Controlling the Humidity with an Embedded Linux System

Using an inexpensive embedded Linux board and a few extra devices, you can control things like room humidity.

Jeffrey Ramsey

Charles Darwin, in his *Beagle Diary* that led to the book *Voyage of the Beagle*, wrote while in Peru, "On the hills near Lima, at a height but little greater, the ground is carpeted with moss, and beds of beautiful yellow lilies, called Amancaes. This indicates a very much greater degree of humidity, than at a corresponding height at Iquique." Like Darwin, I always have been conscious of humidity. For years, I've struggled with the humidity in my music room, as my Carlos Pina concert-grade classical guitar went out of tune frequently with wild swings in humidity. Pennsylvania winters are cold and dry, summers hot and humid, and this plays havoc on my classical guitar.

Commercially available humidifiers and dehumidifiers have humidity sensors that are far too coarse for certain applications. One such application is the humidity control for my music room. Being an embedded developer for my entire career, with a particular interest in embedded applications for Linux, I decided to build my own humidity controller for my music room. After a bit of research, I settled on a hardware architecture that includes a Cirrus EP9301 ARM9-based controller, several solid-state relays and a capacitive humidity/temperature sensor. Linux was my selection as the embedded OS, and with several Linux device drivers to control the relays and monitor the humidity and temperature, the basis of a humidity controller was born.

I decided to use the humidifying and dehumidifying capability of my retail humidifier and dehumidifier units. The humidity controller that I built switches power on and off to the humidifier and dehumidifier, essentially assuming the role of the humidity sensor. To finish off the humidity controller, I added a Web interface that allows me to monitor and control the system through any network-attached browser, such as Firefox.

Before I began developing the embedded humidity controller, I had to decide on the system-level requirements. Even though this was for personal use only, it's always good practice to do a bit of systems engineering on the front end of the design process. I decided on the following requirements:

- The humidity control system should control humidity with a minimum range of plus or minus 3.5% rH.
- Humidifier and dehumidifier control will be through switching of 120V AC and neutral power lines.
- Current humidity and temperature will be displayed through a browser interface.
- Configuration of the desired humidity setting will be done through a browser interface.
- All humidity and temperature settings will be stored persistently in an SNMP MIB.
- All software will operate in an embedded Linux environment.

Figure 1 shows the overall embedded hardware architecture of the humidity controller. The ARM9-based controller I selected is the TS-7200 from Technologic Systems. In addition to the controller board, I used a TS-RELAY8 peripheral board connected to the TS-7200's PC/104 bus. The daughter board contains eight SPDT relays. To house the system, I used a TS-ENC720 enclosure. Figure 2 shows the main board and peripheral board mounted on the back plate of the enclosure.

The capacitive humidity/temperature sensor is a Sensirion SHT11, which is controlled through a two-wire data/clock interface. The SHT11 control interface connects to two of the TS-7200's discrete I/O pins. Switching power on and off is accomplished with the single pole double throw (SPDT) relays on the peripheral board. I used a pair of relays for the humidifier and another pair for the dehumidifier. I used a pair as it

