Digital Raspberry Pi temperature and humidity (& pressure) logger/webserver

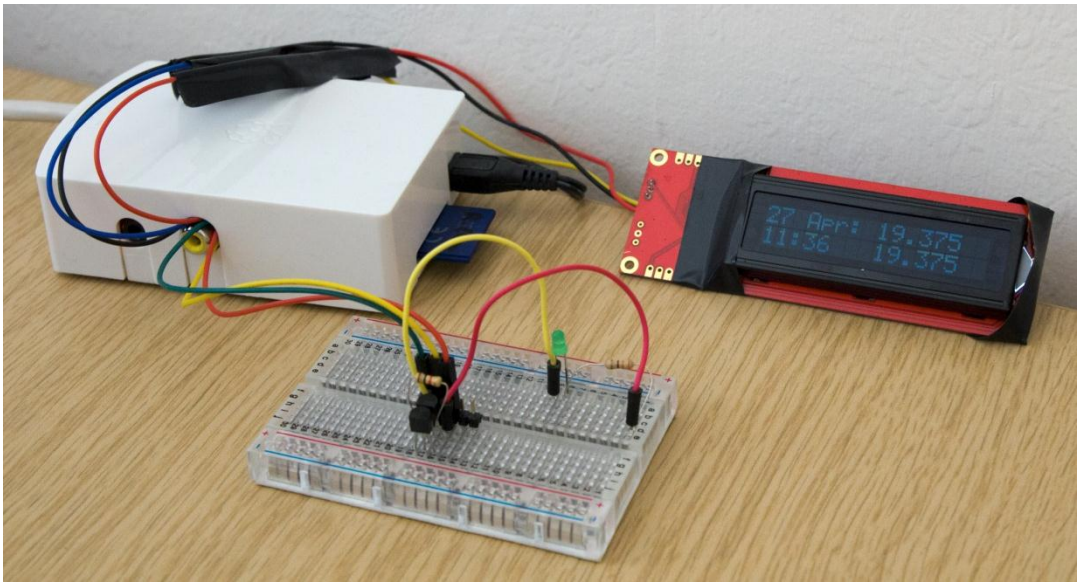


Figure 1- Raspberry Pi logger - interim build using Sparkfun serial LCD, 2xDS18B20s and an LED

Background

The Raspberry Pi is a single board, low cost, low power computer system running Linux (or several other operating systems) on an ARM processor architecture. The system boots from an SD card and an 8 GB SD card is more than enough to store the operating system and lots of data. The power requirements are very modest and a simple 5 V phone charger with micro USB plug is enough to run the system. The software is very stable (many units have been running non-stop for several months) and there are many general purpose I/O pins (GPIO) available for interfacing to external devices. The device is a reasonably powerful[1] computer on a credit card sized PCB.

Why build yet another small Pi-based temperature measuring system? I wanted a relatively cheap, fairly high resolution temperature logger for use at work where our temperature – controlled labs are held between 19.9 °C and 20.1 °C – the ±0.1 ° range is smaller than many temperature logger units' resolution, but I want to look for slight trends in the data at this level. In particular I wanted to detect any excessive heating or cooling and to generate automated alerts for such occurrences. Further, I wanted to have remote access to see what was happening and to have long-term storage of data for archival and laboratory best-practice (due to our accreditation status).

For an explanation why we need accurate temperature control, try chapters 7 and 8 of my thesis.

Also I wanted to learn a bit of another computer language (in this case Python) as well as re-visit some of my previous experiences of Linux. The Raspberry Pi along with some reasonably cheap sensors fulfils all these requirements.

[1] From www.raspberrypi.org/faqs: "… graphics capabilities are roughly equivalent to Xbox 1 level of performance. Overall real world performance is something like a 300MHz Pentium 2, only with much, much swankier graphics."

Parts

To build one of these systems, you need a Raspberry Pi model B (since you really need the network socket and the model A does not have one, only USB – so you could use a WiFi adaptor or USB-based network socket with a model A, but it is overall easier to use a model B); the temperature and humidity sensors; a power supply; an SD card; an LCD display to show live data; some form of web server software to give access to live data; and some form of scripting to keep it all together and run it. Optionally, but recommended are the niceties of a decent case, proper board mounting, ribbon cables and sockets/connectors for the interfaces. You can run a Pi without the case *etc*., but it might be less reliable if connections short together or wires break. The case and board mount give a secure base for everything. You can also miss out the LCD if you just want to read the temperatures into the Pi and perhaps make them available via the web server.

I looked around for various options and suppliers, but at work we have a default supplier of RS Components so that is where I ordered most of the parts from. For just over £100 you can get all the parts except pre-soldered/mounted temperature sensors (available from eBay), however these can be wired up with bare sensors and any suitable 3 way cable and some heat shrink tubing. If you want to save money, you can ditch the case, the Vero board, the ribbon cable and all the nice sockets and try lashing it together with solder and sticky tape for about £60. If you leave out the rather expensive humidity sensor, a simple single channel temperature logger without LCD can be made for around £50 Inc. VAT, with direct connection to the GPIO pins on the Pi using some cheap cables. A completely bare bones system needs the Pi, SD card, a suitable power supply (e.g. phone charger with micro USB) and at least one DS18B20: if you have the phone charger already and an SD card of at least 2 GB capacity, you will be spending about £26 + VAT on the remaining items.

Parts from RS (excluding any negotiated discount)	Part No.	ex VAT	
Raspberry Pi Model B	7568308	£21.60	x1
Micro USB power supply adapter, UK, 1.2A	7263069	£3.89	x1
Un-mounted DS18B20 temperature sensor	5402805	£4.17	x2 (or below)
SHT75 humidity sensor	6675271	£21.80	x1
16x2 transflective STN blue/white, 80x36 LCD display	5326436	£6.17	x1
Black Aluminium case with sliding top and removable end caps	7732981	£12.18	x1
8 GB Class 10 SDHC card	7582574	£11.53	x1
Vero board 95 mm wide	5280661	£6.83	x1
Straight 26 way IDC header for Vero	4738298	£1.23	x1
IDC 26 way connectors	458516	£3.59	x2
IDC cable 26 way (5 m)	2899925	£7.96	(part of)
Vero mount 3 way screw terminal	2204276	£0.69	x2 (comes as 5)
Narrow pitch sockets for SHT75 sensors	7023041	£0.35	x1 (comes as 5)
3 way PCB header	4838477	£0.23	x2 (comes as 10)
3 way socket housing	2964940	£0.21	x2 (comes as 5)
crimp pins	467598	£0.059	x10 (comes as 100)
4 way PCB header	4838483	£0.40	x1 (comes as 10)
4 way socket housing	2964956	£0.25	x1 (comes as 5)

Parts from Amazon (or make up directly)
Long lead DS18B20 sensors
Amazon: B008RIOGP2
eBay: http://www.ebay.co.uk/itm/10x-3M-55-to-125-DS18B20-Waterproof-Digital-Temperature-Probe-Sensor-/370802007896?pt=UK_BOI_Electrical_Components_Supplies_ET&hash=item5655859f58
or use un-mounted ones above with 4 way low power cable, RS 1681105 £35.66 (comes 30 m)

Miscellaneous from any electronics store
Also need a 10 k ohm pull up resistor for the SHT75 (one per SHT)
Also need a 4k7 k ohm pull up resistor for the DS18B20 sensors (one per Pi)

Photographs of the main parts

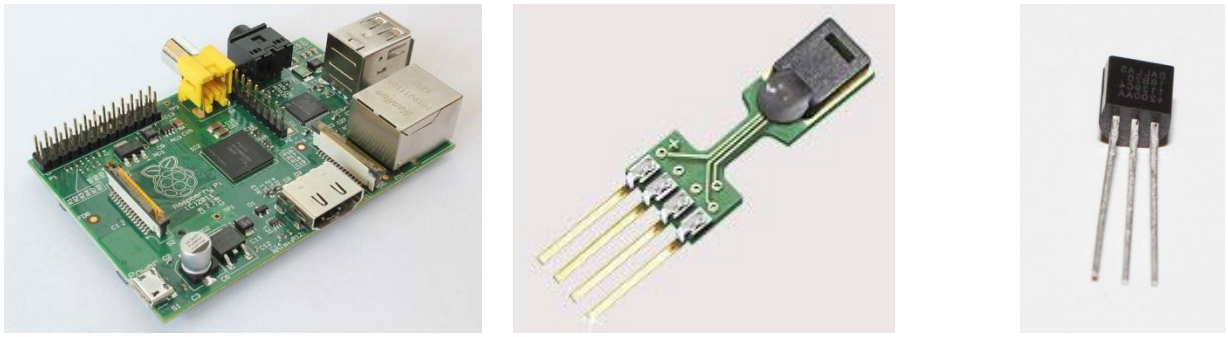


Figure 2 - Raspberry Pi, SHT75 humidity sensor, and DS18B20 temperature sensor

Figure 3 - 16 x 2 display, aluminium case, Veroboard

Software/hardware design process

Probably the longest delay in this project was working out what software to use to achieve all the goals. As well as software, hardware also played an important part in the overall design process. The problem was that no one piece of software did all that I wanted and some items of software depended on specific hardware. It took some time to sort my way through the options, trying out different routes to see how they worked and which items were compatible with one another – for example, could I run both an LCD display and some 1-Wire sensors at the same time? Conversely, certain items of hardware naturally leaned towards the use of specific software. So the software design and hardware design processes actually merged into one. This made it a bit harder to find the optimum solution.

Requirements

- Sense temperatures with better than 0.1 °C resolution.
- Sense humidity (mostly as % RH) with 1 % RH to 5 % RH resolution.
- Store readings onto SD card in a format that retains original resolution and allows relatively easy retrieval.
- Display web graphs of temperature trends especially the last day or week or two weeks.
- Allow for printouts of graphs.
- Detect out of tolerance conditions and send email alerts to defined recipients.

Since no one software module could do all of these, it seemed that I would have to combine several modules with some scripting. The Pi allows for a wide range of programming and scripting languages to be used. But what language to use?

Essentially the software options included:

- C or C++ scripting/programming
- Python scripting
- Perl scripting
- Lighttpd or Apache webserver
- Nettemp
- Digitemp
- Cacti
- Dygraphs
- Cosm external data store

Hardware options included:

- Digitemp-compatible USB 1-Wire interface
- Direct 1-Wire access via GPIO pins
- SPI interface to LCD or sensors
- I2C interface to LCD or sensors
- Serial port driven LCD
- Classic Hitachi 44780 LCD interface via GPIO pins

With hindsight it is quite easy to see the optimum way forwards but at the time, trying to learn Raspberry Pi's Debian O/S, some Python and a bit of Perl as well as the PHP source code of Nettemp, it seemed like there were too many options to consider. A particular problem was that certain hardware items were incompatible with each other and this led to me buying two Raspberry Pi devices so that I could evaluate hardware in parallel, without having to keep uninstalling everything all the time. At £25 each, this is a nice feature to exploit!

The choice of sensor was pretty easy. After some initial experiments with SHT1X and similar devices, I realised they had limited resolution and it was not easy to have several devices connected at once. Far better were the Dallas DS18B20 units, each one individually readable on the 1-Wire bus, and with sub 0.1 °C resolution and factory calibrated 0.5 °C accuracy. A bit further reading, actually at a later stage in the project, but valid nevertheless, revealed that the SHT75 sensors are probably the best ones for higher resolution and accuracy humidity measurement. They include sensors for both humidity and temperature, since the calculations to convert between relative humidity and dew point require a temperature value as well.

I tried using the Nettemp software for a while but it had the disadvantage that it relied on Digitemp which in turn required certain USB-based 1-wire adaptors. These cost around £25 each. Nettemp source and display are in Polish, so I had a go over Christmas at re-writing bits in English to understand the operation. I got it working but eventually decided that the reasons I had chosen Nettemp: email alerting, onscreen graphics, configurable, were all achievable using some Python scripting in association with Cacti – there was no need to join these functions into a single software module – they could all exist separately. I abandoned Nettemp. As of writing in May 2013, Nettemp now supports GPIO mounted DS18B20 sensors – I could have waited and then used the latest version, but it's always easier to modify your own code than adapt existing systems written for a different purpose, especially when it comes to debugging.

I experimented for a while using a Sparkfun serially addressed LCD unit, so that I could conserve the GPIO pins for other uses. This device takes just ground, power and a single data line and performs on-board conversion from serial commands into LCD display codes using a custom programmable chip embedded on the board. However, to get this to work requires some deep delving into the operating system and turning off the default serial port which is where the Pi normally send debugging and status information during boot time. That was not too difficult to achieve, but, coupled with the extra cost and size of the display, made it less attractive than using a generic LCD based on the Hitachi 44780 chipset, for which several Python libraries had already been written. The downside of using a 44780 chipset LCD is that it needs 6 GPIO pins to drive it, in addition to 5 V power and ground. In other experiments, I worked out that I only needed 3 pins to drive the DS18B20

temperature sensors and 4 pins to drive the SHT75 humidity sensors, and this left enough for the 44780 LCD. But the decision also needed to take into account the software aspects.

I started off with some simple Perl and Python scripts found on various websites. They could read a 1-Wire sensor fine and display the values on the terminal, and another script could be used to address either a serial LCD or GPIO LCD unit. Combining the two allowed for temperature read/display capability. Adding in a bit more code allowed for detection of IP address and current time and date, for further output to the LCD.

The main system software I settled on is the Raspbian build of Debian. This is available directly from the Raspberry Pi foundation as an SD Card image that needs to be written to the SD card. On top of Raspbian, we need to run a web server, some Python scripting to read the sensors and update the LCD. Aside from this simple scripting, the Cacti software module is installed and this produces the web pages which are served up by Apache. Cacti sets up regular logging and produces interactive graphs of data over various time frames. It stores the data in a Round Robin Database (RRD file) which stores multiple averages of data, averaged over different time frames, *e.g*. minute averages for 1 day, hourly averages for a week, *etc*. Because this data is always being averaged, the total amount of data stored in the RRD file remains the same: as newer high resolution data becomes available, older data loses time resolution as it gets averaged, thereby taking up less space. This is good from a data display and storage perspective but not so good if you want to look back several years and see the original data, because it will have been averaged out. Instead, this is why an additional simple script logs the raw data to a simple text file. The data is logged with a time stamp, followed by the temperature and humidity data. All of the Python scripting is automated by setting up a **crontab** job (a task that is run at nominated regular intervals).

So, with all the software ready and the hardware design completed, how do we make one of these devices?

Manufacturing instructions

Hardware

The hardware design is based on a single board of Vero (strip board) cut down to the right size to fit the case, with the Raspberry Pi mounted onto it using short standoff pillars I found lying around. Sockets for the various sensors are also mounted on the Vero, as is a socket for the IDC ribbon cable (26 way) that neatly routes the signals from the Pi to the Vero. Some wires on the top of the Vero swap the signals to the correct locations for the sensor sockets and get them in a nice line for taking to the LCD. The nicest way to send all the LCD signals to the LCD is using correctly spaced ribbon cable – I found some 6 wire cable with the right spacing - but a selection of single strand wires is sufficient.

- Cut out the case panels as per diagrams – end panel cut for Raspberry Pi connectors, sensor connectors/holes and power feed through; top panel for LCD display.
- Cut Vero to 120 mm length so that it will fit in the case. Observe the number of columns as per figure 5 as well as the number of rows.
- Cut the Vero tracks as indicated in the diagram. This is necessary because the ribbon connector brings two signals to each Vero track so we have to split the tracks underneath the connector. Also we use the tracks on the left side to be different from those near the ribbon connector so some small splits are required there to. The splits away from the ribbon connector can be made using a track cutting tool (like a drill bit) whereas those underneath the middle of the ribbon connector will need a sharp knife, scalpel or a Dremel tool.
- Mount Raspberry Pi onto Vero ready to slide into new case to check the fit with the case end plate. Use insulating standoff pillars to avoid electrical contact between the Pi and the circuit board. Remove and continue soldering.
- Mount the IDC plugs onto the ribbon cable and connect the Pi to the Vero using the ribbon – cut to a suitable length for a neat routing.

