Using .NET Gadgeteer with Lab of Things

Contents

Overview

Lab of Things is a platform that enables researchers to share and collect data from a wide variety of sensors and devices. The .NET Gadgeteer platform offers a modular plug-and-play ecosystem for developing custom devices with a broad range of capabilities. You can use the .NET Gadgeteer platform to build your own custom devices that interact with Lab of Things.

This guide walks you through building a Wi-Fi-enabled remote temperature/humidity sensor. Reference implementations for the sensor device and its driver are included, along with a demonstration of how to use the sensor with existing HomeOS apps. Finally, we'll cover some tips for writing your own custom HomeOS app.

This guide assumes that you are familiar with developing in C#, however previous knowledge of HomeOS and Gadgeteer are not assumed. In order to complete this tutorial you will need to set up a Home Hub (a Windows computer that runs the HomeOS platform). See the documentation at http://www.lab-of-things.com/getstarted.html for more information about how to set up a Home Hub.

Requirements

This section describes the hardware and software required to complete the tutorial.

Software

- A computer (preferably a laptop) set up as a Home Hub.
- A local copy of the Lab of Things source code
- .NET Micro Framework 4 SDK
- Gadgeteer tools (an account with GHI is required)

Gadgeteer Modules

- FEZ Spider mainboard – there are around 20 different Gadgeteer mainboards available at the time of writing, however only the FEZ Spider is currently supported by HomeOS due to its support for Wi-Fi using the native TCP/IP stack.
- A USB power module and associated cable
- A Wi-Fi RS21 module
- A serial USB module and associated cable
- A button (used for resetting the Wi-Fi connection, optional)

- A multicolor LED (used to represent the Wi-Fi state, optional)
- A temperature and humidity sensor

How does the Home Hub interact with Gadgeteer?

- The HomeOS Home Hub runs on your computer.
- HomeOS can create a Wi-Fi network called `setup` by running `starthostednetwork.bat` in the Output folder of your Hub (Hub/output). Gadgeteer devices can then use the setup network to fetch the Wi-Fi credentials of the Home Hub's main (secured) Wi-Fi network.
- A component of HomeOS, Gadgeteer Scout, searches for Gadgeteer devices and sends the Wi-Fi credentials either via the USB serial line or via the setup Wi-Fi network.
- The Gadgeteer device will attempt to connect to the Wi-Fi network.
- Upon successful connection, the Gadgeteer device will use the HomeOS Gadgeteer libraries to create a web server. You are then able to handle web requests from HomeOS to either provide data to HomeOS or control hardware (e.g. displays, LEDs, relays).
- A driver that you write will communicate between HomeOS and the Gadgeteer devices, periodically checking whether each one has anything interesting to report and if so it will report the data to HomeOS.
- You can then write an app on top of HomeOS that will allow you to interact with your Gadgeteer device. HomeOS apps can also use the Windows Azure SDK to store data in the cloud so that you can easily view the data without direct access to the home hub.

Getting started with .NET Gadgeteer

This section introduces .NET Gadgeteer and shows you how to create the remote sensor device. If you have never worked with .NET Gadgeteer before we recommend following the steps in this section. You are also encouraged to try out some of the examples on the Gadgeteer website.

Confirm your firmware version

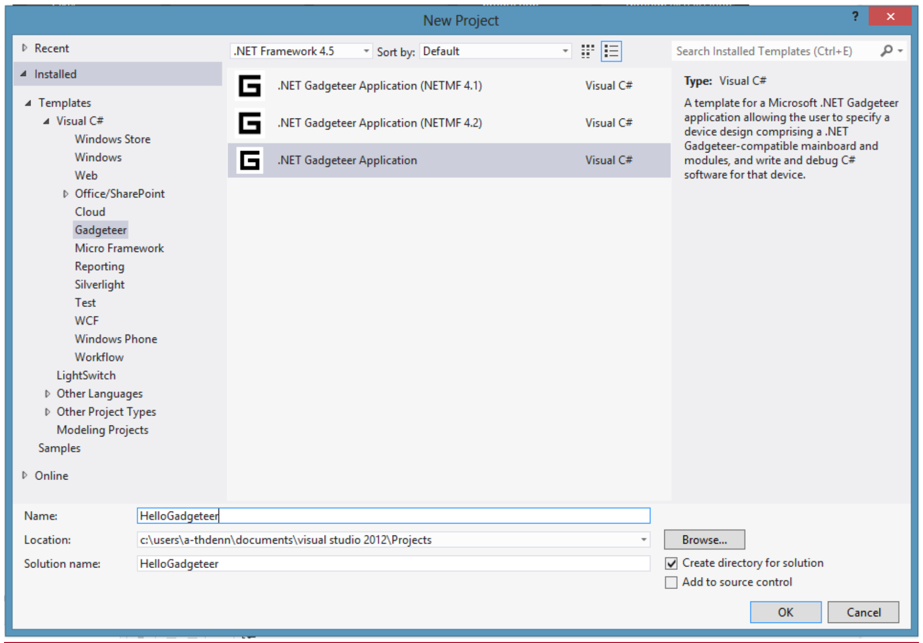
Before diving in, connect your FEZ Spider mainboard to your computer and use the FEZ Config utility to update to the latest version of .NET Micro Framework. Follow these instructions first. If you have trouble with the FEZ Config Utility, try using MFDeploy instead. If you're still having issues, visit the Gadgeteer Troubleshooting page.

Create a 'Hello World' device

This section explains how to build and test a simple Gadgeteer device which forms the basis for the next steps in this walkthrough. If you have not already set up Visual Studio 2012 with the **.NET MF 4.3 SDK** and **Gadgeteer tools**, you should do so now (see the previous section Requirements). You'll also need the Gadgeteer modules listed in the requirements section.

Create the 'HelloGadgeteer' project

Open Visual Studio and create a new Visual C# project. Select the **.NET Gadgeteer Application** type (the one with no version number) and name the project `HelloGadgeteer`.



.NET Gadgeteer Application (NETMF 4.1)	Visual C#
.NET Gadgeteer Application (NETMF 4.2)	Visual C#
.NET Gadgeteer Application	Visual C#

Type: Visual C#

A template for a Microsoft .NET Gadgeteer application allowing the user to specify a device design comprising a .NET Gadgeteer-compatible mainboard and modules, and write and debug C# software for that device.

Name: HelloGadgeteer

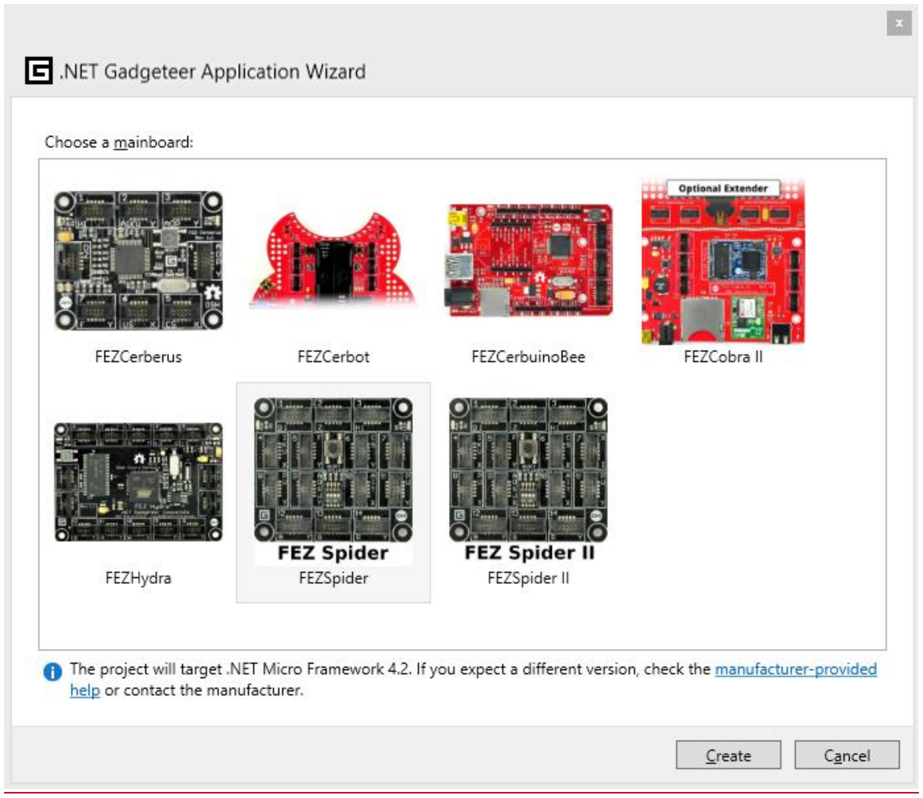
Location: c:\users\a-thdenn\documents\visual studio 2012\Projects

Solution name: HelloGadgeteer

[✓] Create directory for solution

[ ] Add to source control

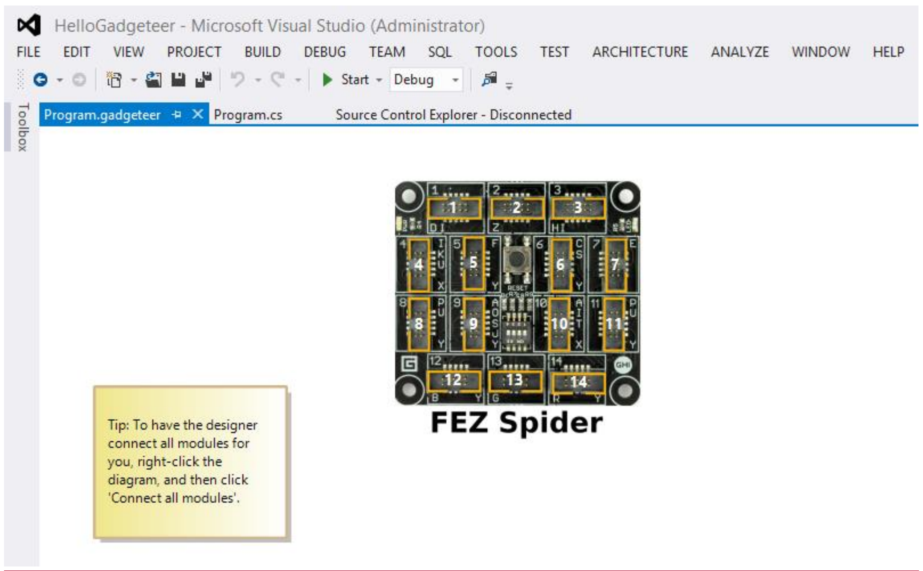
After creating the project you will be presented with the following prompt. Select the Fez Spider as your mainboard, then click **Create**.