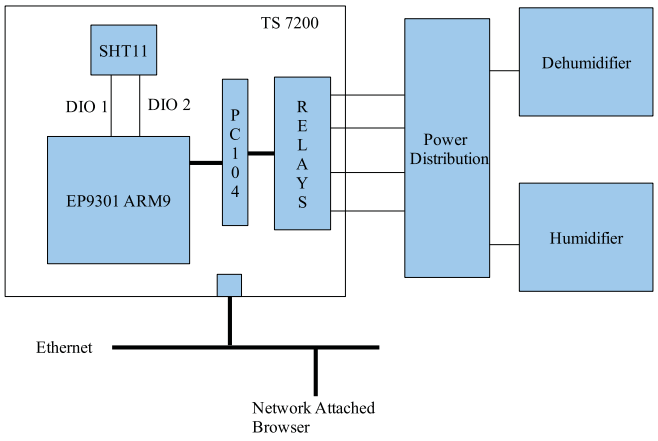

Figure 1. Hardware Architecture

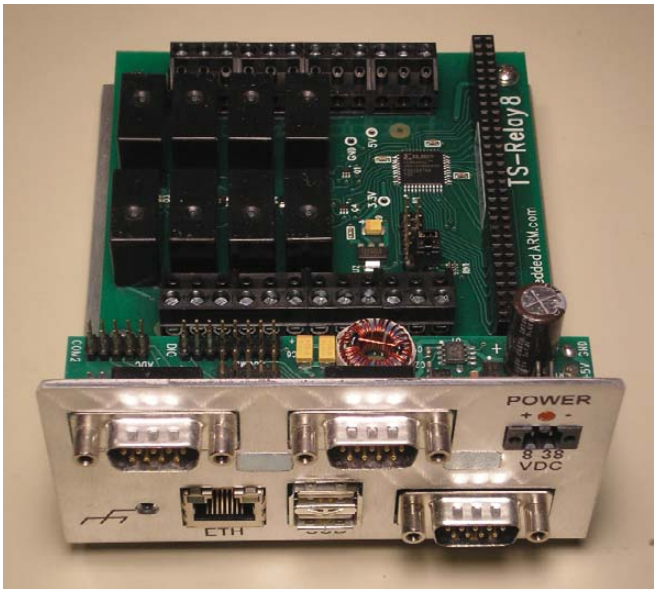

Figure 2. Hardware

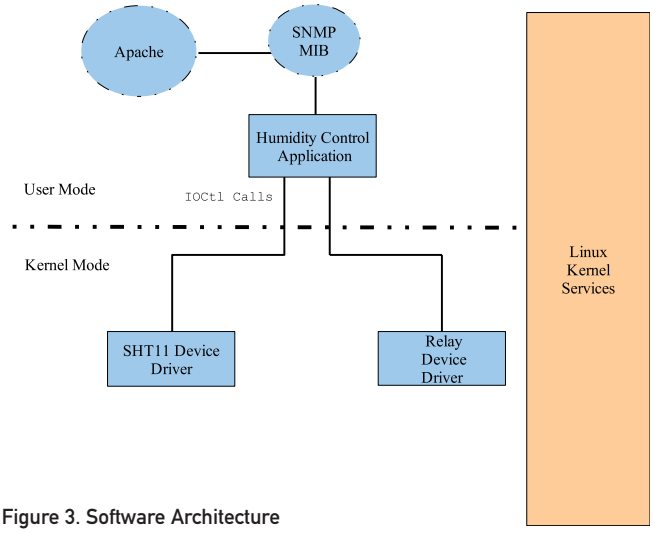

Figure 3. Software Architecture

Listing 1. Generate SHT11 Start Transmission Sequence

```
void writeSHT1xTransmissionStartSequence(void)
{
     writeSHT1xOne(DATA_SHT);
     writeSHT1xZero(SCK_SHT);
     udelay(2);
     writeSHT1xOne(SCK_SHT);
     udelay(2);
     writeSHT1xZero(DATA_SHT);
     udelay(2);
     writeSHT1xZero(SCK_SHT);
     udelay(2);
     udelay(2);
     udelay(2);
     writeSHT1xOne(SCK_SHT);
     udelay(2);
     writeSHT1xOne(DATA_SHT);
     udelay(2);
     writeSHT1xZero(SCK_SHT);
     udelay(2);
}
```

seemed much safer to switch both the 120V and neutral lines, rather than just the 120V.

The TS-7200 single-board computer (SBC) runs Linux on an ARM9-based processor. The system's software architecture is shown in Figure 3. Two Linux drivers are required: one to sense the humidity (and temperature, which came almost free) and the second to control the position of the relays. A user-mode application on top of the drivers periodically polls the humidity and temperature data, and controls the relay position depending on SNMP MIB configuration settings. The SNMP MIB is managed by the Linux snmpd dæmon. The SNMP MIB also serves as the basic bridge to an Apache custom module that exposes the MIB data to a Web browser for control and monitoring of the entire humidity control system. Each component of the humidity control system is described in more detail later in this article.