- Solder the screw terminals or push fit terminals for sensors.
- Solder the wires that route signals around the board.
- Solder the two resistors – these are pull-up resistors for the data connectors for the humidity and temperature sensors.
- Solder the ribbon cables or wires for the LCD. This is best done last as otherwise the LCD and the wires get in the way during the other steps.

If wiring directly rather than using the Vero board, simply use a bit of ribbon cable with the appropriate connector to bring the GPIO lines off the Pi and then connect the DS18B20 sensor(s) as per the following (the pin out is from http://blog.petrockblock.com/2012/07/03/snesdev-rpi-a-snes-adapter-for-the-raspberry-pi/):

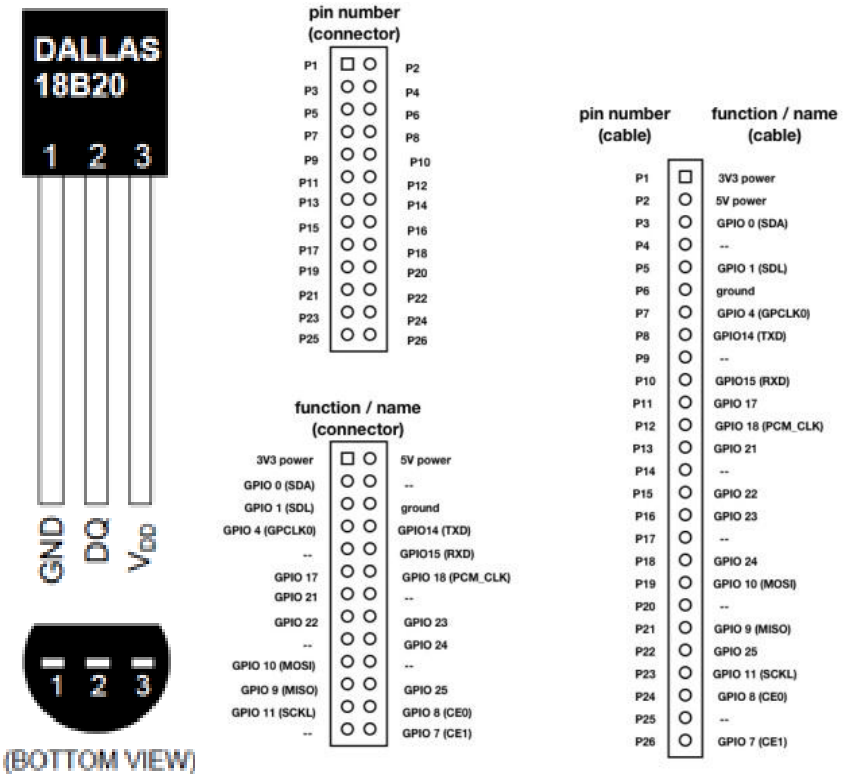


Figure 4 - DS18B20 and R-Pi pinouts

- Connect pin 1 (GND) on the DS18B20 to pin 6 (GND) on the Pi.
- Connect pin 2 (DQ) on the DS18B20 to pin 7 (GPIO4) on the Pi.
- Connect pin 3 (V_{DD}) on the DS18B20 to pin 1 (3V3) on the Pi.
- Finally, put a 4k7 resistor between Pins 2 and 3 on the DS18B20.

Below is an exported Fritzing image showing the strip board with all the components – this is for those who want to use strip board. If building a simpler version, then all you need to be able to measure temperatures is just to connect the three pins of the DS18b@0 to the relevant pins on the R-Pi and join the 3V3 and data pins with a 4k7, *i.e.* 4700 ohm, resistor. If using the pre-wired DS18B20 sensors available from eBay (example suppliers: this or this) then you need some way to attach to the R-Pi – for example consider the AdaFruit Cobbler breakout kit from Amazon.co.uk which brings the Pi's GPIO outputs to a small board with pins which may be easier to solder to. Assembly instructions are on the AdaFruit website:
http://learn.adafruit.com/adafruit-pi-cobbler-kit/solder-it

When using ribbon connectors – be sure to get the orientation correct – always put the red colour wire of the ribbon at the end marked pin 1 or the end with the small triangle symbol, when mounting the ribbon into the connector.

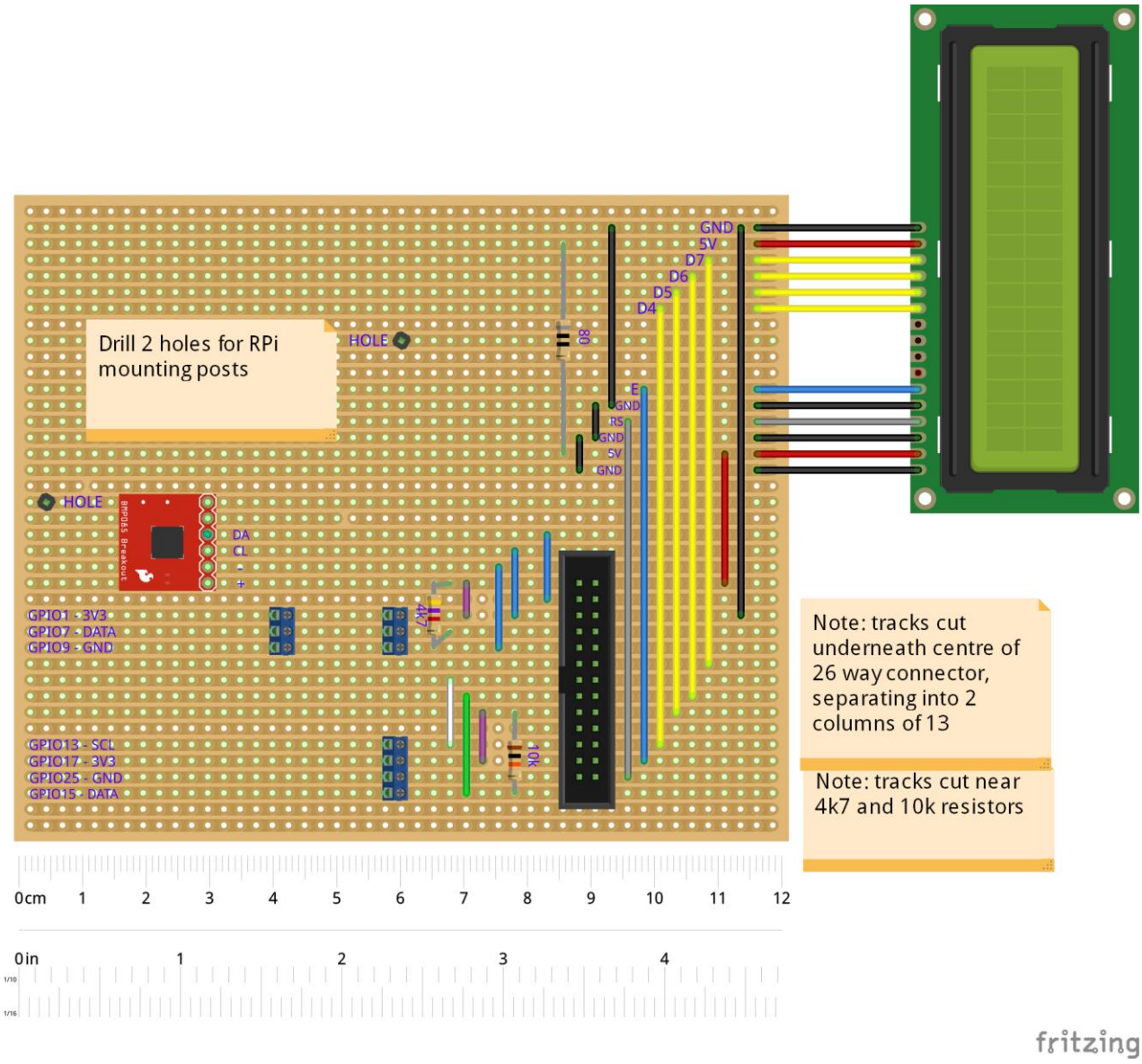


Figure 5 - Strip board layout. Copper tracks are actually on the underside, but they are shown here on the topside to indicate layout.

The first version of the layout did not include the 80 ohm resistor on the LCD backlight power line – it used a direct wiring from the 5 V supply. The first few copies of this device quickly burnt out the LED backlight, and I burnt my fingers on the LED once. Whoops – forgot the current limiting resistor! New in V3 – locations and connections for a SparkFun BMP180 or BMP085 breakout board pressure sensor.

The case requires a few holes to be cut or milled out. The top panel needs a rectangular hole for the LCD panel together with the four corner holes for the LCD mounting screws. One removable end panel needs holes for the USB and Ethernet sockets, the power cable and the sensors. After a bit of experimentation, it works out OK if the Veroboard is slid into the lowest channel moulded into the side walls of the case and then the dimensions given in figures 6 are OK for the end plate modifications. The LCD mounts in the top panel and location is unimportant provided the cables to it from the circuit board are long enough. If they are too long they are difficult to squeeze into the box.

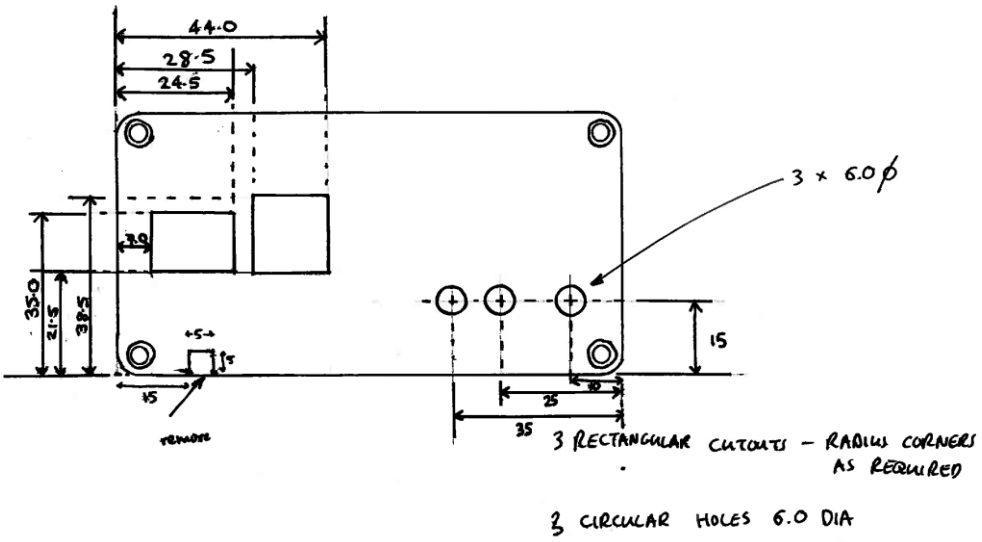


Figure 6 - The engineering modifications to the end panel (sketch)

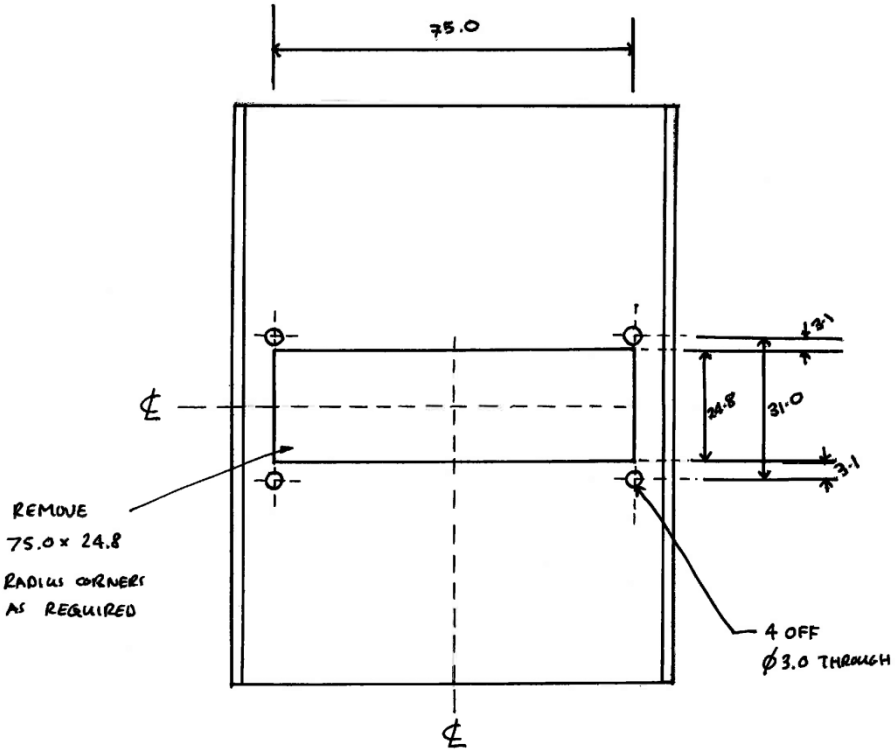


Figure 7 - The engineering modifications to the top panel (sketch)

Software

Installing the operating system

First step is to install the O/S. This project uses **Raspbian** – in particular any version dated 2013-02-09 (as used here) or later is fine since they will include the Python GPIO modules already built in.

Download the latest Raspbian image from `http://www.raspberrypi.org/downloads`. Be sure to download the normal one and not the soft-float one which has poorer performance.

Raspbian "wheezy"

If you're just starting out, **this is the image we recommend you use**. It's a reference root filesystem from Alex and Dom, based on the Raspbian optimised version of Debian, and containing LXDE, Midori, development tools and example source code for multimedia functions.

Torrent	2013-02-09-wheezy-raspbian.zip.torrent
Direct download	2013-02-09-wheezy-raspbian.zip
SHA-1	b4375dc9d140e6e48e0406f96dead3601fac6c81
Default login	Username: pi Password: raspberry

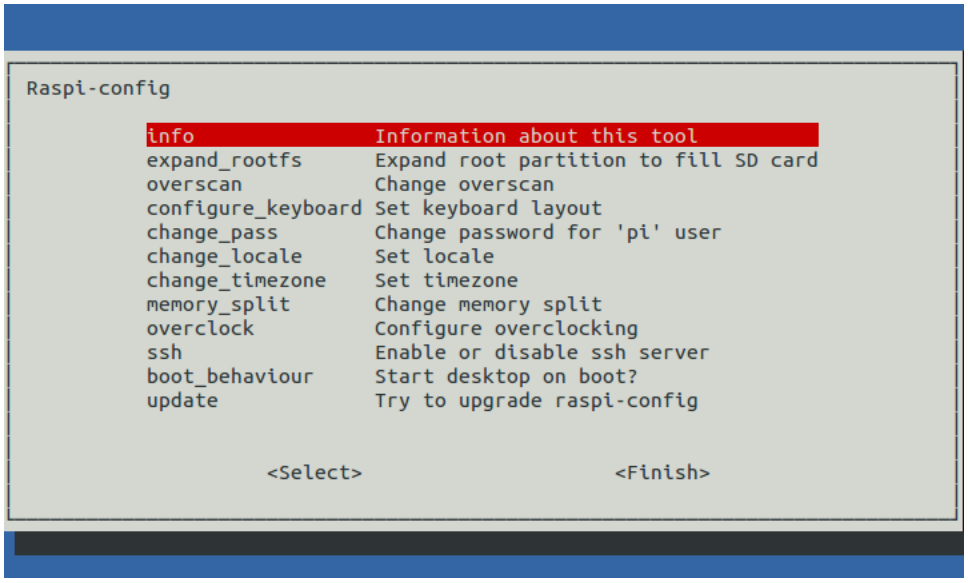
Save the file to a hard drive then expand the zip file to reveal the image file. This needs to be written to the SD Card which you will use in the Pi. Choose an SDHC card of class 6 or 10 (for best performance) and 8 GB capacity, so you can store lots of data. (Minimum is a class 6 card with 2 GB). The OS itself takes up a small space and we will expand the image later to fill the whole SD card – a bit like stretching a Windows partition to fill all of a drive.

