A .NET Gadgeteer Based Fermentation Temperature Controller

James L. Haynes

Introduction

This article is intended to demonstrate the design and creation of an automated fermentation temperature controller using a combination of commercial and custom .NET Gadgeteer compatible hardware and the .NET Micro Framework, a.k.a. NETMF. I chose to go with the .NET Gadgeteer platform for a variety of reasons including ease of use and the ability to program in C#. The .NET Gadgeteer is a good platform for this project as there are a wide variety of compatible main boards and modules available. It is also relatively easy to create your own custom modules which we will be doing in this project.

The purpose of a temperature controller is to maintain proper fermentation temperature for home brewing. A common practice in advanced home brewing is to use a residential refrigerator as a fermentation chamber to maintain the wort (unfermented beer) at the proper temperature during the fermentation process. For ales the ideal temperature range is between 65 and 70 °F, however, most refrigerators are not able to be set that warm, hence the need for a separate temperature controller. These temperature controllers are typically placed in-line with the refrigerator's power cord so that they can switch power on and off to the refrigerator as necessary in order to maintain the internal temperature in the desired range. The temperature controller will have some sort of remote temperature probe that is placed inside the refrigerator, and possibly in a thermowell inside the fermenter, in order to sense the temperature of the wort. There are a variety of temperature controllers already on the market including both analog and simple digital controllers. Most digital controllers consist of an LED or LCD display of current temperature, and a couple of buttons for adjusting the temperature set point. Some of these controllers can be configured to control a heater as well, but typically they can only control one device at a time. These controllers perform their function adequately; however, I want to build a controller that has a touch screen interface and the ability to control both a heater and refrigerator at the same time. The capability to heat and cool could be useful if your fermentation chamber is in a location such as a garage or shop where at certain times of the year outdoor temperature variations could result in the wort getting too cold at certain times of the day, and too warm at other times.

System Hardware

A .NET Gadgeteer system is composed of a microprocessor mainboard and a variety of modules which connect to the mainboard through a simple plug-and-play interface. The first step will be to select and purchase the hardware that we need to build the system.

To start out, we need a main board to build our system around. The FEZ Spider mainboard from GHI is a good starting point. It is based on a 72 MHz ARM7 processor and has 14 .NET Gadgeteer compatible expansion sockets. This board also supports touch input, which not all mainboards do.
The mainboard requires a red USB module in order to power it and to connect it to a computer for programming. Several are available, but I chose the GHI DP (dual power) module because it allows the mainboard to be powered from USB or from an external DC power supply.
Next we need a touchscreen module for the user interface. The GHI TE35 320x200 3.5" LCD Color display module with touch screen looks like it will suit our needs quite well.
In order to control a refrigerator we need to be able to turn the AC power on and off to it. This is a job for a relay. For this project we need 2 relays; one to control a heater and one to control a refrigerator. There are several .NET Gadgeteer relay modules available, but they contain from 4 to 16 relays, and most of those relays are rated to carry 10 amps or less. So rather than buying one of those modules, we will design our own custom relay module instead.
The whole purpose for this project is to control a refrigerator based on its internal temperature so we need some way to remotely sense that temperature. I initially chose to use GHI's Thermocouple module which comes with a 2 meter long k-type thermocouple that can be used from 0-400 degrees Celsius. Unfortunately, my initial experiments with this module showed it to have poor repeatability and accuracy in the temperature range I was interested in. It simply won't meet my needs. Since there are no other suitable modules currently available, we will need to design our own custom temperature sensor module as well.
Finally we need some sort of enclosure to put the system into so that we end up with a nice professional looking package. Plastic enclosures are available in a wide variety of shapes, sizes and colors. However, at this point we really don't know what size of enclosure to buy. Once we purchase the commercial modules and get further along in the design of the custom modules we should be better able to estimate the size of enclosure we will need.

Programming

The first step to beginning the software development is to install and setup all of the necessary development tools and libraries. We will be using Visual Studio 2010 and the .NET Micro Framework 4.2 SDK. Instructions for setting up the necessary software can be found at http://www.netmf.com/gadgeteer/get-started.aspx. We also need to install the .NET Gadgeteer Builder templates that can be found on the Gadgeteer CodePlex site. The MSI installs three templates. We will be using the kit and module templates for building our custom module drivers. Windows Installer XML (WiX) tools will also need to be installed for this project to work correctly. They can be installed from http://wix.codeplex.com/

Software Architecture

Now that we are ready to being writing software, we need to consider the software architecture. For this project I really want to separate the user interface from the controller operation as much as possible. This is a relatively common approach and there are several different architectures that have

been designed with that concept in mind. I have chosen to use the Model-View-Controller (MVC) architecture, which I feel would be a very good fit for this application.

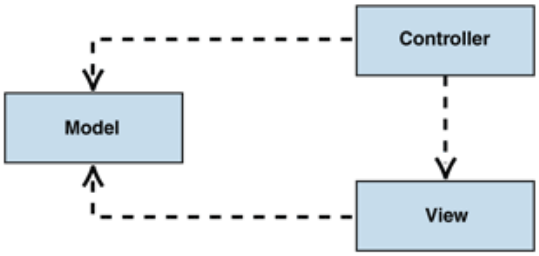

Figure 1: The MVC Architecture

In this architecture the data, the application behavior, and the user interface are separate. The Model contains the application data and the business logic. The views provide the user with a representation of the underlying data and application state. The controller, as you would expect, controls the application and defines the behavior by sending commands to the model.
We'll start by creating a new project using the .Net Gadgeteer Application template and name it TemperatureController as shown here.

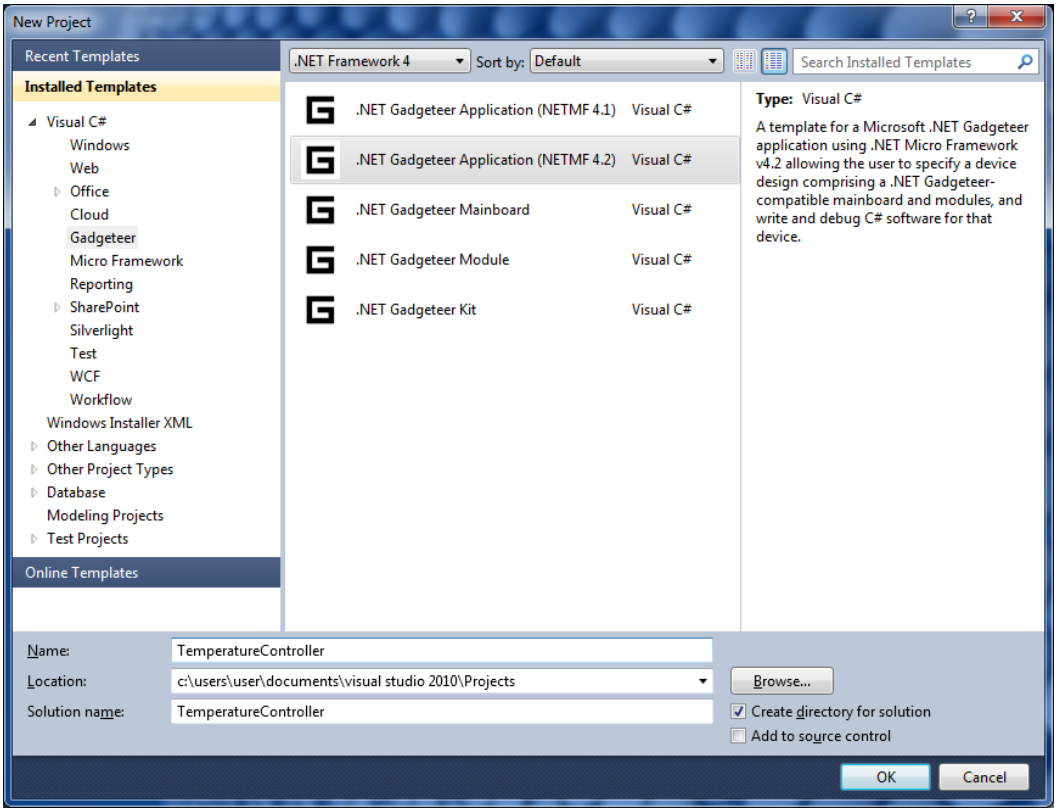

Figure 2: Creating a new .NET Gadgeteer project in VS2010

Using the Solution Explorer create directories for the views, the controller, and the model as well as custom controls.