Linux Device Drivers

The two required Linux drivers, which I designed as loadable modules, are rather basic as far as Linux drivers go. They both are character devices with ioctl interfaces that provide access to the SHT11 sensor and control of the power relays. The SHT11 driver requires only two ioctl functions:

- SHT1X_IOC_READ_HUMIDITY: read the current SHT11 humidity.
- SHT1X_IOC_READ_TEMPERATURE: read the current SHT11 temperature.

With both the temperature and humidity, I have the option of calculating the dew point (even though the system is indoors, and the last thing I expect is dew to form on the components). The SHT11 driver reads humidity and temperature using a two-wire interface that is well defined in

Listing 2. Transmit Command Sequence

```
void writeSHT1xCommand(int iMode)
{
     unsigned char  ucBitToCheck;
     unsigned char  ucAckBit;

     driveDataLine(DATA_SHT);
     /* All 3 address bits always zero
      * so start with those */
     writeSHT1xZero(DATA_SHT);
     udelay(2);
     writeSHT1xOne(SCK_SHT);
     udelay(2);
     writeSHT1xZero(SCK_SHT);

     writeSHT1xZero(DATA_SHT);
     udelay(2);
     writeSHT1xOne(SCK_SHT);
     udelay(2);
     writeSHT1xZero(SCK_SHT);

     writeSHT1xZero(DATA_SHT);
     udelay(2);
     writeSHT1xOne(SCK_SHT);
     udelay(2);
     writeSHT1xZero(SCK_SHT);

     /* Now transmit the 5 command bits,
      * in the order of MSb to LSb */
     for (ucBitToCheck=0x10; ucBitToCheck != 0;)
     {
          if (iMode & ucBitToCheck)
               writeSHT1xOne(DATA_SHT);
          else
               writeSHT1xZero(DATA_SHT);
          udelay(2);
          writeSHT1xOne(SCK_SHT);
          udelay(2);
          writeSHT1xZero(SCK_SHT);
          ucBitToCheck >>= 1;
     }

     /* Now tri-state the data DIO so the
      * device can ACK the transfer */
     tristateDataLine(DATA_SHT);
     udelay(2);
     writeSHT1xOne(SCK_SHT);
     udelay(2);
     ucAckBit = readSHT1x(DATA_SHT);
     writeSHT1xZero(SCK_SHT);
     mdelay(250);
}
```

the Sensirion SHT11 data sheet. The clock has no real minimum frequency, but has a maximum frequency of 10MHz. I had no reason to run the clock at the maximum rate. In fact, the messages required to transfer the temperature and/or

humidity data are so short, the clock rate could be anything within reason, so I decided to run the clock at 250KHz.

Accessing the SHT11 is relatively straightforward. A start and end sequence for each transfer is achieved using a prescribed combination of data and clock discrete I/O transitions. For example, in order to request the current humidity or temperature, a start of transmission sequence is issued that consists of the sequence of data and clock transitions as shown in Listing 1.

In Listing 1, note the use of udelay kernel calls. The timing requirements of the SHT11 two-wire access is satisfied using delays in the microseconds and, in some cases, milliseconds. This is most easily achieved using the kernel udelay call, and when millisecond delays are required, the mdelay call. I suppose there are some developers who shudder at the use of busy loops, but remember, this is a dedicated, embedded system. It does nothing but read humidity and check whether relays need to be switched on or off, and it repeats this forever.

After the start transmission sequence, the driver is free to write an 8-bit command sequence that identifies the operation to the humidity sensor, such as measure the humidity or temperature. A second procedure actually transmits the 8-bit command sequence and is shown in Listing 2.

Listing 2 not only demonstrates the bit-twiddling necessary to drive a two-wire interface solely with software, but it also reveals how the sensor acknowledges receipt of a valid command. The data DIO must be tri-stated (that is, not driven to either a 0 or a 1 by the ARM) in order for this two-wire interface to permit slave devices, such as the SHT11, to transmit back to the two-wire interface master—in this case, the SHT11 device driver in the ARM. In addition, note that the last line of code in the procedure will cause a 250-millisecond delay. This is because the SHT11 takes a good deal of time, relatively speaking, to measure either the temperature or humidity. The specification requires 210 milliseconds for the most accurate form of measurement, with a +–15% tolerance. This puts the worst-case delay at 241.5 milliseconds, which I increased to 250 milliseconds, just to be safe.