The project will target .NET Micro Framework 4.2. If you expect a different version, check the manufacturer-provided help or contact the manufacturer.

Note—You should see the message "The project will target .NET Micro Framework 4.2". If you do not see this prompt, or if the message refers to a different version, you will need to ensure you are creating a ".NET Gadgeteer Application" (no version number).

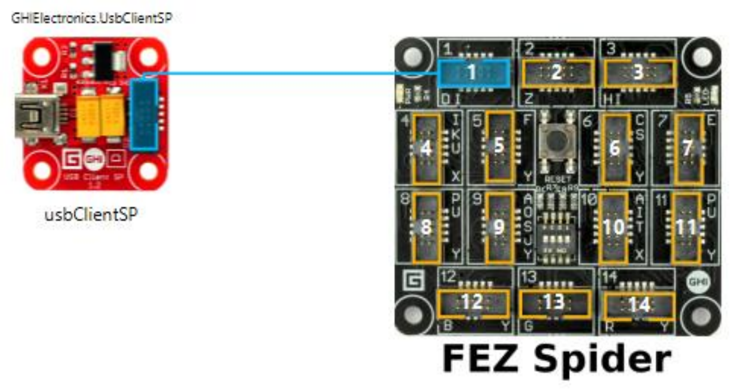
Next, you will be presented with the Gadgeteer designer. The designer allows you to connect Gadgeteer modules to your mainboard. Note that you must recreate the diagram that you create in the designer in real life with your modules and cables in exactly the same way:


HelloGadgeteer - Microsoft Visual Studio (Administrator)
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM SQL TOOLS TEST ARCHITECTURE ANALYZE WINDOW HELP
Start Debug
Toolbox
Program.gadgeteer
Program.cs
Source Control Explorer - Disconnected
FEZ Spider

Tip: To have the designer connect all modules for you, right-click the diagram, and then click 'Connect all modules'.

Connect the modules in the designer

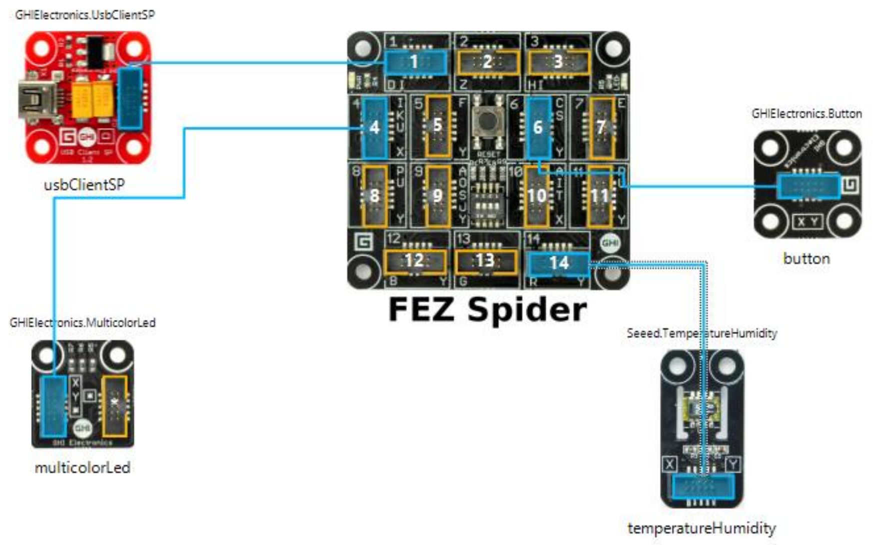
To add a new module to the designer, click on the Toolbox and drag it into the designer. Begin by dragging in the **USB Client SP** (SP is short for Single Power, you should have purchased this module already). The modules in the Toolbox are listed by manufacturer, so you should find this under **GHI Electronics**. Once you have dragged it, you can connect it to the mainboard by clicking and dragging from the socket on the power module to Socket 1 on the mainboard. You should find that the socket is highlighted when dragging:



Next, add the following modules and connect them to the listed sockets:

- GHI Electronics Button → Socket 6
- GHI Electronics Multicolor LED → Socket 4
- Seeed Temperature Humidity Sensor → Socket 14

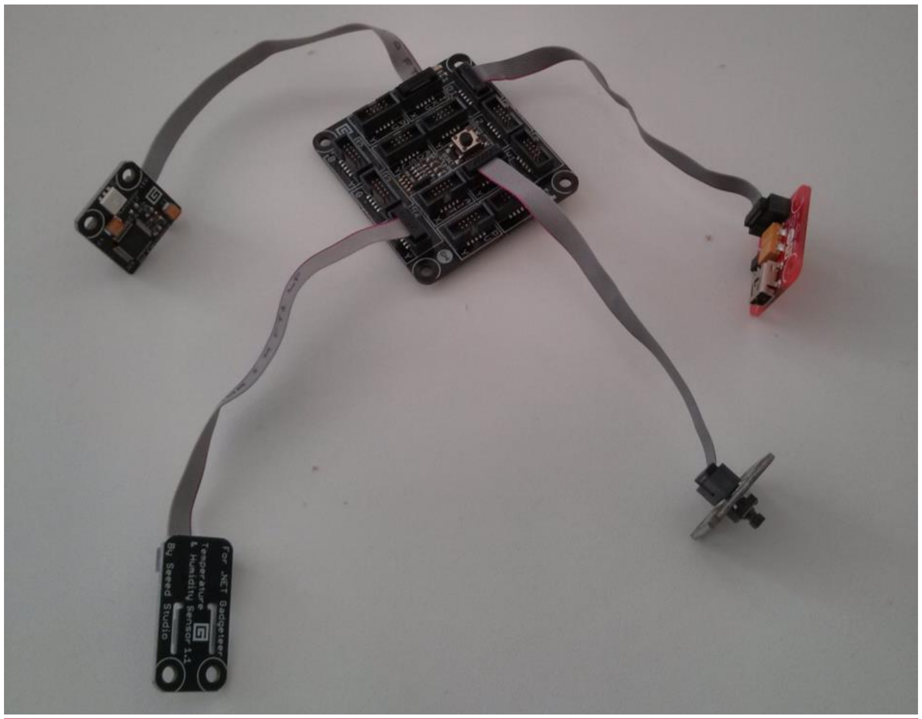
Note—This particular socket arrangement was selected to make it easier to see where things are connected.



Before continuing be sure to save the Gadgeteer designer file. This makes it possible to access the modules from your code.

Physically connect the modules

In this step you will connect the modules to the FEZ Spider mainboard, as shown in the Gadgeteer designer file. **Always unplug the FEZ Spider mainboard from your PC prior to connecting or removing modules!**



Your USB Client SP module includes a mini-USB cable. Use this cable to connect the device to your computer. Windows will automatically install the required drivers. If you have already installed the Gadgeteer drivers correctly you will receive a message confirming that drivers are properly installed.

You should now open Program.cs from the Solution Explorer. This file will contain the code of your application. The `ProgramStarted()` method is called when the device starts up. Replace the contents of the Program class with the following:

```
    public partial class Program
    {
        // This method is run when the mainboard is powered up or reset.
        void ProgramStarted()
        {
            button.ButtonPressed += button_ButtonPressed;
            temperatureHumidity.MeasurementComplete +=
temperatureHumidity_MeasurementComplete;
            temperatureHumidity.StartContinuousMeasurements();
            Debug.Print("Program Started");
        }

        // This method is run when the temperature sensor updates
        void temperatureHumidity_MeasurementComplete(GTM.Seeed.TemperatureHumidity
sender, double temperature, double relativeHumidity)
        {
            Debug.Print("Received temperature reading of " + temperature.ToString() + "
and humidity reading of " + relativeHumidity.ToString());
        }

        // This method is run when the button is pressed
        void button_ButtonPressed(GTM.GHIElectronics.Button sender,
GTM.GHIElectronics.Button.ButtonState state)
        {
            Random r = new Random();
            multicolorLed.SetRedIntensity(r.Next(256));
            multicolorLed.SetGreenIntensity(r.Next(256));
            multicolorLed.SetBlueIntensity(r.Next(256));
        }
    }
```

Note—Not all of the same classes of the .NET Framework are available in the .NET Micro Framework, however many equivalents exist.

The first two lines of code in `ProgramStarted()` create event handlers for the button being pressed and the temperature/humidity sensor reading a value. The sensor then begins reading values at regular intervals after `StartContinuousMeasurements()` is called.