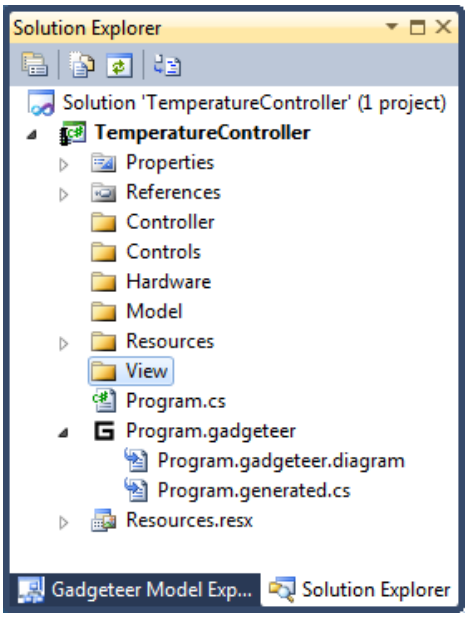

Figure 3: Project Folders

Next set the default namespace to NETMFTemperatureController by double-clicking on the Properties item in the Solution Explorer.

The Model

The first thing we'll do is create the model. The model will contain the application data and the business logic. This means that all of the user settings and current temperature are stored here, as well as the functionality such as temperature averaging and Celsius to Fahrenheit conversion. First, add a new class in the Model directory and name it TemperatureControlModel. Place this class in the NETMFTemperatureController.Model namespace.

The first thing we need is storage for the important system parameters and temperature measurements. As the internal temperature of our refrigerator gets to the temperature set point we don't want it to cycle on and off rapidly as that could eventually damage it. To prevent this we will add some hysteresis. This basically just means that the turn-on and turn-off temperatures are different. To keep things simple we will use a hysteresis value of 1 degree. Thus in cooling mode, the refrigerator will turn off when the temperature has reached the set point, and won't turn on again until the temperature rises to 1 degree above the set point.

```
public class TemperatureControlDataModel
{
   private const minTemp = 40;
   private const maxTemp = 90;
   public TemperatureControlMode ControllerMode { get; set; }
   public TemperatureControllerState CurrentState{ get; set;}
   public string TemperatureDegreeSymbol { get; private set; }
   public int hysteresisValue = 1;
   public int TemperatureSetPointHigh { get; private set; }
```

```
   public int TemperatureSetPointLow { get; private set; }

   private int temperatureSetPoint;
   public int TemperatureSetPoint
   {
      get { return temperatureSetPoint; }
      set
      {
         temperatureSetPoint = (maxTemp < value) ? maxTemp : (value <
            minTemp) ? minTemp : value;
         TemperatureSetPointHigh = (temperatureSetPoint + hysteresisValue);
         TemperatureSetPointLow = (temperatureSetPoint - hysteresisValue);
      }
   }
   …
```

Temperature averaging

In order to keep the temperature display from jumping around we will incorporate a running average of several temperature samples. Averaging the last 16 temperature samples should be more than adequate to ensure a smoothly changing display. This is easily accomplished with a `Queue` and a sample accumulator as follows:

```
private int averageCount = 16;
private Queue temperatureSamples = new Queue();
private float sampleAccumulator = 0;
public float AverageTemperature { get; private set; }

/// <summary>
/// Keep a running average of temperature samples
/// </summary>
/// <param name="sample"></param>
public void RecordTemperatureSample(float sample)
{
   sampleAccumulator += sample;

   temperatureSamples.Enqueue(sample);

   if (temperatureSamples.Count > averageCount)
   {
      sampleAccumulator -= (float)temperatureSamples.Dequeue();
   }

   AverageTemperature = sampleAccumulator / temperatureSamples.Count;
}
```

By using an accumulator with the Queue in this fashion the calculation of average temperature will always require the same amount of time regardless of the number of samples that are being averaged.

The Views

For the UI I would like to have a dashboard type overview of operating parameters. I also want the user to be able to adjust certain parameters such as temperature set point and operating mode. As I will be

using a relatively small 3.5" touch screen, things could get rather crowded if I try to put too much information on the screen at once. Due to limited real estate we should have several different screens or views for the different functions.
The NETMF does not support XAML, or the WPF designers, so the UI will have to be created in code using WPF derived classes, but considering the small size of this application that won't be too painful. The Visual Studio WPF designers can be used to help design/layout the views, however. Another issue is that in order to maintain a small footprint, the NETMF does not support all of the standard WPF elements. As such, some of the more useful layout elements of WPF such as `Grid` and `DockPanel` are, unfortunately, not available. We can achieve the layouts we want using only `StackPanels`, however, it takes a bit more work.
As I mentioned earlier, I want a dashboard view and a configuration or setup view. I would also like to have a separate idle view that will be displayed the majority of the time which simply displays current temperature and possibly an icon, such as a snowflake or a flame, to represent the controller's current operating state. In this project I am going to implement a basic Windows-like UI with a title bar and glassy buttons. Glassy looking buttons can be easily created in Photoshop by following any one of the many tutorials available online. All of the images used will be added to the project as resources and will be placed in the project's Resources directory.

View base class

We will begin by creating an abstract base class that all views will derive from. This will be a subclass of Canvas to make drawing/adding child elements easier. This class will provide the basic layout and functionality for all of the other views. Using the solution explorer, add a new class in the View directory and name it ViewBase. Place this class in the NETMFTemperatureController.View namespace. The ViewBase class is based on an example by Marco Minerva (http://mikedodaro.net/2011/11/18/windows-like-interface-for-the-net-gadgeteer-display-english/) with the addition of a couple of overloaded helper functions and no status bar. The series of overloaded helper functions will make it easier to add UI child elements such as `StackPanels`, `Text`, and `Images`.

```
public abstract class ViewBase : Canvas
{
   protected Model.TemperatureControlDataModel theModel;

   public ViewBase(Model.TemperatureControlDataModel model)
   {
      theModel = model;
      displayHeight = SystemMetrics.ScreenHeight;
      displayWidth = SystemMetrics.ScreenWidth;
   }

   public Border AddTitleBar(string title, Font font, GT.Color foreColor,
      Brush backgroundBrush)
   {
      Border titleBar = new Border();
      titleBar.Width = displayWidth;
      titleBar.Height = titlebarHeight;
      titleBar.Background = backgroundBrush;
```

```
            Text text = new Text(font, title);
            text.Width = displayWidth;
            text.ForeColor = foreColor;
            text.SetMargin(marginSize);
            text.TextAlignment = TextAlignment.Left;

            titleBar.Child = text;
            AddChild(titleBar);

            return titleBar;
        }
/// <summary>
/// Add a UIElement to the canvas at the origin (0,0)
/// </summary>
/// <param name="element"></param>
        public void AddChild(UIElement element)
        {
            this.Children.Add(element);
            Canvas.SetTop(element, 0);
            Canvas.SetLeft(element, 0);
        }
/// <summary>
/// Add a UIElement to the canvas at the specified location
/// </summary>
/// <param name="element"></param>
/// <param name="top"></param>
/// <param name="left"></param>
        public void AddChild(UIElement element, int top, int left)
        {
            this.Children.Add(element);
            Canvas.SetTop(element, top);
            Canvas.SetLeft(element, left);
        }
```

…

Dashboard view

The Dashboard view should give us a high-level view of important information about our temperature controller. The primary information of interest for this controller is of course the current temperature in our fermentation chamber so we definitely want to display that. I would also like to see additional information such as the temperature set point, the current mode and the current state of operation: whether it is currently heating, cooling, or idle.

Add a new class in the View directory named DashboardView. Derive this class from ViewBase and place it in the namespace NETMFTemperatureController.View

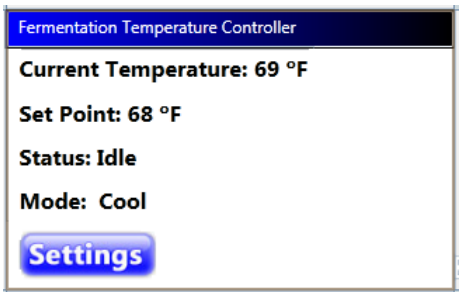

Figure 4: The Dashboard View

The Dashboard view will contain several `Text` elements and will update its display periodically to reflect the current state of the model. To accomplish this we will create a timer tick event handler that updates the display elements when the timer fires, but only if the view is currently visible. The timer itself exists in the Controller.

```
public class DashboardView : ViewBase
{
   private Text currentTempLabel;
   private Text setPointLabel;
   private Text stateLabel;
   private Text modeLabel;
   private Image settingsButton;

   public void timerDisplayUpdate_Tick(object sender, EventArgs e)
   {
      if (this.Visibility == Visibility.Visible)
      {
         UpdateDisplay();
      }
   }

   public void UpdateDisplay()
   {
      currentTempLabel.TextContent =
         Resources.GetString(Resources.StringResources.currentTempLabel) +
         theModel.AverageTemperature.ToString("F1") +
         theModel.TemperatureDegreeSymbol;
      setPointLabel.TextContent =
         Resources.GetString(Resources.StringResources.setpointLabel) +
         theModel.DesiredTemperature.ToString() +
         theModel.TemperatureDegreeSymbol;
      stateLabel.TextContent =
         Resources.GetString(Resources.StringResources.statusLabel) +
         theModel.CurrentStateStringTable[(int)theModel.CurrentState];
      modeLabel.TextContent =
         Resources.GetString(Resources.StringResources.currentModeLabel) +
         theModel.ControllerModeStringTable[(int)theModel.ControllerMode];
   }
...
```

