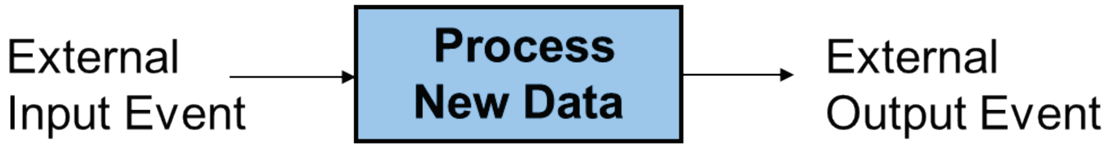
ECE 2036 Lab #4 Build an mbed thermostat

Check-Off Deadline: **Section A – Tuesday, March 10th**
Section B – Thursday, March 12th

1. Real-Time Systems

Many embedded devices are also real-time systems. A real-time system needs to read in data from external sensors, perform computations based on new sensor data, and update the control outputs within a fixed period of time.



A Real-Time System reads in sensor data, processes the data, and updates control outputs

Examples in cars include airbags, antilock brakes, and engine controls. Autopilots in airplanes and spacecraft are another example. In many of these systems, responding too slowly will cause system failure. Most real-time systems need a response time on the order of milliseconds or even microseconds.

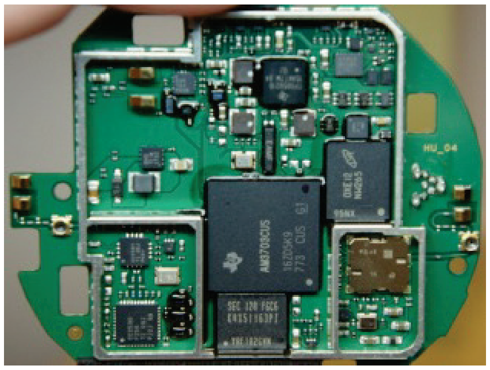
Frequently, the control system software found in an embedded device's firmware requires a complex state machine coded in C or C++ that has additional timing constraints. Most processors have hardware timers or perhaps even a real-time clock that software can use to provide accurate time delays. Some systems may also require more complex control system calculations such as a proportional-integral-derivative controller or PID controller.

2. The New Nest Thermostat

Programmable home heating and cooling thermostats are also embedded devices. Thermostats could perhaps also be considered a real-time system, but with a response in range of minutes that is several orders of magnitude slower than most applications. The new Nest thermostat seen on the next page was designed by an engineer that previously designed Apple's iPods. It uses a color LCD similar to those in cellphones and a rotating ring for user input. Menus are used to change settings and display data. This device is now available at your neighborhood Lowes, Home Depot, the Apple Store, and Amazon. Nest Labs is one of Silicon Valley's newest startup success stories. Google just bought Nest over a year ago for $3.2B. It is an interesting example of how even common household devices are currently being re-engineered using new technology. Honeywell now has a similar thermostat, but the Nest came out first.

The Nest IoT Thermostat

As seen in the printed circuit board (PCB) image below, the nest contains an ARM Cortex A8 32-bit processor, temperature, light, motion, and humidity sensors and Wi Fi. The motion sensor senses when no one is home to save energy. A learning algorithm monitors your inputs to setup automatic scheduling. Wi Fi can be used for remote control via the web and for automatic firmware updates. There is even a new smart smoke alarm that networks with Nest and other smoke alarms called Nest Protect. Devices like Nest, using networking for monitoring and control are part of the new “Internet of Things” or “IoT”. Even IoT light bulbs and switches such as the WeMo are now available. Small kitchen appliances such as crock pots and coffee pots are starting to appear with IoT features.