The temperatureHumidity_MeasurementComplete method uses the Debug class (part of the Gadgeteer framework) to log the received values to the Output window of Visual Studio. When debugging Gadgeteer devices it is very useful to have this window open (you can view it using View, Output or by processing Ctrl + O, W).

Finally, when the button is pressed a random RGB color is generated and the LED displays the specified color.

Note—In some versions of the GHI drivers for the multicolor LED there is a bug where the Green/Blue values are the wrong way round so do not worry if your LED turns bright green when calling `SetBlueIntensity(255)`!

Now that you have designed and created the device as well as writing the code for it you will be able to press Start in Visual Studio to begin debugging the Gadgeteer device. Depending on which version of the driver you are using it may take up to a minute to deploy to the device. You should see something like the following in the Output Window:

```
Using mainboard GHI Electronics FEZSpider version 1.0

Program Started

Received temperature reading of 25.030000000000001 and humidity reading of 46.82

Received temperature reading of 25 and humidity reading of 46.75
```

You should also be able to press the button and see the LED turn a random color.

If this doesn't run correctly, see the Troubleshooting section for possible solutions.

Create a Lab of Things Gadgeteer device

In this section we'll use the HelloGadgeteer project as a starting point, adding the usbSerial and Wi-Fi_RS21 modules, and code to enable wireless communication between the sensor device and the Home Hub.

Update the HelloGadgeteer project

1. Open Visual Studio, and then open the HelloGadgeteer project you created earlier (if you have not created the HelloGadgeteer project yet, follow these instructions to do so now).
2. From Toolbox, drag the following devices into the Gadgeteer Designer file (Program.Gadgeteer) and connect them as follows:
    - usbSerial → Socket 11
    - Wi-Fi_RS21 → Socket 9
3. Verify that the FEZ Spider is unplugged from your computer, and then physically connect the usbSerial and Wi-Fi_RS21 modules to the indicated sockets on the FEZ Spider mainboard.
4. Right-click the solution name in Solution Explorer, click **Add** then click **Existing Project...**. Add the **HomeOSGadgeteerLibrary** project to your solution. This can be found in the Lab of Things source code at labofthings/Gadgeteer/HomeOSGadgeteerLibrary/HomeOSGadgeteerLibrary.csproj.
5. Right-click **References**, click **Add Reference**, select **Projects**, and then select HomeOSGadgeteerLibrary.
6. Press F6 to save and build the solution.

Expand the HelloGadgeteer code

You will need to add some new code so that the Home Hub can support the Gadgeteer component.

1. Open Program.cs and add the following using directive.

    ```
    using HomeOSGadgeteer;
    ```

2. Replace the contents of the Program class with the following:

```
     public partial class Program
     {
         HomeOSGadgeteerDevice hgd;
         double currentTemperature = 0;
         double highThreshold = 30;
         double lowThreshold = 20;

         // This method is run when the mainboard is powered up or reset.
         void ProgramStarted()
         {
             hgd = new HomeOSGadgeteerDevice("YourOrganization", //This should be the
name of your organization, or your name
             "TemperatureSensor", //The name of the sensor
             "abcdefgh", //A 'well known secret' - keep this value as the default for
now
             wifi_RS21, //The name of the Wi-Fi chip that HomeOS can connect with
             null, //The LED to be used to display the current status
             null, //An optional display (not used in this example) for displaying the
current display
```

```
            usbSerial.SerialLine.PortName, //The USB serial port to receive the Wi-Fi
credentials from
            null, //Optional parameter
            null, //Optional parameter
            () => { return GT.Timer.GetMachineTime() < TimeSpan.Zero; }, //Should it
control the LED?
            true); //Should it check for Wi-Fi credentials?

            hgd.SetupWebEvent("temp").WebEventReceived += Program_WebEventReceived;
            hgd.SetupWebEvent("led").WebEventReceived += Led_WebEventReceived;

            temperatureHumidity.MeasurementComplete +=
temperatureHumidity_MeasurementComplete;
            temperatureHumidity.StartContinuousMeasurements();

            // Use Debug.Print to show messages in Visual Studio's "Output" window
during debugging.
            Debug.Print("Program Started");
        }

        void temperatureHumidity_MeasurementComplete(GTM.Seeed.TemperatureHumidity
sender, double temperature, double relativeHumidity)
        {
            currentTemperature = temperature;
            updateLED();
        }

        void Program_WebEventReceived(string path,
            HomeOSGadgeteer.Networking.WebServer.HttpMethod method,
            HomeOSGadgeteer.Networking.Responder responder)
        {
            string response = "{\"DeviceId\":\"" +
                hgd.IdentifierString + "\"," +
                "\"Temperature\":" + currentTemperature.ToString() +
                "}";
            Debug.Print("Web event received from " + responder.ClientEndpoint + " -
response: " + response);
            responder.Respond(response);
        }

        void Led_WebEventReceived(string path,
            HomeOSGadgeteer.Networking.WebServer.HttpMethod method,
            HomeOSGadgeteer.Networking.Responder responder)
        {
            string lowString = responder.GetParameterValueFromURL("low");
            string highString = responder.GetParameterValueFromURL("high");
            if (lowString != null && lowString != "")
            {
                this.lowThreshold = (double)int.Parse(lowString);
            }
            if (highString != null && highString != "")
            {
                this.highThreshold = (double)int.Parse(highString);
            }
            responder.Respond("");
            updateLED();
        }
```

```
            void updateLED()
            {
                if (this.currentTemperature > this.highThreshold)
                {
                    multicolorLed.TurnRed();
                }
                else if (this.currentTemperature < this.lowThreshold)
                {
                    //Swap to TurnBlue() if your LED works correctly
                    multicolorLed.TurnGreen();
                }
                else
                {
                    //Swap to TurnGreen() if your LED works correctly
                    multicolorLed.TurnBlue();
                }
            }
        }
```

The program starts by creating a new instance of the HomeOS Gadgeteer Library with the appropriate information (such as the device name) and the various modules you are using. This will then immediately begin searching on the USB Serial Line for incoming Wi-Fi credentials from the HomeOS Home Hub and connect to the network.

Commented [AB1]: It would help comprehension to add some line numbers in the above code and then reference them in appropriate places here

Commented [TD(S2R1]: I will try and see if VS offers an easy way of doing this. I guess the alternative is to have comments with //Point 1 or something at various points in the code

Commented [AB3]: This is the point where I realized this was the usb setup option.

You also create a custom responder for web events – in this case when the request is at http://ip.of.your.device:80/temp. Furthermore, you also begin updating the temperature regularly and assign another event to this.

You may notice that the response is in JSON format. JSON is a widely used format that makes it easy to communicate complex data structures between different services and languages. It also provides a lightweight alternative to XML. If you have not developed with JSON before, you may wish to read more about it.

Before you can run this new Gadgeteer device you will need to get HomeOS set up, however if you have already done this you can skip to the Running your Gadgeteer Lab of Things device section.

Start the Home Hub

In this step we'll build the HomeOS project and make sure it runs correctly.

1. In Visual Studio open labofthings/Hub/Core.sln.
2. Press F6 to build the solution.
3. Open an instance of PowerShell with Administrator privileges.
4. Navigate to labofthings\Hub\output.
5. Run the following command:

   ```
   .\startplatform.bat
   ```

   The console will indicate whether the platform has started. Keep the PowerShell window open.

6. Open a web browser and navigate to the following URL:

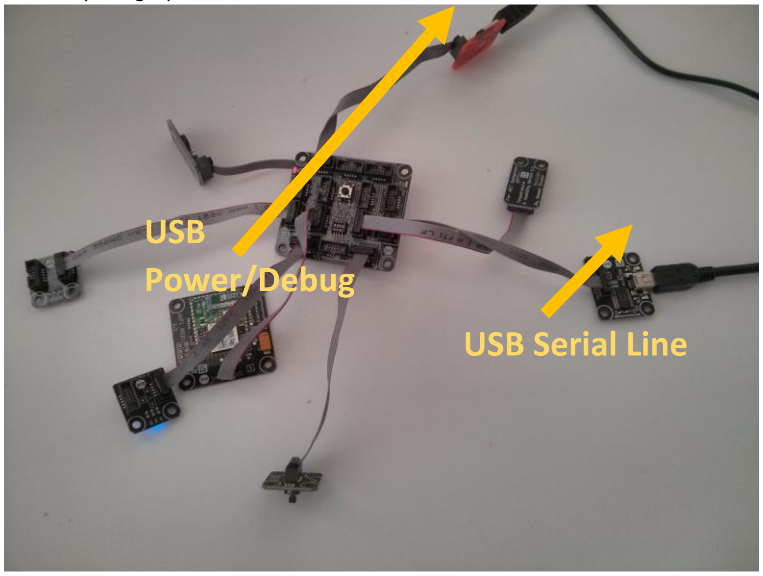
   ```
   http://localhost:51430/guiweb/
   ```

7. If this is your first time running the dashboard, you will be prompted for your local network's security key. Your Home Hub computer must be connected to the same Wi-Fi network as the one you entered the key for. Note that when testing you will not be able to use corporate networks that require you to enter user credentials.
8. To stop the server, type 'exit' at the prompt (no quotes).
9. Refer to the Lab of Things Getting Started Guide for further guidance.