Settings view

From the settings view we should be able to change the temperature set point, toggle temperature units between Fahrenheit and Celsius, and change the control mode to heat, cool or both.
Add a new class in the View directory and name it SettingsView. Again derive this class from ViewBase and place it in the NETMFTemperatureController.View namespace.
In order to change the temperature set point and toggle temperature units and modes we need to have some buttons. For temperature it makes sense to have buttons for increasing and decreasing the values. These could either be labeled with '+' and '-' symbols or with up and down arrows. For the temperature units and modes '+' and '-' symbols really don't make sense. In order to present a consistent look I'll use arrows for all of the buttons.

Custom Components – Up-Down button

The `TouchDown` event handler for each button image can be programmed to affect the appropriate variables but it would be nice to have a single component to interact with for each parameter rather than two separate buttons and multiple text labels. To make the UI easier to work with and look more consistent we will create a custom up/down button component. This component will have two images for buttons and two text components, all placed in a `StackPanel`. The component will display the parameter name and current value, and pressing the buttons will change the value. Here's what our new component will look like:

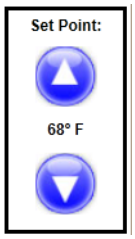

Figure 5: The Up/Down Button Component

Our new component will notify the system of user input by publishing two custom events; `UpButtonTouchDown` and `DownButtonTouchDown`. Basically, we will intercept the `TouchDown` event from each button image and then fire our own Up or Down touch event as necessary.
Add a new class in the Controls directory and name it UpDownButton. This class will derive from `ContentControl` and will be in the namespace NETMFTemperatureController.Controls.
First, we need to declare two events based on the `TouchEventHandler` delegate as follows:

```
public event TouchEventHandler UpButtonTouchDown;
public event TouchEventHandler DownButtonTouchDown;
```

Next, we need methods to raise the events:

```
protected virtual void UpButton_TouchDown(TouchEventArgs e)
{
   TouchEventHandler handler = UpButtonTouchDown;
   if (handler != null)
```

```
    {
        handler(this, e);
    }
}
protected virtual void DownButton_TouchDown(TouchEventArgs e)
{
    TouchEventHandler handler = DownButtonTouchDown;
    if (handler != null)
    {
        handler(this, e);
    }
}
```

And finally, in order to raise our new events we need to create event handlers to intercept the touch event for the images.

```
void arrowUpImage_TouchDown(object sender, TouchEventArgs e)
{
    UpButton_TouchDown(e);
}

void arrowDwnImage_TouchDown(object sender, TouchEventArgs e)
{
    DownButton_TouchDown(e);
}
```

So now when the user touches an arrow button on our control, the image's `TouchDown` event will fire, which in turn will raise one of our custom events.
The Settings view can now use this new component for displaying and changing system parameters.

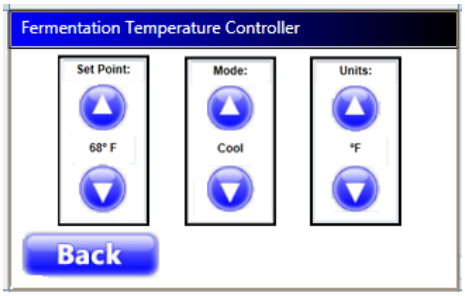

Figure 6: The Settings View

This view will not only reflect the current state of the model, but also be able to change it. The Set Point Up/Down control will simply increase or decrease the temperature set point. For sanity we limit the range of acceptable set point values in the Model's `TemperatureSetPoint` property. The Units Up/Down control will simply toggle back and forth between Fahrenheit and Celsius. The Mode Up/Down control, however, will not be quite so easy. This control will actually cycle through a list of operating modes and will wrap around when it gets to the end. To accomplish this we will have to create a couple of functions in the Model which cycle up or down through the list of available operating modes. The text for each mode will be stored as string resources.

```
public class TemperatureControlDataModel
{
/// <summary>
/// Cycle down through operating modes
/// </summary>
   public void OperatingModePrevious()
   {
      switch (ControllerMode)
      {
         case TemperatureControlMode.modeOff:
            ControllerMode = TemperatureControlMode.modeHold;
            break;
         case TemperatureControlMode.modeHold:
            ControllerMode = TemperatureControlMode.modeHeat;
            break;
         case TemperatureControlMode.modeHeat:
            ControllerMode = TemperatureControlMode.modeCool;
            break;
         case TemperatureControlMode.modeCool:
            ControllerMode = TemperatureControlMode.modeOff;
            break;
         default:
            throw new NotSupportedException("Invalid
            TemperatureControlMode");
         }
}

/// <summary>
/// Cycle up through operating modes
/// </summary>
   public void OperatingModeNext()
   {
      switch (ControllerMode)
      {
         case TemperatureControlMode.modeOff:
            ControllerMode = TemperatureControlMode.modeCool;
            break;
         case TemperatureControlMode.modeCool:
            ControllerMode = TemperatureControlMode.modeHeat;
            break;
         case TemperatureControlMode.modeHeat:
            ControllerMode = TemperatureControlMode.modeHold;
            break;
         case TemperatureControlMode.modeHold:
            ControllerMode = TemperatureControlMode.modeOff;
            break;
         default:
            throw new NotSupportedException("Invalid
            TemperatureControlMode");
      }
}
…
```

Our event handlers for the Mode Up/Down control will then call one of these functions as follows:

```
public class SettingsView : ViewBase
{
…
    void ModeChangeButtons_DownButtonTouchDown(object sender, TouchEventArgs
    e)
    {
        theModel.OperatingModePrevious();
        ModeChangeButtons.ButtonValue =
            theModel.ControllerModeStringTable[(int)theModel.ControllerMode];
    }

    void ModeChangeButtons_UpButtonTouchDown(object sender, TouchEventArgs e)
    {
        theModel.OperatingModeNext();
        ModeChangeButtons.ButtonValue =
            theModel.ControllerModeStringTable[(int)theModel.ControllerMode];
    }
```

Idle view

The idle view will be displayed 99% of the time while the controller is just doing its job of monitoring and controlling the refrigerator's internal temperature. This view is intended to provide a minimum of information at a glance when no user interaction is required. I envision this view being a very simple display of current temperature and a status icon on a black background. To create this view add a new class in the View directory and name it IdleView. Place it in the NETMFTemperatureController.View namespace and derive it from ViewBase.

When the temperature controller is calling for cooling it will display a snowflake icon, and when it is calling for heat it will display a flame icon. Otherwise, the icon should be blank.

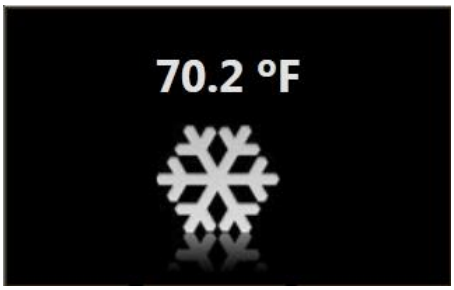

Figure 7: The Idle View

Much like the other views, the Idle view contains a timer tick event handler that updates the display elements when the timer fires, if and only if the view is currently visible. The timer itself exists in the controller.

```
public class IdleView : ViewBase
{
    public void timerDisplayUpdate_Tick(object sender, EventArgs e)
    {
        if (this.Visibility == Visibility.Visible)
        {
            UpdateDisplay();
```

```
        }
    }

    public void UpdateDisplay()
    {
        if (theModel.ControllerMode == Model.TemperatureControlMode.modeOff)
        {
            temperatureDisplayLabel.TextContent =
                    Resources.GetString(Resources.StringResources.offModeText);
        }
        else
        {
            temperatureDisplayLabel.TextContent =
                theModel.AverageTemperature.ToString("F1") +
                theModel.TemperatureDegreeSymbol;
        }

        switch (theModel.CurrentState)
        {
            case Model.TemperatureControllerState.stateIdle:
                statusImage.Bitmap = idleBitmap;
                break;
            case Model.TemperatureControllerState.stateHeating:
                statusImage.Bitmap = heatBitmap;
                break;
            case Model.TemperatureControllerState.stateCooling:
                statusImage.Bitmap = coolBitmap;
                break;
            default:
                throw new NotSupportedException("Invalid
                      TemperatureControlStatus");
        }
    }
    …
```

Error view

We need a view to display system errors to the user. This will be a very simple window with an error message displayed in red text.