The image can be written to the SD card using **Win32DiskImager** which is available from **SourceForge**: **http://sourceforge.net/projects/win32diskimager/**.

Full instructions on this part of the process are given on the **eLinux** website – just be careful as the process will erase the SD card, or whatever drive you tell it to use, including various other devices – check before you push the button, otherwise you might erase Windows : **http://elinux.org/RPi_Easy_SD_Card_Setup**.

After copying the O/S to the SD card, put the card into the Pi, connect a keyboard and monitor as well as a network cable and then power up the Pi by plugging in the USB charger cable. Again, full details are available in the **Raspberry Pi QuickStart Guide** which also contains useful diagrams of all the connectors on the Pi: **http://www.raspberrypi.org/wp-content/uploads/2012/12/quick-start-guide-v1.1.pdf**

The main steps during the first boot and setup are as follows. On first boot you will come to the **Raspi-config** window.



Change settings such as **timezone** and **locale** if you want. It is a good idea here to change the password for the default 'pi' user, and to ensure **SSH** is enabled. In the UK, the **timezone** is usually already correct. Finally, select the second choice:

expand_rootfs

and then select

update

let the update finish You might receive an updated **raspi-config** tool – follow it through again and you might also have the option under **Advanced Options** to set the hostname directly here rather than later. When all configuration changes are made, select **Finish** (you might need to use the TAB key to move to this item). Then say 'yes' to a reboot if asked. If reboot is not requested, manually trigger one by typing

```
sudo reboot
```

The Raspberry Pi will reboot and you will see **`raspberrypi login:`**

Type:
pi

'pi' is the default username. You will be asked for your Password. The default is 'raspberry'.

Type:
raspberry

You will then see the prompt:

pi@raspberry ~ $

This is the command prompt – you are now working in the command console. The Pi is up and running. If you forgot to adjust any of the initial settings such as expanding the file system to fill the SD card, use the command

```
sudo raspi-config
```

to bring up the menu again. You should definitely expand the file system (otherwise you will only be able to use a fraction of your SD card's space) and you should set the **timezone**, since we will be displaying local time on the LCD display.

Next, a bit of security – all SD card images come with the same default username and password – you should change the password of the default Pi user to something else (if not set during initial configuration).

```
passwd
```

This will ask for your old password and then ask you to type a new password twice to confirm.

Future login sessions can occur via the SSH protocol remotely, but in order to do that, we need to know the IP address of the Pi.

```
sudo ifconfig
```

This will list the interfaces on the Pi, typically including the `eth0` Ethernet interface – note the IP address given here – typically on the line starting **inet addr: 192.168.7.100**. Use a remote SSH client like `Putty` to SSH into the Pi on this address. After confirming working SSH access, logout of the Pi on the main terminal and monitor session

```
exit
```

and continue through use of the SSH connection. The monitor and keyboard can now be disconnected. If you are using a model A R-Pi, you don't have the network adaptor so you need to continue directly connected to the Pi with the screen and keyboard.

Next, create another user which you will use as your normal login.

To set up a new user (here we add a user called john):

```
sudo adduser john
```

Follow the prompts for information.

```
sudo visudo
```

Find the line

pi ALL=(ALL) NOPASSWD: ALL

add the new user on the next line

john ALL=(ALL) NOPASSWD: ALL

Once the changes are made press **ctrl + o** to save the **sudoers** file, then press **ctrl + x** to exit **visudo**. It is also worth adding your new user to the **sudoers** list (this is the list of super users that are allowed to perform admin tasks on the OS).

```
sudo adduser john sudo
```

If any of the above commands fail to work, ensure you are prefixing them with the **sudo** command to force them to run at an elevated privilege level, *e.g.*

```
sudo adduser john sudo
```

We now want to set up the correct networking on the device. We use the nano text editor to make the changes. It uses cursor keys to move around and uses **ctrl+x** to quit and prompt for changes to be saved. The networking settings are stored in some text files. Firstly we tell the Pi what its network name is.

```
sudo nano /etc/hostname
```

The only content of this file should be the network name of the Pi. Edit as required.

```
sudo nano /etc/hosts
```

Confirm that the entry for **127.0.0.2** uses the same hostname as above.

If you want to ensure that the Pi always has the same IP address (called a static address) then edit the network interfaces file as follows.

```
sudo nano /etc/network/interfaces
```

Look for the line which says

```
iface eth0 inet dhcp
```

and change it to

```
iface eth0 inet static
```

Below this line enter the following.

```
address 192.168.100.1
netmask 255.255.255.0
network 192.168.100.0
broadcast 192.168.100.255
gateway 192.168.100.254
```

Note that you should use the correct values for your own network. In the above example we set the Pi to sit on the private network address 192.168.100.1 and for which the router (to the outside world) is 192.168.100.254.

Save the file and exit the editor.

We now should have a working network connection. We use this to update the on-board list of modules and upgrade the Pi to the latest version of the OS (a bit like Windows Update).

```
sudo apt-get update && sudo apt-get upgrade
```

Answer 'Y' (ENTER) when prompted. This step can take quite a while – after the various updated modules have been downloaded, they are installed. Sometimes one or more of the software mirror servers is busy and there is an error. Wait a moment and try again.

Corporate firewall problems

If setting this up inside a corporate firewall, it can cause some problems. First we perform a quick check that we can tunnel through any firewall that exists. Try the following:

```
sudo-apt get update
```

If error messages are generated concerning inability to resolve addresses or to access mirror locations, it may be necessary to set up information on the firewall in use on the network so that we can tunnel through it. We need to edit a file using the nano text editor.

```
sudo nano /etc/apt/apt.conf
```

Ensure the file contains the following, substituting actual values.

```
APT::Get::AllowUnauthenticated 1;
Acquire::http::Proxy "http://user:password@proxy.company.co.uk:port/";
Acquire::ftp::Proxy "ftp://user:password@proxy.company.co.uk:port/";
Acquire::https::Proxy "https://user:password@proxy.company.co.uk:port/";
```

Save the file and reboot the Pi. If the firewall still blocks WGET access (see below), any files required can be copied to the Pi after installing the Samba daemon (see below).

Installing GPIO module for Python

We will be using the Python language to write the scripts that are needed to run the hardware data capture and drive the LCD unit. Luckily (!) Python is pre-installed in the latest versions of Raspbian. We can test this by simply typing

```
python
```

and we see the Python interpreter prompts.

```
Python 2.7.3 (default, Jan 13 2013, 11:20:46)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

To quit from the interpreter simply type

```
exit()
```

and press ENTER.

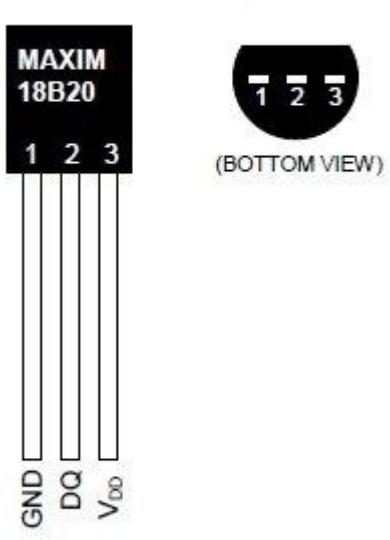
However, there are some add-on modules that we need that are not installed by default in Python, and we need to install them manually. The first module is the GPIO module which we will use for communicating with the sensors and also the LCD.

```
sudo apt-get install python-rpi.gpio
```

Note, on latest versions of Raspbian, this is already installed – asking for the installation as per the above command is not a problem – you just get a note that “python-rpi.gpio is already the newest version”.

Installing the modules for the 1-Wire temperature sensors

Ensure that the DS18B20 sensor(s) are connected as per the hardware assembly instructions. Particularly, ensure that pin 1 is connected to ground (GPIO pins 6, 9, 14, 20, 25), pin 2 is connected to DATA (GPIO pin 7) and pin 3 is connected to 3V3 (GPIO pin 1). Ensure a 4k7 resistor is between pin 2 (data) and pin 3 (3V3).



The sensor interfacing is performed using the One Wire Filing System (OWFS) which mounts the sensors as if they were directories in the filing system.

Next we install the One Wire File System and its shell support.

```
sudo apt-get install owfs ow-shell i2c-tools
```

We could just load the kernel driver modules temporality for testing using

```
sudo modprobe w1-gpio
sudo modprobe w1-therm
```

But it is better to make them automatically load at power up.

```
sudo nano /etc/modules
```

Ensure that the following lines are present in this file of kernel modules to load at boat time:

```
snd-bcm2835
w1-gpio
w1-therm
```

```
sudo reboot
```

Now, ensuring that the hardware is connected (*i.e.* the DS18B20 sensors), we can check that the modules have loaded properly and that the sensors are being mounted into the file system.

```
lsmod
```

You should see the following modules in the list:

`w1_term, w1_gpio, wire`

amongst others. We now check for the presence of the DS18B20 devices, which appear as individual directories below the higher `/sys/bus/w1/devices` directory.

```
cat /sys/bus/w1/devices/*
```

We should see something like:

```
cat: /sys/bus/w1/devices/28-00000349aeea: Is a directory
cat: /sys/bus/w1/devices/28-00000349d77b: Is a directory
cat: /sys/bus/w1/devices/w1_bus_master1: Is a directory
```

And we can now actually get readings (with debugging information added) from the devices:

```
cat /sys/bus/w1/devices/28-*/w1_slave
```

gives output such as :

```
32 01 4b 46 7f ff 0e 10 1e : crc=1e YES
32 01 4b 46 7f ff 0e 10 1e t=19125
50 01 4b 46 7f ff 10 10 49 : crc=49 YES
50 01 4b 46 7f ff 10 10 49 t=21000
```

In other words, the first sensor is reading 19.125 °C and the second one is reading 21.000 ° and the CRC check is OK for both devices, showing that the readings are valid. So we now have functioning temperature sensors. With some simple Python code we can read and manipulate the values. But first, let's set up the humidity sensor.

Installing the modules for the SHT75 humidity sensor

The SHT1X library is not so simple – we have to download it using `wget` and then install it using Python. Afterwards we tidy up the stuff that gets left behind in our user folder. The installer copies the relevant files into the Python system.

```
cd ~
wget https://pypi.python.org/packages/source/r/rpiSht1x/rpiSht1x-1.2.tar.gz
sudo gunzip rpiSht1x-1.2.tar.gz
sudo tar -xvf rpiSht1x-1.2.tar
cd rpiSht1x-1.2/
sudo python setup.py install
cd ..
rm rpiSht1x-1.2.tar
sudo rm -rf rpiSht1x-1.2/
```

If the call to `sudo python setup.py install` fails, it is probably because a firewall is blocking a python download, most probably the `python distribute` system code.

If so, you will need to download it separately from:

http://pypi.python.org/packages/source/d/distribute/distribute-0.6.28.tar.gz

then copy it to your home directory on the Pi (requires Samba to be installed and running – see below). Now we have to process the file and install manually:

```
tar -xzvf distribute-0.6.45.tar.gz
cd distribute-0.6.45
sudo cd ..python setup.py install
cd ..
rm distribute-0.6.45.tar.gz
sudo rm -rf rpiSht1x-1.2/sudo rm -rf dist
sudo rm -rf distribute-0.6.45/
```

(It would be much easier if the corporate firewall was a properly transparent proxy rather than requiring authentication, but anyway…). Having solved the issue, go back to the above install of the SHT1x modules and continue with the remaining steps:

```
cd rpiSht1x-1.2/
sudo python setup.py install
cd ..
rm rpiSht1x-1.2.tar
sudo rm -rf rpiSht1x-1.2/
```

Ensure the sensor is connected properly:

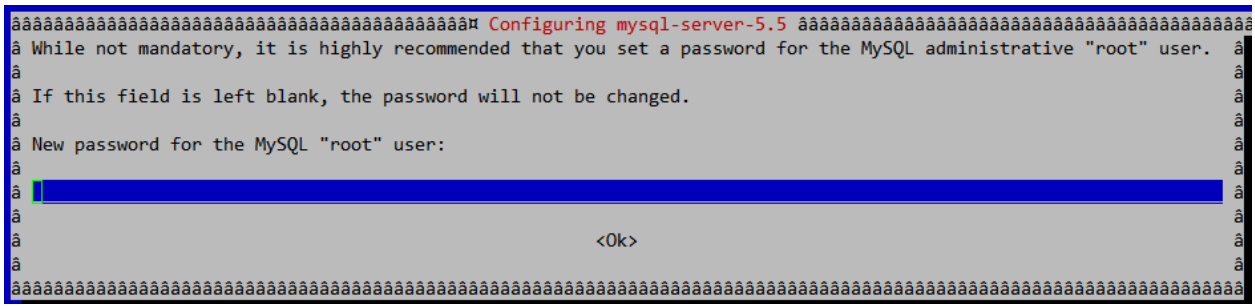
Figure 8 - Pin out for SHT75 humidity sensor

Connect pin 1 to the CLK pin chosen in the GPIO connector (GPIO pin 13), pin 2 goes to the 3V3 supply (GPIO pin 1), pin 3 connects to any GND (GPIO pins 6, 9, 14, 20, 25) and pin 4 connects to the chosen DATA pin on the GPIO (GPIO pin 15). If you get it the wrong way round, *i.e.* VDD and GND inverted, you will get a hot burning smell…. This is not good, but it should work OK after cooling down a bit…mine did !

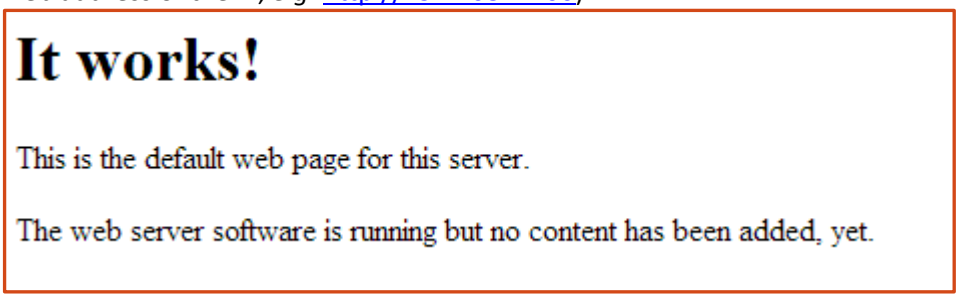
Installing the webserver

We will use the Apache 2 web server.

```
sudo apt-get install apache2 php5 libapache2-mod-php5 php5-mysql mysql-server mysql-client
```



Leave MySQL root user password blank (three times). We now have a working webserver (you can check the web address of the Pi, *e.g.* http://192.168.7.100).



Optimise the Apache 2 web server with php

The default install of Apache 2 and PHP uses quite a lot of processor power. We need to install the **PHP-APC** module to speed things up. APC is 'Alternative PHP Cache'.

```
sudo apt-get install php-apc
```

```
sudo nano  /etc/php5/conf.d/20-apc.ini
```

File contents:

```
extension=apc.so
apc.enabled=1
apc.shm_size=12M
```

Then force the service to restart:

```
sudo service apache2 restart
```

You can also check that the APC module is installed and running within PHP.