The third and final required piece of code necessary to read data from the SHT11 humidity sensor is shown in Listing 3. The Read Sensor Data procedure will read 16 bits of data from the sensor after it has measured either the humidity or the temperature. The SHT11 has the option of sending an 8-bit CRC at the end of the 16 bits of data, but I opted not to check the CRC, as it is unlikely the data ever will be corrupted due to environmental effects in my music room.

The procedures shown in Listings 1, 2 and 3 form the core of the SHT11 two-wire interface device driver code. When the driver receives an ioctl requesting the humidity, the three instructions shown in Listing 4 are all that is needed to read the current humidity from the sensor.

The second device driver controls the relays and switches the 120V AC and neutral lines to the humidifier and dehumidifier. The ioctl interface for the relay driver required

Listing 3. Read Sensor Data

```
unsigned int readSHT1xData(void)
{
     int           iLoop;
     unsigned int  uiBitRead;
     unsigned int  uiMSB=0;
     unsigned int  uiLSB=0;
     unsigned int  uiRetValue;

     /* Read MSB from SHT1x */
     for (iLoop = 0; iLoop < 8; iLoop++)
     {
          uiMSB <<= 1;
          writeSHT1xOne(SCK_SHT);
          uiBitRead = readSHT1x(DATA_SHT);
          udelay(2);
          writeSHT1xZero(SCK_SHT);
          udelay(2);
          if (uiBitRead)
              uiMSB |= 1;
     }
     /* Acknowledge sequence; must drive data
      * line as it is tri-stated at this point */
     driveDataLine(DATA_SHT);
     writeSHT1xZero(DATA_SHT);
     udelay(2);
     writeSHT1xOne(SCK_SHT);
     udelay(2);
     writeSHT1xZero(SCK_SHT);
     tristateDataLine(DATA_SHT);
     udelay(2);
     /* Read LSB from SHT1x */
     for (iLoop = 0; iLoop < 8; iLoop++)
     {
          uiLSB <<= 1;
          writeSHT1xOne(SCK_SHT);
          uiBitRead = readSHT1x(DATA_SHT);
          udelay(2);
          writeSHT1xZero(SCK_SHT);
          udelay(2);
          if (uiBitRead)
              uiLSB |= 1;
     }
     /* Don't acknowledge last byte so the device
      * doesn't transmit the 8-bit CRC as it isn't
      * really necessary for this application */
     uiRetValue = u8tou16(uiMSB, uiLSB);

     return(uiRetValue);
}
```

Listing 4. Read Humidity Sequence

```
writeSHT1xTransmissionStartSequence();
     writeSHT1xCommand(SHT1x_MEASURE_HUMIDITY);
     uiHumidity = readSHT1xData();
```

the following ioctl functions:

- RELAY8_IOC_READ_RELAYS: read the current relay settings.
- RELAY8_IOC_WRITE_RELAYS: set the relays to the supplied state.

Reading the relay settings is used to ensure that the relays are in the desired position. The relay hardware actually includes eight relays, and all eight relay values are written in one shot. The data register used to control and report the relay positions consists of one 8-bit register. This register either is read to report the current relay settings or written to change the relay settings. Unlike the SHT11 driver, the relay driver can affect a change in a relay state with one writeb Linux driver C instruction. Listing 5 shows the relay read and write procedures, along with an excerpt from the ioctl processing that differentiates between read and write. It doesn't get much simpler than this!