Add a new class in the View directory and name it ErrorView. The error message will be assigned to a field in the constructor, so when we need to display an error to the user we will instantiate an ErrorView.

```
public class ErrorView : ViewBase
{
    private Text errorMessageText;
    public ErrorView(Model.TemperatureControlDataModel model, string message)
        : base(model)
    {
        SetupUI();
        errorMessageText.TextContent = message;
    }
    …
```

Splash screen

Finally, I want to add a simple splash screen at startup that will display the program name and version number for a few seconds.

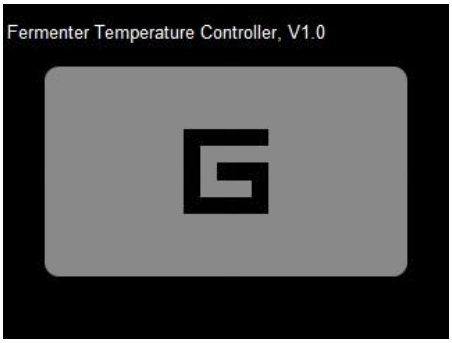

Figure 8: The Splash Screen View

Add new class in View directory and name it SplashView. This view will be very simple with just a `Text` component and an `Image`.

```
public class SplashView : ViewBase
{
   public SplashView(Model.TemperatureControlDataModel model)
      : base(model)
   {
      Image backgroundImage = new
         Image(Resources.GetBitmap(Resources.BitmapResources.StartScreen));
      this.AddChild(backgroundImage);

      AddTitleBar("Fermenter Temperature Controller, " +
         Resources.GetString(Resources.StringResources.appVersion),
         Resources.GetFont(Resources.FontResources.NinaB),
      GT.Color.White, GT.Color.Black);
   }
}
```

The Controller

The controller will control the overall system and will contain the system timers and associated tick event handlers. The controller will be responsible for switching between views as well as controlling the relays based on the temperature reading.

Create a new class in the Controller directory and name it TemperatureController and place it in the NETMFTemperatureController.Controller namespace.

```
public class TemperatureController
{
   private Window mainWindow;

   private Model.TemperatureControlDataModel theModel;
```

```
    private View.SettingsView settingsView;
    private View.DashboardView dashboardView;
    private View.SplashView splashView;
    private View.IdleView idleView;
    private View.ErrorView errorView;
    private DispatcherTimer startupTimer = new DispatcherTimer();
    private DispatcherTimer controllerTimer = new DispatcherTimer();
    private DispatcherTimer idleTimer = new DispatcherTimer();
…
```

The Controller will create the model as well as all of the views. This is done in the constructor as follows:

```
public TemperatureController(Window mainWindow)
{
    this.mainWindow = mainWindow;
    theModel = new Model.TemperatureControlDataModel();

    splashView = new View.SplashView(theModel);
    dashboardView = new View.DashboardView(theModel);
    idleView = new View.IdleView(theModel);
    settingsView = new View.SettingsView(theModel);
…
```

The basic operation of the controller will be to display the Splash view for 5 seconds, then switch to the `dashboardView`. If there has been no user input for at least 10 seconds the controller should automatically switch to `idleView`. Touching the screen while the `idleView` is displayed should cause the controller to return to the `dashboardView`. This can be accomplished in two steps. In the Controller class we create a `DispatcherTimer` named `idleTimer` with a delay of 10 seconds. Next we create an associated tick event handler that changes the current view to the Idle view when fired.

```
public void idleTimer_Tick(object sender, EventArgs e)
{
    if (!settingsView.IsVisible)
    {
        idleTimer.Stop();
        mainWindow.Background = new SolidColorBrush(GT.Color.Black);
        mainWindow.Child = idleView;
    }
}
```

Next, create an event handler for the `TouchDown` event of the main window. This handler will simply restart `IdleTimer` each time someone touches the screen.

```
mainWindow.TouchDown += new TouchEventHandler(mainWindow_TouchDown);

public void mainWindow_TouchDown(object sender, TouchEventArgs e)
{
    idleTimer.Stop();
    idleTimer.Start();
```

```
}
```

In order to change system parameters we need to be able to get to and from the Settings view. To do this we again create event handlers in the Controller class, this time for the button events:

```
public void settingsButton_Click(object sender, TouchEventArgs e)
{
   ChangeView(settingsView);
}

public void backButton_Click(object sender, TouchEventArgs e)
{
   ChangeView(dashboardView);
}
```

Next, the `TouchDown` events for the `settingsButton` in the Dashboard view and the `backButton` in the Settings view needs to be assigned their respective an event handlers. Each of the timer intervals will be also need to be set and their event handlers will need to be assigned. We will do all of this in a method called `InitializeController()`. This is also a good place to assign the window touch event handlers.

```
public void InitializeController()
{
   startupTimer.Interval = new TimeSpan(0, 0, theModel.StartupInterval);
   startupTimer.Tick += new EventHandler(startupTimer_Tick);

   controllerTimer.Interval = new TimeSpan(0, 0,
      theModel.ControllerInterval);
   controllerTimer.Tick += new EventHandler(controllerTimer_Tick);
   controllerTimer.Tick += dashboardView.timerDisplayUpdate_Tick;
   controllerTimer.Tick += idleView.timerDisplayUpdate_Tick;

   idleTimer.Interval = new TimeSpan(0, 0, theModel.IdleScreenTimeout);
   idleTimer.Tick += new EventHandler(idleTimer_Tick);

   dashboardView.settingsButton.TouchDown += settingsButton_Click;
   settingsView.backButton.TouchDown += backButton_Click;

   mainWindow.TouchDown += new TouchEventHandler(mainWindow_TouchDown);
   idleView.TouchDown += new TouchEventHandler(idleView_TouchDown);
}
```

The Controller will need to call for heating or cooling depending on the temperature as well as the currently selected mode. We also want to ensure that only one relay will be on at a time. We don't want to make the fridge and the heater fight it out. This will be done in the `SetControllerState()` method.

```
public void SetControllerState()
{
   if(((theModel.ControllerMode == Model.TemperatureControlMode.modeCool) ||
```

```
        ((theModel.ControllerMode == Model.TemperatureControlMode.modeHold) &&
        (theModel.CurrentState == Model.TemperatureControllerState.stateIdle)))
        &&(theModel.AverageTemperature >= theModel.TemperatureSetPointHigh))
    {
        theModel.CurrentState = Model.TemperatureControllerState.stateCooling;
    }
    else if (((theModel.ControllerMode ==
              Model.TemperatureControlMode.modeHeat) ||
        ((theModel.ControllerMode == Model.TemperatureControlMode.modeHold) &&
        (theModel.CurrentState == Model.TemperatureControllerState.stateIdle)))
        &&(theModel.AverageTemperature <= theModel.TemperatureSetPointLow))
    {
        theModel.CurrentState = Model.TemperatureControllerState.stateHeating;
    }

    else if (((((theModel.ControllerMode ==
              Model.TemperatureControlMode.modeCool) ||
        ((theModel.ControllerMode == Model.TemperatureControlMode.modeHold) &&
        (theModel.CurrentState ==
           Model.TemperatureControllerState.stateCooling))) &&
        (theModel.AverageTemperature <= theModel.TemperatureSetPoint)) ||
        (((theModel.ControllerMode == Model.TemperatureControlMode.modeHeat) ||
        ((theModel.ControllerMode == Model.TemperatureControlMode.modeHold) &&
        (theModel.CurrentState ==
           Model.TemperatureControllerState.stateHeating))) &&
        (theModel.AverageTemperature >= theModel.TemperatureSetPoint))) ||
        (theModel.ControllerMode == Model.TemperatureControlMode.modeOff))
    {
        theModel.CurrentState = Model.TemperatureControllerState.stateIdle;
    }

    switch (theModel.CurrentState)
    {
        case Model.TemperatureControllerState.stateIdle:
            CallForCooling(false);
            CallForHeating(false);
            break;
        case Model.TemperatureControllerState.stateHeating:
            CallForCooling(false);
            CallForHeating(true);
            break;
        case Model.TemperatureControllerState.stateCooling:
            CallForCooling(true);
            CallForHeating(false);
            break;
        default: //Don't allow both to be on at same time.
            throw new NotSupportedException("Invalid TemperatureControlState");
    }
}
```

The `CallForCooling()` and `CallForHeating()` methods will eventually be used to turn our relays on and off. For now we will just create then as empty methods.