```
php -m
```

The APC module will be in the listed of modules that scrolls past.

If you need to shut off the Pi at any time, use the following command and then wait until the lights stop flashing and only the red LED remains lit. This takes about 20 seconds. Then turn off the power or unplug.

```
sudo shutdown -hP now
```

Enabling virtual hosts

We will not be mounting the web pages below the usual /var/www root location so we need to allow the web server to operate virtual hosts.

```
sudo nano /etc/apache2/apache2.conf
```

Ensure that the following lines (near the end) are OK and present:

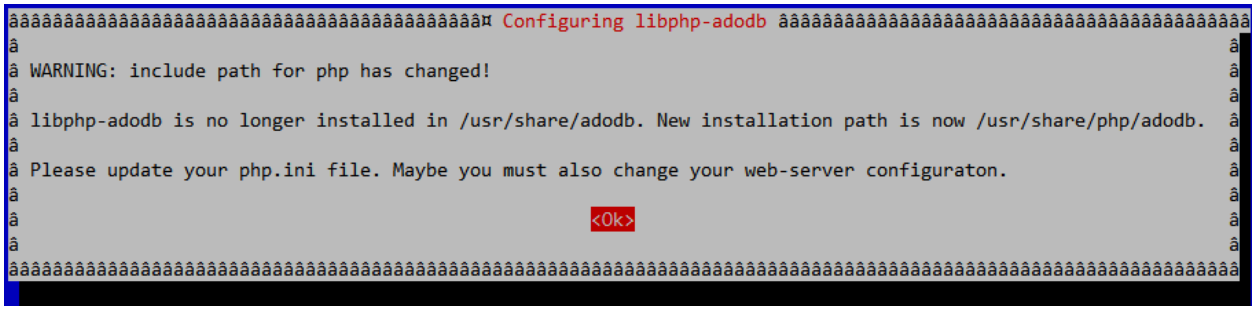
```
# Include generic snippets of statements
Include conf.d/

# Include the virtual host configurations:
Include sites-enabled/
```

Installing Cacti

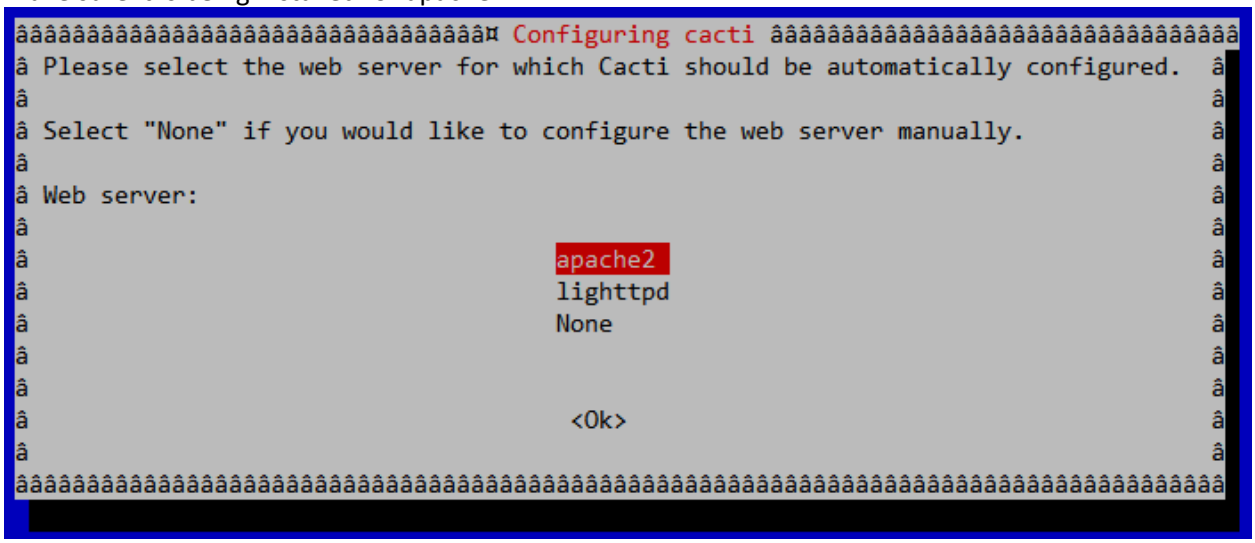
Cacti is the data gathering and graphing tool. It uses **RRDTOOL** for its data storage and generates the various graphs that are displayed as well as handing user administration. It is a bit difficult to get to grips with initially. Luckily, for a default Raspbian operating system, it installs very easily.

```
sudo apt-get install cacti
```

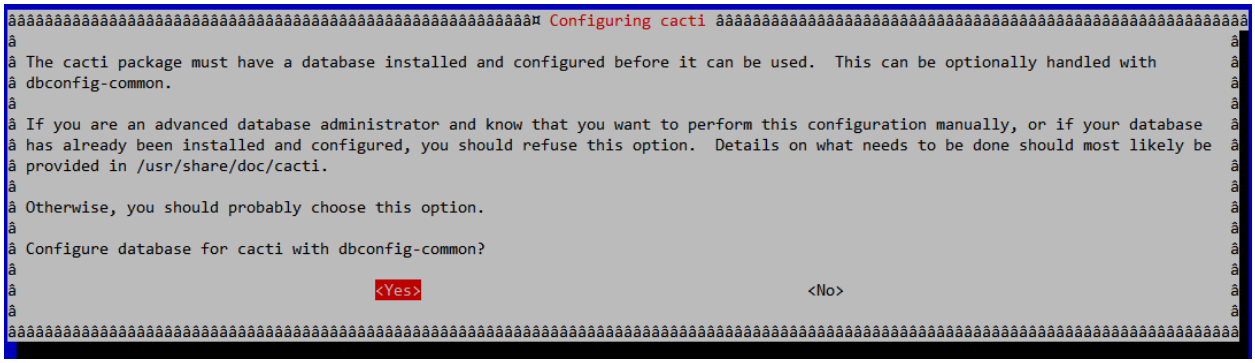
A few screens will pop up asking for input. Basically, since we are not going to be particularly security conscious, we accept the defaults on all screens by simply pressing ENTER.



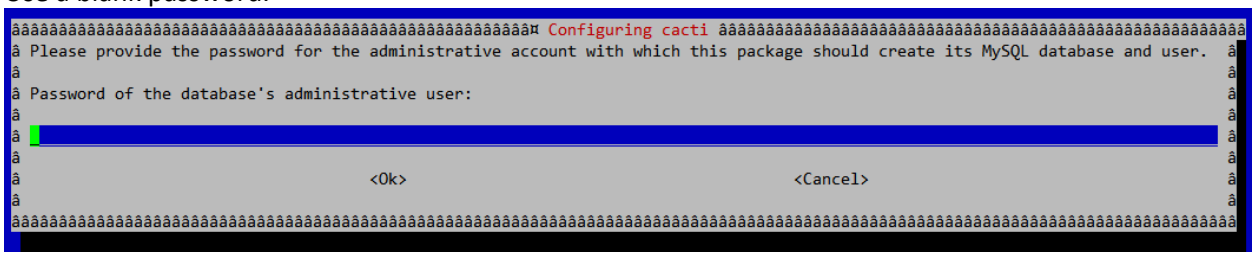
Make sure it is being installed for apache2.



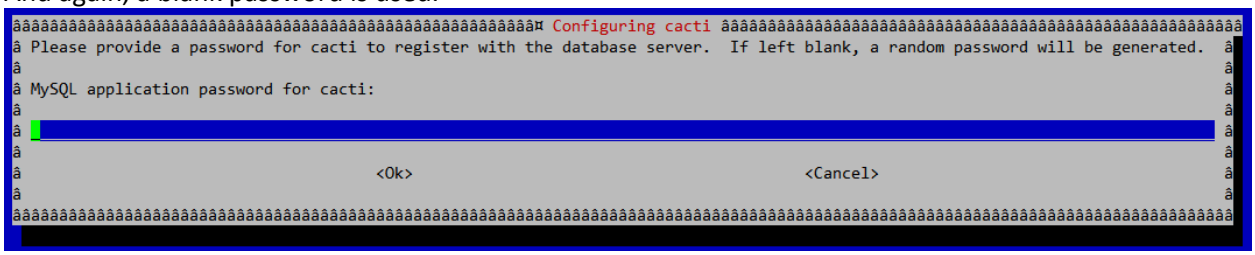
Let it set up default databases.



Use a blank password.



And again, a blank password is used.



When the install is complete, point a web browser at the Pi's address with the **/cacti/** suffix, for example:

http://192.168.7.100/cacti/

You will now be taken through the basic Cacti install process via the web browser.

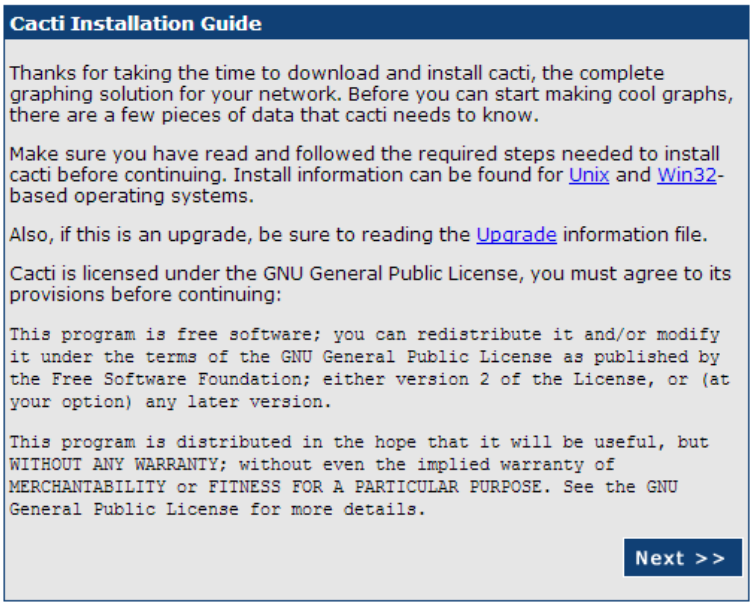
If this fails to appear, check that the file

/etc/apache2/conf.d/cacti.conf

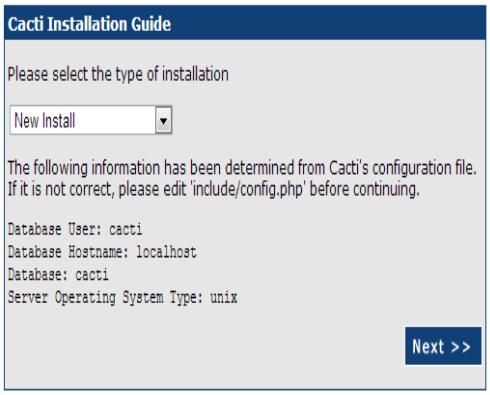
exists and contains the following, which declares a virtual host:

```
Alias /cacti /usr/share/cacti/site

<Directory /usr/share/cacti/site>
        Options +FollowSymLinks
        AllowOverride None
        order allow,deny
        allow from all

        AddType application/x-httpd-php .php

        <IfModule mod_php5.c>
                php_flag magic_quotes_gpc Off
                php_flag short_open_tag On
                php_flag register_globals Off
                php_flag register_argc_argv On
                php_flag track_vars On
                # this setting is necessary for some locales
```

```
                php_value mbstring.func_overload 0
                php_value include_path .
        </IfModule>

        DirectoryIndex index.php
</Directory>
```

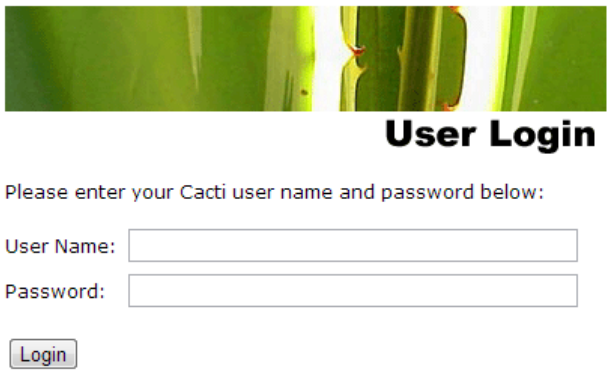


Press the **Next >>** button.



It is definitely a New Install.

That's nice to see – it has found all the necessary software tools already installed. Press the Finish button. You now need to login to Cacti.

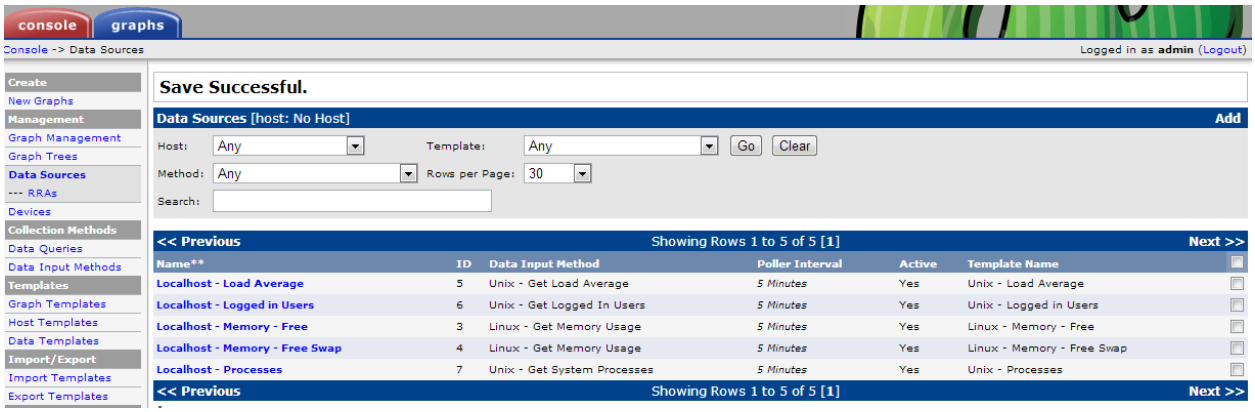


Login with username: **admin** and password: **admin**.

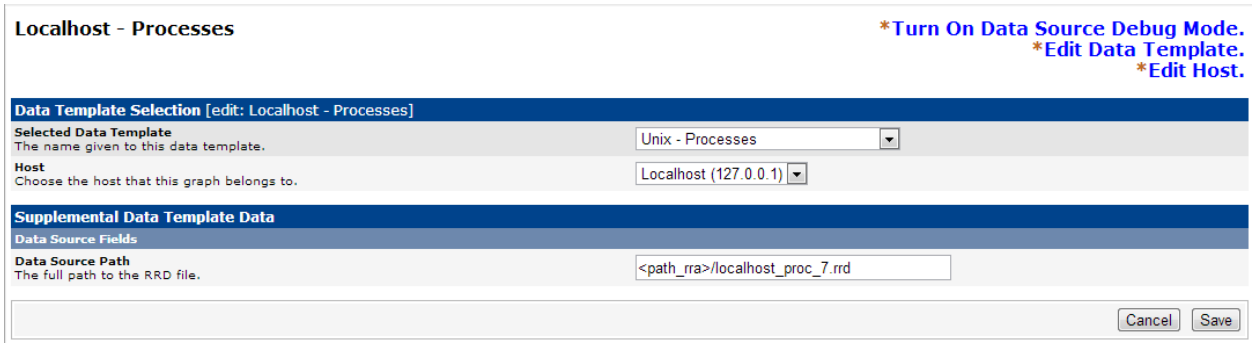
You will then be forced to change the password for the admin account.
You are now presented with the main Cacti window.

If you click on the blue graphs tab, a set of default graphs for a Linux host appears. For a short while these graphs will be empty or not display properly – Cacti is gathering data on a 5 minute cycle and the graphs will not display until at least 2 data cycles are complete.

However, it is necessary to start the round robin databases (RRDs) before anything will appear – although Cacti sets these up for you during the install, it does not actually start them running – in the **console** tab, select the **Data Sources** menu item and then click on each data source one by one and press save.



Select each one, one at a time, *e.g.* **Localhost - Processes**. In the screen that comes next, simply press **Save**.



Then wait 5 to 10 minutes for the graphs and their databases to populate. Click on the blue **Graphs** tab to see them.

Configuring Cacti's poller process

First, we want to change the default user account that the Cacti poller uses to gather data. Normally it is set to **www-data** but some of the humidity code which uses GPIO pins cannot run as this user, it requires root privileges.

