APPLICATION NOTE

AN400

Rabbit Races Across Wi-Fi Ethernet Bridge

By Ingo Cyliax

Ever wondered how to make your embedded application wireless? Well, if you already have an Ethernet based solution, it's easy. Just add an Ethernet to Wi-Fi bridge. This paper will describe a Wi-Fi enabled embedded application. First, let's cover some background.

Wi-Fi Background

Wi-Fi, a popular name for 802.11b, is one of the wireless schemes available in the 802.11 suite. 802.11b describes the media access and link layer control for a 2.4 GHz implementation which can communicate at a top bit-rate of 11 Msps. There are some proprietary extensions to 22 Msps. Other standards describe a faster implementation (54 Msps) in the 2.4 GHz (802.11g) and a 54 Msps implementation in the 5.6 GHz band. Currently, Wi-Fi (802.11b) is the most common implementation.

The 802.11 standard also describes how these devices access each other. In the simplest scheme, called the ad-hoc, each device sets a channel number and a code. Once they match, they can talk with each other. This works fine, when all of the devices can hear each other, or there are only two stations.

802.11 also defines a scheme called infrastructure. Here an access point arbitrates and manages devices. It allocates timing and bandwidth to devices. When a device wants to join a work group, it will listen for an access point and then announce itself to it to join.

For our application, we will use a Wi-Fi bridge. It translates standard Ethernet packets into 802.11b packets, which can be received by another 802.11b device. Most Wi-Fi Ethernet bridges can operate in ad-hoc or infrastructure mode. For this application we will use it in ad-hoc mode to communicate with a Wi-Fi card in a PDA or notebook computer.

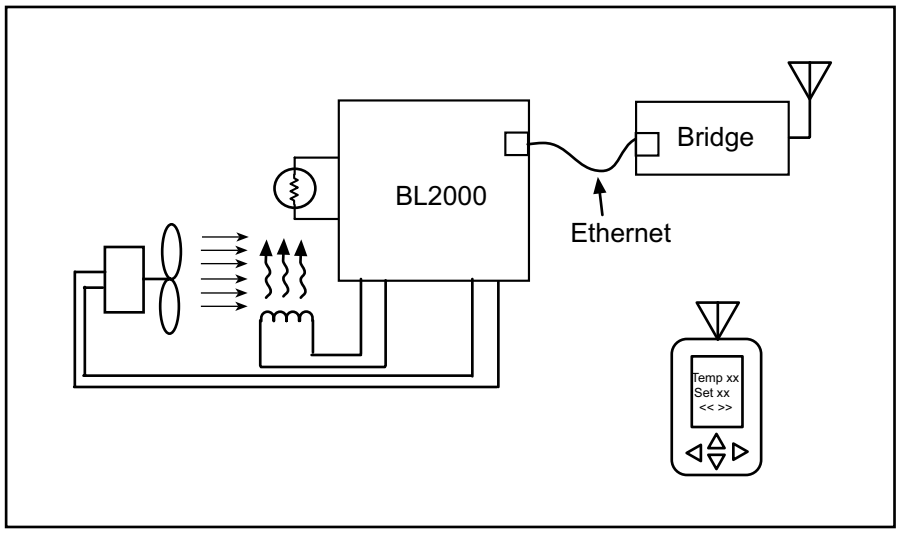
For more information on using wireless networking with Rabbit-based products, please see technical note TN230, "Off-the-Shelf Solutions for 802.11b Wireless Networking." This document is available online at:

`http://www.rabbitsemiconductor.com/support/techNotes_whitePapers.shtml`

The Application

We're implementing a Wi-Fi based thermostat. The controller will sense the temperature using a thermistor, and drive a heater output and a fan output to try to control the temperature to match a preset temperature. The user interface is entirely implement as an embedded Web server application, so any Wi-Fi enabled device capable of HTML browsing will be able to act as our user interface device. Figure 1 shows a block diagram of the system.

Figure 1. Block Diagram of Wi-Fi Based Thermostat

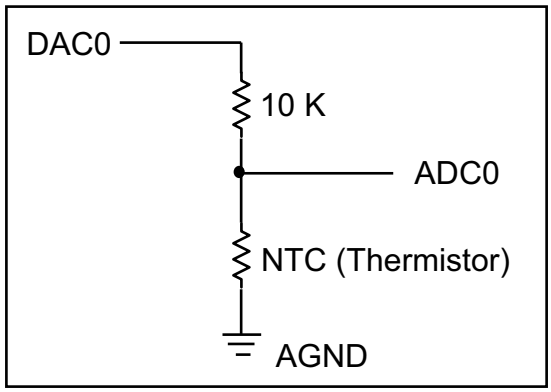


Hardware Solution

I ended up using a BL2000 because of its compact shape, Ethernet and high current output drivers. Other good choices include the BL2100, the BL2500 and the OP7200.