In order to tie everything together into a working program we need to create a main window and the controller as soon as the program starts, and then call the controler's `Start()` method. We do this inside the `Program::ProgramStarted()` method.

```
public partial class Program
{
   public Window mainWindow;

// This method is run when the mainboard is powered up or reset.
   void ProgramStarted()
   {
// Setup Main Window
      mainWindow = display_TE35.WPFWindow;
      mainWindow.Background = new SolidColorBrush(GT.Color.White);

// Create Controller
   Controller.TemperatureController controller = new
      Controller.TemperatureController(mainWindow);

// Start the Dispatcher
   controller.Start();
   }
}
```

At this point the software can be compiled and deployed to the mainboard. The UI is fully functional and the system will display the views and will respond to touchscreen inputs approprately. We can even change the temperature set point and system mode, although the temperature displayed will never change.

Hardware

Now that we have some basic software written and working let's get into designing the custom hardware modules.

Power Supply

For this project I will be using the USB Client DP Module from GHI to power the Spider main board. This module will allow us to power the main board from a USB host or from an external DC power supply. Since this controller will be switching power to a refrigerator it needs to be plugged into a wall outlet in-line with a refrigerator. I really don't want to have a separate wall-wart type power supply in order to power the microcontroller, so we need some sort of internal power conditioning to provide the necessary DC power from the available 120 VAC. According to GHI, the DP module will function from 7 V to 30 V, but they recommend using a power supply capable of delivering 12V at 1A or more. I envision the custom hardware being built on a circuit board, so I would like to find a PCB mount power supply. I had initially decided to go with a 9 VDC power supply module. A quick search through Mouser, Digikey, and Newark for PCB mount AC-DC power supply modules yielded a plethora of choices. Some are encapsulated and some are open frame. There are many choices of output voltages and power ratings as well. A 9 Volt 10 watt module seems like a good choice at first, but it is a non-stocked item. So I

would have to wait 8-10 weeks for one. Many non-stocked items also have a minimum quantity requirement, so I might have to order 200 or more of them. This is not something I want to do. Since the input voltage of the DP module is pretty flexible I limited my search to normally stocked items that can be ordered in individual quantities. One of the most common voltages of normally stocked power supplies is 12 V, which happens to be what GHI recommends, so that is what we will go with. Of all of the available power supplies, I chose the CUI Inc. VSK-S10-12UA which is a 12 Volt, 10 watt encapsulated power module that has over current and short circuit protection built in. Since we are only going to be running the mainboard, a small screen, a low power temperature probe, and a couple of relays, a 10 Watt power module is more than sufficient.

Relays

As we will be controlling power to a refrigerator we will need a relay that can switch mains power at 120 VAC. Most residential refrigerators typically draw about 5 Amps, but the inrush current can be higher. Also, if we are to control some type of heater, the current draw could be much higher. For safety reasons I prefer to use a relay that can handle the full current of a typical household circuit, which is 15 Amps. There are a wide variety of relays types available; both mechanical and solid state. Solid state relays have several nice features such as logic level control and zero-crossing operation; however, they are consequently more expensive than their mechanical brethren. In order to keep costs under control for this project we will stick with traditional mechanical relays.

The mainboard we have selected has 3.3 volt digital I/O pins. Searching through the vendor catalogs we find that there are very few relays available with 3.3V coils that can switch 120 VAC. Most of these have contacts that are rate at 10 A or less, and most are non-stocked items. Given that the available choices for low coil voltage relays is rather limited, we need to look at relays with higher voltage coils and use a separate driver circuit to control the relay.

Continuing our search, we find that relays with 12 V coils are quite common and many are standard stocked items, and relatively inexpensive as well. This is an ideal choice since we have chosen to use a 12 V power supply. Typically a relay is chosen that operates from the available supply voltage and that has contacts that are capable of switching the expected current to the load that you want to control. A driver circuit is then designed that can control the current to the relay's coils. After looking at many different options and form factors I have chosen the Omron G5CA series relay with quick connect terminals. This relay has contacts rated to 15A so it is ideal for switching power in household appliances. It also has a low profile so it should fit nicely in whatever enclosure we decide to use.

Relay Driver Circuit - Transistor control

As the relays we have chosen require 12 VDC to operate, we can't directly switch them on and off with our 3.3 V microcontroller. In this case a driver circuit is needed to be able to control the relays. This can be done with little more than a simple transistor. Transistors are often used as small signal amplifiers, but they can also be used as a switch by driving them into saturation. By applying a small current to the base of a transistor it can be used to control the current through the relay's coil, which will actuate the contacts to switch power on and off to the refrigerator.

I have chosen to use a simple NPN bipolar junction transistor (BJT) for this project as I happened to have many 2N3904 transistors on hand, however, a field effect transistor (FET) could also be used. In order to use the BJT to control the relay we will use the so-called common emitter configuration as shown in the following diagram.

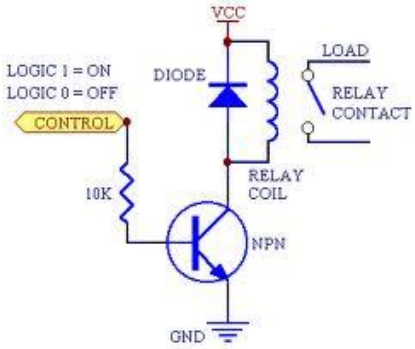

Figure 9: Relay Driver

We will place the transistor with the collector connected to one side of the relay coil and the emitter connected to ground. The other side of the relay coil will be connected directly to the power supply. The transistor's base will be connected to our microcontroller through a suitable resistor. The value of the resistor is chosen to set the amount of current into the transistor's base. This allows us to drive the transistor into saturation (fully turned on) when our microcontroller's digital output pin goes high.
To find the value of the resistor we first look at the specifications for the relay to determine how much pull-in and hold current that it requires. According to the data sheet, the 12V relay that we chose has a coil resistance of 720 ohms and draws 16.7 mA. We need to provide enough base current to make sure the transistor remains in saturation at this current level. To do this, the base current should be greater than the collector current divided by the transistor's DC current gain, also known as Beta or h_{FE}.

$$base\ current \geq \frac{I_c}{h_{FE}}$$

According to the data sheets ,the 2N3904 has a DC current gain of 100 at a collector current of 10 mA, and a base-emitter saturation voltage of 0.65V. So we need to provide a base current of at least

$$base\ current \geq \frac{16.7\ mA}{100} = 0.17mA$$

To ensure full saturation we'll round that up to 0.25 mA. As our microcontroller operates at 3.3V we need a resistor value of approximately

$$R_b = \frac{V_{in} - V_{be}}{I_b}$$

$$= \frac{3.3\ V - 0.65\ V}{0.25\ mA}$$

$$= 10.6\ k\Omega$$

This is of course a non-standard resistor value, so we'll use a standard 10 kΩ, 5% tolerance resistor instead. Depending on the actual value of resistance, this will provide a base current of between 0.25 mA and 0.28 mA.

Protection diode

The final component to our relay driver circuit is a flyback protection diode connected antiparallel across the relay coil. This diode protects the transistor by shunting current from the back-EMF pulse that is generated by the relay's coil when the current is turned off. An inexpensive power diode, such as a 1N4001 will work fine. Without this diode, back-EMF voltages can reach -300 volts which can easily damage the transistor.

Temperature Sensing

Since the GHI thermocouple module didn't pan out, we will design our own custom temperature measurement module. A popular temperature sensor is the Dallas/Maxim DS18B20. The DS18B20 is a 9 to 12 bit programmable resolution, 1-Wire® digital thermometer. This device is accurate to +/- 1°C, and at 12 bits you can get 0.0625 °C resolution. These devices are available as individual components, or packaged as waterproof remote temperature probes. For this project I have selected the DS18B20 6" Stainless Steel Waterproof Probe from brewinghardware.com. The main drawback to this sensor is the 1-Wire serial bus interface. The 1-Wire bus uses a single wire for power and data transfer, so serial communications are somewhat complex and require very tight timing control. In fact a standard 1-wire time slot is 60 µS wide, while an overdrive timeslot is only 8 µS. With a non-deterministic system like NETMF, it is nearly impossible to consistently satisfy the timing requirements of a 1-Wire bus by "bit-banging" in software. Luckily there is another way: protocol conversion. The FEZ Spider mainboard has an I^2C bus interface so it is possible, and relatively easy, to use a Dallas/Maxim DS2482-100 I^2C-to-1-Wire Bridge to interface between the microprocessor and the temperature probe. It is quite simple to communicate with the DS2482 using the I^2C bus and the DS2482 internally generates the time-critical 1-Wire waveforms necessary to communicate with the DS18B20. The main thing to remember is that all communications with the thermometer are **through** the DS2482 and can only be accomplished with commands **to** the DS2482.