Listing 5. Read/Write Relays

```
unsigned char readRelay8(int iRelay8Address)
{
     /* Read Relay8 register and return the value */
     return(readb(iRelay8Address));
}

void writeRelay8(int iRelay8Address, unsigned char ucValues)
{
     /* Write Relay8 register with the values */
     writeb(ucValues, iRelay8Address);
}

// Excerpt from ioctl function:
     switch(cmd) {

     case RELAY8_IOC_READ_RELAYS:
          /* Read Relay8 relay values */
          ucRelayValues = readRelay8(relay8_base + RELAY8_CONTROL);
          if (copy_to_user((typeof(relay8_relayValues)) arg,
                           &ucRelayValues,
                           sizeof(relay8_relayValues))) {
              ret = -EFAULT;
          }
          break;

     case RELAY8_IOC_WRITE_RELAYS:
          /* Write Relay8 relay values */
          writeRelay8(relay8_base + RELAY8_CONTROL,
                      *(typeof(relay8_relayValues)) arg);
          break;

     default:
          ret = -ENOTTY;
     }
```

User-Mode Application

I wrote a user-mode application that periodically polls the SHT11 device driver for the current humidity and temperature using the ioctl SHT1X_IOC_READ_HUMIDITY and SHT1X_IOC_READ_TEMPERATURE, respectively. Depending on the desired humidity setting, the application determines whether the current humidity is either too high or too low, taking into account the tolerance of plus or minus 3.5% rH. If an actionable event is determined, the specific relays are turned either on or off using the relay device driver RELAY8_IOC_WRITE_RELAYS ioctl function. For example, when the user-mode application reads the humidity and determines that the humidifier must be turned on, it issues an ioctl RELAY8_IOC_WRITE_RELAYS function to switch on both relays that are dedicated to the 120V A/C and neutral lines of the humidifier. At the same time, the application also ensures that the two relays associated with the 120V A/C and neutral lines of the dehumidifier are switched off. Relay control can be one of three options: 1) the humidifier is turned on, and the dehumidifier is turned off; 2) the dehumidifier is turned on, and the humidifier is turned off; or 3) both the humidifier and dehumidifier are turned off. The application never turns both the humidifier and dehumidifier on at the same time. The application is loaded at Linux boot time and, like most embedded applications, runs perpetually.

Along with controlling the humidifier and dehumidifier relays, the application accumulates and saves statistics. In this control system, the actual data that is acted upon is required to be persistent—that is, the humidity data must be saved somewhere for later use. The user-mode application is responsible for saving the data for later use by a browser, and it does so with the use of the SNMP (Simple Network Management Protocol) support provided by the net-snmp Linux package.

SNMP is a standard set of protocols and policies for managing networks and devices. The net-snmp implementation of SNMP consists of an agent, which runs as a Linux dæmon snmpd, and a database called a Management Information Base, or MIB. A MIB is structured as a tree, with branches grouping together similar items. I extended the standard Linux MIB that is shipped with the net-snmp package and added a new branch off of the "enterprises" node, which includes all the humidity controller items that I need (Figure 4). The snmpd agent acts on the MIB at the request of SNMP clients—that is, the agent reads/writes data from/to the MIB on behalf of client get and set requests. In this architecture, there are two clients: the user-mode application and the Web browser.

In order to adapt SNMP to any application, a MIB must be defined in a standard MIB ASN.1 format. I defined a MIB for my humidity controller and called it HUMIDITYCONTROLLER-MIB, which gets loaded when the snmpd dæmon runs during the Linux boot process. The MIB contains data items that are represented by object identifiers, or OIDs. An example of an OID definition from my MIB for the humidity controller targetHumidity variable is shown below:

```
targetHumidity OBJECT-TYPE
    SYNTAX      Integer32(0..2147483647)
    MAX-ACCESS  read-write
    STATUS          current
    DESCRIPTION
    "Target humidity."
    ::= { humidityEntry 3 }
```

Listing 6. mib2c-Generated C Code

```
netsnmp_register_scalar(
    netsnmp_create_handler_registration(
        "targetHumidity",
        handle_targetHumidity,
        targetHumidity_oid,
        OID_LENGTH(targetHumidity_oid),
        HANDLER_CAN_RWRITE));


int
handle_targetHumidity(netsnmp_mib_handler           *handler,
                      netsnmp_handler_registration  *reginfo,
                      netsnmp_agent_request_info    *reqinfo,
                      netsnmp_request_info          *requests)
{
    switch (reqinfo->mode) {

    case MODE_GET:
        break;

    case MODE_SET_RESERVE1:
        break;

    case MODE_SET_COMMIT:
        break;
    }

    return SNMP_ERR_NOERROR;
}
```

The previous ANS.1 MIB definition phrase generates an OID with the value .1.3.6.1.4.1.2200.2.3. This rather cryptic-looking sequence of numbers is a scheme used to identify a leaf in the MIB tree. The branch that I added to the enterprises node is identified by the integer 2200. Under the 2200 node is the node identified by a 2, which contains all of the overall humidity controller items that the system needs. The leaf node identified by a 3 is the targetHumidity.