Hardware found inside the Nest

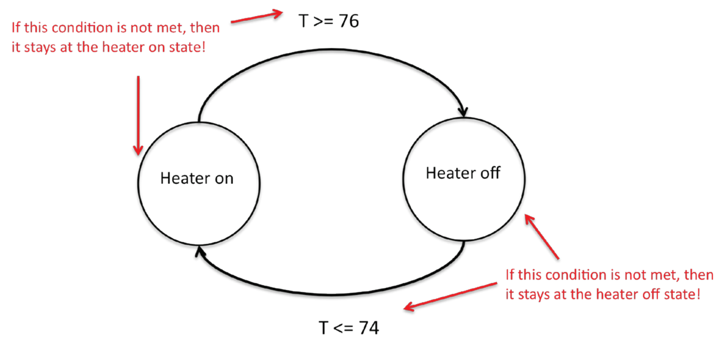
Most home heating systems use 24V AC relays to turn the fan, heater, and cooling on and off. A driver circuit (http://mbed.org/users/4180_1/notebook/relays1/) converts a digital logic output signal from the processor to the higher voltage and current levels required to trigger the relays.

II. Building a Thermostat Using Mbed

For this assignment, we will use the ARM processor on the mbed module and develop C/C++ code to build a basic thermostat. The IoT networking features will not be included to reduce the scope of the assignment, but the mbed can support them. Instead of hooking up a driver circuit for the heating system’s 24V relays, the control outputs will just be displayed using the mbed’s built-in LEDs. Mbed’s I/O devices all have been setup with easy to use C++ Application Programming Interface (API) calls that are described in the mbed handbook (http://mbed.org/handbook/Homepage). The API code provided with mbed uses C++ classes and methods to make them easier to use. These are similar to a simple device driver and it is not necessary to understand the hardware details of each I/O device to develop complex C/C++ applications on mbed. Follow the hyperlinks provided in this document for additional documentation and C/C++ code examples.

A TMP36 analog temperature sensor (http://mbed.org/users/4180_1/notebook/lm61-analog-temperature-sensor/) will be used to read the temperature. Using mbed's built-in analog to digital convertor and a C/C++ I/O Application Programming Interface (API) call, the voltage from the sensor will be converted from analog to a floating point value returned in a C/C++ variable. This temperature value is then checked to determine when to turn on and off the heating and cooling systems. Two of mbed's built in LEDs will be used to indicate the value of the output signals that control the heating (LED4) and cooling (LED3). It will not be attached to an actual HVAC system or require a driver circuit to control the relays.

Thermostats and control systems must use hysteresis to avoid rapidly switching on and off. Mechanical systems take time to respond and wear out faster if they are constantly turned on and off. They need to run for a longer period of time to avoid excessive mechanical wear. They are also more energy efficient with longer run times. In the case of a thermostat for a heater, with hysteresis the heater would turn on at the temperature control point, but then not turn off until the temperature has increased a couple degrees. It would not turn back on until the temperature drops a couple degrees. This requires state information (i.e. heater is on or off) and then checking a ***different*** temperature threshold when turning the device on versus turning it off. The figure below shows a simple state transition diagram for a thermostat set at 75^{o}. The plus and minus 1 degree provide the hysteresis to give an average temperature of 75^{o}. It should start in the state where the heater is off.


If this condition is not met, then it stays at the heater on state!
T >= 76
Heater on
Heater off
If this condition is not met, then it stays at the heater off state!
T <= 74

State Diagram for a Basic Thermostat set to 75

1. Modeling a State Machine in C/C++

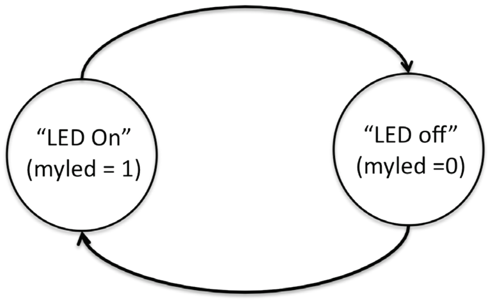
Nested control structures are required to implement a complex state machine in software. As an example, a simple state machine is shown below in C/C++ that blinks an LED. When the program runs, LED1 changes every .33 seconds.

```
include "mbed.h"
DigitalOut myled(LED1);

enum Statetype { LED_off = 0, LED_on };

int main()
{
    Statetype state = LED_off;
    while(1)
    {
        switch (state)
        {
            case LED_off:
                // calculate the Moore-Type Outputs
                myled = 0;
                // calculate the next state
                state = LED_on;
                break;
            case LED_on:
                // calculate the Moore-Type Outputs
                myled = 1;
                // calculate the next state
                state = LED_off;
                break;
        }
        wait(0.33);
    }
}
```

A simple state machine in C/C++ that blinks an LED



A simple state diagram for state machine in above C code

An infinite while loop containing a *switch()* statement helps structure the code. The *switch()* statement selects one of several cases based on the current state. Each case (i.e., state) then checks the appropriate conditions to assign the next state. An enumerated type is used for state. ***You must use this***

overall structure in your code for the thermostat's state machine. It will be more structured, readable, and easier to understand. For just two unconditional states, an *if..else* statement could be used, but with a case statement adding more states and input conditions will be easier as the state machine gets more complex (and it will in this lab). A default case could be added to jump to the initial state just in case it ever entered an undefined state. *Wait()* actually slows down the state machine clock to around three clocks per second. Without the *wait()* the LED would change so fast it would only look a bit dimmer. **Note that this simple state machine does not check inputs. For a more complex state machine with inputs, inside each case *if..else* statement(s) could be used to check inputs to determine the next state. Each transition arrow leaving a state corresponds to one if statement.**

2. Reading the Temperature Sensor

An TMP36 wiki page is provided that shows how to connect the TMP36 sensor and read the temperature using C/C++ on the mbed module. Be careful it looks just like the 2N3904 transistor in the kit, so check the tiny marks on the case's flat area for the part number. Start by wiring up the TMP36 sensor to mbed and getting the demo code at the wiki page to work as shown. If you did not use the TMP36 in the earlier mbed lab, use the compiler import code link (i.e., box after source code) on the TMP36 wiki page to copy over the complete project demo files for the TMP36. This will verify your hardware connections and sensor operation before trying your new code.

See the breadboard theory wiki, if you have not used a breadboard before and **be very careful whenever inserting or removing the mbed module**, as the pins can easily bend and break. If that happens, you will need a new mbed module. It is a good idea to put the mbed on your breadboard once and leave it there. A larger version of the TMP36 breadboard setup photograph is also available using the link under the image on the TMP36 wiki page. Device placement is often a critical first step in building a circuit on a breadboard and also in designing a printed circuit board (i.e. shorter wires with less wires crossing). Typically placing devices with a large number of connections close together simplifies everything.

Looking ahead at the hardware needed for this laboratory assignment and consulting the mbed pin layout (colored card in kit), the LCD needs a serial connection. Since the SD card (might be used in a later lab) and the older kit's Shiftbrite RGB LED need an SPI (Serial Peripheral Interface) interface, it probably makes sense to put the LCD on the right side of the mbed using serial pins on the right side of the mbed module. The SD card and Shiftbrite could then go on the left nearer the SPI pins. The TMP36 can use any AnalogIn pin, Pushbuttons can use any GPIO pin, and the speaker will need a PWM pin or the analog output pin, but these devices need only one pin and are not as critical. The RGB LED needs three PWM pins (lower right side of mbed). Pins are passed as arguments when the pin I/O functions are initially setup, so as long as the pin supports the hardware function needed they can be moved around with a one line change in your C/C++ code. Details on pin assignment suggestions can be found in the ***Skeleton code*** provided for use with this assignment. Instructions on downloading this code are provided later near the end of this document. Check the card with pin functions first before moving pins. ***Using a pin for a non-supported hardware function does not cause a compile error; it generates a run time error which flashes the LEDs once the code runs.*** Be sure to setup the breadboard bus power strips. There is only one open pin on mbed for gnd and 3.3V (Vout) and power will be needed for several devices.

Double check power pin connections before inserting the USB cable for power, ***if they are incorrect that is the easiest way to destroy an IC***. If the mbed's single LED goes out, there is a power short.

If you did NOT do the earlier prelab, you will need to follow the instructions provided in your mbed box to setup a user account for access to the C/C++ compiler and space to save source files. You have to logon to use the compiler, but not the wiki pages.

The temperature sensor outputs a voltage that indicates the current temperature. An analog-to-digital converter inside the processor converts the analog voltage input to a 12-bit integer value at a maximum sample rate of 200Khz. The mbed C++ API AnalogIN reads a 3.3V max analog input and scales it to a float value between 0 and 1.0 (i.e., a 1.0 would be 3.3V and 0 would be 0V). DigitalOut can be used to control the LEDs. Use LED4 to indicate the state of the heater (i.e., LED4 on means the heater is on).

The mbed I/O APIs use C/C++ classes to enable users to work at a higher level of abstraction without dealing with the low-level hardware details. The API is basically a set of useful C functions and classes, just like any additional functions or classes you may write yourself. As example, here is what happens automatically when you use the AnalogIn API C++ class to read an analog input pin:

1. The pin is configured for analog input on initial use of AnalogIn in the constructor. Most I/O pins can have one of several different functions that are controlled by pin configuration registers. I/O registers on RISC processors are in memory addresses in a special range reserved just for I/O devices . Pins are a critical resource on an IC, so most I/O pins are configurable on new ICs. A pin can cost as much as 100K transistors inside the chip.
2. Whenever the AnalogIn variable is read, the eight input analog multiplexor is set to select the correct analog channel to read the pin. This requires writing a value to another I/O control register along with a small delay for the analog voltage to stabilize.
3. The A/D convertor is then started by writing a start A/D conversion bit in the A/D control register.
4. The software then waits in a while loop for the A/D to report back that the conversion is complete. The hardware sets a bit in an A/D status register when this happens. The A/D can only read 200K samples per second. This is a lot slower than instruction execution times, so code needs to wait for the A/D conversion complete bit.
5. The AnalogIn value is then read in from the A/D data register, scaled from 0.0 to 1.0, and returned as the float value of the AnalogIn variable.

A lot of development time is saved by using AnalogIn, there is no need to find and understand all of these hardware details in the 800 page LPC1768 User's Manual to get the I/O device to work. This is why I/O device drivers are typically used in larger systems.

Since temperature changes slowly, it is engineering overkill to check the temperature as fast as possible in a tight while loop. In an actual system, running slower can save power or provide extra CPU time for other tasks. On mbed, a time delay can be added using *wait()*. Check the temperature sensor and update states ***no more than three times a second***.

With the temperature sensor connected and the program running, you can simulate the effect of the heater turned on by lightly pushing down on the sensor with your finger (being careful not to knock it loose from the protoboard). Your finger is typically a bit cooler than your core body temperature and it does not cover the entire sensor, but you should be able to warm the sensor up to the high 80s after a few seconds. So you can test the thermostat by setting the control point a bit higher than room temperature so that the heat comes on (LED3). Then by warming it up a couple degrees with your finger you should see the heat turn off after a couple seconds. Monitor the temperature using your LCD display that you got working in the earlier prelab while this is happening. It is also possible to cool down the sensor using an ice cube in a small plastic bag (be careful not to get water on the breadboard or sensor leads). Spraying the sensor with an aerosol can of “dust-off” cleaner can even be used to cool the sensor below freezing.

3. Skeleton Project Code to Get Started

Several steps are need to add all of the libraries and *.h files to a project and the required includes in main.cpp for all of the I/O devices that will be used. To save you time on the assignment, a sample project with all of these already added and the simple state machine code example that just blinks an LED has been provided for your use. The skeleton code project setup is available at:

http://mbed.org/users/4180_1/code/mythermostat/

It is not required to use the skeleton code, but it will likely save you some time finding all of the *.h files and adding them one at a time. To use the skeleton code, plug in your mbed, logon to mbed.org, open a new tab in the browser window and go to the URL above. On the web page that appears click the **Import this program** box (upper right). It will open a compiler window and import all of the files into your project area and setup a new copy of the project.

It also has some sample I/O code just to show that things are setup OK for all of the new I/O devices. It should compile without errors after import (a warning or two is OK), but if you run it you will get a **runtime error on SD file I/O** (i.e., all four mbed LEDs will flash) until you delete the SD card I/O example code or attach the SD card breakout and insert an SD card. You will not use the SD card for this lab, so you will not need this part of the code for this lab; however, you may use this in future labs. You will also see an error message on the PC if you have a terminal application program running hooked up to the virtual com port. Once that code is deleted, if you recompile and download the state machine will blink the LED.