The I^2C bus uses a data line (SDA) and a clock signal (SCL) for communication. Both of these lines are open collector, which means they require a connection to positive supply voltage through a pullup resistor. According to the .NET Gadgeteer Module Builders Guide, all Gadgeteer mainboards should provide pullup resistors on the SDA and SCL lines, so we shouldn't need to provide them. Optionally, we could provide 1 kΩ pullup resistors connected to removable jumpers in case a different microcontroller is used that doesn't provide its own. I^2C is a serial bus protocol which can support multiple simultaneously connected slave devices. Individual device addresses are used in order to communicate with any particular device. Each I^2C slave device has a 7 bit address, however, 8 bits are sent over the bus. The 8^{th} bit tells the device if the operation is a read or a write. The DS2482-100 only provides two pins for address selection; the most significant 5 address bits are fixed as '00110'. If we tie both address

lines to ground then the chip's address will be '0011000', or 0x18. The I²C interface uses Gadgeteer socket type I which, according to the module builders guide, uses pin 8 for data and pin 9 for the clock. The DS18B20 probe that we are using supports a dedicated power line so we can avoid the complications of issuing a Strong Pull-up command that is required when using 'parasite power'. Also since we only have one device on the 1-Wire bus, we don't need to worry about probe's identification. We can simply issue the "Skip ROM" command prior to issuing any function commands.

Schematic

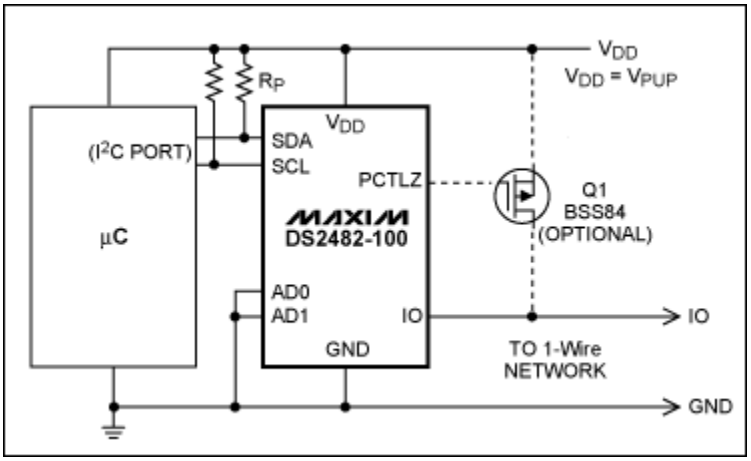

Figure 10: DS2482 Schematic

Enclosure

Now that we have selected our hardware and know the sizes of all of the components we can estimate a required size for the enclosure. After quickly laying the parts out on the table and playing with arrangement, I estimate that an enclosure sized about 5"x6"x2" should be sufficient to put all of our hardware into. There are an almost unlimited selection of plastic enclosures available but I finally settled on a black Serpac model 172. I think it will give us a nice looking final product and we can easily add a couple panel mount AC plugs to the front to plug our refrigerator and heater into.

Putting it all together

The next step is to cut holes in the lid for the touchscreen and the panel mount AC plugs. For the prototype I will build the circuitry on a generic Radio Shack perf board. We will cut the board to fit inside the enclosure, then lay out the parts ensuring that there are no clearance issues from the touchscreen or plugs mounted in the lid. Once everything is laid out, the circuit can be soldered together with hookup wire.

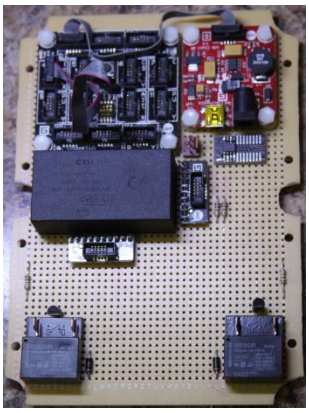

Figure 11: Populated Perf Board

Custom Module Drivers

For our custom relay and temperature probe modules we will reference “Building a .NET Gadgeteer Compatible Hardware and Software Module” by Pete Brown, as well as the .NET Gadgeteer Module Builder Guide.

Project Setup

First, create a new project using the .Net Gadgeteer Kit template and name it MyGadgeteerModules as shown here.

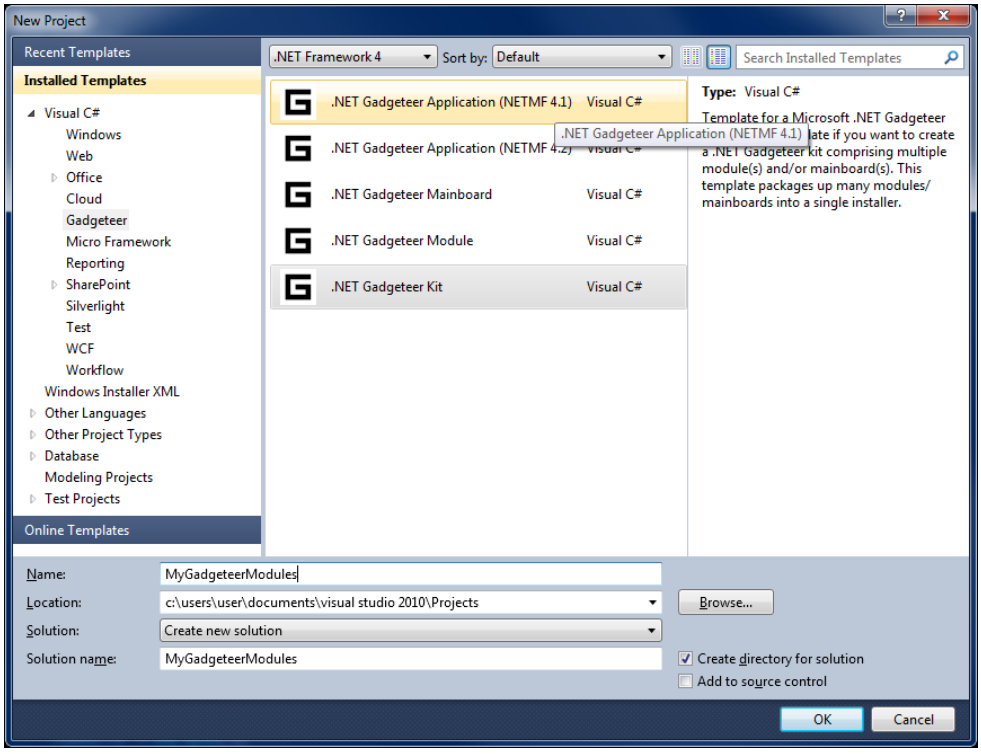

Figure 12: Kit Project

Next, add two separate projects to the solution, one for each module we will create. To do this, right click on the Solution item in the solution explorer and select Add->New Project and select the .NET Gadgeteer Module template. Name one project ControlRelays and the other TemperatureProbe

A module project creator window will appear. Deselect the NETMF 4.1 and 4.3 versions

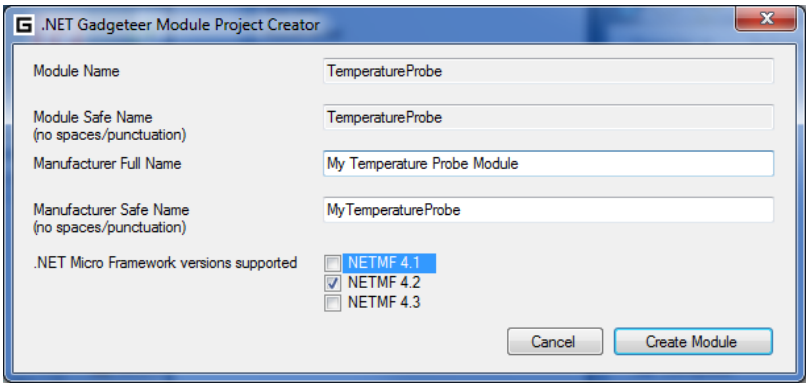

Figure 13: Project Creator

When you select Create Module, an error will appear. This is just informing you that you will not be supporting version 4.3 of the NETMF. Dismiss the error window.