Running your Gadgeteer Lab of Things device

Do the following to run your Gadgeteer code (you won't need to do all of this every time):

- Ensure that all of your Gadgeteer modules are connected in the same way in real life as there are in your diagram.
- Run the Hub (follow the steps in Start the Home Hub). Leave the PowerShell window open.
- Go to the 'Add devices' page of HomeOS (we won't do anything yet, but this page runs the scanner that will send the Wi-Fi credentials to the Gadgeteer device).
- Plug the Gadgeteer device into your computer via the red USB client power module *and* the USB serial module. This only needs to be done once, so that the Hub can use the serial connection to load data about the wireless connection onto the Gadgeteer device. It is recommended that you do this by using a powered USB hub.



-
- Run the Gadgeteer solution from Visual Studio. This will deploy the solution onto your Gadgeteer device and restart it. If you receive an error such as `'Device not found'` you should either try again or confirm that your device is correctly configured and that your build settings are correct (sometimes setting the startup target to `Program.ProgramStarted` in the properties window of the project can help).

- Open the Output window of Visual Studio. You should see the `Program started` message and the device will begin searching for Wi-Fi networks.
- If you don't see a message in the format '`Received new network credentials: ssid WI_FI_NETWORK_NAME_YOU_ENTERED_IN_HOMEOS key WI_FI_PASSWORD`' in the Output Window of Visual Studio then you should press 'Scan Again' in HomeOS
- Once the Gadgeteer device has connected to the Wi-Fi you should find it logs a message confirming this and that the LED glows blue (if you have chosen to use an LED)
- The Gadgeteer device will also log its IP address. Copy and paste this into your browser and open the page at http://ip.of.your.device:80/temp. If everything is configured correctly you should find that something similar to the following is shown:

```
{"DeviceId":"HomeOSGadgeteerDevice_TemperatureSensor_YourOrganization_25864053255964212759","T
emperature":24.800000000000001}
```

If this is correct you will be ready to begin writing a HomeOS driver for your device. If not, read the Troubleshooting Gadgeteer section.

Creating a HomeOS driver

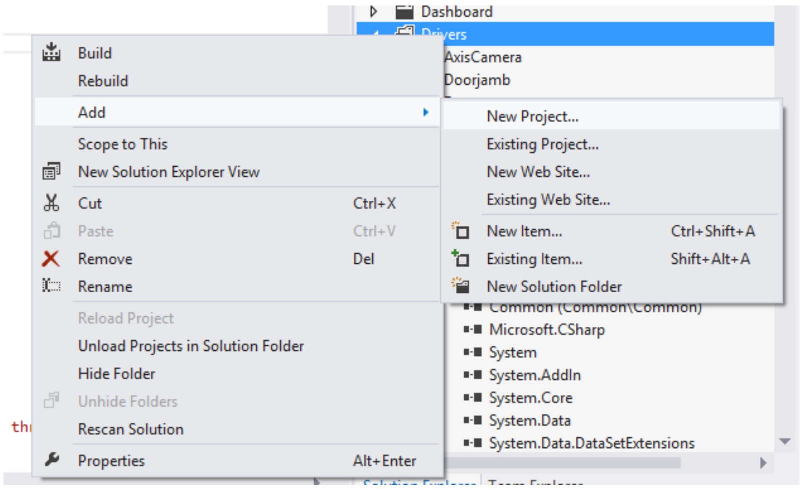
So far we've been able to create a Gadgeteer device that you can poll at regular intervals to check the temperature. At this stage you could write a very simple app to process the JSON from the device and store it in a database, however this would miss out on some key benefits offered by Lab of Things.

The HomeOS architecture dictates that each different kind of device has a 'driver' that HomeOS, and apps running on HomeOS, can poll at regular intervals to get data from the device in an abstract way. This means that there only needs to be one 'webcam' app, however several different drivers can exist that interact with cameras from several manufactures, for example.

Note—Several sections in this part of the guide are also explained in the Lab of Things developer guide found at http://labofthings.codeplex.com/releases/.

Creating the driver project

1. Open the Hub solution from the HomeOS directory.
2. In the Solution Explorer, right click on Drivers, click **Add** and then click **New Project:**



3. Create the new project with the following settings:
   - Project type: Visual C# Class Library
   - Project name: `Gadgeteer.YourOrganization.TemperatureSensor` (this should be the same as the values you provide when you create the Gadgeteer device)
   - Ensure that the project location is the `Hub/Drivers/` directory **(this is very important later!)**
4. In the Solution Explorer, add the following references:
   - Projects → **Common** and **Views**
   - .NET → **System.AddIn** and **System.Runtime.Serialization**
5. In References, right-click on **HomeOS.Hub.Platform.Views**, select **Properties** and set the **Copy Local** property to **False**.
6. Go to the Properties pane of your driver project, change the Output path to `..\..\output\binaries\Pipeline\AddIns\HomeOS.Hub.Drivers.Gadgeteer.OrganizationName.TemperatureSensor` (you may wish to click 'Browse' to ensure that you have got this correct – you will know if there are other similarly named drivers)
7. You will need to add the `DriverGadgeteerBase.cs` class to your project. This file is in `Hub/Drivers/GadgeteerBase`. Important: use the **Add as link** option when adding the class.
8. Rename `Class1.cs` to `DriverYourOrganizationTemperatureSensor.cs` and check Yes when it asks you if it should also rename the class
9. Add using directives in the source file for:
   - `HomeOS.Hub.Common`
   - `HomeOS.Hub.Platform.Views`
   - `System.Net`
   - `System.Runtime.Serialization`
   - `System.Runtime.Serialization.Json`
10. Change the namespace to `HomeOS.Hub.Drivers.Gadgeteer.YourOrganization.TemperatureSensor`

11. Change the `DriverYourOrganizationTemperatureSensor` class to inherit `DriverGadgeteerBase`
12. Add this attribute to your class:
    ```
    [System.AddIn.AddIn("HomeOS.Hub.Drivers.Gadgeteer.YourOrganization.TemperatureSens
    or")]
    ```
13. To make debugging easier, add your Driver project as a project dependency for the Platform project
14. To ensure that your driver is loaded correctly by HomeOS you should add the following to your `Modules.xml` configuration file:
    ```
    <Module
    FriendlyName="HomeOS.Hub.Drivers.Gadgeteer.YourOrganization.TemperatureSensor
    for TemperatureSensor" AppName="driver for TemperatureSensor"
    BinaryName="HomeOS.Hub.Drivers.Gadgeteer.YourOrganization.TemperatureSensor"
    AutoStart="1" Background="1">

        <Args Count="1"
    val1="HomeOSGadgeteerDevice_TemperatureSensor_YourOrganization_711215724423903
    89996" />

    </Module>
    ```

At this stage your source file should look like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Json;
using System.Text;
using System.Threading.Tasks;

using HomeOS.Hub.Common;
using HomeOS.Hub.Platform.Views;
namespace HomeOS.Hub.Drivers.Gadgeteer.YourOrganization.TemperatureSensor
{

[System.AddIn.AddIn("HomeOS.Hub.Drivers.Gadgeteer.YourOrganization.TemperatureSensor")]
    public class DriverYourOrganizationTemperatureSensor : DriverGadgeteerBase
    {
    }
}
```

Writing the driver

In the next step we'll implement the driver. As a reminder, all Modules (i.e. Apps and Drivers) must implement the following functions:

```
void Start()

void Stop()

void PortRegistered(VPort port)

void PortDeregistered(VPort port)
```

HomeOS simplifies this through the DriverGadgeteerBase class, which has the effect of reducing the amount of code you need to write.

Source code for the driver

Note—Be sure to update all of the "Your Organization" references to your organization's name, as you have done previously in the Gadgeteer device code.

```
[DataContract]
    public class Response
    {
        [DataMember(Name = "DeviceId")]
        public string DeviceId { get; set; }
        [DataMember(Name = "Temperature")]
        public double temperature { get; set; }
    }


[System.AddIn.AddIn("HomeOS.Hub.Drivers.Gadgeteer.YourOrganization.TemperatureSensor")]
    public class DriverYourOrganizationTemperatureSensor : DriverGadgeteerBase
    {
        int temp = 0;

        protected override void WorkerThread()
        {
            while (true)
            {
                try
                {
                    string url = string.Format("http://{0}/temp", deviceIp);

                    HttpWebRequest webRequest = (HttpWebRequest)WebRequest.Create(url);
                    HttpWebResponse response = (HttpWebResponse)webRequest.GetResponse();

                    if (response.StatusCode != HttpStatusCode.OK)
                        throw new Exception(String.Format(
                        "Server error (HTTP {0}: {1}).",
                        response.StatusCode,
                        response.StatusDescription));
                    DataContractJsonSerializer jsonSerializer = new
DataContractJsonSerializer(typeof(Response));
                    object objResponse =
jsonSerializer.ReadObject(response.GetResponseStream());
                    Response jsonResponse = objResponse as Response;

                    response.Close();

                    if (jsonResponse.temperature > 0)
                        logger.Log("Gadgeteer Temperature: {0}",
jsonResponse.temperature.ToString());

                    temp = (int)jsonResponse.temperature;

                        IList<VParamType> retVals = new List<VParamType>();
                        retVals.Add(new ParamType(temp));

                        devicePort.Notify(RoleSensor.RoleName, RoleSensor.OpGetName,
retVals);
```

```
                }
                catch (Exception e)
                {
                    logger.Log("{0}: couldn't talk to the device. are the arguments
correct?\n exception details: {1}", this.ToString(), e.ToString());

                    //lets try getting the IP again
                    deviceIp = GetDeviceIp(deviceId);
                }