```
sudo nano /etc/cron.d/cacti
```

Change the line:

```
*/5  *  *  *  *  www-data  php  --define  suhosin.memory_limit=512M  /usr/share/cacti/site/poller.php  2>&1
>/dev/null | if [ -f /usr/bin/ts ] ; then ts ; else tee ; fi >> /var/log/cacti/poller-error.log
```

to use **root** instead of **www-data** and **1** instead of **5** minutes:

```
*/1 * * * * root php --define suhosin.memory_limit=512M /usr/share/cacti/site/poller.php 2>&1 >/dev/null |
if [ -f /usr/bin/ts ] ; then ts ; else tee ; fi >> /var/log/cacti/poller-error.log
```

This file that we just edited is one of the inputs to the **crontab** system – the Linux scheduler. This file is set up so that every 5 minutes (*/5) it calls the php system (as root) and runs the Cacti poller, which is located at **/usr/share/cacti/site/poller.php** . This poller calls all the necessary scripts that read in the periodically updated data. Any errors are logged. We change it so that it is run every minute as well.

Improving Cacti display of numbers in small decimal range

By default, Cacti is not very good at displaying graph axis labels when the range of data is only a few digits in the first decimal point. (In fact it is the RRDTOOL that Cacti calls that has this limitation). We can improve this by editing one of the Cacti setup files.

```
sudo nano /usr/share/cacti/site/lib/rrd.php
```

This file is quite big. Scroll down several screens and look for function rrdtool_function_graph($local …). Another 3 screens below that are the lines:

```
if ($graph["auto_scale"] == "on") {
        switch ($graph["auto_scale_opts"]) {
            case "1": /* autoscale ignores lower, upper limit */
                $scale = "--alt-autoscale" . RRD_NL;
                break;
```

after the --alt-autoscale option, add --alt-y-grid to make the line read:

```
$scale = "--alt-autoscale --alt-y-grid" . RRD_NL;
```

and then save the file. The, selection of this option in the Cacti graph configuration screen ("--alt-autoscale") will set a better grid labelling.

For now, we will leave Cacti alone as we set up the software modules to interface to the sensors.

Creating the Python script for Cacti to use to read temperatures

The Cacti poller is a fairly simple beast – it is designed to call other scripts to do the hard work of reading sensors. So, we need to create a script, in Python, that the poller can call periodically. The script needs to return (to **stdout**) the values from the sensors in the format required by the poller. This is usually the data name followed immediately by a colon, then the value. A space separates the value from the next data name, and so on, *e.g.* `item1:12.345 item2:6.789`.

Simply use the nano text editor to create the file `/usr/share/cacti/site/scripts/new_temperatures.py` with the following contents:

```
#!/usr/bin/python

# This file is: /usr/share/cacti/site/scripts/new_temperatures.py

import os, glob, time, sys, datetime

#set up the location of the two DS18B20 sensors in the system
device_folder = glob.glob('/sys/bus/w1/devices/28*')
device_file = [device_folder[0] + '/w1_slave', device_folder[1] + '/w1_slave']


def read_temp_raw(): #a function that grabs the raw temperature data from the sensors
    f_1 = open(device_file[0], 'r')
    lines_1 = f_1.readlines()
    f_1.close()
    f_2 = open(device_file[1], 'r')
    lines_2 = f_2.readlines()
    f_2.close()
    return lines_1 + lines_2


def read_temp(): #a function that checks that the connection was good and strips out the temperature
    lines = read_temp_raw()
    while lines[0].strip()[-3:] != 'YES' or lines[2].strip()[-3:] != 'YES':
        time.sleep(0.2)
        lines = read_temp_raw()
    equals_pos = lines[1].find('t='), lines[3].find('t=')
    temp = float(lines[1][equals_pos[0]+2:])/1000, float(lines[3][equals_pos[1]+2:])/1000
    return temp


temp = read_temp() #get the temp
```

```
print('T1:'+str(temp[0])+' T2:'+str(temp[1]))
```

This script will find the first two DS18B280 temperature sensors (alphabetically ordered) and read them until they return valid temperatures. The two channel numbers (T1, T2) will be output followed by a colon and the temperature. A space separates the two outputs.

```
sudo chmod 777 /usr/share/cacti/site/scripts/new_temperatures.py
```

We can then check the script works by calling it:

```
sudo /usr/share/cacti/site/scripts/new_temperatures.py
```

This is the type of output we expect:

T1:23.187 T2:23.187

Each sensor is followed, after a colon, by its reading. A space separates this sensor from the data from the second sensor. The sensors are allocated as T1 and T2 based on the alphabetical sorting of their serial numbers. If you replace a sensor at a later date, the ordering may change (depending on serial numbers)!

Creating the Python script for Cacti to use to read humidity

Again, we need a Python script to prepare the data in the format required by the Cacti poller. Simply use the nano text editor to create the file `/usr/share/cacti/site/scripts/new_humidity.py` with the following contents:

```
#!/usr/bin/env python

# This file is /usr/share/cacti/scripts/humidity.py

import string
import os
import sys

# some imports for the humidity monitoring using an SHT75 device (uses same code as SHT1x)
from sht1x.Sht1x import Sht1x as SHT1x
dataPin = 15
clkPin =  13
sht1x = SHT1x(dataPin, clkPin, SHT1x.GPIO_BOARD)

# Read the SHT device to get humidity and temperature
SHTtemperature = sht1x.read_temperature_C()
SHThumidity = sht1x.read_humidity()
SHTdewPoint = sht1x.calculate_dew_point(SHTtemperature, SHThumidity)
sys.stdout.write('RH:')
hval=format(SHThumidity, "2.1f")
sys.stdout.write(hval)
sys.stdout.write(' ')
#print("RH:"+format(SHThumidity, "2.1f")),
```

This script will read the SHT75 sensor, taking a reading of the humidity and the temperature and use these to calculate the relative humidity and the dew point temperature. The data label ‘RH’ will be output followed by a colon and the relative humidity.