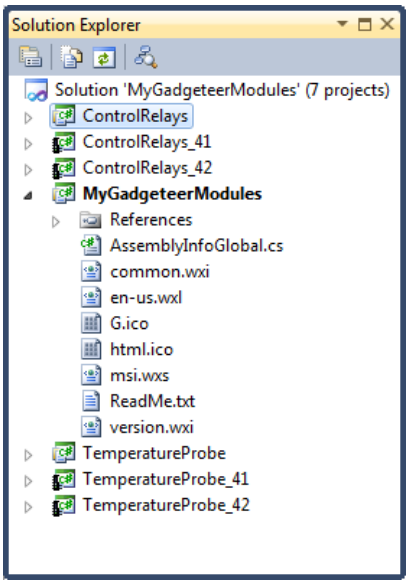

Figure 14: Module Kit Solution Explorer

Relay Module

In order to control the relays, we will use general purpose digital I/O (GPIO) pins. An examination of the socket type table in the module builders guide indicates that socket types X and Y are ideal for our use. Both of these sockets types have GPIO on pins 3 and 4.

First, create storage for a socket and two digital output pins.

```
public class ControlRelays : GTM.Module
{
```

```
    #region Fields
    private Socket socket;
    private GTI.DigitalOutput relay1;
    private GTI.DigitalOutput relay2;
…
```

Next, create properties for each relay that will set and record the relay's state.

```
private bool relay1State = false;
public bool Relay1
{
    get { return relay1State; }
    set
    {
        relay1State = value;
        relay1.Write(relay1State);
    }
}

private bool relay2State = false;
public bool Relay2
{
    get { return relay2State; }
    set
    {
        relay2State = value;
        relay2.Write(relay2State);
    }
}
```

The module's constructor will be used to instantiate the socket and GPIO pins.

```
public ControlRelays(int socketNumber)
{
    socket = Socket.GetSocket(socketNumber, true, this, "XY");
    relay1 = new GTI.DigitalOutput(socket, Socket.Pin.Three, false, this);
    relay2 = new GTI.DigitalOutput(socket, Socket.Pin.Four, false, this);
}
```

Temperature Probe Module

Our temperature probe module will use the I^2C interface, which is socket type I. As with the relays, the socket will be instantiated in the constructor.

```
public class TemperatureProbe : GTM.Module
{
    public TemperatureProbe(int socketNumber)
    {
        socket = Socket.GetSocket(socketNumber, true, this, "I");
    }
…
```

Before we can begin taking temperature measurements we must initialize the temperature probe. This involves resetting the DS2482 interface, checking the DS18B20's power supply state, and setting the temperature resolution.

```
/// <summary>
/// User selectable resolution of the DS18B20 ADC.
/// Bits 5 and 6 of the config. register are used to set the resolution.
/// Bits 0-4 and 7 are reserved. Config register is as follows:
/// Bit:    7  6  5  4  3  2  1  0
/// Value:  0  R1 R0 1  1  1  1  1
/// </summary>
public enum ADCResolution
{
   NineBit = 0x1F,
   TenBit = 0x3F,
   ElevenBit = 0x5F,
   TwelveBit = 0x7F
}


public void Initialize(ushort bridgeAddress, int clockFrequency,
   ADCResolution adcResolution, Module probe)
{
   this.bridgeAddress = bridgeAddress;
   this.adcResolution = adcResolution;

   i2cBus = new I2CBus(socket, bridgeAddress, clockFrequency, probe);

   switch (adcResolution)
   {
      case ADCResolution.NineBit:
         temperatureResolution = 8 * minTempResolution;
         convertionTime = maxConvertionTime / 8;
         break;
      case ADCResolution.TenBit:
         temperatureResolution = 4 * minTempResolution;
         convertionTime = maxConvertionTime / 4;
         break;
      case ADCResolution.ElevenBit:
         temperatureResolution = 2 * minTempResolution;
         convertionTime = maxConvertionTime / 2;
         break;
      case ADCResolution.TwelveBit:
         temperatureResolution = minTempResolution;
         convertionTime = maxConvertionTime;
         break;
      default:
         break;
   }

   try
   {
      ResetDS2482();
      ReadDS18B20ROMCode();
      ReadDS18B20PowerSupply();
      ConfigureDS18B20();
```

```
    }
    catch (Exception)
    {
        throw;
    }
}
```

Configuring the DS18B20 involves writing configuration data into the device's scratch pad, then copying that data into its EEPROM.

```
private void ConfigureDS18B20()
{
    Reset1WireBus();
    Write1WireByte((byte)DS18B20ROMCommands.SkipROM);
    Write1WireByte((byte)DS18B20FunctionCommands.WriteScratchPad);
    Write1WireByte((byte)0x00);  // zero Th
    Write1WireByte((byte)0x00);  // zero Tl
    Write1WireByte((byte)adcResolution);

    Reset1WireBus();
    Write1WireByte((byte)DS18B20ROMCommands.SkipROM);
    if (usingParasitePower)
    {
        IssueStrongPullup();  // Issue Strong Pullup
    }

    Write1WireByte((byte)DS18B20FunctionCommands.CopyScratchPad);
    if (usingParasitePower)
    {
        Thread.Sleep(10); // Delay for 10 mS while copy operation in progress
    }
}
```

The response time of the DS18B20 temperature probe depends on the temperature resolution chosen. As such I would like the module to fire an event when a temperature measurement is ready. Our module will only begin performing a temperature measurement and conversion when a request is received.
First, we will need a delegate for the event handler.

```
public delegate void MeasurementCompleteEventHandler(TemperatureProbe sender,
float temperatureCelsius);
```

Next, declare the event using the delegate as the type.

```
public event MeasurementCompleteEventHandler MeasurementComplete;
```

Next, we wrap the event invocation inside a protected virtual method.

```
protected virtual void OnMeasurementComplete()
{
```

```
  MeasurementCompleteEventHandler handler = MeasurementComplete;
  if (handler != null)
    {
     handler(this, temperatureCelsius);
    }
}
```

Temperature measurements are retrieved from the temperature probe by reading the probe's 9-byte scratch pad. The actual temperature is contained in bytes 0 and 1 as a 16 bit sign-extended two's complement number. The 4 least significant bits represent the fractional value. For 12-bit resolution all bits in the register contain valid data. For 11-bit resolution, bit 0 is undefined. Similarly for 10-bit resolution, bits 1 and 0 are undefined.

	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
LS BYTE	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
	BIT 15	**BIT 14**	**BIT 13**	**BIT 12**	**BIT 11**	**BIT 10**	**BIT 9**	**BIT 8**
MS BYTE	S	S	S	S	S	2^6	2^5	2^4

S = SIGN

Figure 15: Temperature Register

To be useful outside of this module, the value in the temperature register must be converted into a floating point value as follows:

```
private float GetTemperatureFromBytes(byte[] temperatureBytes)
{
   float fractionalPart = 0;
   short wholePart = 0;
   short temperatureSignMultiplier = 1;
   float temperatureReading;

   uint uintValue =
      Microsoft.SPOT.Hardware.Utility.ExtractValueFromArray(temperatureBytes,
      0, 2);

   if ((uintValue & 0x8000) == 0x8000)     // is negative
   {
      uintValue = ~uintValue + 1;          // 2's complement
      temperatureSignMultiplier = -1;
   }

   fractionalPart = (float)(((uintValue & 0x0000000F) >> (3-
      (int)this.adcResolution >> 5))) * temperatureResolution);
   wholePart = (short)(uintValue >> 4);
   temperatureReading = (wholePart + fractionalPart) *
      temperatureSignMultiplier;

   return temperatureReading;
}
```

The DS18B20 provides temperature readings in degrees Celsius, so the values will have to be converted into Fahrenheit if the user selects the Fahrenheit display mode.

Now we need a method to respond to requests for temperature measurements. This will initiate a temperature conversion and fire the `MeasurementComplete()` event when finished.

```
public void RequestMeasurement()
{
   try
   {
      Reset1WireBus();
      Write1WireByte((byte)DS18B20ROMCommands.SkipROM);

   // Convert T command to DS18B20
      if (usingParasitePower)
      {
         IssueStrongPullup();  // Issue Strong Pullup
      }
      Write1WireByte((byte)DS18B20FunctionCommands.ConvertT);

      Thread.Sleep((int)convertionTime);

      do
      {
         ReadDS18B20ScratchPad();
      } //if CRC matches, continue else read again
      while ((byte)CRCCalculator.CalculateChecksum(ds18B20ScratchPad, 0, 8)
         != ds18B20ScratchPad[8]);

      temperatureInBytes[0] =
         ds18B20ScratchPad[(int)DS18B20ScratchPadLocation.TempLSB];
      temperatureInBytes[1] =
         ds18B20ScratchPad[(int)DS18B20ScratchPadLocation.TempMSB];
// Decode Temperature
      temperatureCelsius = GetTemperatureFromBytes(temperatureInBytes);

// Fire MeasurementCompleted event
      OnMeasurementComplete();
   }
   catch (Exception)
   {
      throw;
   }
}
```

We now need to modify several WIX project files with socket information as well as module and kit version numbers. These modifications are fully explained in the Module Builders Guide as well as the Readme.txt files in the project and module folders. Once that is finished build the solution. The kit project's `\bin\Installer` folder should now contain an MSI installer file. Run this MSI to install the module drivers. You can verify installation by checking the programs list in the control panel. Our

custom modules are now available so we can add them to the main project and connect them to the mainboard using the Program.gadgeteer designer interface.