                System.Threading.Thread.Sleep(4 * 1000);
            }
        }

        /// <summary>
        /// The demultiplexing routing for incoming
        /// </summary>
        /// <param name="message"></param>
        protected override List<VParamType> OnOperationInvoke(string roleName, String
opName, IList<VParamType> parameters)
        {
            switch (roleName.ToLower())
            {
                case RoleSensor.RoleName:
                    {
                        switch (opName.ToLower())
                        {
                            case RoleSensor.OpGetName:
                                {
                                    List<VParamType> retVals = new List<VParamType>();
                                    retVals.Add(new ParamType(temp));

                                    return retVals;
                                }
                            default:
                                logger.Log("Unknown operation {0} for {1}", opName,
roleName);
                                return null;
                        }
                    }
                case RoleActuator.RoleName:
                    {
                        switch (opName.ToLower())
                        {
                            case RoleActuator.OpPutName:
                                {
                                    try
                                    {
                                        string url =
string.Format("http://{0}/led?low={1}&high={2}", deviceIp, (int)parameters[0].Value(),
(int)parameters[1].Value());

                                        HttpWebRequest webRequest =
(HttpWebRequest)WebRequest.Create(url);
                                        HttpWebResponse response =
(HttpWebResponse)webRequest.GetResponse();
```

```
                                    }
                                    catch (Exception e)
                                    {
                                        logger.Log("{0}: couldn't talk to the device. are 
the arguments correct?\n exception details: {1}", this.ToString(), e.ToString());

                                        //lets try getting the IP again
                                        deviceIp = GetDeviceIp(deviceId);
                                    }
                                    return new List<VParamType>();
                                }
                            default:
                                logger.Log("Unknown operation {0} for {1}", opName, 
roleName);
                                return null;
                        }
                    }
                default:
                    logger.Log("Unknown role {0}", roleName);
                    return null;
            }
        }

        protected override List<VRole> GetRoleList()
        {
            return new List<VRole>() { RoleSensor.Instance, RoleActuator.Instance };
        }
    }
```

Source code explanation

The Response class

This class holds the data that will be parsed from the JSON response received from the Gadgeteer device. It uses attributes to enable the JSON parser to correctly set the property values. If you are writing more complex data structures you may wish to write custom getter/setter methods.

WorkerThread()

This method is first called by the background thread and runs in an infinite loop every four seconds until it is halted by `Stop()`. It works like this:

1. Figure out the URL that it needs to fetch from (i.e. the IP address of your Gadgeteer device).
2. Make an HTTP request against that address. When this runs the Gadgeteer device logs that it has received a request from your computer.
3. Decode the JSON response into the `Response` class.
4. Logs the recorded temperature to the logger.
5. If the newly received temperature differs from the previously received temperature (this number is rounded) then it should notify any apps that are subscribed to receive notifications from your driver.

OnOperationInvoke()

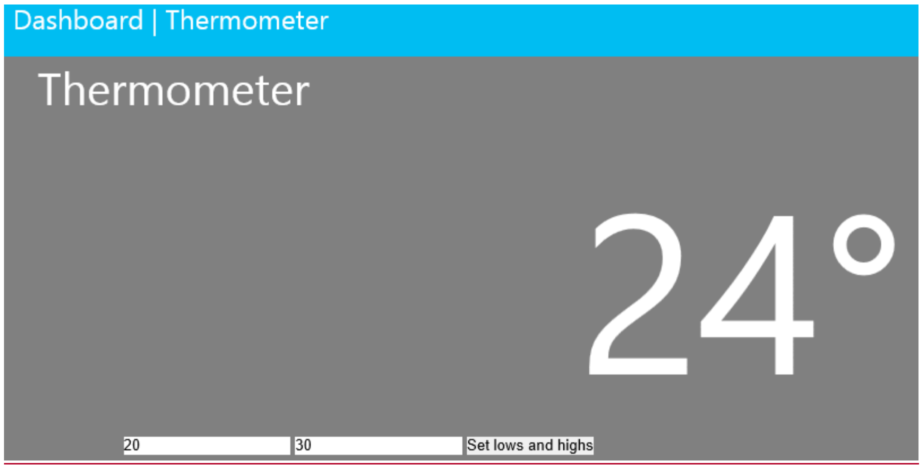
This method is called when the module (driver) receives a notification from apps that can work with it. The standard `OpGetName` operation will return a list with the current temperature, whereas the

`OpPutName` operation will be used by the app later on to change the LED colors based on the temperature.

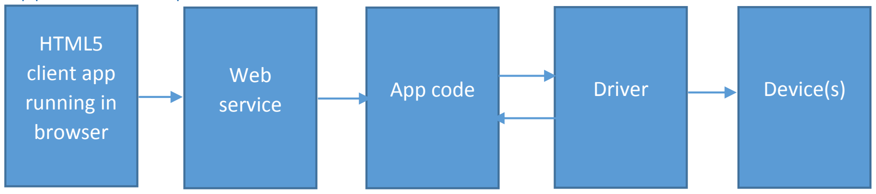
Note—You **will not** be able to use this driver with the Alerts app in its current form because this driver returns the exact value for the temperature, whereas Alerts will only trigger if it receives a value of 255.

Creating a custom HomeOS app

In its current state the temperature sensor is not particularly useful because it has no way of receiving interaction from HomeOS. The next stage of the project is to write a simple app for HomeOS that can display the current recorded temperature from the driver, as well as allowing the user to set a low threshold (i.e. if the temperature falls below that point the color of the LED on the device will change to blue) and a high threshold (i.e. if the temperature goes above that point the LED should change red).



App interaction pattern



In the preceding diagram, arrows indicate the direction in which messages are sent.

- The driver requests the current temperature from the device and reports this value to the app.
- The client requests the current temperature from the web service which then gets the value from the app.
- To set the LED color the client app makes a POST request to the web service. This causes the app to notify the driver, which then makes a request to the device.

Creating the app project

Creating an app is fairly similar to creating a driver:

1. Open the Hub solution from the HomeOS directory.
2. In the Solution Explorer, right click on Apps, click **Add** and then click **New Project.**
3. Create the new project with the following settings:
    - Project type: Visual C# Class Library
    - Project name: `Thermometer`
    - Ensure that the project location is the `Hub/Apps/` directory **(this is very important later!)**
4. In the Solution Explorer, add the following references:
    - Projects → **HomeOS.Hub.Common** and **HomeOS.Hub.Platform.Views**
    - .NET → **System.AddIn** and **System.Runtime.Serialization**
5. In References, right-click on **HomeOS.Hub.Platform.Views**, go to **Properties** and set **Copy Local** to **False**.
6. Go to the Properties pane of your driver project, change the Output path to `..\..\output\binaries\Pipeline\AddIns\HomeOS.Hub.Apps.Temperature` (you may wish to click 'Browse' to ensure that you have got this correct – you will know if there are other similarly named drivers)
7. Rename `Class1.cs` to `AppThermometer.cs` and check Yes when it asks you if it should also rename the class.
8. Add using directives in the source file for:
    - `HomeOS.Hub.Common`
    - `HomeOS.Hub.Platform.Views`
    - `System.Net`
    - `System.Runtime.Serialization`
    - `System.Runtime.Serialization.Json`
9. Change the namespace to `HomeOS.Hub.Apps.Thermometer.`
10. Change the class to inherit `ModuleBase.`
11. Add this attribute to your class: `[System.AddIn.AddIn("HomeOS.Hub.Apps.Thermometer")]`
12. To make debugging easier, add your Driver project as a project dependency for the Platform project
13. Add the following XML to your `Modules.xml` file (in `Hub/Output/Configs/[the_config_you_are_using]`):

```
<Module FriendlyName="Thermometer" AppName="Thermometer"
BinaryName="HomeOS.Hub.Apps.Thermometer" AutoStart="1" Background="0">
 <RoleList Optional="False">
<Role Name=":sensor:" />
</RoleList>
</Module>
```

14. You will also need to add the following XML to `ModuleDb.xml in Hub/Platform/HomeStore` (this notifies HomeOS that it can install the app):

```
<Module AppName="ThermoAlerts" BinaryName="HomeOS.Hub.Apps.Thermometer"
Description="Sample application for Gadgeteer guide" Rating="5">
    <RoleList>
      <Role Name=":sensor:" />
```

Commented [WF(AI4]: Manually updating config files: need to make sure folks know that if they reset these settings get killed.

Commented [TD(S5]: Is this actually necessary?

```
            </RoleList>
        </Module>
```

Writing the app code

HomeOS apps have four key elements:

- A class that handles connections between devices and HomeOS.
- A class that runs a web service that can respond to HTTP requests.
- An HTML page that uses a standard template for HomeOS apps to create your apps' UI.
- A JavaScript file that contains the code that interacts between the server (i.e. HomeOS) and the client side HTML page – for simple apps this can be included within the HTML file, however if your app becomes larger you may wish to split it out into several files or using something like TypeScript for module management.

AppThermometer.cs

This is the main class for handling connections between devices.

```
    [System.AddIn.AddIn("HomeOS.Hub.Apps.Thermometer")]
    public class AppThermometer : ModuleBase
    {
        AppThermometerService service;
        SafeServiceHost serviceHost;
        WebFileServer webUiServer;

        Dictionary<VPort, VCapability> registeredSensors = new Dictionary<VPort,
VCapability>();
        Dictionary<VPort, VCapability> registeredActuators = new Dictionary<VPort,
VCapability>();

        /// <summary>
        /// The most recently fetched temperature from the sensor
        /// </summary>
        public int Temperature = 0;

        public override void Start()
        {
            logger.Log("Started: {0}", ToString());

            service = new AppThermometerService(this, logger);

            serviceHost = AppThermometerService.CreateServiceHost(logger, this, service,
moduleInfo.BaseURL() + "/webapp");

            serviceHost.Open();

            webUiServer = new WebFileServer(moduleInfo.BinaryDir(), moduleInfo.BaseURL(),
logger);

            logger.Log("{0}: service is open for business at {1}", ToString(),
moduleInfo.BaseURL());

            //... get the list of current ports from the platform
            IList<VPort> allPortsList = GetAllPortsFromPlatform();
            if (allPortsList != null)
            {
                foreach (VPort port in allPortsList)
                {
                    PortRegistered(port);
                }
            }
        }

        public override void Stop()
        {
            serviceHost.Abort();
        }

        public override void PortRegistered(VPort port)
        {
            lock (this)
            {
                if (Role.ContainsRole(port, RoleSensor.RoleName))
                {
                    VCapability capability = GetCapability(port, Constants.UserSystem);

                    if (registeredSensors.ContainsKey(port))
                        registeredSensors[port] = capability;
                    else
                        registeredSensors.Add(port, capability);

                    if (capability != null)
                    {
                        port.Subscribe(RoleSensor.RoleName, RoleSensor.OpGetName,
                               this.ControlPort, capability, this.ControlPortCapability);
                    }
                }
                if (Role.ContainsRole(port, RoleActuator.RoleName))
                {
                    VCapability capability = GetCapability(port, Constants.UserSystem);

                    if (registeredActuators.ContainsKey(port))
                        registeredActuators[port] = capability;
                    else
                        registeredActuators.Add(port, capability);

                    if (capability != null)
                    {
                        port.Subscribe(RoleActuator.RoleName, RoleActuator.OpPutName,
this.ControlPort, capability, this.ControlPortCapability);
                    }
                }
            }
        }

        public void setLEDs(double low, double high)
        {
            foreach (var port in registeredActuators.Keys)
            {
                if (registeredActuators[port] == null)
                    registeredActuators[port] = GetCapability(port,
Constants.UserSystem);
```

```
                if (registeredActuators[port] != null)
                {
                    logger.Log(string.Format("Set LEDs {0},{1}", low, high));
                    IList<VParamType> parameters = new List<VParamType>();
                    parameters.Add(new ParamType((int)low));
                    parameters.Add(new ParamType((int)high));

                    port.Invoke(RoleActuator.RoleName, RoleActuator.OpPutName,
parameters, ControlPort, registeredActuators[port], ControlPortCapability);
                }
            }
        }

        public override void PortDeregistered(VPort port)
        {
            lock (this)
            {
                if (Role.ContainsRole(port, RoleSensor.RoleName))
                {
                    if (registeredSensors.ContainsKey(port))
                    {
                        registeredSensors.Remove(port);
                        logger.Log("{0} removed sensor port {1}", this.ToString(),
port.ToString());
                    }
                }
                if (Role.ContainsRole(port, RoleActuator.RoleName))
                {
                    if (registeredActuators.ContainsKey(port))
                    {
                        registeredActuators.Remove(port);
                        logger.Log("{0} removed actuator port {1}", this.ToString(),
port.ToString());
                    }
                }
            }
        }

        public override void OnNotification(string roleName, string opName,
IList<VParamType> retVals, VPort senderPort)
        {
            logger.Log("Notitification from {0} for {0}", roleName, opName);
            if (retVals.Count >= 1)
            {
                this.Temperature = (int)retVals[0].Value();
            }
            else
            {
                logger.Log("{0}: got unexpected retvals [{1}] from {2}", ToString(),
retVals.Count.ToString(), senderPort.ToString());
            }
        }
    }
```

Implementation details

- The Thermometer app class is a standard module in the same way that drivers are, and it is loaded once the app has correctly been installed.
- The `Start()` function creates a new web server (described below) that is able to respond to certain requests.
- The `Stop()` function aborts the service when HomeOS is shutting down or when the app is uninstalled/removed from the home hub.
- The `PortRegistered()` function is called when a new port connection is opened between the driver and the app. The device that we created earlier works as both a sensor – i.e. it can record values which can then be sent to HomeOS – and an actuator – i.e. it can receive values from HomeOS and change how it runs, such as changing the low and high thresholds for the LEDs. It is important to check for whether it is a Sensor or an Actuator as not all Gadgeteer devices are going to be both or either.
- The `SetLEDs()` function is called by the web service (described below) when the user wishes to change the LED thresholds. It searches through all of the connected Actuator ports, and sends a message containing the new values to the driver.
- The `PortDeregistered()` function handles the disconnection of devices and removes them appropriately from the app's internal dictionary of devices.
- The `OnNotification()` function is called when a new value is received from the driver and updates the currently recorded temperature.

AppThermometerSvc.cs

This is the service file that acts as a simple HTTP webserver. Note that this will require the same `using` declarations as the `AppThermometer.cs` source file.

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)]
    public class AppYourOrganizationThermometerService : 
ISimplexThermometerNotifierContract
    {
        private VLogger logger;
        private AppThermometer thermometerApp;

        public AppYourOrganizationThermometerService(AppThermometer thermometerApp, 
VLogger logger)
        {
            this.logger = logger;
            this.thermometerApp = thermometerApp;
        }

        public static SafeServiceHost CreateServiceHost(VLogger logger, ModuleBase 
moduleBase, ISimplexThermometerNotifierContract instance,
                                                     string address)
        {
            SafeServiceHost service = new SafeServiceHost(logger, moduleBase, instance, 
address);

            var contract = 
ContractDescription.GetContract(typeof(ISimplexThermometerNotifierContract));

            var webBinding = new WebHttpBinding();
            var webEndPoint = new ServiceEndpoint(contract, webBinding, new
EndpointAddress(service.BaseAddresses()[0]));
            webEndPoint.EndpointBehaviors.Add(new WebHttpBehavior());

            service.AddServiceEndpoint(webEndPoint);

            service.AddServiceMetadataBehavior(new ServiceMetadataBehavior());

            return service;
        }

        public double GetTemperature()
        {
            return thermometerApp.Temperature;
        }

        public string SetLEDs(double low, double high)
        {
            thermometerApp.setLEDs(low, high);
            return "";
        }
    }

    [ServiceContract]
    public interface ISimplexThermometerNotifierContract
    {
        [OperationContract]
        [WebInvoke(Method = "POST", BodyStyle = WebMessageBodyStyle.Wrapped,
ResponseFormat = WebMessageFormat.Json)]
        double GetTemperature();

        [OperationContract]
        [WebInvoke(Method = "POST", BodyStyle = WebMessageBodyStyle.Wrapped,
ResponseFormat = WebMessageFormat.Json)]
        string SetLEDs(double low, double high);
    }
```

Implementation details

- The interface defined at the bottom of the source code listing describes the functions that are offered by this web service. When a request is made to the server it will firstly check if it is `GetTemperature` or `SetLEDs` and if so it will call those functions for a JSON response (defined in the attribute of each function – `ResponseFormat = WebMessageFormat.JSON`), however if it is another request it will search for the file on the local disk (such as `index.html`) and serve that instead so you don't have to concern yourself with it. Note that it may not seem correct that the HTTP method for `GetTemperature` is a `POST` request however the HomeOS client JavaScript library handles all requests as `POST`s.
- The instantiation function creates a link to both the logger and the active instance of the app.
- The `CreateServiceHost()` function creates a new server instance and begins responding to requests.
- The `GetTemperature()` function returns the current temperature in the app and the value will be encoded as JSON before it is sent to the client.
- The `SetLEDs()` function will read the 'low' and 'high' HTTP parameters from a request from the app and then set them in the app itself.

index.html

After adding index.html to the project:

- Right-click index.html, select Properties, and set the **Copy to Output Directory** property to **Copy if Newer**.
- Copy the Styles folder from the Alerts app into the Thermometer app project, and set the **Copy to Output Directory** property for AppAlertsStyleSheet.css to **Copy if Newer**.

```
<script type="text/javascript" src="../GuiWeb/homeos-1.1.js"></script>
<!DOCTYPE html>
<html lang="en" class="js-disabled home">

<head>
  <title>Thermometer | HomeOS</title>
  <meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1" />
   <script type="text/javascript" src="../GuiWeb/jquery-1.7.1.js"></script>
  <script type="text/javascript" src="../GuiWeb/common.js"></script>
  <link rel="Stylesheet" type="text/css" href="Styles/AppAlertsStyleSheet.css" />

    <script>
        var currentTemperature = 0;
        var lowThreshold = 20;
        var highThreshold = 30;

        $(document).ready(
            function () {
                RefreshTemperature();
                setInterval(function () { RefreshTemperature() }, 2000);
            }
        );

        function RefreshTemperature() {
            new PlatformServiceHelper().MakeServiceCall("webapp/GetTemperature", "",
RefreshTemperatureCallback);
        }

        function RefreshTemperatureCallback(context, result) {
            currentTemperature = result.GetTemperatureResult;
            $("#temperatureValue").text(currentTemperature.toString());
            updateColors();
        }

        function updateColors() {
            if (currentTemperature < lowThreshold) {
                $("#bigtemperaturebox").css('color', 'light-blue');
                console.log("set blue");
            }
            else if (currentTemperature > highThreshold) {
                $("#bigtemperaturebox").css('color', 'red');
                console.log("set red");
            }
            else {
                $("#bigtemperaturebox").css('color', 'white');
                console.log("set white");
            }
            console.log("Updated color to " + $("#bigtemperaturebox").css("color"));
        }

        function setLEDs(e) {
            lowThreshold = parseInt($("#lowEntry").val());
            highThreshold = parseInt($("#highEntry").val());
            updateColors();
            var json = JSON.stringify({ low: lowThreshold, high: highThreshold });
            new PlatformServiceHelper().MakeServiceCall("webapp/SetLEDs", json ,
LEDCallback);
            return false;
        }

        function LEDCallback(context, result) {
            updateColors();
        }
    </script>
</head>
<body>
        <div class="dashboard_toolbar">
            <div class="homeID_title"><a href="../GuiWeb/index.html">Dashboard</a> |
Thermometer </div>
        </div>

        <div class="page">
            <div class="row">
                <div class="page_title col">Thermometer</div>
            </div>
            <div class="row">
                <div id="bigtemperaturebox" style="text-align:center;width:100%;font-
size:200px">
                    <span id="temperatureValue"></span>&deg;
                </div>
            </div>
            <div class="row">
                <div style="padding-left:100px">
                        <input type="number" value="20" id="lowEntry" placeholder="Low
value" />
                        <input type="number" value="30" id="highEntry" placeholder="High
value" />
                        <input type="submit" value="Set lows and highs"
onclick="setLEDs()" />
                    </div>
            </div>
        </div>

        <!--Used for debugging messages, set style="display:none" to turn off showing
them, common.js will put text into div with id =divDebugInfo when calls are made to
service -->
        <div id="divServiceDebug" style="display:block">
        Service Status Debug Info: <p id="divDebugInfo"></p>
        </div>


</body>
</html>
```