```
sudo chmod 777 /usr/share/cacti/site/scripts/new_humidity.py
```

We can then check the script works by calling it (ensure the sensor is connect correctly before running the script).

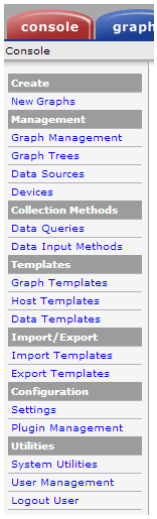
```
sudo /usr/share/cacti/site/scripts/new_humidity.py
```

This is the type of output we expect:

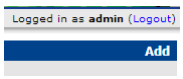
```
RH:45.6
```

Updating the Cacti configuration to read in the temperature and humidity data

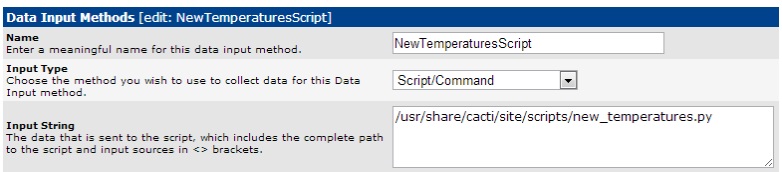
Having created the necessary Python scripts, and checked that they output the correct data in the format that Cacti can read, we now tell Cacti about these scripts and set up the relevant data input methods and data sources and their output graphs. First, we need to tell Cacti about the scripts we created earlier that can perform the data capture for us. Log in to Cacti, as before, ensuring you log in with administrator privileges. Go into the red console tab near the top of the browser window. Running down the left side is the main menu of links we will use.



First click on **Data Input Methods**. Then click on the Add link at the top right of the screen.

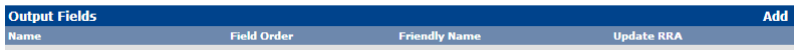


We'll add the script for the temperature readings. Name the method `NewTemperaturesScript`. The input type is `Script/Command` and the Input String is simply the full path to the script:
/usr/share/cacti/site/scripts/new_temperatures.py

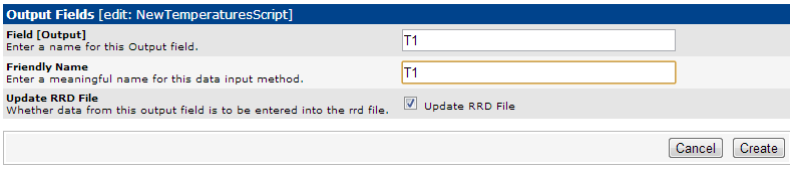


With the settings as above, press the **Create** button and we are presented with two new sections to this screen: **Input fields** and **Output fields**. There are no input fields (*i.e.* the script `new_temperatures.py`

does not expect any input parameters). It does, however, produce two outputs and we need to enter these. Press the **Add** link at the top right of the **Output Fields** separator:

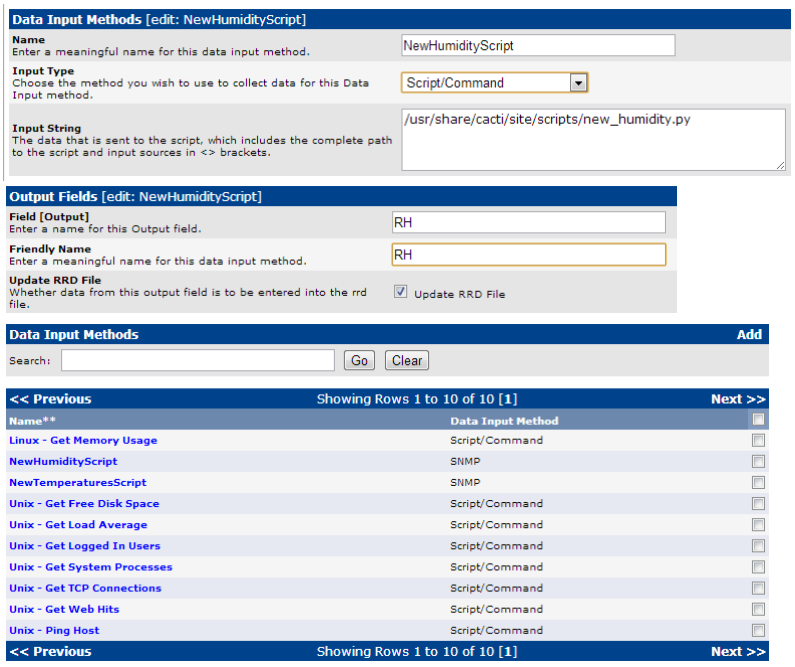


We might as well use consistent naming, using both T1 for the **Output field** and the **Friendly Name**:



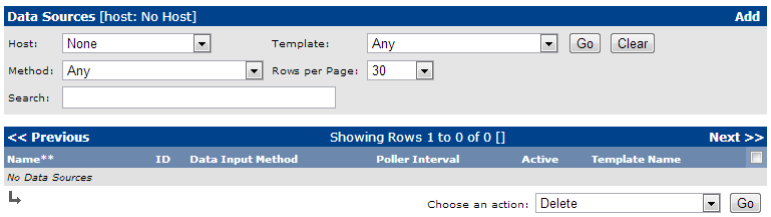
Then push '**Create'**. Repeat the process with 'T2' so we end up with Cacti set up to call the correct script and to expect two return parameters, T1 and T2. (This is why we used the `T1:19.876 T2:23.456` formatting in the script output – Cacti will parse the returned line for these two entries, and extract their numerical values). Press the **Save** button on the web page and then the **Return** button. You will be presented with a list of all the known **Data Input Methods** – some are default ones already entered during the initial setup of Cacti, but joining them is our **NewTemperaturesScript**.

Now repeat the process to add another **Data Input Method** for the humidity. Here are the various screens that we encounter, filled out as required. Note that, as per the temperature script, the **Output Fields** names we set up in Cacti must match those output by the script.



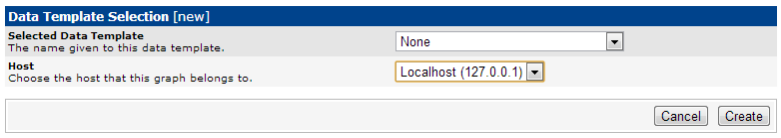
So, that's the scripts entered into Cacti. Now we have to go to the next step which is creating Data Sources, one for the temperatures, one for the humidity – Cacti uses Data Sources to plot graphs, so we won't be able to generate graphs until we set up the two Data Sources.

Select **Data Sources** from the left menu.

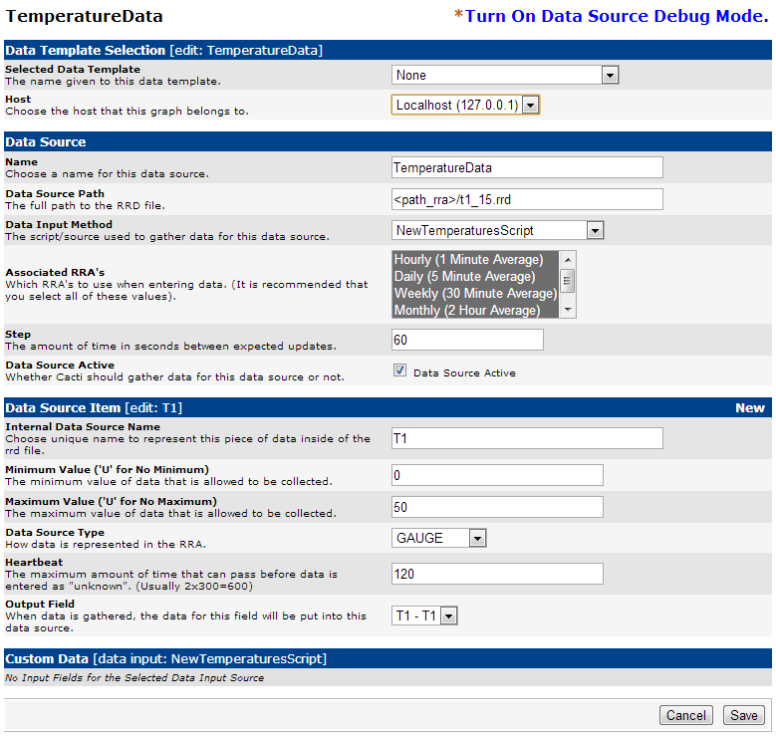


Click the **Add** button near the top right.

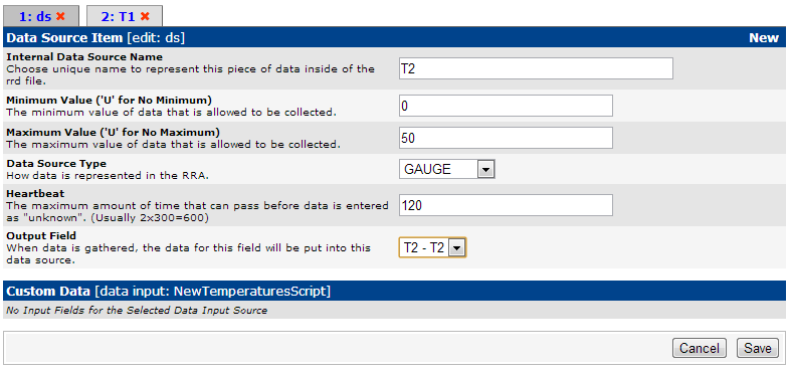
Leave the **Data Template** set to **None** and set **Host** to **Localhost** and Click **Create**.



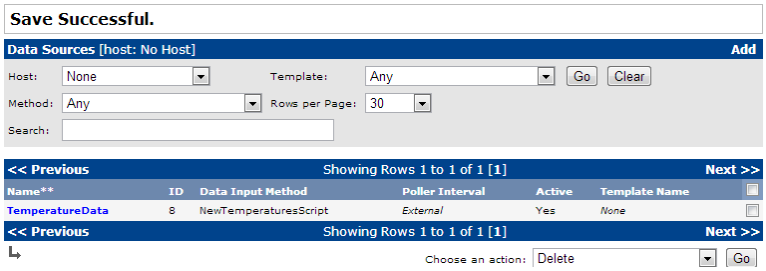
Name the temperature source **TemperatureData**. Leave the **Data Source Path** blank. The **Data Input Method** is the script we created earlier – **NewTemperaturesScript**. Set the **Step** to **60** (we expect to be updating these values every 60 seconds). The section below, entitled **Data Source Item** is used to set up the entries, one by one – we need to set up two. First set up T1, as per below. Ensure all the input values are as given in this example then push the **Create** Button. Cacti fills out the **Data Source** Path entry for us and allows us to add more **Data Source Items**. Push the '**New'** link 2/3 the way down the window.



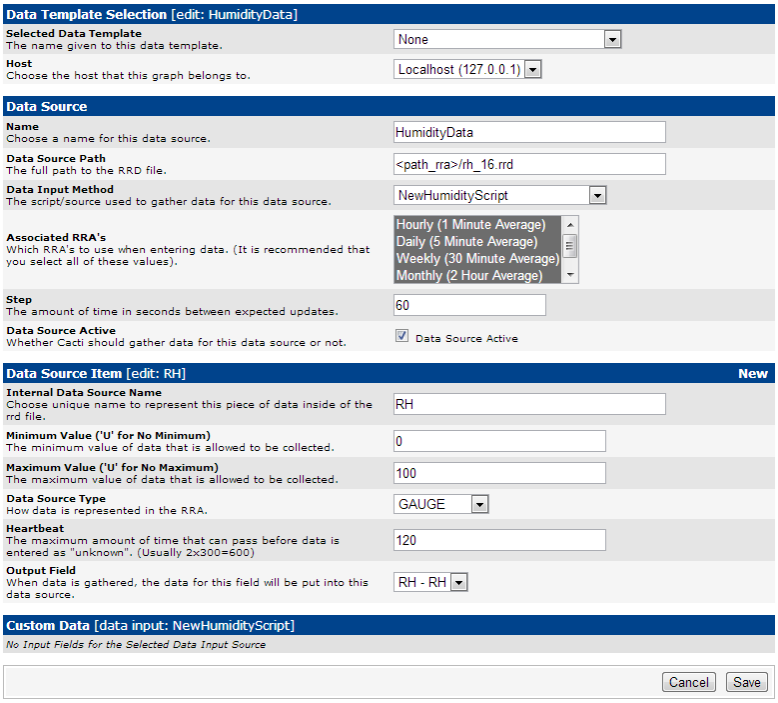
A second data source item, with the default name of '**ds**' is created – there are now two tabs labelled 'T1' and 'ds'. The T1 is the one we created a moment ago, we now update the values in the 'ds' tab and press **Save**:



This closes the entry screen and takes us back one step – you can select the **TemperatureData** entry again and go back to check that the T1 and T2 values were correctly entered.

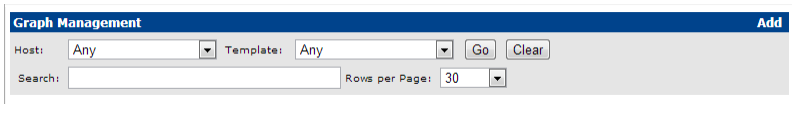


We now have the temperature data source correct, do the same thing with the humidity one, though this time there will only be one data source item, RH.

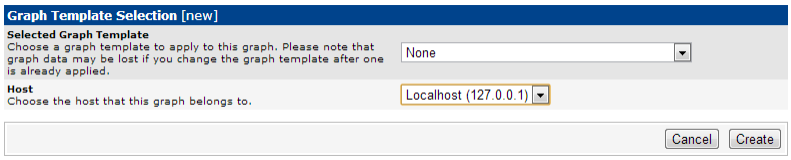


No need to add a second item.

Now that we have the Data Sources set up, we can finally add the graphs. Click **Graph Management**, then the **Add** link at the top right.

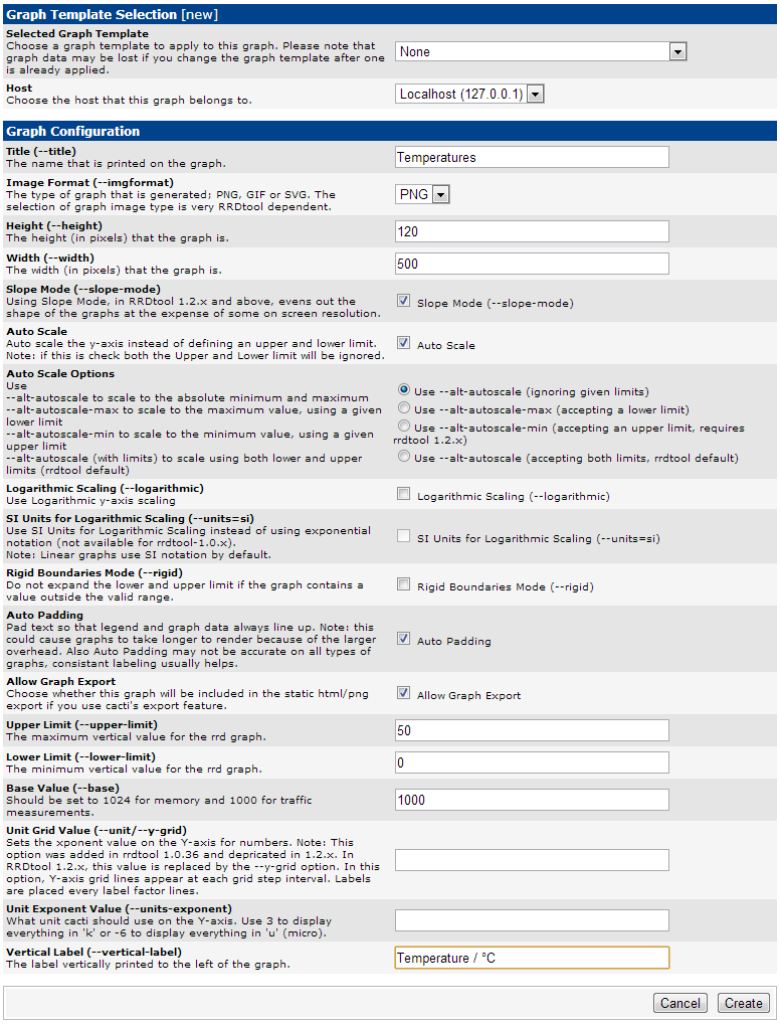


Choose **None** for both the **Selected Graph Template** and for the **Host** choose **Localhost** and press the **Create** button.

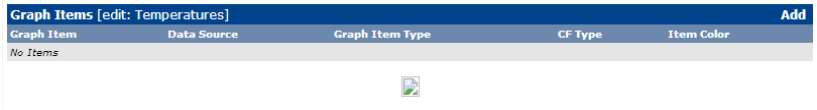


Give the graph a title, such as **Temperatures**. Leave the **Image Format**, **Height**, **Width**, **Slope Mode**, **Auto Scale** as defaults (see below) but ensure to check the first **Auto Scale Option** (**Use –alt-autoscale ignoring given limits**) – remembering that we adjusted this earlier by editing the PHP file to ensure better display of decimalised units. Leave the next few items as defaults, and give suitable Upper Limit and Lower limits for the graph. The only other item to change is the **Vertical Label**: **Temperature / °C**.

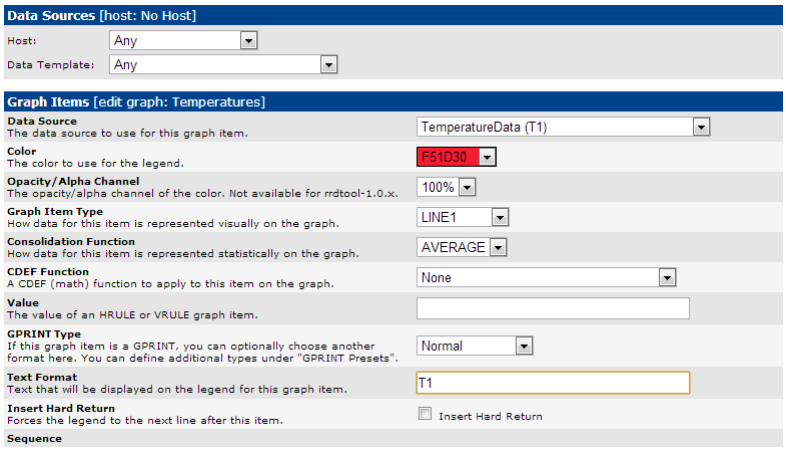
When these are all entered correctly (see screen shot below), click **Create**. This creates the overall graph, but does not add any data sources to it. You are returned to the same screen but there is now a **Graph Items** section as the second section in the display. This is where we add the individual items such as current values and the line graphs.

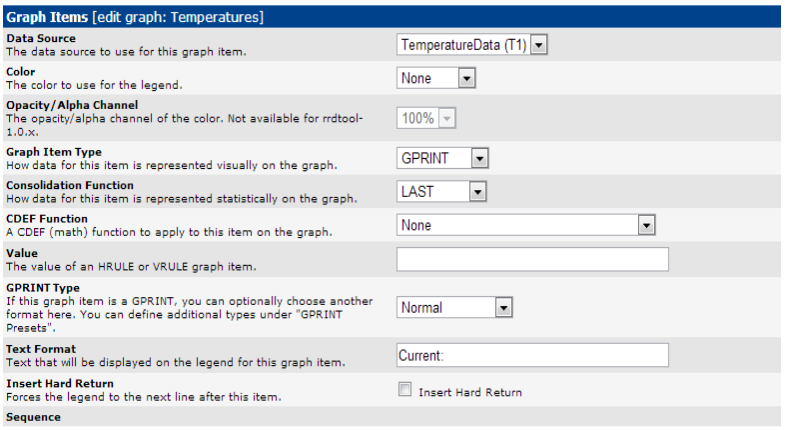


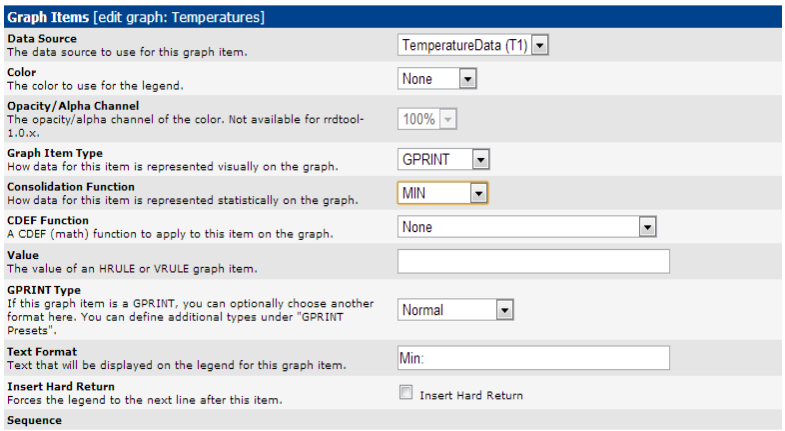
Click on the **Add** link at the top right of this section.

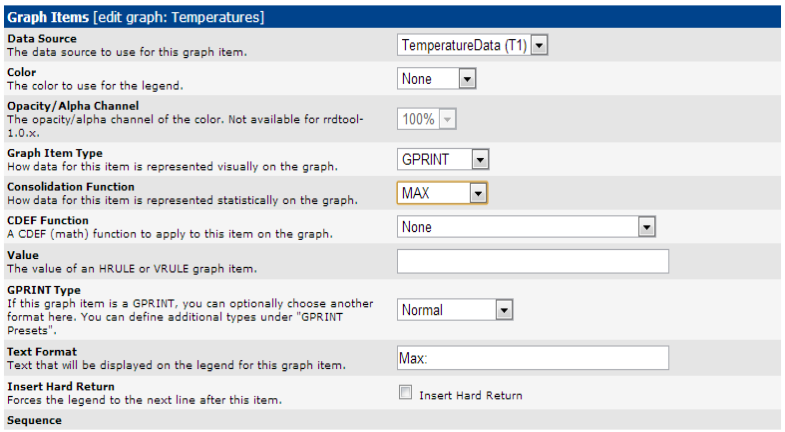


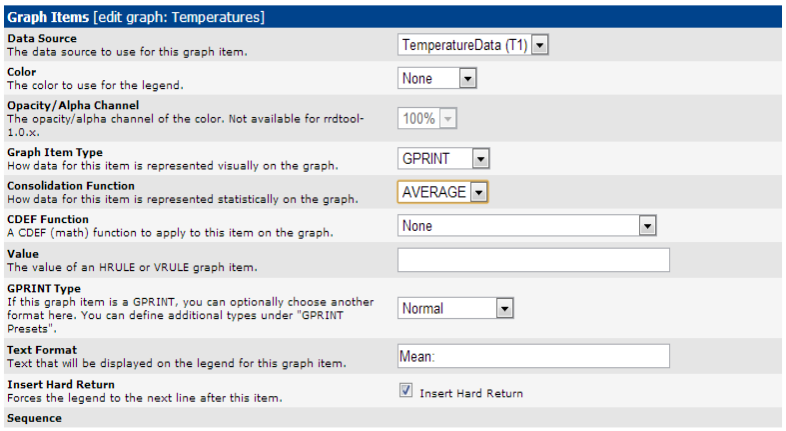
We now have a screen where we can enter the various graph items that we want to appear. We want to add two line graphs (T1 & T2) as well as the **Max**, **Min**, **Average** and **Current** values for each temperature. To save space, the screens below only show the entries for T1 – repeat the same way for T2. Use **None** for the **Host** and **Any** for the **Data Template** at the top of the screen (these are defaults). You need to use the **Add** link for each item to be added. If the relevant **Data Source** is not listed, check the filter at the top of the Window - it may need to be set to **Host: Any** or **Host: Localhost.**



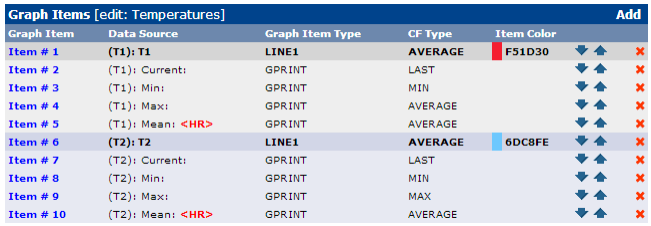








Obviously, choose a different colour for the T2 line graph – green here. After adding all the items, for both T1 and T2, there should be 10 items listed:



Creating a Python script for obtaining readings and driving the LCD

So, we now have Cacti happily logging the data and generating graphs which are served by Apache. We now need a combined script that can drive the LCD, check for tolerances being exceeded and send email alerts, as well as logging raw data to a file.

Simply use the nano text editor to create the file `/usr/local/bin/lcd.py` with the following contents:

```
#!/usr/bin/env python

# File: /usr/local/bin/lcd.py

# IMPORTANT USER EDITABLE DEFINITIONS
# ***********************************
# Define the temperature corrections for both sensors (in deg C)
T1_correction = 0.144                          # calibrated versus known temperature
```

```
T2_correction = 0.144                          # calibrated versus known temperature

raw_logfile = '/usr/local/bin/temps.dat'     # file for raw data logging

SendEmails = True                       # send email when tolerance is exceeded
AlertMaximum = 35                       # max temperature before alerting
AlertMinimum = 5                        # min temperature before alerting
smtpserver =  "mailrelay.company.co.uk"     # SMTP email server
sender =      'me@some.co.uk'               # the 'From:' field
destination = ['who@some.co.uk']            # the 'To:' field
username =    'me@some.co.uk'               # SMTP server login, if needed
password =    'password'                        # SMTP server login, if needed

# Raspberry Pi temperature/humidity logger
# Multi purpose, run by crontab each minute: * * * * * /usr/local/bin/lcd.py
# HD44780 LCD output, Twin DS18B20 temperature inputs, SHT75 humidity input
# Raw logging to file (Cacti does the RRD logging for web pages)
# Email alert when tolerances exceeded
# Using LCD code from Matt Hawkins,http://www.raspberrypi-spy.co.uk
# Additional IP, email, temperature raw logging and LCD code from Andrew Lewis

# The wiring for the LCD is as follows:
# 1 : GND                     - GPIO PIN 6 (GND)
# 2 : 5V                      - GPIO PIN 2 (+5V)
# 3 : Contrast (0-5V)*        - GPIO PIN 6 (GND)
# 4 : RS (Register Select)    - GPIO PIN 26 (SP10 CE1 N)
# 5 : R/W (Read Write)        - GPIO PIN 6 (GND)
# 6 : Enable or Strobe        - GPIO PIN 24 (SP10 CE0 N)
# 7 : Data Bit 0              - NOT USED
# 8 : Data Bit 1              - NOT USED
# 9 : Data Bit 2              - NOT USED
# 10: Data Bit 3              - NOT USED
# 11: Data Bit 4              - GPIO PIN 22 (GPIO 25)
# 12: Data Bit 5              - GPIO PIN 18 (GPIO 24)
# 13: Data Bit 6              - GPIO PIN 16 (GPIO 23)
# 14: Data Bit 7              - GPIO PIN 12 (GPIO 18)
# 15: LCD Backlight +5V**     - GPIO PIN 2 (+5V)
# 16: LCD Backlight GND       - GPIO PIN 6 (GND)

# The wiring for the DS18B20 temperature probes is as follows:
# 1:  GND                     - GPIO PIN 6 (GND)
# 2:  DATA                    - GPIO PIN 7 (GPIO4)
# 3:  VCC (3V3)               - GPIO PIN 1 (+3V3)
# with a 4k7 resistor between DATA and 3V3

# The wiring for the SHT75 humidity sensor is as follows:
# 1:  SCK                     - GPIO PIN 13 (GPIO 21)
# 2:  VDD                     - GPIO PIN 1 (3V3)
# 3:  GND                     - GPIO PIN 6 (GND)
# 4:  DATA                    - GPIO PIN 15 (GPIO 22)
# with a 10k resistor between DATA and 3V3

# Imports
import string, time, socket, datetime, os, glob, sys, pickle
from sht1x.Sht1x import Sht1x as SHT1x
import RPi.GPIO as GPIO

# Define GPIO to LCD mapping, device constants, timing constants
LCD_RS = 7
LCD_E  = 8
LCD_D4 = 25
LCD_D5 = 24
LCD_D6 = 23
LCD_D7 = 18
LCD_WIDTH = 16    # Maximum characters per line
LCD_CHR = True
LCD_CMD = False
LCD_LINE_1 = 0x80 # LCD RAM address for the 1st line
LCD_LINE_2 = 0xC0 # LCD RAM address for the 2nd line
E_PULSE = 0.00005
E_DELAY = 0.00005
```

```
################### MAIN CODE BLOCK ###################
def main():
    # Main program block
    GPIO.setmode(GPIO.BCM)       # Use BCM GPIO numbers
    GPIO.setup(LCD_E, GPIO.OUT)  # E
    GPIO.setup(LCD_RS, GPIO.OUT) # RS
    GPIO.setup(LCD_D4, GPIO.OUT) # DB4
    GPIO.setup(LCD_D5, GPIO.OUT) # DB5
    GPIO.setup(LCD_D6, GPIO.OUT) # DB6
    GPIO.setup(LCD_D7, GPIO.OUT) # DB7

    # Initialise display
    lcd_init()

    # Display the IP address briefly
    myip = getNetworkIp()
    lcd_byte(LCD_LINE_1, LCD_CMD)
    lcd_string(myip)
    # 2 second delay
    time.sleep(2)

    # Read the SHT device to get humidity and temperature, only display humidity
    # Define the SHT75 (based on SHT1x) pin mappings in use, instantiate module
    SHT75datPin = 15
    SHT75clkPin = 13
    sht1x = SHT1x(SHT75datPin, SHT75clkPin, SHT1x.GPIO_BOARD)
    SHTtemperature = sht1x.read_temperature_C()
    SHThumidity = sht1x.read_humidity()
    SHTdewPoint = sht1x.calculate_dew_point(SHTtemperature, SHThumidity)

    # Cleanup the GPIO to avoid clash between LCD and SHT75
    GPIO.setmode(GPIO.BCM)       # Use BCM GPIO numbers
    GPIO.setup(LCD_E, GPIO.OUT)  # E
    GPIO.setup(LCD_RS, GPIO.OUT) # RS
    GPIO.setup(LCD_D4, GPIO.OUT) # DB4
    GPIO.setup(LCD_D5, GPIO.OUT) # DB5
    GPIO.setup(LCD_D6, GPIO.OUT) # DB6
    GPIO.setup(LCD_D7, GPIO.OUT) # DB7

    lcd_init()
    lcd_byte(LCD_LINE_1, LCD_CMD)
    h_string = format(SHThumidity, "2.1f")
    lcd_string("Hum: " + h_string + " %RH")
    lcd_byte(LCD_LINE_2, LCD_CMD)
    h_string = format(SHTdewPoint, "2.1f")
    lcd_string("DP : " + h_string + " C")
    # 3 second delay
    time.sleep(3)

    # Read the two calibration corrected temperatures into an array
    temp = read_temp()
    # Read the current timedate
    now = datetime.datetime.now()

    # Send basic date and temperature 1
    lcd_byte(LCD_LINE_1, LCD_CMD)
    t_string = format(temp[0], "0.3f")
    lcd_string(now.strftime("%d %b ")+'T1='+t_string)

    # Send basic time and temperature 2
    lcd_byte(LCD_LINE_2, LCD_CMD)
    t_string = format(temp[1], "0.3f")
    lcd_string(now.strftime("%H:%M  ")+'T2='+t_string)
    #lcd_string(str(temperature))

    time.sleep(0.5) # 0.5 second delay before GPIO cleanup

    # Now write the raw values to a text file for future retrieval
    # print '%f, %f, %f'%(time.time(), read_temp()[0], read_temp()[1])
```

```
    # Now to check for out of tolerance temperatures
    if (SendEmails == True):
        f =  open('/usr/local/bin/lcd.pickle')
        MaxExceeded, MinExceeded = pickle.load(f)
        f.close()

        print(MaxExceeded, MinExceeded, AlertMaximum, AlertMinimum)

        if (MaxExceeded == False) and ((temp[0] > AlertMaximum) or (temp[1] > AlertMaximum)):
            MaxExceeded = True
            SendEmailViaSMTP('Maximum limit exceeded')
        if (MinExceeded == False) and ((temp[0] < AlertMinimum) or (temp[1] < AlertMinimum)):
            MinExceeded = True
            SendEmailViaSMTP('Minimum limit exceeded')
        if (MaxExceeded == True) and ((temp[0] < AlertMaximum) and (temp[1] < AlertMaximum)):
            MaxExceeded = False
            SendEmailViaSMTP('Cooled to within limits')
        if (MinExceeded == True) and ((temp[0] > AlertMinimum) and (temp[1] > AlertMinimum)):
            MinExceeded = False
            SendEmailViaSMTP('Warmed to within limits')

        f = open('/usr/local/bin/lcd.pickle', 'w')
        pickle.dump([MaxExceeded, MinExceeded], f)
        f.close()

    GPIO.cleanup()




# All the function definitions now follow


def getNetworkIp():
    # Function to determine real IP address, to display at each cycle
    # Requires open IP route to the named device
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.connect(('wgate.npl.co.uk', 0))
    return s.getsockname()[0]


def lcd_init():
    # Initialise display
    lcd_byte(0x33,LCD_CMD)
    lcd_byte(0x32,LCD_CMD)
    lcd_byte(0x28,LCD_CMD)
    lcd_byte(0x0C,LCD_CMD)
    lcd_byte(0x06,LCD_CMD)
    lcd_byte(0x01,LCD_CMD)


def lcd_string(message):
    # Send string to display
    message = message.ljust(LCD_WIDTH," ")
    for i in range(LCD_WIDTH):
        lcd_byte(ord(message[i]),LCD_CHR)


def lcd_byte(bits, mode):
    # Send byte to data pins
    # bits = data
    # mode = True  for character
    #        False for command
    GPIO.output(LCD_RS, mode) # RS
    # High bits
    GPIO.output(LCD_D4, False)
    GPIO.output(LCD_D5, False)
    GPIO.output(LCD_D6, False)
    GPIO.output(LCD_D7, False)
```

```
    if bits&0x10==0x10:
        GPIO.output(LCD_D4, True)
    if bits&0x20==0x20:
        GPIO.output(LCD_D5, True)
    if bits&0x40==0x40:
        GPIO.output(LCD_D6, True)
    if bits&0x80==0x80:
        GPIO.output(LCD_D7, True)
    # Toggle 'Enable' pin
    time.sleep(E_DELAY)
    GPIO.output(LCD_E, True)
    time.sleep(E_PULSE)
    GPIO.output(LCD_E, False)
    time.sleep(E_DELAY)
    # Low bits
    GPIO.output(LCD_D4, False)
    GPIO.output(LCD_D5, False)
    GPIO.output(LCD_D6, False)
    GPIO.output(LCD_D7, False)
    if bits&0x01==0x01:
        GPIO.output(LCD_D4, True)
    if bits&0x02==0x02:
        GPIO.output(LCD_D5, True)
    if bits&0x04==0x04:
        GPIO.output(LCD_D6, True)
    if bits&0x08==0x08:
        GPIO.output(LCD_D7, True)
    # Toggle 'Enable' pin
    time.sleep(E_DELAY)
    GPIO.output(LCD_E, True)
    time.sleep(E_PULSE)
    GPIO.output(LCD_E, False)
    time.sleep(E_DELAY)


def read_temp_raw():
    # Function that grabs the raw temperature data from two sensors ready for
        # use by read_temp()
    # Set up location of 2 DS18B20 sensors, then find their filenames
        # Results are ordered by alpha
    device_folder = glob.glob('/sys/bus/w1/devices/28*')
    device_file = [device_folder[0] + '/w1_slave', \
                       device_folder[1] + '/w1_slave']
    f_1 = open(device_file[0], 'r')
    lines_1 = f_1.readlines()
    f_1.close()
    f_2 = open(device_file[1], 'r')
    lines_2 = f_2.readlines()
    f_2.close()
        # Returns the 4 read lines, 2 from each sensor, containing result strings
        # as well as validity information
    return lines_1 + lines_2


def read_temp():
    # Function to read 2 temperatures. Repeats until good connection then
        # reads inputs and strips out the two temperatures
    lines = read_temp_raw()
    while lines[0].strip()[-3:] != 'YES' or lines[2].strip()[-3:] != 'YES':
        time.sleep(0.2)
        lines = read_temp_raw()
    equals_pos = lines[1].find('t='), lines[3].find('t=')
        # Caclulate temperatures from strings and apply calibration offsets
    temp = T1_correction + float(lines[1][equals_pos[0]+2:])/1000, \
               T2_correction + float(lines[3][equals_pos[1]+2:])/1000
        # Return both temperatures in an array with corrections applied
    return temp


def SendEmailViaSMTP(message='test'):
    # Function that sends email alerts, takes the email text as input
```

```
    import smtplib
    import time

    From = sender
    To = destination
    Date = time.ctime(time.time())
    Subject = "NPL G3-L14 Logger - ALERT"
    Text = message

    # Format the message
    mMessage = ('From: %s\nTo: %s\nDate: \
            %s\nSubject: %s\n%s\n' %(From, To, Date, Subject, Text))

    print 'Connecting to Server'
    s = smtplib.SMTP(server)
    rCode = s.sendmail(From, To, mMessage)
    s.quit()

    if rCode:
        print 'Error Sending Message'
    else:
        print 'Message sent OK'#


#function to read internal temp and pressure from BMP085 in hPA (i.e. mbar)
#Needs additional import libraries

def read_pressure():
    (chip_id, version) = bus.read_i2c_block_data(addr, 0xD0, 2)
    cal = bus.read_i2c_block_data(addr, 0xAA, 22)
    ac1 = get_short(cal, 0)
    ac2 = get_short(cal, 2)
    ac3 = get_short(cal, 4)
    ac4 = get_ushort(cal, 6)
    ac5 = get_ushort(cal, 8)
    ac6 = get_ushort(cal, 10)
    b1 = get_short(cal, 12)
    b2 = get_short(cal, 14)
    mb = get_short(cal, 16)
    mc = get_short(cal, 18)
    md = get_short(cal, 20)
    bus.write_byte_data(addr, 0xF4, 0x2E)
    sleep(0.005)
    (msb, lsb) = bus.read_i2c_block_data(addr, 0xF6, 2)
    ut = (msb << 8) + lsb
    bus.write_byte_data(addr, 0xF4, 0x34 + (oversampling << 6))
    sleep(0.04)
    (msb, lsb, xsb) = bus.read_i2c_block_data(addr, 0xF6, 3)
    up = ((msb << 16) + (lsb << 8) + xsb) >> (8 - oversampling)
    x1 = ((ut - ac6) * ac5) >> 15
    x2 = (mc << 11) / (x1 + md)
    b5 = x1 + x2
    t = (b5 + 8) >> 4
    b6 = b5 - 4000
    b62 = b6 * b6 >> 12
    x1 = (b2 * b62) >> 11
    x2 = ac2 * b6 >> 11
    x3 = x1 + x2
    b3 = (((ac1 * 4 + x3) << oversampling) + 2) >> 2
    x1 = ac3 * b6 >> 13
    x2 = (b1 * b62) >> 16
    x3 = ((x1 + x2) + 2) >> 2
    b4 = (ac4 * (x3 + 32768)) >> 15
    b7 = (up - b3) * (50000 >> oversampling)
    p = (b7 * 2) / b4
    #p = (b7 / b4) * 2 # Use this verison for very high pressures
    x1 = (p >> 8) * (p >> 8)
    x1 = (x1 * 3038) >> 16
    x2 = (-7357 * p) >> 16
    p = p + ((x1 + x2 + 3791) >> 4)
```

```
    return p / 100.0

if __name__ == '__main__':
    main()
```

We also need to set up a blank file ready to receive the logged data.

```
sudo touch /usr/local/bin/temps.dat
sudo chmod 777 /usr/local/bin/temps.dat
```

We also need a pickle file that can be used to store alert status between calls to the script. The script `reset_pickle.py` creates the pickle file. It sets the alert triggers status flags to false. The `lcd.py` script will set these to true whenever an alert is triggered. The code in **lcd.py** check these flags to see if an alert was already triggered to prevent it sending an email every minute. The logic is that when an alert is triggered, it immediately sends an alert email and sets the relevant flag to be true. Next time round, it check the flags – it only triggers an alert if the flag was false and the alert condition is now fulfilled. If the flag is true, it does nothing until the alert condition is no longer fulfilled and then it sends an email to say things are back to normal and resets the alert flag.