The Linux SNMP package contains a very useful tool called mib2c. mib2c takes a MIB definition, such as HUMIDITYCONTROLLER-MIB, and generates C code that can be used to extend the standard Linux snmpd agent. Several options exist when generating code with mib2c. I used the more general option for generating C code from a MIB with the mib2c.scalar.conf configuration, which causes code to be generated for general-purpose scalar OIDs, as opposed to table-based OIDs. The generated C code is used by the snmpd dæmon. Listing 6 is a distilled example of the generated C code from mib2c for the targetHumidity OID that shows the code framework needed to support the SNMP get

To finish off the humidity controller, I added a Web page interface that includes a recipe that uses a tad of HTML, a smattering of JavaScript and a pinch of AJAX with server-side scripting to create an end-user browser interface.

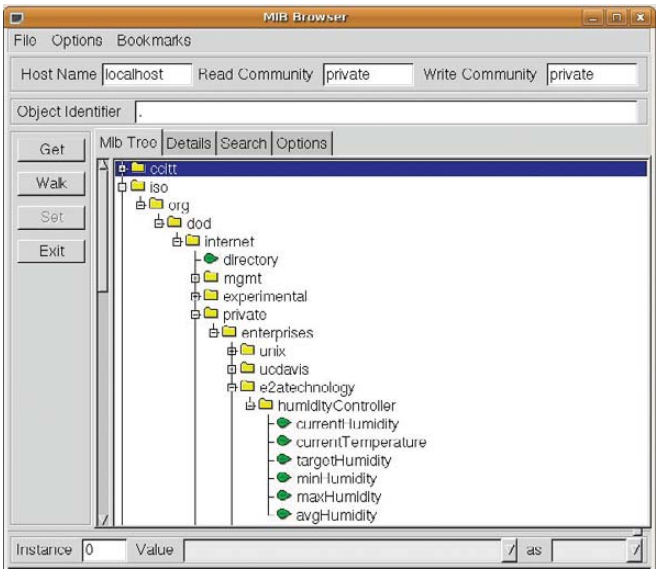

Figure 4. mbrowse Screenshot

Listing 8. Perl Script setTargetHumidity

```
use CGI;
$query = new CGI;
$targetH = $query->param('targetH');
$SNMP_SET_CMD = "snmpset -v 1 -c private";
$SNMP_TARGET = "localhost";
$SNMP_TARGETHUM_OID = "targetHumidity .0";
$SNMP_TYPE = "i";

chomp($retVal = `${SNMP_SET_CMD} ${SNMP_TARGET}
      ➥$SNMP_TARGETHUM_OID $SNMP_TYPE $targetH`);
```

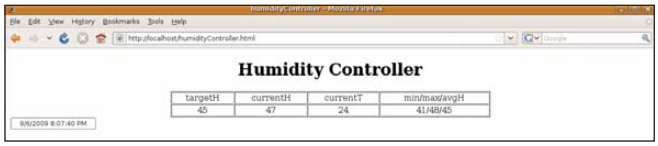

Figure 5. Humidity Controller and Firefox

(MODE_GET) and SNMP put (MODE_SET_RESERVE1 and MODE_SET_COMMIT) operations.

The code shown in Listing 6 makes reference to the generated callback procedure, handle_targetHumidity, which is supplied in skeletal form only by mib2c. Not much code is needed in order to support scalar OIDs, which the humidity controller uses exclusively. Anytime a specific OID, in this case the targetHumidity OID .1.3.6.1.4.1.2200.2.3, has an operation performed, the snmpd dæmon will invoke this callback procedure with an indication of the requested operation being performed on the OID.

I rebuilt the snmpd dæmon so that the newly created humidity controller MIB structure and generated framework code could be supported. Before rebuilding the snmpd dæmon, the new MIB must be configured into the build environment. This is accomplished easily with the following command:

```
$ ./configure --with-mib-modules="humidityController"
```

Once configured, the entire net-snmp package was rebuilt with the make command. Once the snmpd dæmon was rebuilt, I tested the new MIB structure by using the net-snmp command-line interface utilities snmpset and snmpget. For example, in order to set the targetHumidity OID to 50% rH, the following command can be issued:

```
$ snmpset -Ovqe -v 1 -c private localhost targetHumidity.0 i 50
```

Note the use of relative, symbolic OIDs in the snmpset command. The actual OID .1.3.6.1.4.1.2200.2.3 could be used as well, because it's statically defined and should never change. But, I prefer symbolic references where possible, as it helps in readability. The -Ovqe switch controls the output format that results from the snmpset. Although I built the net-snmp package to support all three major versions of SNMP (1, 2 and 3), I really needed only basic version 1 support, which is why the -v 1 switch appears. The SNMP community string is indicated by the -c private switch and appears in set operations because only private communities are permitted to set OID values (this is a one-time option when the snmpd dæmon is configured).

The humidity controller MIB can be viewed with a tool included in net-snmp called mbrowse. mbrowse is a GUI that bolts onto the system MIB structure and permits manipulation of specific OIDs. Figure 4 shows a screenshot of mbrowse and the humidity controller MIB tree branch.

Once the snmpd dæmon was complete with support for the newly added humidity controller OIDs, I was able to complete the user-mode application code. Listing 7 contains the complete user-mode application, and it is too long to print here, but it is available on the *LJ* FTP site (see Resources). It is very typical of an embedded application, as it perpetually reads data and then takes actions on the data. Note the use of snmpget and snmpset. The net-snmp package does include APIs for both C and Perl, but I decided it was simpler to leverage the existing snmpget and snmpset utilities.

To finish off the humidity controller, I added a Web page interface that includes a recipe that uses a tad of HTML, a smattering of JavaScript and a pinch of AJAX

Figure 6. Completed Humidity Controller with Humidifier and Dehumidifier Connected

Figure 7. Carlos Pina Classical Guitar

with server-side scripting to create an end-user browser interface. The humidity controller in a Firefox browser looks like what is shown in Figure 5. The targetHumidity (targetH) cell in the table has a JavaScript function associated such that editing is possible, and when a new value is entered, it is POSTed to Apache. Apache will invoke a Perl script to set the target humidity in the SNMP MIB. Listing 8 is an excerpt from the Perl code that shows the SNMP actions. The other cells are read-only and are refreshed periodically with values from the SNMP MIB with the help of a second Perl script, humidityController.cgi. This second Perl script pushes out only the data necessary to generate the table of values shown in Figure 5.

The humidity controller (Figure 6) has been keeping my music room within a humidity range that makes my Carlos Pina classical guitar quite happy (Figure 7). The work involved to build the system was a real pleasure. But the best part is sitting down to play the opening arpeggios in Bach's *Prelude* and hearing the notes ring true without retuning my guitar. It not only makes me smile, but I think it would make Bach smile as well.■

Jeffrey Ramsey has been an embedded developer his entire career, and when not pouring through Linux kernel and driver source code, he can be found plucking a guitar. Jeffrey can be contacted at jeffreyramsey@e2atechnology.com.

Resources

Listing 7 (the Complete User-Mode Application):
ftp.linuxjournal.com/pub/lj/listings/issue188/10534.tgz

TS-7200 Main Board (from Technologic Systems):
www.embeddedarm.com/products/board-detail.php?product=TS-7200

TS-RELAY8 Daughter Card (from Technologic Systems):
www.embeddedarm.com/products/board-detail.php?product=TS-RELAY8

TS-ENC720 Enclosure (from Technologic Systems):
www.embeddedarm.com/products/board-detail.php?product=TS-ENC720