Implementation details

- This is a simple HTML file that will be served as the main UI for the app. This example uses HTML5 as a method of presentation for an app, but you could conceivably use another technology such as WPF/Windows 8 style apps.
- Several JavaScript and CSS files are included to provide the app with standard HomeOS JavaScript functions and a basic app style.
- When the page is loaded it refreshes the temperature once every two seconds – this is half the four second interval that the driver refreshes at to ensure that it picks up on every single temperature reading from the device; if it was exactly four seconds it would fall out of sync and miss potentially interesting readings.
- The `RefreshTemperature()` function creates a call to the GetTemperature function of the service, as defined earlier. JavaScript tends to use callbacks for asynchronous web requests, so a `RefreshTemperatureCallback()` function is also defined to receive the values from the service and present them on screen
- `updateColors()` is a simple JavaScript function that updates the CSS text color of the temperature box so that it shows the same color as the actual device when there is a temperature change
- The `setLEDs()` function is called when a user presses a button in the form to change the LED thresholds. It parses integers from the currently entered values in the form and then calls the `setLEDs` endpoint on the web service.

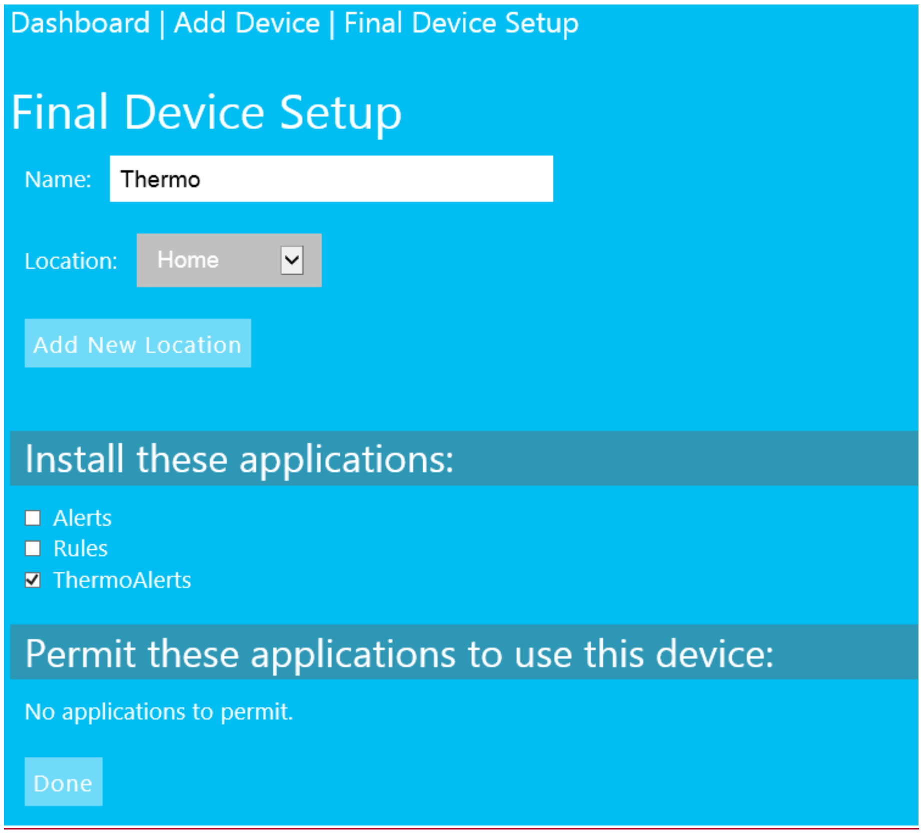
Running and Installing the app

Now you have created your first HomeOS app! Do the following in order to run the app:

1. Rebuild the entire HomeOS Hub solution.
2. Run the hub.
3. Wait for 'Waiting for commands' to show on the command line.
4. Open the Dashboard: http://localhost:51430/GuiWeb/.
5. Click 'Add devices'.
6. Wait for your temperature sensor to appear in the list before clicking on it.
7. Configure the device with your Wi-Fi credentials, if necessary.
8. Give the device a name and configure it with the following options:

Commented [WF(AI6]: All of this and no satisfaction. If I reset the hub I can get it to discover the sensor and take me to the settings page, but then I'm stuck. Subsequent attempts I do not see the sensor unless I reset the hub and start over.

Also none of this worked unless the Gadgeteer project was running with two USB cables. How does one get to the point where they can disconnect from USB and actually use the Wi-Fi? I assume that's the ultimate goal.


Dashboard | Add Device | Final Device Setup

Final Device Setup

Name: Thermo

Location: Home

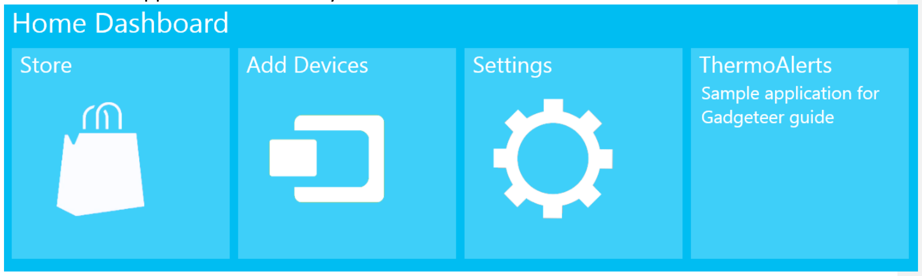
Install these applications:

- [ ] Alerts
- [ ] Rules
- [x] ThermoAlerts

Permit these applications to use this device:

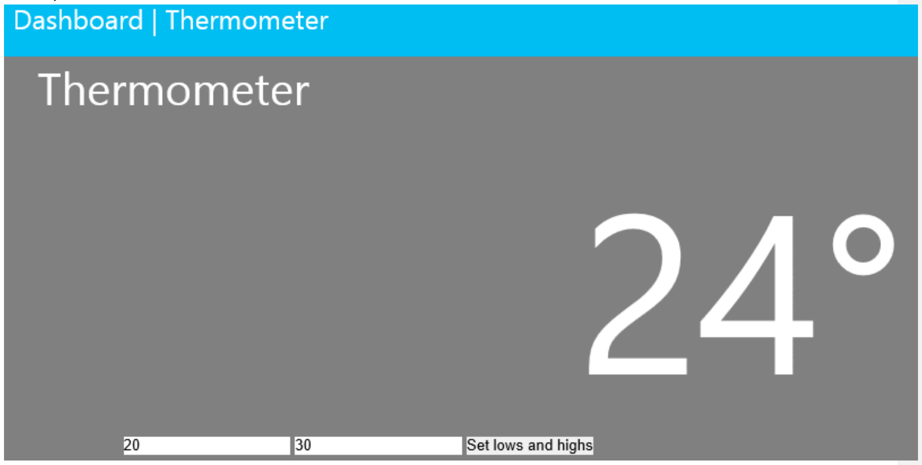
No applications to permit.

10. When you click 'Done' you will return to the HomeOS Dashboard and you should see that the ThermoAlerts app has been correctly installed:



11.
12. If you then launch the app you should then be redirected to a page like the following with the current temperature that your device is reading as well as the option to change the low threshold (i.e. the device will show a blue light if the temperature drops below the 'cold' value)

and the high threshold (the device will show a red light if the temperature goes above that hot value):

13. 

14. If you change the values for the low and high thresholds you should find that the color of the LED also changes on the device

Thermometer: Unplugged

At this point it is possible to run the Gadgeteer device as the autonomous wireless device it was meant to be. To do this, follow these steps:

1. Eject the Gadgeteer from your PC and unplug it (it appears as "EMX" in the USB device list).
2. Plug the device in using USB Client SP only.
3. Open a browser and check the IP for your Gadgeteer device (http://ip.of.your.device:80/temp). If the device is working you will see the default server page.
4. Reset the platform (on the command line type `reset.bat [name of config]`).
5. Start the Hub.
6. Follow the steps in Running and Installing the app.
7. Click the ThermoAlerts app. Now you are seeing wirelessly delivered sensor updates!

Troubleshooting

In theory if you have followed these instructions everything should run correctly however there are a few common pitfalls that may occur when attempting to run Gadgeteer code.

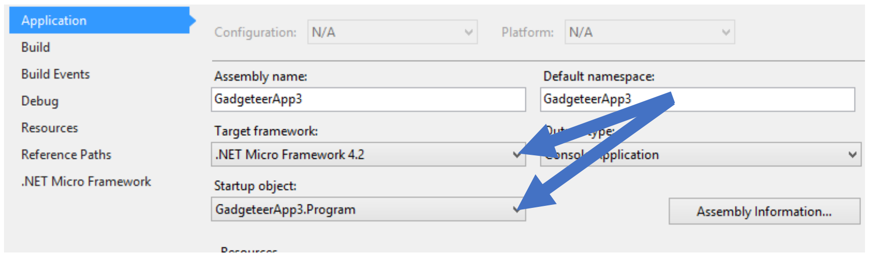
Device not found by Visual Studio debugger

Occasionally Visual Studio has problems deploying to the device, even if it is plugged in, correctly configured and you have the driver installed. This is usually a rare occurrence however you may find that it prints something like this in the Output window:

```
Iteration 0

Iteration 1
Iteration 2
. . .
Iteration 60
```

Usually this is because the build target has not been correctly set. Validate that the following is true in the properties window of Visual Studio:



Configuration: N/A Platform: N/A

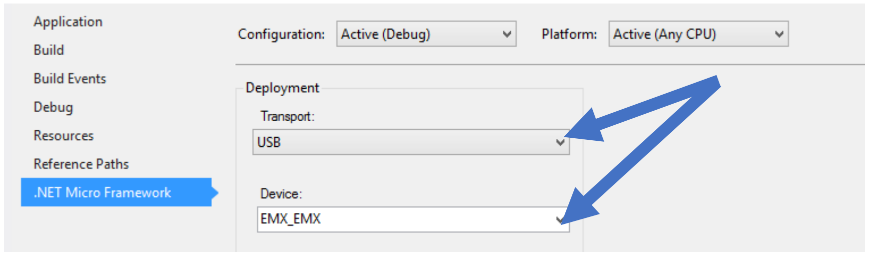
Assembly name: GadgeteerApp3

Default namespace: GadgeteerApp3

Target framework: .NET Micro Framework 4.2

Output type: Console Application

Startup object: GadgeteerApp3.Program

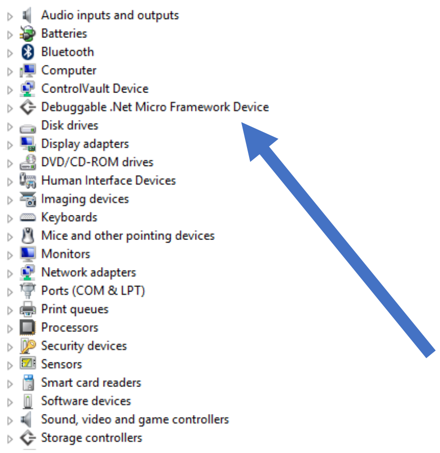


Configuration: Active (Debug) Platform: Active (Any CPU)

Transport: USB

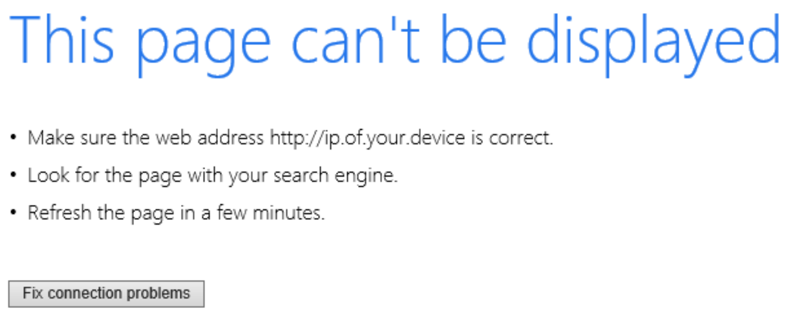
Device: EMX_EMX

If you are still having issues, it may be worth checking in Device Manager (accessible in Windows by press Start and typing 'Device Manager' – on Windows 8 and above it will appear in the settings category of the search pane) to confirm that 'Debuggable .Net Micro Framework Device' appears in the list of devices (if not, you probably haven't installed the drivers correctly):



404, timeout or page not found

If, when you try to go to http://ip.of.your.device:80/temp you get a page like the following then your Gadgeteer device probably isn't connected to your Wi-Fi network properly:



This page can't be displayed

- Make sure the web address http://ip.of.your.device is correct.
- Look for the page with your search engine.
- Refresh the page in a few minutes.

You must ensure that your Home Hub and your Gadgeteer device are both connected to the same Wi-Fi network and that this is the same Wi-Fi network that you have entered the credentials for in HomeOS. Furthermore, bear in mind that Gadgeteer is currently only compatible with 'normal' Wi-Fi networks; you won't be able to connect to a corporate Wi-Fi network that requires both a key for the Wi-Fi network and a username/password combination before connecting.

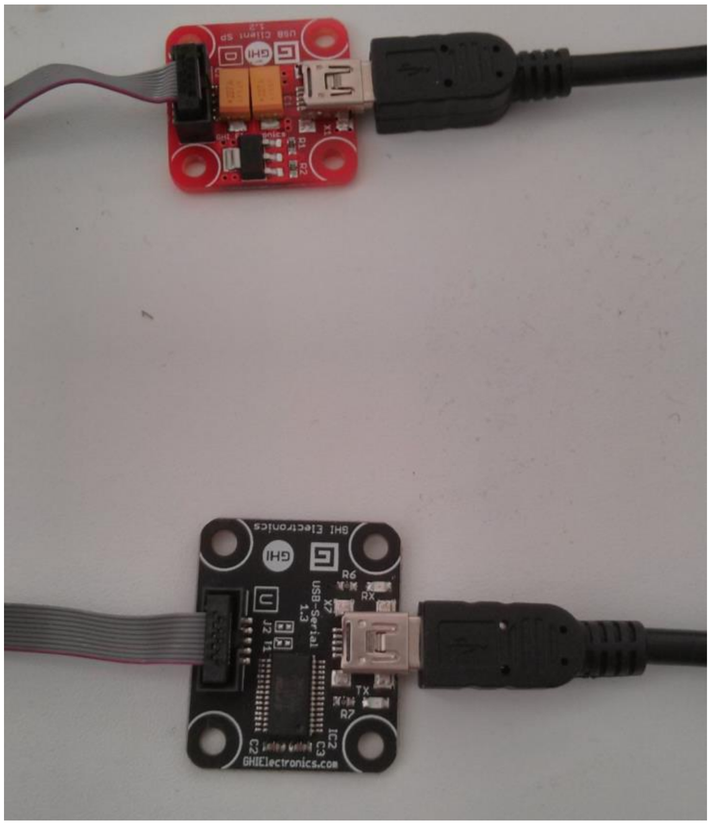
.NET Micro Framework error

You may find, if you have an old FEZ Spider board, that you have an error regarding the .NET Micro Framework (NETMF) version. This can occur because you haven't already installed the .NET Micro Framework SDK, so you may wish to validate this is installed on your PC. On the other hand, if NETMF is correctly installed, it is likely because your Gadgeteer mainboard has an old version of GHI Electronics' firmware. If this is the case you should follow GHI's instructions for updating your mainboard (after reading the introduction you will need to follow the **EMX instructions**).

HomeOS doesn't find my device

This is likely because you don't have the USB serial (the black – not the red – module plugged in as well), the device hasn't connected to Wi-Fi or you are not connected to the same Wi-Fi network on your Home Hub as you are on the device. Ensure that both the USB power/debug and the USB serial are plugged in as follows:

Commented [AB7]: Picture?



The red module is the power/debug, the black module is the USB serial line

The device shows up and I can add it however I get an error saying the device is not configured, press 'retry'

Unfortunately in this instance pressing retry probably isn't going to help. This is likely because you haven't added the `StartModule` call to `Start` in Program.cs or because you have not got the build path correctly set up. If you have confirmed that the former is true, go to Hub/output/binaries/Pipeline/AddIns and confirm that there is a folder called HomeOS.Hub.Drivers.Gadgeteer.YourOrganization.TemperatureSensor.

Commented [WF(AI8]: Huh? The example code does not show this anyplace.

Temperature shows as 0/The console shows an error saying the socket communication failed

This is probably because the device cannot be found or its IP address has changed since it first connected with HomeOS. Simply redeploying the Gadgeteer solution should cause it to correctly run.

'Service/Module not started, retrying' error in the main HomeOS dashboard when trying to load the app

Usually this message will disappear fairly quickly and the app will appear, however there are usually a few common solutions:

- The app is not listed in Modules.xml and ModulesDb.xml
- You have not rebuilt the solution fully after adding/making changes to the app
- Add a breakpoint at the end of the `Start()` method of your app to confirm whether it is running. If it is not one of the above is probably true

Unknown class RoleActuator

You are most likely using an old copy of the HomeOS source code. It is recommended that you get the latest version from Codeplex.

Commented [TD(S9]: Currently this is a lie because RoleActuator isn't in Codeplex (and neither is the Gadgeteer base class for drivers)