A thermistor based temperature sensor was interfaced to ADC0 and DAC0, in a voltage divider circuit using a 10K resistor. At 25ºC the input to ADC0 will read 50% of the voltage applied with DAC0.

Figure 2. Voltage Divider Circuit



A 12 V DC brushless fan was attached between the RAW power supply and OUT2, which is configured in a current sink mode. The fan draws 0.16 A, which is within the 200 mA current limit of the output.

To compute the minimum resistance to implement a resistive "heater":

```
R = 12/0.200 = 60 Ohm

P = 12*0.200 = 2.4 Watt
```

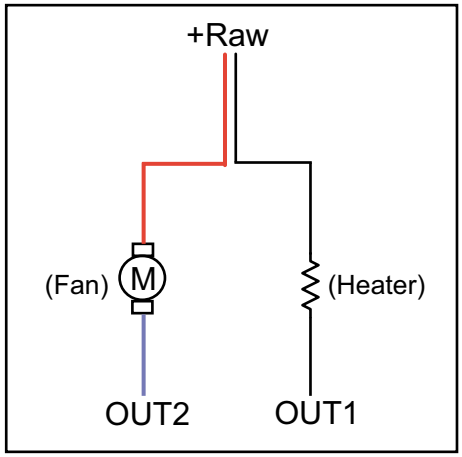
A larger resistance can be used to match the power rating of the resistor you have available. For example, if you want to use a 1 Watt resistor, then the minimum resistance would be:

$$P = V^2/R$$

```
R = 144/1 = 144 Ohm
```

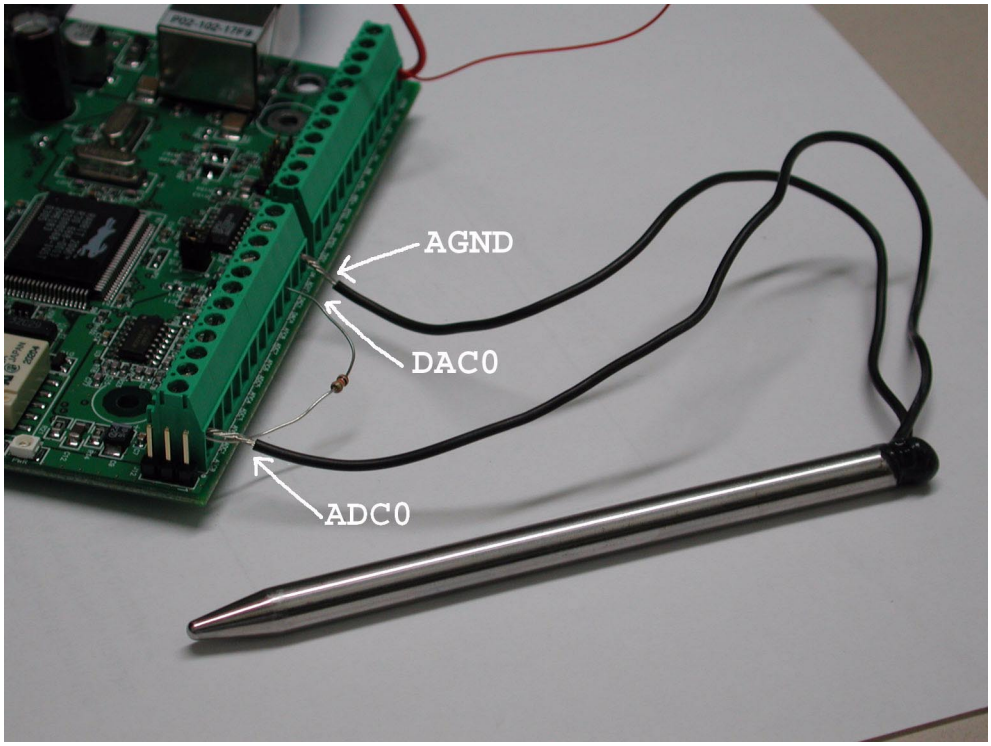
You can use a smaller power rating for the resistor if the object you are heating can sink the extra power. The "heater" is wired between RAW and OUT1 on the BL2000.

Figure 3. Fan and heater wired to controller



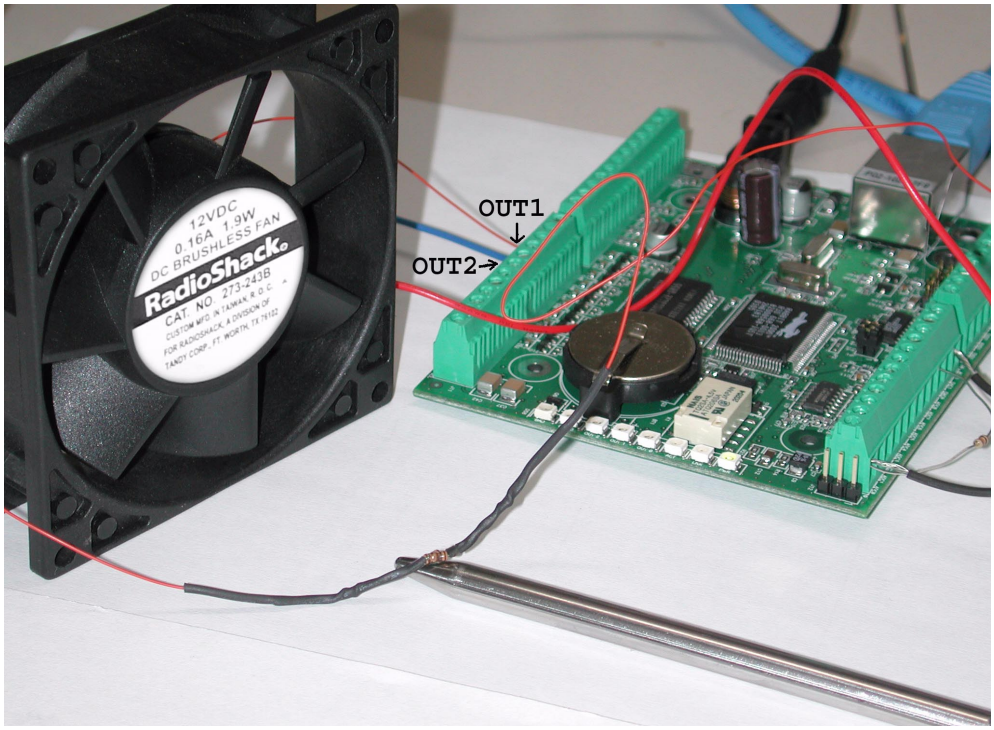
The following photograph shows a voltage divider that matches the wire diagram in Figure 2

Figure 4. Voltage Divider.



The next photograph adds the heater and fan from Figure 3.

Figure 5. Heater, Fan and Thermistor wired to the BL2000



Now that the BL2000 is wired up as a Wi-Fi enabled thermostat, we are ready to turn our attention to the application software and the Wi-Fi devices we will be using.

Software Solution

We will need to download some software to the BL2000. A zip archive (`AN400.zip`) with the application software can be downloaded from:

```
http://www.rabbitsemiconductor.com/support/appNotes.shtml
```

Run the Dynamic C editor[1] and then open the source file `Temp_bl.c`. Attach the "Prog" connector on the programming cable to the BL2000. Attach the serial connector on the other end of the programming cable to a COM port on the PC that is running Dynamic C.

Before downloading `Temp_bl.c` to the target there are two things you must do.

1. Set the controller's IP address. To do this, open the file `/lib/tcpip/tcp_config.lib` and edit the macro `_PRIMARY_STATIC_IP`. You need to use an IP address that can be seen from the Wi-Fi bridge, i.e., on the same network. E.g., the Wi-Fi bridge used in this application had a default IP address of 192.168.1.30, so the macro `_PRIMARY_STATIC_IP` was set to 192.168.1.75. Consult the documentation in `tcp_config.lib` for more information.
2. Identify the pathname of the HTML page and graphic that are used for the web interface. In the source code, find the lines:

   ```
   #ximport "h:/examples/web/pages/temp.shtml" index_html
   #ximport "h:/examples/web/pages/rabbit1.gif" rabbit1_gif
   ```

   and change them to match the location of wherever you put these files when you extracted them from AN400.zip.

Now you are ready to download `Temp_bl.c` to the BL2000. Press the F9 key and the program will compile, download and run. After this you may disconnect the programming cable.

1. If you are using a Dynamic C version later than 7.33 go to http://www.rabbitsemiconductor.com/support/downloads/downloads_prod.shtml and download the patch in `IDBlock.zip`. The patch is required for this demo unless you are using Dynamic C 7.33.

Software Details

The software consists of three logical components, and is divided into two functions and one library call. The library call implements the poll loop the TCP/IP and Web server needs to process incoming, outgoing packets and various timers it needs to function. The other two functions `update_temp()` and `update_outputs()` do what their names imply.

The function `update_temp()` sets the desired output voltage of the DAC0 using the BL2000 library routine `anaOutVolts()`, which in our case is 1.0 V. The `anaInVolts()` library call is used to read the input voltage, which is the result of a voltage divider network using the thermistor and a fixed/known resistor. (See Figure 2.)

To compute the resistance of the thermistor, we use the equation:

$$Rth = V_{ADC0} \cdot R_{EXT} / (V_{DAC0} - V_{ADC0})$$

Where:

V_{ADC0} - voltage read by ADC0
V_{DAC0} - voltage applied by DAC0 (1.0 V)
R_{EXT} - external resistor (10k)
Rth - computed resistance of thermistor

Once we know the resistance, we can use the constants provided for the thermistor to compute the temperature.

$$T = 1/(A + ((1/Bth) \cdot \log(Rth)))$$

Bth is a constant provided for the thermistor. A is computed with this equation:

$$A = 1/T_K - 1/Bth \cdot \log(Rth)$$

where

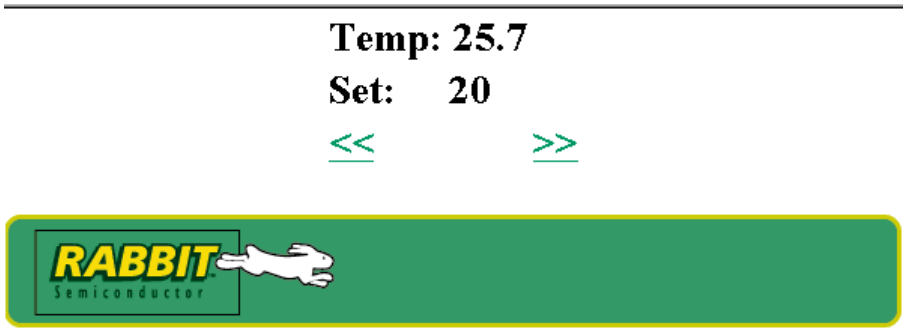
T_K = 298°K (25°C)
Rth = 10k (25°C)

We average 10 readings before we update the temperature variable used by the web interface and our output routine.

The output routine simply compares the current averaged temperature with the set point and sets the outputs OUT1, OUT2 correspondingly. If the integer representation of the temperature and the set point match, then none of the outputs are on. This implements a dead band so that the controller isn't constantly seeking between heating and cooling.

The Web interface is based on a simple HTML web page. A table construct organizes the display into 3 rows and 2 columns.

Figure 6. HTML Page Served by the BL2000



The current temperature and set point are substituted when the page is served using server side includes (SSI). The left and right arrows are links that point to two CGI functions, `setlower.cgi` and `sethigher.cgi`, which are mapped as function call backs in the application software. The functions simply increment and decrement the set point value by one.

The HTML page also contains a META refresh tag that instructs the browser to reload the same page every 2 seconds. This is a simple mechanism for implement dynamic web pages, without resorting to using facilities like JavaScript or Java applets.

```
<META HTTP-EQUIV="refresh" contents="2;URL=" >
```

This application will run the same whether it's Wi-Fi enabled or on the Ethernet. If you plan to use this application in an Ad-hoc Wi-Fi network, you can use the supplied address, as long as your Wi-Fi enabled device which plans to communicate with it, has an IP address on the same network.

Setting Up Wi-Fi

First you need to configure the bridge you will use. Follow the manufacturers direction on how to configure the bridge to use ad-hoc networking, and pick a channel number (11) and SSID (default) for your network. It doesn't matter what it is, as long as it matches all the devices you plan to use in the network. Optionally, you may want to set the bridge's IP address to one that is on the same network, but not the same address as the thermostat or your PDA/Notebook. This will typically let you use the Web interface on the bridge for further configuration and/or status checks.

Next, you will need to configure your Wi-Fi enabled display device (PDA/Notebook). You will do this by configuring your Wi-Fi card (again following the manufacturers instructions) to set the mode to ad-hoc, the same channel (11) and assigning the same SSID (default) to it. You will also have to set the network interface to use static IP addressing and assign an IP address on the same network as the Wi-Fi enabled thermostat.

The following photograph shows everything wired up.

Figure 7. BL2000 as a Wi-Fi Enabled Thermostat

Open a browser on your PDA/Notebook and go to the IP address that you assigned to the BL2000. You will see the HTML page shown in Figure 6. When the temperature of the heater rises above the set point, the fan will turn on to cool it down. You can adjust the set point using the left/right arrows.

And that's all there is to it. The parts used in this application are listed in the table below. The source code listing is shown in Appendix A.

Table 1. BOM for Wi-Fi Enabled Thermostat

Description	Supplier	Part Number
BL2000	Rabbit Semiconductor	101-0430
10k ohm Thermistor Temperature Sensor	Grauger	Goldline Model 1M-4
10k ohm ¼ W Resistor	Digikey	P10KBACT-ND
12VDC Brushless DC Fan	Radio Shack	273-243
60 ohm 2.5 W Resistor	Digikey	62W-5-ND
Wireless Bridge D-Link DWL-810+	D-Link	D100 1444
Dell Axim (Pocket PC)	Dell	W0772*
Linksys WCF12 CF Wi-Fi Adaptor	CompUSA	298210

* You can buy a refurbished unit (part #9W052) directly from Dell, or go to eBay like we did.

Appendix A: Source Code Listing

```
/*************************************************************

    temp_bl.c
    Rabbit, 2003

    Description

    ===========

    This program demonstrates a controller running a WEB based closed loop control application.
    A thermistor is wired in a voltage divider network to the ADC0 input. The webserver will
    monitor the temperature and allow the user to update the setpoint. Two digital outputs can be
    used to indicate if the setpoint is cooler or hotter than the actual temperature. These outputs
    can be used to implement a cooling device (fan) or a heating device (resistor), in order to con-
    trol the temperature.

*************************************************************/

#class auto
```

```
/************************************
   START OF CONFIGURATION SECTION
************************************/
/*
   Pick the predefined TCP/IP configuration for this sample. See LIB\TCPIP\TCP_CONFIG.LIB
   for instructions on how to set the configuration.
*/
#define TCPCONFIG 1
/*
   TCP/IP modification - reduce TCP socket buffer size to allow more connections. This can be
   increased, with increased performance, if the number of sockets  are reduced. Note that this
   buffer size is split in two for TCP sockets—1024 bytes for send and 1024 bytes for receive.
*/
#define TCP_BUF_SIZE 2048
/*******************************
   Web server configuration
*******************************/
/*
   Define the number of HTTP servers and socket buffers. With tcp_reserveport(), fewer HTTP
   servers are needed.
*/
#define HTTP_MAXSERVERS 2
#define MAX_TCP_SOCKET_BUFFERS 2
/*
   This is the address that a client (e.g., Netscape/IE) uses to access your server. Usually, this is
   your IP address. If you are behind a firewall, though, it might be a port on the proxy, that will
   be forwarded to the Rabbit board. The commented out line is an example of such a situation.
*/
#define REDIRECTHOST _PRIMARY_STATIC_IP
// #define REDIRECTHOST "my.host.com:8080"
/************************************
   END OF CONFIGURATION SECTION
************************************/
```

```
/*
```

REDIRECTTO is used by cgi functions to tell the browser which page to hit next. The default REDIRECTTO assumes that you are serving a page that does not have any address translation applied to it.

```
*/

#define REDIRECTTO "http://" REDIRECTHOST ""

#memmap xmem

#use "dcrtcp.lib"              // include the TCP/IP stack
#use "http.lib"                // include the HTTP server


/*
```

The compiler directive #ximport places the HTML pages that we want our HTTP server to serve in xmem flash on the board that is hosting the server. In this case, a BL2000.

```
*/

#ximport "h:/examples/web/pages/temp.shtml" index_html
#ximport "h:/examples/web/pages/rabbit1.gif" rabbit1_gif

/*
```

The following structure associates a file extensions with the function to handle the extension. The first entry in the structure, in our case the handler function for files with an extension of .shtml, will be used for files that have no extension.

```
*/

const HttpType http_types[] =
{
   { ".shtml", "text/html", shtml_handler},    // ssi
   { ".html", "text/html", NULL},              // html
   { ".cgi", "", NULL},                        // cgi
   { ".gif", "image/gif", NULL}                // gif
};

float temptot;                    // variable to sum temp readings
int tempn;                        // variable to count temp readings


char temperature[15];             // variable on web page
char setpoint[15];                // variable on web page
```

```
/*
```

These 2 functions adjust the web page variable, setpoint, and then redirects the browser back to the original page so that the updated value is visible.

```
*/

int setlower(HttpState* state)
{
   sprintf(setpoint,"%d",atoi(setpoint)-1);
   cgi_redirectto(state,REDIRECTTO);
   return 0;
}

int sethigher(HttpState* state)
{
   sprintf(setpoint,"%d",atoi(setpoint)+1);
   cgi_redirectto(state,REDIRECTTO);
   return 0;
}

/*
```

This structure defines the files, variables and functions that the web server can access.

```
*/

const HttpSpec http_flashspec[] =
{
   {HTTPSPEC_FILE, "/", index_html, NULL,0,NULL,NULL},
   {HTTPSPEC_FILE, "/index.shtml", index_html, NULL,0,NULL,NULL},
   {HTTPSPEC_FILE, "/rabbit1.gif", rabbit1_gif, NULL,0,NULL,NULL},
   {HTTPSPEC_VARIABLE,"temperature",0,temperature,PTR16,"%s",NULL},
   {HTTPSPEC_VARIABLE, "setpoint",0, setpoint, PTR16, "%s", NULL},
   {HTTPSPEC_FUNCTION, "/setlower.cgi", 0, setlower,0,NULL, NULL},
   {HTTPSPEC_FUNCTION, "/sethigher.cgi",0, sethigher,0,NULL, NULL},
};

#define OUT_FAN 2          // this is the output channel the fan is on
#define FAN_ON  0          // fan output is active low
#define FAN_OFF 1

#define OUT_HEATER 1       // the heater is on this output channel
#define HEATER_ON  0       // heater output is active low
#define HEATER_OFF 1

#define ADC_RTC 0          // thermocouple input
#define DAC_RTC 0          // output for thermocouple resistor network
```

```
void update_outputs()
{
   auto int setp, temp;

   setp = atoi(setpoint);                // string to int conversion
   temp = atoi(temperature);             //   of web page variables


   if(setp < temp) {                     // temp higher than setpoint?
      digOut(OUT_FAN,FAN_ON);            //   yes, turn on fan
      digOut(OUT_HEATER,HEATER_OFF);     //   and turn off heater
   }

   if(setp > temp) {                     // temp lower than setpoint?
      digOut(OUT_FAN,FAN_OFF);           //    yes, turn off fan
      digOut(OUT_HEATER,HEATER_ON);      //    and turn on heater
   }

   if (setp == temp) {                   // temp equals setpoint?
      digOut(OUT_FAN,FAN_OFF);           //   yes, turn off fan
      digOut(OUT_HEATER,HEATER_OFF);     //   and heater
   }
}

main()
{
   int i;
   temptot = 0.0;                       // initialize vars to average
   tempn = 0;                           //   the temperature reading


   brdInit();                           // initialize board


   digOut(OUT_HEATER,HEATER_OFF);
   digOut(OUT_FAN,FAN_OFF);

   strcpy(temperature,"20");            // initialize web page vars
   strcpy(setpoint,"20");

   sock_init();                         // initialize TCP/IP stack
   http_init();                         // initialize HTTP server
   tcp_reserveport(80);                 // get port 80 for server


   while (1)                            // set up endless loop
   {
      update_temp();                   // compute temp of heater &
      update_outputs();                // control heater & fan
      http_handler();                  // process server requests
   }
}
```

```
#define Rth 10000.0                    // thermistor T25 resistance
#define Bth 4100.0                     // thermistor B value
#define Rex 10000.0                    // other resistor


update_temp()
{
   float R2,R1,V1,V2,A,T;

   V1 = 1.0;
   R1 = Rex;
   A = 1/298.0 - ((1/Bth)*log(Rth)); // compute thermistor constant


   anaOutVolts(DAC_RTC, V1);          // set output voltage for DAC0


   V2 = anaInVolts(ADC_RTC);          // read voltage from ADC0


   R2 = V2*R1/(V1-V2);                // compute thermistor resistance
   T = 1 / (A + ((1/Bth)*log(R2)));   // compute temp in kelvins


   if(tempn > 10 || tempn < 0){       // have we read temp 10 times?
      sprintf(temperature,"%7.1f",temptot/tempn);  // yes, update var
      temptot = 0.0;                  //    & zero out vars holding
      tempn = 0;                      //    temp sum and count.


   }else{                             // no,
      temptot += (T-273.0);           //    add current temp to sum
      tempn ++;                       //    and increment count.
   }
}
```