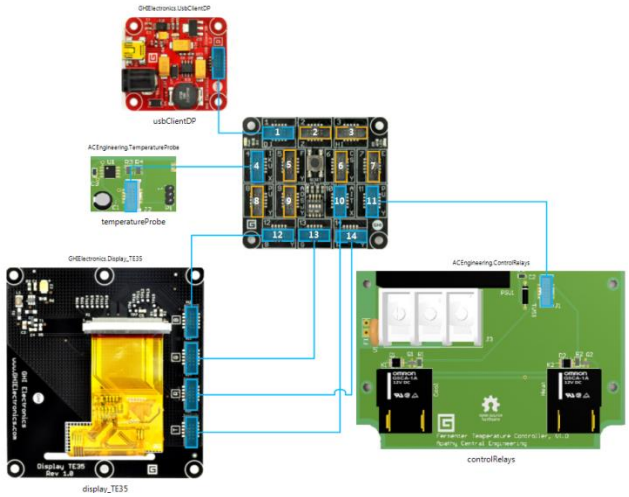

Figure 16: Gadgeteer Designer Interface

The Controller, Part Deux.

In order to make use of our new modules we must revisit the controller. The constructor will need to be revised to store references to the relay and temperature probe modules.

```
public TemperatureController(Window mainWindow, ControlRelays controlRelays,
    TemperatureProbe temperatureProbe)
{
…
    this.controlRelays = controlRelays;
    this.temperatureProbe = temperatureProbe;
}
```

The `InitializeController()` method will be modified to assign an event handler to handle the temperature probe's `MeasurementComplete()` event.

```
public void InitializeController()
{
    …
    temperatureProbe.MeasurementComplete += new
        TemperatureProbe.MeasurementCompleteEventHandler(temperatureProbe_
        MeasurementComplete);
}
```

The `MeasurementComplete()` event handler will simply pass the temperature measurement value to the model by calling the `RecordTemperatureSample()` method.

```
private void temperatureProbe_MeasurementComplete(TemperatureProbe sender,
   float temperatureInCelsius)
{
   theModel.RecordTemperatureSample(temperatureInCelsius);
}
```

The temperature probe module will then be initialized with the interface chip's address and the desired temperature resolution in the controller's `Start()` method.

```
public void Start()
{
   ChangeView(splashView);
   startupTimer.Start();
   InitializeController();

   try
   {
      temperatureProbe.Initialize(ds2482Address, i2CBusSpeed,
         ADCResolution.ElevenBit, temperatureProbe);
   }
   catch (Exception e)
   {
      SystemError(e.Message);
   }
}
```

To make everything run we need to request temperature samples and set the relays on a regular basis. This will be done in the tick event handler for the `controllerTimer`.

```
public void controllerTimer_Tick(object sender, EventArgs e)
{
   try
   {
      temperatureProbe.RequestMeasurement();
      SetControllerState();
   }
   catch (Exception exeption)
   {
      SystemError(exeption.Message);
   }
}
```

Finally we need to be able to turn the relays on and off. We do this inside our `CallForHeating()` and `CallForCooling()` methods:

```
private void CallForCooling(bool coolState)
{
   controlRelays.Relay1 = coolState;
}

private void CallForHeating(bool heatState)
{
```

```
    controlRelays.Relay2 = heatState;
}
```

System Errors

In the event of a system error, such as communication errors with the temperature probe, we want the controller to behave in a safe manner so we should turn off both relays and display an error message to the user.

```
/// <summary>
/// In the event of a system error, stop all timers,
/// put the controller in a safe state (i.e. turn off relays),
/// and display an error message.
/// </summary>
/// <param name="message"></param>
private void SystemError(string message)
{
    controllerTimer.Stop();
    idleTimer.Stop();
    startupTimer.Stop();

    CallForCooling(false);
    CallForHeating(false);

    errorView = new View.ErrorView(theModel, message);
    ChangeView(errorView);
}
```

We now have a fully functional temperature controller that we can use for maintaining proper temperature during fermentation! For a finishing touch we could epoxy a couple of hard drive magnets to the back of the case so that we can mount the controller to the refrigerator.

Figure 17: Final Product

Custom PWB

Now that our software and hardware designs are complete and functional, the next step towards a professional looking product is to design a custom circuit board. This can be done using a circuit board design program like Altium Designer or Eagle PCB.

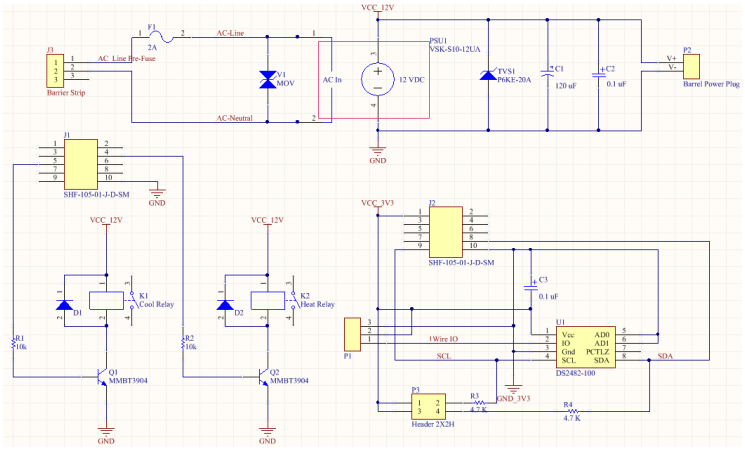

Figure 18: Circuit Schematic

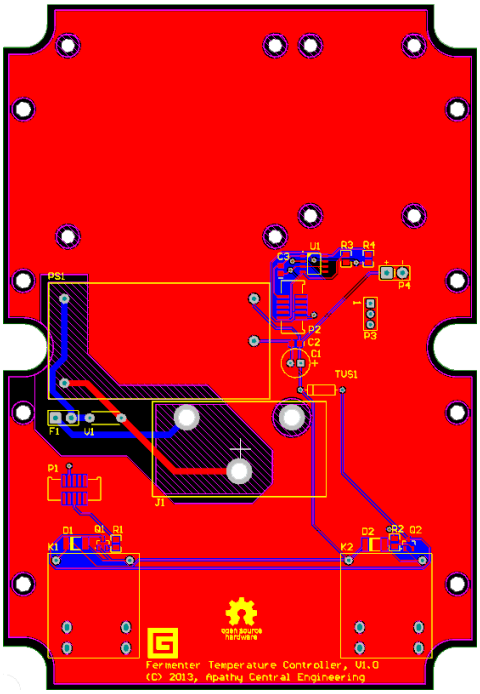

Figure 19: Circuit Board Layout

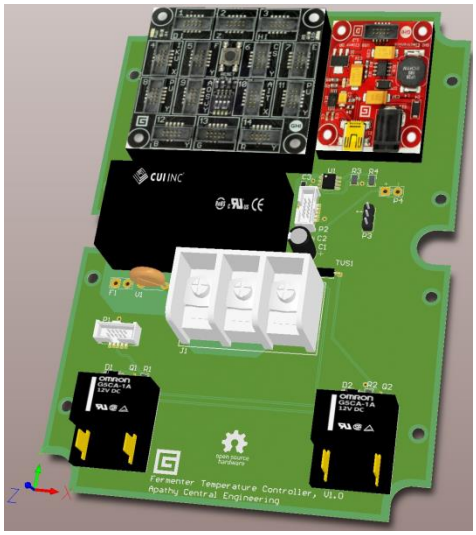

Figure 20: 3D Model of Circuit Board Design

There are many companies that will fabricate prototype or one-off PCBs for a reasonable price. Most simply require uploading the Gerber manufacturing files that are output by the PCB design programs. All of the source code for the software projects as well as the Altera circuit board design project files are available on Github at http://jlhaynes.github.io/TemperatureController/