You will still need to read the handout and wiki pages to figure out how to hookup the hardware and what needs to be done to finish the program to fill in the missing code in main.cpp. The code has pin assignments for the devices based on the earlier breadboard suggestions, but feel free to change them. For those with older kits, *Shiftbrite.h* does not have all of the class code filled in, you will need to edit that file and add it for the final part of the project. Newer kits will need to add code for the RGB LED. If you see some handy code on wiki pages or in the compiler and want to cut and paste it into your compiler source code, highlight it with the mouse, use “control C” to copy, move to the paste location, and use “control V” to paste. **The format button in the compiler is also helpful to clean up nested control structures and auto indent the source code.**

4. Laboratory Requirements

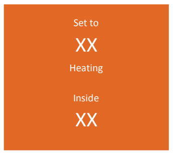
Basic Heater Thermostat (40%)

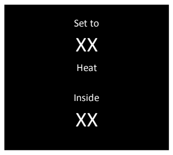
Implement the basic heating thermostat as described in Section 2. This should have two states, HEAT_OFF and HEAT_ON. Have the set-point hardcoded to 75. Your program must implement a Moore state machine. The state transitions are determined by the temperature sensor and set-point values. The fixed output in each state is the output of LED3, which simulates the heater being on or off. You may choose to implement only this "basic thermostat" for a maximum of 75% on the lab, or implement the "improved thermostat" for a maximum of 100%. The first step in implementing this lab should be to get your system working with the basic thermostat, LCD, and pushbuttons. The second step should be to add the speaker and RGB LED to your system. The third and final step should be to implement the "improved thermostat", which consists of modifying your state machine and adding a pushbutton callback. **Note: You will lose points if you do not implement a Moore-type state machine as described in Section II, and build all of the additional state machine code (i.e., new ifs…) inside the case statement with an enumerated state type.** Your main() function should consist of an infinite while loop that contains one switch statement, with one case for each state (2 states for basic thermostat, 5 states for improved thermostat).

Add LCD display (5%)

In the previous mbed lab assignment, you should have successfully gotten your display to work. Now you need to use the LCD to provide useful output to the user, by displaying temperature and thermostat status information. Display the current temperature (in two digits) and current mode. You must display all temperature values in Fahrenheit. Later some user input mechanism to toggle between F and C modes will be added (i.e., a fourth pushbutton). The LCD must also display the thermostat's current state and the current temperature set-point, depending on the current mode.

The LCD must be used to receive any credit for this lab. The 5% is to grade the quality of your LCD output. Since you only have around 16 characters of space on each LCD line to work with, you must carefully design your LCD output to display all necessary information. Your LCD output should be very similar to the following layout idea (it is based on the Nest and Honeywell thermostats):





(HEAT_ON state)   (HEAT_OFF state)

The LCD can be updated using the same rate as the state machine (i.e. one time at the end of the while loop, just before *wait()* is called). There are uLCD member function commands to change text height, width, and color. The command to change the background color needs a clear screen (i.e., cls) to take effect and characters will need to be printed again to change text color. Don't forget about the text background color and mode settings. The temperature display characters must be at least 2X larger than the other characters (even larger if it looks OK) and both character colors and background colors must be the same as those shown. Characters should be roughly centered on the display. Switching to the LCD's highest baud rate will produce faster screen updates.

Do not place any LCD commands, waits, or printfs in interrupt callback routines. This could lock up the LCD since it might be in the middle of an LCD command in main when the pushbutton interrupt occurs.

Add User Input (10%)

Use two pushbuttons to change the temperature set-point. One pushbutton should increase the set-point by one degree each time it is pressed. The other pushbutton should decrease the set-point by one degree each time it is pressed. If you did not implement pushbuttons in Assignment 2, read the pushbutton wiki page and watch the videos for additional help using pushbuttons with mbed.

You should use the PinDetect class to implement pushbuttons. The pushbutton callback option probably makes the most sense for this problem and the last example on the pushbutton wiki page shows two callbacks for two pushbuttons. The usage of the global variable count is similar to that of the set-point in your program. Interrupt and callback routines should be short, fast, and not use *printf()* or *wait()*. The main program can then read this global set-point variable (that is incremented or decremented by the callback routine) , and use the LCD functions *locate()* and *printf()* to display the modified set-point on the LCD. You can re-print the entire 2nd line by locating to (0,1) before calling *printf()*. A better approach that typically results in faster screen updates would be to locate to a specific column in line 2 and use *printf()* to re-print only the set-point, as shown below:

```
uLCD.locate(???, 1);               // Move cursor to column ??? on line 2
uLCD.printf("%2d", heat_setpoint); // Print the heat setpoint
```

Improved Thermostat with both Heating and Cooling (25%)

Build a thermostat that controls either heating or cooling, depending on the mode set by the user. This thermostat has five states: OFF, HEAT_OFF, HEAT_ON, COOL_OFF, and COOL_ON and two different temperature thresholds (i.e., heat_setpoint and cool_setpoint). The OFF state is the default state upon reset, indicating that the thermostat is powered off. The HEAT_OFF state indicates that the thermostat is powered on, but heating is not turned on because the room temperature is above the heat set-point. The COOL_OFF state indicates that the thermostat is powered on, but cooling is not turned on because the room temperature is below the cool set-point.

Add a third pushbutton to be used as the "mode button". Upon reset, the mode is set to OFF. Pressing the mode button once should change the mode to HEAT. Pressing the mode button again should change the mode to COOL. Pressing the mode button again should change the mode back to OFF. Thus, you will a third global variable (i.e., mode) whose value is changed by the user via a pushbutton interrupt. The mode variable should be an integer with three possible values: 0 (OFF), 1 (HEAT), or 2 (COOL). (A good programmer might make a "Modetype" enumeration for mode values similar to the "Statetype" enumeration for state values.) A concise way to have an integer variable (mode) cycle through the values 0, 1, and 2 is as follows:

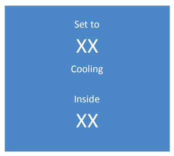
```
mode =  (mode + 1) % 3; // Cycle forward  by 1
```

This method of cycling through a list of values is very useful in many programming situations. If there were four modes to cycle through (0 through 3), you would simply use modulo 4 instead of modulo 3.

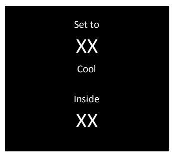
Thus, you will have three global variables (heat_setpoint, cool_setpoint, and mode) which act as *control variables* for your system. The temperature sensor provides a fourth control variable that determines the behavior of your state machine. Your program implements a Moore state machine, in which each state has a fixed set of outputs (the LEDs in this lab). In each state, you look at one or more of the control variables to determine the state transition. For example, in the OFF state, the value of the mode variable determines the transition (if mode == HEAT, go to HEAT_OFF, if mode == COOL, go to COOL_OFF, if mode == OFF, remain in OFF). The HEAT_OFF state can transition back to OFF (depending on mode) or it can transition to HEAT_ON (depending on the temperature sensor and heat_setpoint values). The HEAT_ON state can either stay put or transition to HEAT_OFF (depending on the temperature sensor and heat_setpoint values). The logic for COOL_OFF and COOL_ON is similar to the logic for HEAT_OFF and HEAT_ON.

To determine the outputs in each state, use the following rules. LED3 should be on or off to simulate heating being on or off. LED4 should be on or off to simulate cooling being on or off. LED1 and LED2 should indicate the current mode of the system. LED1 off and LED2 off indicates that the thermostat is powered off. LED1 on and LED2 off indicates that the thermostat is in heat mode. LED1 off and LED2 on indicates that the thermostat is in cool mode.

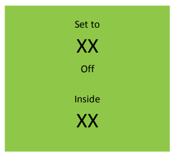
The LCD output display ideas for each of three additional states are shown below:



(COOL_ON state)



(COOL_OFF state)



(OFF state)

Next, add a fourth pushbutton which toggles the temperature display on the LCD from Fahrenheit to Centigrade and Centigrade to Fahrenheit. A "C" or "F" could be added to the LCD temperature displays, but this is not required and is not done on many thermostats. Most products are built for international markets and need to support both along with different languages. This process is sometimes called "localization".

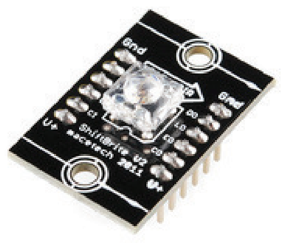
Add Speaker, for Pushbutton and State Feedback (10%)

For user feedback, use the driver transistor and speaker to make a very fast click sound whenever a pushbutton is pressed to increment or decrement the set-point. Also make a short beep when the mode is changed, heating turns on or off, or cooling turns on or off. The PWM speaker PlayNote() function is probably the best choice for this program. You should have used this in the earlier prelab. Three different beep tones should be played to distinguish between mode changes, heat turning on or off, and cooling turning on or off.

In general, interrupt routines (i.e. callback functions) need to be fast, so it would not be a good idea to call PlayNote() within an interrupt routine if you were playing a long note. One option to avoid this is to have a global flag variable that is set in the interrupt routine (to trigger an action) and then serviced in the main loop (perform the action and clear the flag). For this lab, it is acceptable to play the fast click sounds from within the pushbutton callbacks (an extra credit option using a timer interrupt to turn off the sound can fix this later). The longer beep tones must be played in the main loop. A good location for these PlayNote() calls is the same conditional statements that you use to determine state transitions. See the speaker wiki page for additional hardware and software help using speakers with mbed.

Add RGB LED State Display (10%)

The real Nest also uses a passive infrared (PIR) motion sensor to automatically turn off the LCD when no one is around. Many devices also have an ambient light sensor to dim displays in dark rooms. This prevents annoying light pollution in rooms at night, saves some power, and prolongs the life of the display. In such devices, a single RGB LED could be used show status when the display is off. We don't have these light and motion sensors in the kit (saves about ten dollars), but the RGB LED in the kit can be added for this purpose.



Use an RGB LED (left image above in new kits) or the shiftbrite (right image from older kits) to add a color display of the current state like the display (i.e., heating-orange, cooling-blue, not heating or cooling-black, off-green). For this portion, a new C++ class must be developed for a single RGB LED or shiftbrite and placed in a separate *.h file. The shiftbrite should function similar to the existing mbed APIs, it will call the SPI class (and Digital Out) but extend it for a shiftbrite. See the RGB LED or shiftbrite wiki page for additional help using RGB LEDs and shiftbrites with mbed and the mbed "making it a class" example. Reading the code in the classes developed in the speaker wiki page examples will also provide some more details. The RGB LED has a class setup on the wiki page but it is not in a *.h file yet.

For those using old kits with Shiftbrites in the main program, it would work something like this after an #include “Shiftbrite.h”:

```
// declare a new Shiftbrite connected to these pins using a class
Shiftbrite myShiftbrite(penable, platch, pred, pgreen, pblue);
…
// call the class member function to set the LED color
myShiftbrite.write(red, green, blue);
```

After studying the links with the example class code described earlier, for the new Shiftbrite class here is an additional hint. You will need a constructor and a member function (write) to write a new color to the Shiftbrite. The SPI API has three required pin arguments (one is not used but still needed) and the other examples only have one. So here is a line showing how three would work in the class constructor:

```
Shiftbrite::Shiftbrite(PinName pin_e, PinName pin_l, PinName pin_do,
                       PinName pin_di, PinName pin_clk):
           _pin_e(pin_e), _pin_l(pin_l), _spi(pin_do, pin_di, pin_clk)
{
     //setup SPI options and initialize digital outs
}
```

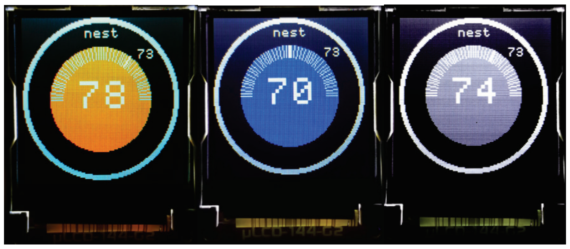
With these also in the class definition:

```
private:
//class sets up the 4 pins
    DigitalOut _pin_e;
    DigitalOut _pin_l;
    SPI        _spi;
};
```

If you are hooking up the Shiftbrite, the pins on the left side (see lettering on PCB) must be used. The pins on the right side go through a logic buffer so that you can connect long chains of Shiftbrites***. Notice that if it is ever plugged into the breadboard upside down, the power pins get reversed, and if the supply does not shutdown from high current the chip burns out!***