```
sudo nano /usr/local/bin/reset_pickle.py
```

```
#!/usr/bin/env python

# This file is /usr/local/bin/reset_pickle.py

import pickle

MaxExceeded = False
MinExceeded = False

f = open('/usr/local/bin/lcd.pickle','w')
pickle.dump([MaxExceeded, MinExceeded], f)
f.close()
```

```
sudo chmod 777 /usr/local/bin/reset_pickle.py
```

Having prepared the script, we now run it to create the pickle file which stores the default state for the email alert triggers since they have not yet been created, but will be read form the pickle file by the script that drives the LCD.

```
sudo /usr/local/bin/reset_pickle.py
```

Setting the LCD script to run periodically

The LCD script has to run on a set interval. Running every minute is not too great a strain on resources, so we set up a `crontab` task to run this for us. Do not do this until: all the hardware is ready and running; the pickle file has been created; the Python scripts have been created and tested OK.

```
sudo crontab -e -u root
```

add the following line

```
* * * * * /usr/local/bin/lcd.py
```

That's it, the LCD should be updated every minute to show IP address, date and time and then humidity and temperatures. Cacti will be logging data into the RRDs and showing it in the consolidated graphs and the raw data will be logged every minute to **/usr/local/bin/temps.dat** – this file can be downloaded to a PC if needed for detailed processing, archiving, *etc*.

Installing Samba to allow file sharing

In order to be able to download data files form the Pi to a Windows network, we need to set up the file server system Samba.

```
sudo apt-get install samba samba-common-bin
```

We now need to configure Samba.

```
sudo nano /etc/samba/smb.conf
```

Most of the file contents is comment, the main lines we need to have in place are as follows. Here we assume that the main user is 'john' and his home directory is to be shared. Use CTRL-K to delete a line.

```
[global]
   workgroup = YOUR_WG_NAME
   server string = %h server
   wins support = yes
   dns proxy = no
   log file = /var/log/samba/log.%m
   max log size = 1000
   syslog = 0
   panic action = /usr/share/samba/panic-action %d
   encrypt passwords = true
   passdb backend = tdbsam
   obey pam restrictions = yes
   unix password sync = yes
   passwd program = /usr/bin/passwd %u
   passwd chat = *Enter\snew\s*\spassword:* %n\n
*Retype\snew\s*\spassword:* %n\n *password\supdated\ss
   pam password change = yes
   map to guest = bad user
   usershare allow guests = yes


[john]
comment = john home directory Share
```

```
path = /home/john
#read only = no
writable = yes
locking = no
force create mode = 0777
force directory mode = 0777
force user = john
create mask = 0777
directory mask = 0777
#guest only = Yes
guest ok = Yes
browseable = yes
write list = john

[root]
comment = root directory Share
path = /
#read only = no
writable = yes
locking = no
force create mode = 0777
force directory mode = 0777
create mask = 0777
directory mask = 0777
#guest only = Yes
guest ok = Yes
browseable = yes
write list = john
```

After saving the modified file, restart the samba service

```
sudo service samba restart
```

Check in a Windows Explorer window that the relevant share can be opened.

That's it, a fully working temperature/humidity logger with LCD display, webserver pages and graphs, raw data log and email alerting.

Using a graphical user interface on the Pi

The Pi is fully capable of running a Windows-like environment. At a command prompt, enter:

```
startx
```

and the system will start X-Windows, which is Linux's version of Windows.

Making sure the clock is correct

The Pi does not have a real-time clock built in – this means that when you power it off, it forgets what time it is. Or rather, it does not keep track of time when powered off – it remembers the time it was shut down, and starts again from there. This is a problem if we want to keep live graphs that are up to date – they will display

the wrong time and date. Whilst real time clock modules (with batteries) are available, the simplest solution for a Pi that is always to be available (i.e. on a network) is to ask it to synchronise its live clock with a reliable time source, such as a server running the Network Time Protocol (NTP). There are many available but I use two servers at NPL as my time source. The relevant software, NTP is already installed and

```
ntpq - p
```

gives useful information on the current list of NTP peers.

To explicitly add particular servers to the list of peers, edit the configuration file

```
sudo nano /etc/ntp.conf
```

Add the servers at the appropriate line, as per below.

```
# You do need to talk to an NTP server or two (or three).
server ntp1.npl.co.uk
server ntp2.npl.co.uk
```

Installing Tightvnc to allow remote desktop

On a Windows network, it is possible to take remote control of a PC and view the desktop as if working directly on the machine using Windows Remote Desktop. The same thing can be done on a Raspberry Pi using the TightVNC system – a server runs on the Pi and a client on the desktop. This is not a required part of the logger, but can be useful for maintenance and general experimentation – sometimes the Pi is located away from the user's desktop and monitor and this is easier than dragging the Pi to a monitor/keyboard.

On the Pi we install the server software first. Full details are already available on the internet:
http://www.penguintutor.com/linux/tightvnc

```
sudo apt-get update
sudo apt-get install tightvncserver
```

Follow the remainder of the instructions on the **penguintutor** website to set up the server and the Windows client. They use the username 'pi' – you should substitute your own. Critical commands to use are as follows:

```
sudo nano /etc/init.d/tightvncserver
```

Enter the contents as given on the website, such as below

```
#!/bin/sh
### BEGIN INIT INFO
# Provides:          tightvncserver
# Required-Start:    $local_fs
# Required-Stop:     $local_fs
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: Start/stop tightvncserver
### END INIT INFO
```

```
# More details see:
# http://www.penguintutor.com/linux/tightvnc

### Customize this entry
# Set the USER variable to the name of the user to start tightvncserver under
export USER='john'
### End customization required

eval cd ~$USER

case "$1" in
  start)
    su $USER -c '/usr/bin/tightvncserver :1 -geometry 1280x1024 -depth 24'
    echo "Starting TightVNC server for $USER "
    ;;
  stop)
    pkill Xtightvnc
    echo "Tightvncserver stopped"
    ;;
  *)
    echo "Usage: /etc/init.d/tightvncserver {start|stop}"
    exit 1
    ;;
esac
exit 0
```

```
sudo chown root:root /etc/init.d/tightvncserver
sudo chmod 755 /etc/init.d/tightvncserver
sudo update-rc.d tightvncserver defaults
sudo reboot
```

You should now install the Windows client software from **http://www.tightvnc.com/**

Adding a pressure sensor

The Bosch BMP085 pressure sensor is east to interface – it connects via the I2C bus and offers 0.01 hPa resolution at decent refresh rates (40 times per second is possible). To use one of these sensors we first have to enable the I2C bus.

```
sudo nano /etc/modules
```

Add these two lines to the end of the file:

```
i2c-bcm2708
i2c-dev
```

```
sudo apt-get install python-smbus
sudo apt-get install i2c-tools
```

```
sudo nano /etc/modprobe.d/raspi-blacklist.conf
```

If you do not have this file then there is nothing to do, however, if you do have this file, you need to edit it and comment out the lines below:

```
blacklist spi-bcm2708
blacklist i2c-bcm2708
```

.. by putting a # in front of them. Power down the Pi and connect the sensor. I bought a pre-mounted one on a breakout board from Proto-Pic (the device is quite small and surface mounted). Connect GND to any suitable GNP pin on the Pi GPIO (e.g. pin 6), 3v3 connects to pin 1, DATA connects to pin 3 and SCLK connect to pin 5. Happily, these are all pins not used by other devices, or ones that can be shared (*e.g.* GND and 3v3).

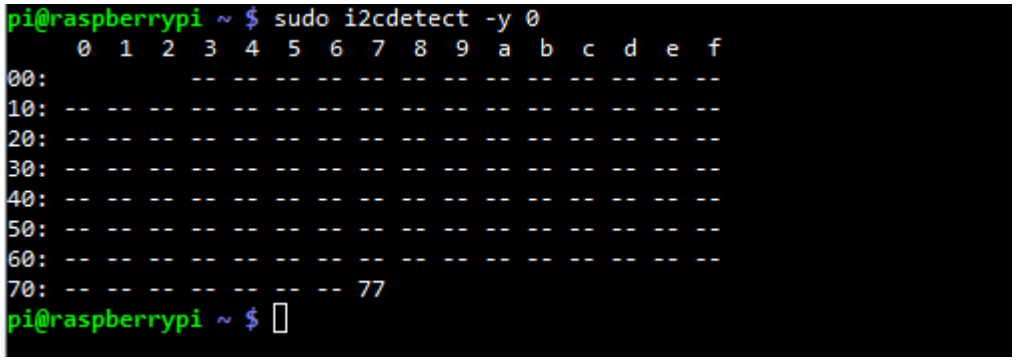
On a revision 1 Pi (256 MB memory) use this to check for the device

```
sudo i2cdetect -y 0
```

On a later Pi revision (512 MB memory) use this instead

```
sudo i2cdetect -y 1
```

You should see a device show up at address 0x77:



The device can now be read using some Python code (the code snippet below was modified from code available here:

http://blog.elevendroids.com/2013/03/barometric-pressure-sensor-bmp085-raspberry-python-and-things-in-the-works/

```
#!/usr/bin/env python

# This file is /usr/share/cacti/site/scripts/pressure.py

from time import sleep
from smbus import SMBus
from ctypes import c_short
addr = 0x77
oversampling = 3        # 0..3 : 3 gives highest resolution
bus = SMBus(0);         # 0 for R-Pi Rev. 1, 1 for Rev. 2

# function to return two bytes from data as a signed 16-bit value
```

```
def get_short(data, index):
    return c_short((data[index] << 8) + data[index + 1]).value

# function to return two bytes from data as an unsigned 16-bit value
def get_ushort(data, index):
    return (data[index] << 8) + data[index + 1]

(chip_id, version) = bus.read_i2c_block_data(addr, 0xD0, 2)
cal = bus.read_i2c_block_data(addr, 0xAA, 22)
ac1 = get_short(cal, 0)
ac2 = get_short(cal, 2)
ac3 = get_short(cal, 4)
ac4 = get_ushort(cal, 6)
ac5 = get_ushort(cal, 8)
ac6 = get_ushort(cal, 10)
b1 = get_short(cal, 12)
b2 = get_short(cal, 14)
mb = get_short(cal, 16)
mc = get_short(cal, 18)
md = get_short(cal, 20)
bus.write_byte_data(addr, 0xF4, 0x2E)
sleep(0.005)
(msb, lsb) = bus.read_i2c_block_data(addr, 0xF6, 2)
ut = (msb << 8) + lsb
bus.write_byte_data(addr, 0xF4, 0x34 + (oversampling << 6))
sleep(0.04)
(msb, lsb, xsb) = bus.read_i2c_block_data(addr, 0xF6, 3)
up = ((msb << 16) + (lsb << 8) + xsb) >> (8 - oversampling)
x1 = ((ut - ac6) * ac5) >> 15
x2 = (mc << 11) / (x1 + md)
b5 = x1 + x2
t = (b5 + 8) >> 4
b6 = b5 - 4000
b62 = b6 * b6 >> 12
x1 = (b2 * b62) >> 11
x2 = ac2 * b6 >> 11
x3 = x1 + x2
b3 = (((ac1 * 4 + x3) << oversampling) + 2) >> 2
x1 = ac3 * b6 >> 13
x2 = (b1 * b62) >> 16
x3 = ((x1 + x2) + 2) >> 2
b4 = (ac4 * (x3 + 32768)) >> 15
b7 = (up - b3) * (50000 >> oversampling)
p = (b7 * 2) / b4
#p = (b7 / b4) * 2 # Use this version for very high pressures
x1 = (p >> 8) * (p >> 8)
x1 = (x1 * 3038) >> 16
x2 = (-7357 * p) >> 16
p = p + ((x1 + x2 + 3791) >> 4)
print "p:"+str(p / 100.0);
```

The code above is suitable for a Cacti data input – however note that it needs to be run as **sudo** because it requires access to hardware. It outputs *e.g.* `p:1013.15` *i.e.* the pressure in 0.01 hPa resolution.

Cloning a working Pi

If you have a Pi that is working fine and you want a backup copy of the SD card, you can either shut it down then use the disk imaging tools to copy the SD card, or you can make a direct copy whilst the Pi is running, provided you have a USB SD card reader that the Pi can recognise. If **git** is not already installed, install it

```
sudo apt-get install git
```

```
git clone https://github.com/billw2/rpi-clone.git
```

```
cd rpi-clone
sudo cp rpi-clone /usr/local/sbin
```

The **rpi-clone** command takes one argument namely the name of the destination disk.

```
sudo rpi-clone sda -f
```

The **–f** switch forces initialisation of the destination partitions. Use the **–v** flag to list all files as they are copied, but this slows down the process somewhat. In this case, the destination CD card is inserted into an SD card reader, plugged into one of the USB sockets on the Pi. It shows up in the **/dev** directory as **sda**.

Graphing the temperature and humidity data

Once you have started to accumulate data in the log file (**/usr/local/bin/temps.dat**) you might want to plot them. Whilst you can do that in Excel, it is also possible in Python using **Matplotlib**. One easy was to do this is to install **Python(x, y)** (https://code.google.com/p/pythonxy/) which is a set of Python libraries for scientific work and includes **numpy** and **Matplotlib** designed to be run on Windows. The nicety is if you use the Spyder IDE that is included, you get a nice programming environment and can interact with the plots in real time.

Alternatively, you can install the necessary libraries on the RPi and then run the code below in a terminal window, however this gives no interaction and all you get is the PNG file output.

```
sudo apt-get install python-matplotlib
```

Here is the code to read in the data, generate the plot and save it to a PNG format file.

```
# Loads the Raspberry Pi logger data and plots it
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.dates as mdates
from matplotlib.ticker import AutoLocator, AutoMinorLocator
import datetime

datafile = 'E:/Desktop/temps.dat'
w, T1, T2, H = np.loadtxt(datafile, delimiter=',', usecols=(0, 1, 2, 3), unpack=True)

# Convert the epoch timestame (seconds) to matplotlib float format
dts = map(datetime.datetime.fromtimestamp, w)
fds = mdates.date2num(dts)

hfmt = mdates.DateFormatter('%Y/%m/%d\n %H:%M')

f1=plt.figure(1)
f1.autofmt_xdate()
sp1=plt.subplot(211)
sp1.plot(fds, T1,'r',fds,T2,'g')
sp1.xaxis.set_major_formatter(hfmt)
plt.xticks(rotation=270)
sp1.grid(True)
for tick in sp1.xaxis.get_major_ticks():
tick.label.set_fontsize(8)
for tick in sp1.yaxis.get_major_ticks():
```

```
tick.label.set_fontsize(8)
sp1.xaxis.set_major_locator(AutoLocator())
sp1.xaxis.set_minor_locator(AutoMinorLocator(12))
plt.tight_layout()

sp2=plt.subplot(212, sharex=sp1)
sp2.plot(fds, H, 'b')
sp2.xaxis.set_major_formatter(hfmt)
plt.xticks(rotation=270)
sp2.grid(True)
for tick in sp2.xaxis.get_major_ticks():
tick.label.set_fontsize(8)
for tick in sp2.yaxis.get_major_ticks():
tick.label.set_fontsize(8)
sp2.xaxis.set_major_locator(AutoLocator())
sp2.xaxis.set_minor_locator(AutoMinorLocator(12))
plt.tight_layout()
plt.savefig('E:/Desktop/temps.png')
```

And this is what it saved in the same directory as the data: