the right edge of the screen (or if you're using a mouse, pointing to the upper-right corner of the screen and moving the mouse pointer down), and then tapping or clicking **Start**.
2. Type cmd, then right-click or press and hold on **Command Prompt**.
3. At the bottom of the screen, click **Run as administrator**.

Or

1. If you're using a keyboard with Windows 8, you can open an elevated command prompt from the Power User Menu. Just press Windows + X and then click on **Command Prompt (Admin)**. Click **Yes** in the User Account Control message that appears.

Software Architecture

In this section we'll discuss how roles, drivers and apps work together.

Modules

The "dummy" configuration runs two modules: *DriverDummy*, and *AppDummy*, and one role *RoleDummy*. These examples are a good reference for understanding how the HomeOS platform works. *DriverDummy* exports a port with the *RoleDummy* role, and two operations "echo" and "echosub", and sends out periodic notifications to other modules that subscribe to its echosub operation. *AppDummy* looks for all ports registered with role "dummy", invokes those ports' echo operation and subscribes to their echosub operation. The following table shows where to find the dummy components.

Module	Source Location
DriverDummy	\Hub\Drivers\Dummy
AppDummy	\Hub\Apps\Dummy
RoleDummy	\Hub\Common\Common\role.cs

Roles

Roles define the services offered by a particular type of device. For example, the role for a video camera with pan and tilt capabilities would include definitions for "up", "down", "left", and "right" while a binary sensor might just have "on" and "off". Roles are not specific to a particular device or app. Instead each role represents a set of capabilities which may be shared by many devices and apps. When writing or extending device drivers and apps, you can use the predefined roles or create a custom role class if necessary.

In the Hub project, some example roles are defined in Hub\Common\Common\role.cs. You can add a custom role class anywhere, as long as that class can be referenced by the driver and application modules that use it. The *RoleDummy* class was created to demonstrate the preferred pattern for creating a role. Scroll down through the code for the roles to see additional examples.

Drivers

Drivers determine how devices communicate with the platform. They work together with roles to bridge the gap between apps and devices. Drivers always inherit from the *ModuleBase* class. One interesting

driver is \Hub\Drivers\AxisCamera (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. \Hub\Apps\SmartCam is designed to interact with all connected cameras. It provides a GUI to view the image received from a camera driver, and to 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. Its interaction with AxisCamera provides an example of how complex objects such as images can be passed across modules.

HomeOS 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, but focus on how HomeOS uses this framework.

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.

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. The 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 light switch 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, image, sound, text, etc.
- Value: captures of the actual object.

Complex types can be passed using this framework. For instance, we pass images as (maintype=image; 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 the `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.

Subscription: This is similar to Invocation, but uses the `Subscribe()` function rather than returning once immediately. The values will be returned later via the `OnNotification()` 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 Code

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

```
IList<VPort> allPortsList = GetAllPortsFromPlatform();
foreach (VPort port in allPortsList)
    PortRegistered(port);
```

PortRegistered() is also called when new ports are registered with the platform. The following snippet shows a possible implementation for an application looking for all switches in a home:

```
public override void PortRegistered(VPort port)
{
    if (Role.ContainsRole(port, "roleswitch"))
    switchPorts.Add(port);
}
```

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

The following snippets show some examples of operation calls on ports. The first example shows calling an operation with no return values in order to turn on a light by setting its dimmer value to 99.

```
IList<VParamType> args = new List<VParamType>();
args.Add(new ParamType(ParamType.SimpleType.range, "0 99", 99, "level"));
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 next example shows calling an operation with no parameters to discover which media is currently being played, and how far into the media it is. Rather than setting the parameters, we must parse the return values

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

The final example shows a subscription and the notification handler `OnNotification()`.