The RGB LED and the Speaker draw a bit more current from the power supply than the other devices. A power surge can cause a momentary voltage drop on the supply and this might cause a low temperature reading the instant that they turn on at a state change. If this is a problem, the TA has some 10uf capacitors that can be connected across the sensor’s power supply pins (right at the sensor pins). The longer lead on the electrolytic capacitor goes to the + supply pin. A small time delay (<1sec?) might also help after turning them on, if the problem persists. Also the temperature reading could be averaged with several of the last readings to minimize this issue. Another option might be to just throw out a temperature reading that changes too fast from the average and take another reading. Some mbeds & PCs are more prone to this issue and when the mbed processor chip is left setup in debug mode (for easy use in our labs on the USB cable) it has a bit more internal analog noise according to the data manuals. If it was not in debug mode (switching the analog things on and off fast inside the chip) and was running off of batteries or and external power supply as it would be in a final product, the analog noise problem goes away.

LCD Artistic Merit, Improving Playnote, and Language Support (Extra Credit up to 8%)



Example Nest-Like Student LCD Display

The LCD supports background and font colors, graphics (i.e., circles, lines…), and different font sizes. So far, it has been used as just a color text LCD. Implement a graphical display more like the real Nest. Photos and videos can be found at www.nest.com. The LCD wiki page demo code shows the additional member functions available for this. For full credit, the following must be included:

1. (+1%) Some nice graphics in the display (perhaps a circle and lines like the real nest?)
2. (+1%) All information displayed somehow (using text, graphics, or colors) as was seen in the original text-based displays.
3. (+1%) Overall quality of artistic design utilizing all of the items above. At discretion of TAs.
4. (+1%) Use a timeout interrupt routine to improve the Playnote function so that it does not wait in the pushbutton callback or main routine for the note to end. The Playsong class code on the using a speaker wiki page shows an example of how this idea works. Playnote would return after starting the sound and use the timer interrupt to turn off the note later (after the note's duration timer counts down and generates another interrupt to turn it off).
5. (+1%) Use a jumper wire tied to a digital input pin to select a different language for the LCD display. The other end of the wire jumper is moved from "gnd" to "Vu" to switch languages at power up. Pick a language that is supported by the default character set. The jumper wire would be working like a switch to select the language. In a final product, a switch or another pushbutton could be used to select the language (but at this point we have already used all of them in the kit).
6. (+1%) Support a language not covered by the LCD's built in character set. It is possible to load new character fonts in the display from the SD card on the back of the LCD, but it takes quite a bit of extra work and a special tool to convert the font files for the SD card. The process is described in the LCD wiki page. Try this extra credit option last and only if you have time!
7. (+2%) For a TA checkoff of the entire lab three days (including any extra credit) before the due date.

ECE 2036 Lab 2 Check-Off Sheet

Student Name: ____

Professor: ____

Demonstration	**TA Signoff**
Basic Thermostat (40%)	
Add LCD data display (5%)	
Add Pushbuttons (10%)	
Improved Thermostat (25%)	
Add Speaker (10%)	
Add RGB LED (10%)	
LCD,Playnote,Language(ExtraCredit<=8%)	
Late Penalty (-20% per day)	
Submission&Demo Late Penalty(-10%per day)	

Lab Submission Instructions:

Each lab must be demonstrated to the TA by the due date. Students should be finished with the lab before the start of TA office hours on the due date, so that they may get the lab checked-off on the due date. There can be long lines on the due date! This can all be avoided by getting the lab checked off before the last day. Students cannot expect to get TA help on lab due dates, as their time will likely be dedicated just to lab check-offs. Only one formal check-off attempt is allowed per student, and the entire lab must be checked-off at once. If the lab gets too busy, the TAs may setup a list of waiting students on the lab's whiteboard. There will be a 20% per week day late penalty on demos.

In the rare event that TAs are unable to finish all checkoffs for students waiting in the lab on the due date when the TA lab hours end, and you get on the TA's list of students in the lab waiting for a checkoff, submit the source code on T-Square (if this does not work or is not setup email the source code in an attachment to a TA). If this happens to you, you still must return to demo using the previously submitted source code during one of the next three TA office hour days. There will be a 10% per day penalty, if you do not return to demonstrate to the TAs within three TA office hour days of the due date (this is all assuming TAs are unable to complete waiting student checkoffs on the due date when the lab closes).