Building an SQLite temperature logger

In this article I'm going to describe how I used a Raspberry Pi to build an SQLite based temperature logging system with a web UI. Follow this link to see the completed Raspberry Pi temperature logger with web UI.

You can download the code for this project from Github: https://github.com/Pyplate/rpi_temp_logger. There's a button to download the project as a zip file at the bottom of the right hand column.

The temperature logger consists of two parts: a script called monitor.py to measure the temperature at 15 minute intervals, and a script called webgui.py that displays temperatures in a web page. Monitor.py is triggered by a cron job. Every 15 minutes, it reads the temperature from a DS18B20 connected to my Pi's GPIO pins, and stores the reading in an SQLite database.

The other script, webgui.py, executes when it is requested by the Apache web server. It queries the database and displays the readings formatted in HTML. Temperatures are displayed in a javascript chart generated by code from Google charts.

Set up the SQLite database

The first thing to do is set up a database. Install SQLite using this command:

```
sudo apt-get install sqlite3
```

Then at a terminal type this command to enter the SQLite shell:

```
$ sqlite3 templog.db
```

In the SQLite shell I entered these commands to create a table called temps:

```
BEGIN;
CREATE TABLE temps (timestamp DATETIME, temp NUMERIC);
```

```
COMMIT;
```

Temps has two fields: a timestamp, which is the date and time when a temperature is entered, and the other field is used to store the temperature. The BEGIN and COMMIT commands ensure that the transaction is saved in the database.

Use the .quit command to exit the SQLite shell, and then use these commands to put the database in /var/www, and set the database's owner to www-data:

```
$ sudo cp templog.db /var/www/
$ sudo chown www-data:www-data /var/www/templog.db
```

The Apache daemon has its own user name, www-data. I changed the database file's owner from pi to www-data so that Apache can read the file.

Writing the monitor script

Monitor.py reads temperatures from a DS18B20 on a breadboard connected to GPIO, and stores them in an SQLite database. The script starts by loading kernel modules for reading from 1 wire devices. Next, it searches for a directory in /sys/bus/w1/devices. The DS18B20 is represented by a directory starting with the digits '28', so searching for /sys/bus/w1/devices/28* finds the device path. To find the device file, we just append '/w1_slave' to the device path.

I wrote a function called get_temp to read from the device file:

```
# get temerature
# argument devicefile is the path of the sensor to be read,
# returns None on error, or the temperature as a float
def get_temp(devicefile):

    try:
        fileobj = open(devicefile,'r')
        lines = fileobj.readlines()
        fileobj.close()
    except:
        return None
```

```
    # get the status from the end of line 1
    status = lines[0][-4:-1]

    # is the status is ok, get the temperature from line 2
    if status=="YES":
        print status
        tempstr= lines[1][-6:-1]
        tempvalue=float(tempstr)/1000
        print tempvalue
        return tempvalue
    else:
        print "There was an error."
        return None
```

When read, the DS18B20 returns a two line string. If the device was read successfully, the end of the first line contains the letters 'YES'. The last five digits on the second line are the temperature in degrees Celsius. I converted them to a float and divided by 1000 to display the temperature with a decimal point in the correct place. If everything goes well, get_temp returns the temp as a float, otherwise it returns None.

The call to get_temp fails sometimes, most likely due to noise picked up by the jumper cables. If get_temp returns None, then we just call it again. See this page for detailed information on reading a DS18B20 temperature sensor on a Raspberry Pi.

Storing readings in the database

Monitor.py contains a function called log_temperature which stores readings in the database. This function connects to the database and creates a cursor. The cursor object is used to execute an SQL command to insert the temperature as a number along with the current date and time. Finally, log_temperature commits the transaction to the database and closes the connection.

```
# store the temperature in the database
def log_temperature(temp):

    conn=sqlite3.connect(dbname)
    curs=conn.cursor()
```

```
    curs.execute("INSERT INTO temps values(datetime('now'), (?))", (temp,))

    # commit the changes
    conn.commit()

    conn.close()
```

See this page for more information on accessing a database in Python.

Installing the monitor script

I saved the script as monitor.py in /usr/lib/cgi-bin/. I used this directory because it's the standard directory for executable scripts in Apache. Note, you have to configure Apache to execute .py files.