```
switchPort.Subscribe("roleswitch", "getdimmer",
                this.ControlPort, this.switchPortCapability,
this.ControlPortCapability);

public override void OnNotification(string roleName, string opName, IList<VParamType>
retVals,
                                    View.VPort senderPort)
 {
     if (roleName.Equals("roleswitch") && opName.Equals("getdimmer"))
     {
        byte newDimmerValue = (byte)retVals()[0].Value();
      }
 }
```

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, ", retvalTypes, canSub)`, where " 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)`. 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 Code

The examples should clarify how drivers respond to operation invocations. The example class *RoleDummy* exports a role with two Operations, "echo" and "echosub". To initialize the module, we can do the following (in Hub\Drivers\Dummy\DriverDummy.cs):

The constructor for class *DummyRole* instantiates the role (in \Hub\Common\CommonRole.cs):

```
protected RoleDummy()
{
    SetName(RoleName);
     _instance = this;

    {
        List<VParamType> args = new List<VParamType>() {new ParamType(0)};
        List<VParamType> retVals = new List<VParamType>() {new ParamType(0)};
        AddOperation(new Operation(OpEchoName, args, retVals));
    }

    {
        List<VParamType> args = new List<VParamType>();
        List<VParamType> retVals = new List<VParamType>() { new ParamType(0) };
        AddOperation(new Operation(OpEchoSubName, args, retVals, true));
    }

}
```

The operation handler could be implemented as shown in Hub\Drivers\Dummy\DriverDummy.cs:

```
public override IList<VParamType> OnInvoke(string roleName, String opName,
IList<VParamType> args)
{

    if (!roleName.Equals(RoleDummy.RoleName))
    {
        logger.Log("Invalid role {0} in OnInvoke", roleName);
        return null;
    }

    switch (opName.ToLower())
    {
        case RoleDummy.OpEchoName:
            int payload = (int)args[0].Value();
            logger.Log("{0} Got EchoRequest {1}", this.ToString(),
payload.ToString());

            return new List<VParamType>() {new ParamType(-1 * payload)};

        default:
            logger.Log("Invalid operation: {0}", opName);
            return null;
    }
}
```

Let's also 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 (Hub\Drivers\Dummy\DriverDummy.cs):

```
public void Work()
{
    int counter = 0;
    while (true)
    {
        counter++;

        //IList<VParamType> retVals = new List<VParamType>() { new ParamType(counter)
};

        //dummyPort.Notify(RoleDummy.RoleName, RoleDummy.OpEchoSubName, retVals);

        Notify(dummyPort, RoleDummy.Instance, RoleDummy.OpEchoSubName, new
ParamType(counter));

        System.Threading.Thread.Sleep(1 * 5 * 1000);
    }
}
```

Device Scouts

Scouts in HomeOS are used to automatically discover devices and integrate them into the running platform. When adding devices, the flow is conducted as follows:

1. The scout discovers the devices in its environment.
2. The scout makes the platform aware of the devices.
3. The user can query the platform for discovered devices that are not already part of the system.
4. The user selects a device to add.
5. The device setup is accomplished via one or more HTML pages.
    a. The initial pages enable device configuration that is specific to the device type.
    b. The final page is the general HomeOS device addition page, where the device's location and associated apps can be configured.

The driver for the device is started as part of loading the final HTML configuration page. The scout is responsible to:

1. Discover devices and report to the platform when queried.
2. Host the UI (HTML pages backed by WCF services) for custom device configuration.
3. Hand over control to the platform by pointing to the generic device addition page.

Writing a Scout

The core requirements for a scout are fairly minimal. Scouts should inherit the IScout interface

```
public interface IScout
{
    void Init(string baseUrl, string baseDir, ScoutViewOfPlatform platform, VLogger
logger);
    List<HomeOS.Hub.Common.Device> GetDevices();
    void Dispose();
}
```

All three functions are invoked by the platform. Init() is called when starting the scout. GetDevices() is called to query for devices in the environment. Dispose() is called when the platform wants to terminate the scout (e.g., when the platform is shutting down).

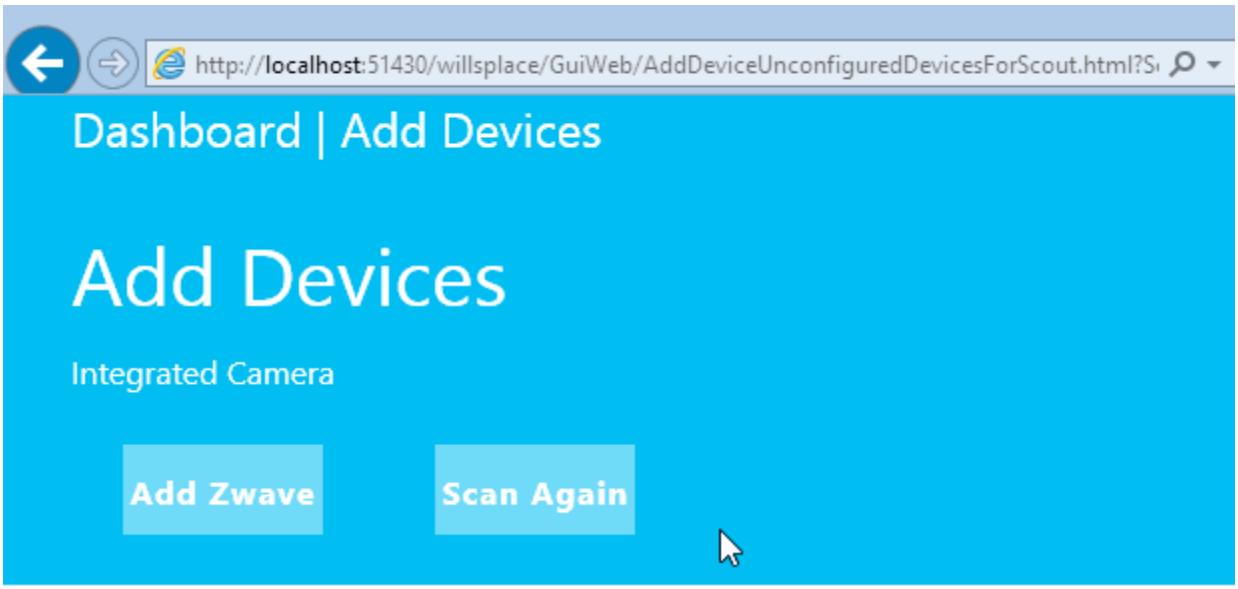
The call to Init() should return quickly. If a scout will take a long time to initialize, the work should be done on a separate thread called SafeThread.

The platform will call GetDevices() to query the scout for discovered devices. This call should return all devices that the scout finds. The scout does not need to worry about whether the device is already configured or not, as the platform will make that determination and act accordingly. However to help make that determination, the scout should assign a unique ID to each device.

Note: Device ID names should be deterministic, that is, the same ID should be consistently assigned to the device across different invocations of the scout and platform. For example, you could incorporate the device's MAC address into the ID.
The GetDevices() call should also return quickly, as the user may be waiting for its results. If triggering discovery for the devices is time consuming, we recommend having scout discover devices in the background and keep a list of recently discovered devices. This list can then be cloned and returned to platform when GetDevices() is called. The Foscam scout uses this approach. If triggering the discovery is quick, then for simplicity the scout may do that in response to the GetDevices() call. The webcam scout takes this approach. The result of GetDevices() is a list of objects of type Device (defined in Common\Common\Device.cs).

Hosting Device Configuration UI

This section discusses how to set up a user interface for device configuration. This is how you will expose configuration options to users in the dashboard. The Add Devices page ships by default as part of the dashboard, and is located here:
\Hub\Dashboard\DashboardWeb\AddDeviceUnconfiguredDevicesForScout.html.



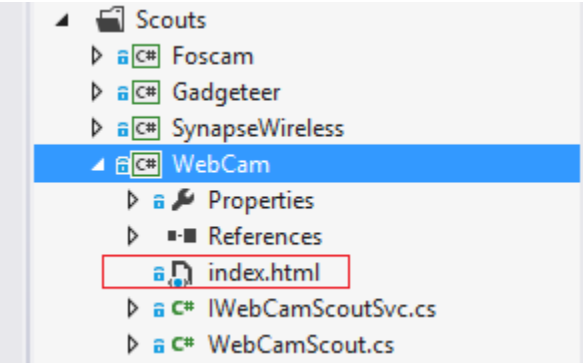
The JavaScript on this page gets a list of unconfigured devices:

```
//Expect URL to be called with HomeId and ScoutName parameters, this gets
unconfigured devices for that scout
//e.g.
http://localhost:51430/GuiWeb/AddDeviceUnconfiguredDevicesForScout.html?HomeId=Brush&
ScoutName=HomeOS.Hub.Scouts.WebCam
$(document).ready(
    function () {
        var qs = getQueryStringArray();
        if ((qs.ScoutName != undefined)) { //if ScoutName was passed as a parameter
get it
            SCOUTNAME = qs.ScoutName;
        }
        GetUnconfiguredDevices(SCOUTNAME);
    }
);

function GetUnconfiguredDevices(sName) {
    var url2 = "";
    var data2 = "";
    if (sName == "") {
        //Get them all
        url2 = "webapp/GetAllUnconfiguredDevices";
    }
    else {
        url2 = "webapp/GetUnconfiguredDevicesForScout";
        data2 = '{"scoutName": "' + sName + '"}';
    }
    new PlatformServiceHelper().MakeServiceCall(url2, data2,
GetUnconfiguredDeviceCallback);
}
```

In this example the only unconfigured device is the built-in webcam, represented by the Integrated Webcam hyperlink:
`http://localhost:51430/willsplace/scouts/HomeOS.Hub.Scouts.WebCam/index.html?DeviceId=Integrated%20Camera`

The index.html page is located in the \Scouts\WebCam project folder:



All setup tasks are implemented on the index.html page for the device. The following example shows a JavaScript snippet from the WebCam's index page.

```
var qs = getQueryStringArray();
var url = "../../GuiWeb/AddDeviceFinalDeviceSetup.html?DeviceId=";
if (qs.DeviceId !== 'undefined' && qs.DeviceId) {
    window.location.href = encodeURI(url + qs.DeviceId); //reroute
}
else {
    alert("Could not extract DeviceId from the URL " + window.location);
}
```

Since the WebCam doesn't require any additional setup, this code just gets the device ID of the cam then does a redirect. For an example of a more elaborate setup, take a look at `\Scouts\Foscam\index.html`.

Once the user has finished configuring the device, they are taken to the final setup page: `\Hub\Dashboard\DashboardWeb\AddDeviceFinalDeviceSetup.htm`.

Add a Scout Config Entry

In order for the platform to recognize a scout, you must add an entry for your scout in `\Configs\Config\Scouts.xml`. These are the default entries:

```
<Scout Name="HomeOS.Hub.Scouts.WebCam" DllName="HomeOS.Hub.Scouts.WebCam.dll"/>
<Scout Name="HomeOS.Hub.Scouts.Foscam" DllName="HomeOS.Hub.Scouts.Foscam.dll"/>
<Scout Name="HomeOS.Hub.Scouts.Gadgeteer" DllName="HomeOS.Hub.Scouts.Gadgeteer.dll"/>
```

Utilities

This section describes some utilities that module writers may 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

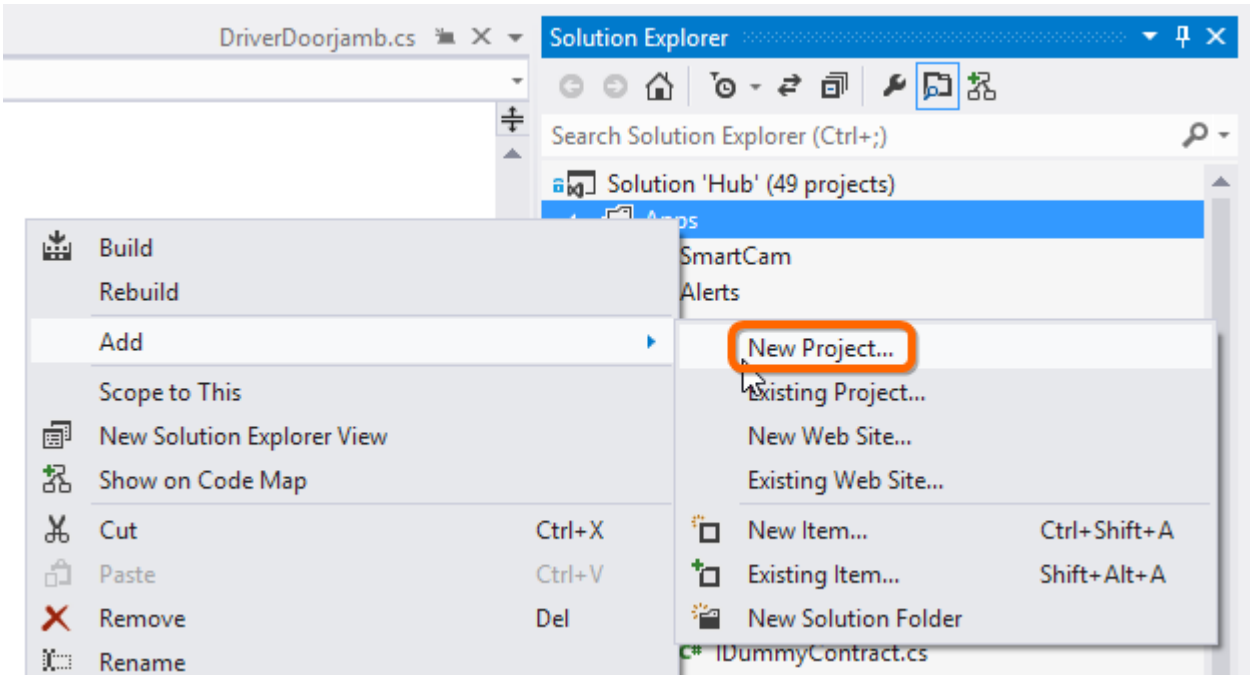
The description above focuses mostly on the communication between modules. In HomeOS modules are isolated from each other so that if one of them crashes, it doesn't impact others or the platform itself. Care has been taken in our design to retain this property but one exception where it fails is if a 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 manner similar to c# threads:

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

Creating a New HomeOS Module in Visual Studio 2012

1. Launch Visual Studio 2012.
2. In Solution Explorer, right-click the Apps folder, click **Add** and then click **New Project**.



3. In the **Add New Project** dialog box:
   - Choose the Visual C# Class Library template.
   - Enter a name for the project. In this example the name is AppNew.
   - Set the location to either the Apps or Drivers directory under homeos.
4. In Solution Explorer, expand the new project and right-click **References**.
5. In the **Add Reference** dialog box, click the **Projects** tab and select **Common** and **Views**.
6. In the **Add Reference** dialog box, click the **.NET** tab and select **System.AddIn**.
7. In Solution Explorer, expand the **References** item, right-click **Views**, go to **Properties** and set the **Copy Local** property to **False**.
8. In Solution Explorer, right-click the AppNew project and select **Properties**.
9. On the Build properties page, set the Output path to `..\..\output\binaries\Pipeline\AddIns\AppNew`.
10. In Solution Explorer, right-click the Class1.cs and rename it to AppNew.cs. Click **Yes** when prompted to rename the class in the file.
11. Add using directives for HomeOS.Hub.Common and HomeOS.Hub.Platform.Views.
12. Your main class should:
    - Inherit Common.ModuleBase
    - Add the attribute `[System.AddIn.AddIn("AppNew")]`.
    - Implement the functions marked abstract in ModuleBase. Place your cursor on "ModuleBase", then go to **Edit**, **IntelliSense**, **Implement Abstract Class**. The following code will be generated:

```
public override void Start()
{
    throw new NotImplementedException();
}

public override void Stop()
{
    throw new NotImplementedException();
}

public override void PortRegistered(VPort port)
{
    throw new NotImplementedException();
}

public override void PortDeregistered(VPort port)
{
    throw new NotImplementedException();
}
```

The Start() method is where the module is initialized. Be sure that the control never falls out of this function, otherwise the module will be unloaded. If your module doesn't do anything active but instead reacts to events from other modules, use System.Threading.Thread.Sleep(System.Threading.Timeout.Infinite) as the last line of Start().

Stop() should contain any cleanup actions.

PortRegistered() is called when a port is registered with the platform.

PortDeregistered() is called when a port is unregistered with the platform.

13. Finally, here are a few tips that may make things easier when building and running your application.
    - Add a project dependency for your project from the application so that it is automatically compiled before the platform is run. In Solution Explorer, right-click the Platform project and click **Project Dependencies**. Select **AppNew** from the list.

Running Your HomeOS Modules

There are three ways to run your module:

- Debug
- Modify Platform.cs
- Modify the Configuration Files

Debug a Module

1. Launch Visual Studio as an Administrator.
2. Open Hub.sln.
3. Ensure that Platform (\Hub\Platform\Platform) is set as the startup project.
4. Open modules.xml for the configuration you are using (\Hub\Output\configs).

5. Add an entry for your module as shown in the following example. Be sure to set the Autostart attribute to "1".

```
<Module FriendlyName="MyApp" AppName="MyApp"
BinaryName="HomeOS.Hub.Apps.MyApp" AutoStart="1" Background="1"
Version="1.0.0.0"></Module>
```

6. Press F5 to start debugging.

Modify Platform.cs

You can modify Platform.cs to run a module at launch. To do this, open \Hub\Platform\Platform\Platform.cs and navigate to the Start() function. Add the following line immediately before the closing bracket of Start() (substitute with the info for your own module):

```
StartModule(new ModuleInfo("friendly name is newnewapp", "app name is AppNew",
"AppNew", null, false, "no arguments here"));
```

Modify the Configuration Files

You can also modify the configuration files to launch your module when the platform is launched. To do this, add an entry for your module to \Hub\Platform\Configs\Config\Modules.xml.

The following example shows the module entries for *AppDummy* and *DriverDummy*:

```
<Modules>
  <Module FriendlyName="DriverDummy" AppName="DriverDummy"
BinaryName="HomeOS.Hub.Drivers.Dummy" AutoStart="1" Background="1" Version="1.0.0.0">
    <Args Count="1" val1="Hero"/>
  </Module>
  <Module FriendlyName="AppDummy" AppName="AppDummy"
BinaryName="HomeOS.Hub.Apps.Dummy" AutoStart="1" Version="1.0.0.0">
    <Args Count="1" val1="Zero"/>
    <RoleList>
      <Role Name=":dummy:"/>
    </RoleList>
  </Module>
</Modules>
```

If the AutoStart flag is set to 1, your module will start when the platform is launched, through InitAutoStartModules() (\Hub\Platform\Platform\Platform.cs).

IMPORTANT: If you modify any of the config files, be sure to first build the solution (F6) before running or debugging the platform. This ensures that the correct modifications will be applied to the output directory.

Troubleshooting

This section covers commonly encountered issues, and provides possible solutions.

My Module Won't Run

Build Directory is Improperly Set

A common error is that the build output directory is pointing to the wrong location. Most of the time, this can be fixed by taking the following steps:

1. In Solution Explorer, right-click the AppNew project and select **Properties**.
2. On the **Build** properties page, ensure that the **Output path** is set correctly relative to your app's directory (`homeos2\Hub\Apps\<your app>`).

   For example, if the AppNew app is located here:

   `C:\HomeOS2\homeos2\Hub\Apps\AppNew`

   the build output directory should be:

   `..\..\output\binaries\Pipeline\AddIns\AppNew`

HomeID Is Not Set

If no modules start, you may need to set the HomeID for the platform. To do this, take either one of the following steps:

- Start the platform, navigate to http://localhost:51430/guiweb/, and enter the information as prompted.
- Start the platform and type `guigeneric SetHomeIdWeb <homeid> <password>` on the platform console (replacing `<homeid>` and `<password>` with your own values).

Output Directory Contains Views Assembly

If the output directory for your module contains HomeOS.Hub.Platform.Views.dll, other modules won't run correctly. To fix this, take the following steps:

1. In Visual Studio, open Hub.sln.
2. In Solution Explorer, expand **Platform/References**, right-click **Views**, go to **Properties** and set the **Copy Local** property to **False**.
3. Delete HomeOS.Hub.Platform.Views.dll from the output directory.
4. Build the solution.