The following commands give monitor.py executable permissions, and change its owner to Apache's user name, www-data:

```
$ sudo chmod +x /usr/lib/cgi-bin/monitor.py
$ sudo chown www-data:www-data /usr/lib/cgi-bin/monitor.py
```

Now a cron job needs to be set up to trigger monitor.py every 15 minutes. I did this by editing the user crontab file for www-data. This command opens the crontab file in the nano editor:

```
$ sudo crontab -u www-data -e
```

Crontab can be used with other editors, but nano is the default. I added this line at the end of www-data's crontab:

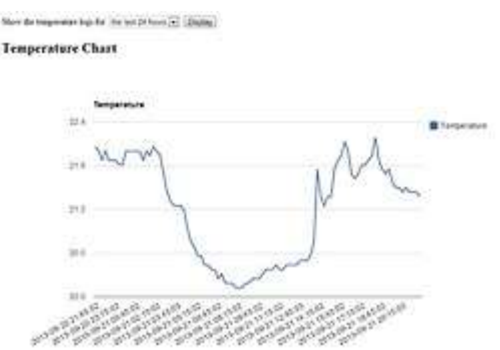
```
*/15 * * * * /usr/lib/cgi-bin/monitor.py
```

To save the file, I pressed Control O, and just hit return when prompted for the file name. The updated file will be saved to a temporary file and then automtically installed in the proper place. Press Control X to exit nano.

In the next post, I'll describe how I built the web UI for the Raspberry Pi temperature logger.

http://raspberrywebserver.com/cgiscripting/rpi-temperature-logger/

Building a web user interface for the temperature monitor



In my last post, I wrote about [logging temperatures in an SQL database](). In this article, I'm going to build a [web based user interface]().

You can download the code for this project from Github: https://github.com/Pyplate/rpi_temp_logger. There's a button to download the project as a zip file at the bottom of the right hand column.

The UI is generated by a script called webgui.py. When webgui.py executes, it searches the database and returns a list of records. Webgui.py begins execution in the main function. The first thing it does is call get_option() to see if any options were passed to the script, and recover them if available. The first time a user opens the UI in their browser, no options will be passed to the script. As you'll see a few paragraphs down, users can select an option in the UI and reload the script. When this happens, an option is passed to the script, and get_option() returns that option so that it can be used in the rest of the script.

Next, webgui.py searches the database for records within the time limit imposed by any options that were passed. Function get_data() handles this:

```
# get data from the database
# if an interval is passed,
# return a list of records from the database
def get_data(interval):

    conn=sqlite3.connect(dbname)
    curs=conn.cursor()

    if interval == None:
        curs.execute("SELECT * FROM temps")
```

```
    else:
        curs.execute("SELECT * FROM temps WHERE
             timestamp>datetime('now','-%s hours')" % interval)

    rows=curs.fetchall()

    conn.close()

    return rows
```

It connects to the database and creates a cursor. If no option was passed to the script, then all records in the database are returned. If a time limit was specified, the database is queried for records where the timestamp is greater than the current time minus the interval.

Print the HTTP header

Before anything else happens, the HTTP header is sent to the browser. This is just a string that tells the browser that we're about to send it some HTML:

```
Content-type: text/html\n\n
```

There must be a blank line after the header, so the string is terminated with two new line characters.

The list of records returned by get_data is then passed to create_table(), a function that formats the data as a javascript table as follows:

```
['2013-09-19 10:30:03', 20.062],
['2013-09-19 10:45:02', 20.687],
['2013-09-19 11:00:02', 21.125]
```

This table will be embedded in a javascript snippet later on. Note that the last line in the table does not have a comma at the end. This is important because of the format of the code that the table is embedded in.

Printing the HTML head section

Now we're ready to print the rest of the page. We start with the HTML tag, and then the head section. The head section of a web page contains meta data, javascript and CSS styling. In this example, there's no need to use meta data or CSS, so we just need to print the page title tags and the javascript for the Google chart. The javascript

snippet is printed by function print_graph_script(). The table generated earlier is passed to print_graph_script() so that it can be embedded in the javascript code.

```
    <script type="text/javascript"
src="https://www.google.com/jsapi"></script>
    <script type="text/javascript">
      google.load("visualization", "1", {packages:["corechart"]});
      google.setOnLoadCallback(drawChart);
      function drawChart() {
        var data = google.visualization.arrayToDataTable([
          ['Time', 'Temperature'],
['2013-09-19 20:15:02', 21.187],
['2013-09-19 20:30:02', 21.375],
['2013-09-19 20:45:02', 21.625],
['2013-09-19 21:00:02', 21.812],
['2013-09-19 21:15:02', 21.875],
['2013-09-19 21:30:02', 22],
['2013-09-19 21:45:02', 22.187],
['2013-09-19 22:00:02', 22.062],
['2013-09-19 22:15:03', 22.125]

        ]);

        var options = {
          title: 'Temperature'
        };

        var chart = new google.visualization.LineChart
                        (document.getElementById('chart_div'));
        chart.draw(data, options);
      }
    </script>
```

Printing the page body

Near the top of the UI, there's a drop down list where users can select whether they view data from the last 6, 12 or 24 hours. The list is an HTML form (see http://raspberrywebserver.com/cgiscripting/web-forms-with-python.html for more information). When the submit button is pressed, the script runs again and the chosen value is passed to the new instance of the script. Function get_option() uses Python's CGI library to get the value of the option. When the list is displayed again, it

needs to be displayed with a default value to indicate which option was selected. This is done by embedding 'selected="selected"' in the string for the chosen option:

```
    <form action="/cgi-bin/webgui.py" method="POST">
        Show the temperature logs for
        <select name="timeinterval">
            <option value="6" selected="selected">the last 6 hours</option>
            <option value="12">the last 12 hours</option>
            <option value="24">the last 24 hours</option>
        </select>
        <input type="submit" value="Display">
    </form>
```

Displaying the temperature chart

The code for the chart appears in the head section, but the chart itself is drawn in the page body. The javascript code references an HTML div called chart_div. The function show_graph() prints this line of HTML:

```
<div id="chart_div" style="width: 900px; height: 500px;"></div>
```

When the entire page is loaded in a browser, the javascript code executes and draws the chart in chart_div.

Display statistics

The final function to execute is show_stats. This function connects to the database, creates a cursor, and runs three queries. The queries search the database for the minimum, maximum and average temperatures.

```
    conn=sqlite3.connect(dbname)
    curs=conn.cursor()

    curs.execute("SELECT timestamp,max(temp) FROM temps
                 WHERE timestamp>datetime('now','-%s hour')" % option)
    rowmax=curs.fetchone()
    rowstrmax="{0}   {1}C".format(str(rowmax[0]),str(rowmax[1]))

    curs.execute("SELECT timestamp,min(temp) FROM temps
                 WHERE timestamp>datetime('now','-%s hour')" % option)
    rowmin=curs.fetchone()
    rowstrmin="{0}   {1}C".format(str(rowmin[0]),str(rowmin[1]))
```

```
curs.execute("SELECT avg(temp) FROM temps
                 WHERE timestamp>datetime('now','-%s hour')" % option)
rowavg=curs.fetchone()
```

The first two queries use the max() and min() SQL functions to find the minimum and maximum entries in the database. They return a record containing the timestamp and a numeric temperature value. The third query in show_stats uses the avg() function to find the average value in the database. This query just returns a record containing the average time without a timestamp. All of these queries select records where the timestamp is greater than some offset from the current time. The data returned from the three queries is formatted and printed.

Show_stats executes another query to get records from the last hour. These records are displayed as a table at the bottom of the page.

It's not practical for me to leave the temperature logger set up indefinitely (I need to use my Pi for other things), so I've modified the SQL queries in the scripts that are used to display the temperature logger on this site. I unplugged the breadboard at around 21:30, so I replaced 'now' with the timestamp '2013-09-19 21:15:02'. Instead of selecting records that are greater than 'now minus an interval', I modified webgui.py to select records where the timestamp is greater than '2013-09-19 21:15:02' minus an interval AND less than '2013-09-19 21:15:02'.

In webgui.py, there are several lines that contain hardcoded dates so that you can use the script with the sample database provided. There is an equivalent version of each of these lines that uses 'now' instead of a hardcoded timestamp. If you want to view data you've collected yourself, you should uncomment the lines that use 'now', and comment out the lines that have a hardcoded date. See webgui.py, lines 45, 117, 122, 127 and 148.

See also:

- http://docs.python.org/2/library/sqlite3.html
- http://sqlite.org/lang_aggfunc.html
- http://www.sqlite.org/lang_datefunc.html
- http://www.tutorialspoint.com/sqlite/sqlite_useful_functions.htm