Beer Fermentation Control System – Final Documentation

David Sorber
525.743
May 6, 2013

Project Description

The purpose of this project is to develop a control system to control the temperature of a batch of home-brewed beer while it is fermenting. Different types of beer require different fermentation temperatures. Ales typically ferment between 68°F to 72°F, while lagers typically ferment between 45°F to 55°F. This control system will allow either type of beer to ferment at the optimum temperature regardless of the ambient temperature. This project assumes that the fermenter will always be placed in a location where ambient temperature is higher than the desired fermentation temperature. As such heating the fermentor to account for lower ambient temperatures is not considered.

After the beer has been transferred into a fermenter, the fermenter will be placed into a mini refrigerator which will be used to cool the beer while it is fermenting. The control system will then monitor the temperature of the beer inside the fermenter as well as outside the fermenter (but inside the refrigerator). The control system will also provide both local and remote system monitoring capabilities.

Capabilities

- monitor the temperature of beer while it is fermenting
- enable power to the refrigerator when the beer temperature exceeds a predefined threshold temperature
- disable power to the refrigerator when the beer temperature falls below a predefined threshold temperature
- provide a means to locally monitor system status (i.e. while physically standing next to the system)
- provide a means to remotely monitor system status (i.e. while not physically standing next to the system)

Limitations

This project is not intended to interface directly with the control circuitry of the mini refrigerator. Instead, the refrigerator will be set on its highest (i.e. coldest) setting and power to the refrigerator will be turned on and off as needed in order to regulate the internal temperature. Also both the local and remote monitoring functions are not intended to provide manual override controls.

Approach

The control system will use a Raspberry Pi computer. The Raspberry Pi is a small, low cost, ARM-based computer which is powerful enough to run a standard desktop version of the Linux operating system. There is a large hobbyist community that actively provides support for all things related to the Raspberry Pi. The Raspberry Pi includes an ARM CPU, 512MB of RAM, 2 USB ports, 10/100 Ethernet, HDMI video output, component video output, analog audio output, and a 26 pin header that includes power, ground, I^2C, SPI, and several GPIOs. An IEEE 802.11 wireless (i.e. “WIFI”) adapter will be added to the Raspberry Pi to allow remote communication for monitoring purposes. The Raspberry Pi will periodically send temperature and relay status information to a remote web server which will allow the status of of the entire control system to be monitored remotely.

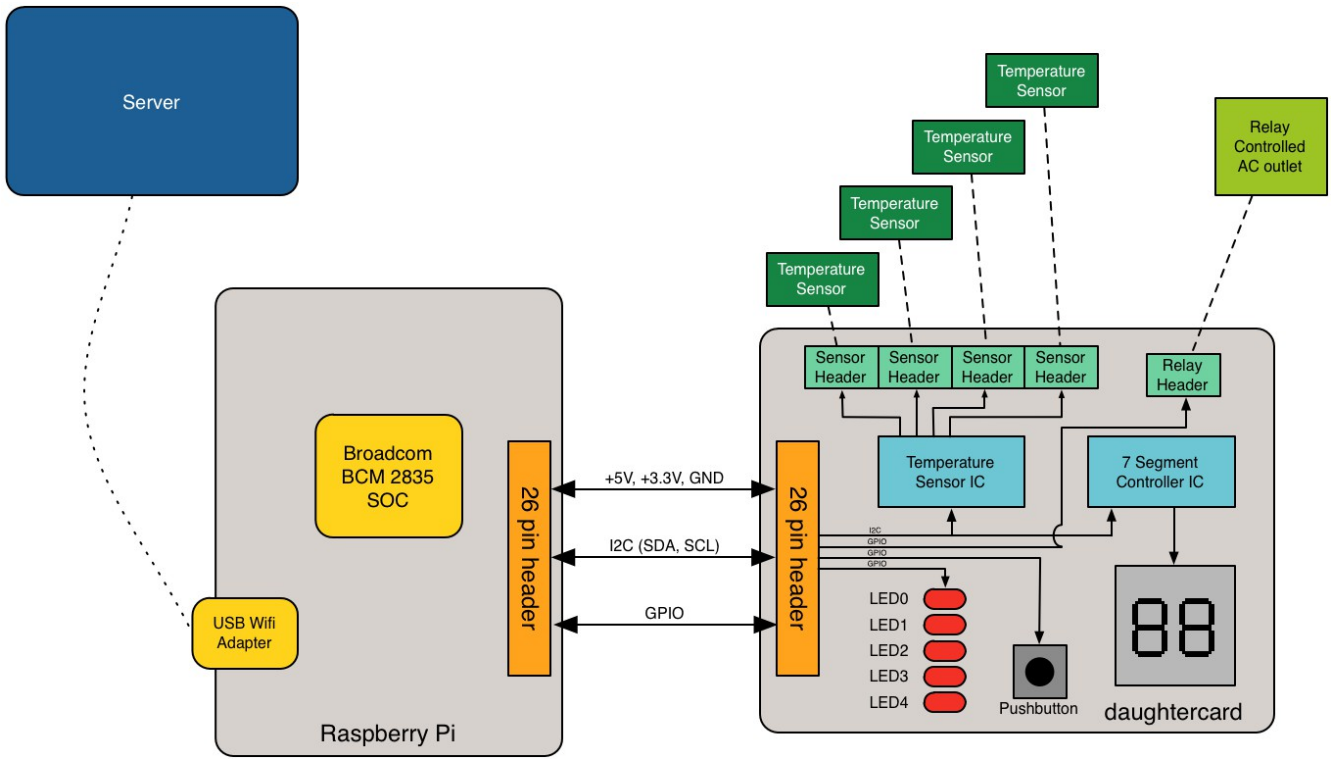
A “daughtercard” will be developed to provide functionality that is specific to the control system but is not found on the Raspberry Pi. The daughtercard will be attached to the Raspberry Pi using the 26 pin header mentioned above. The daughtercard will contain components necessary for remotely sensing temperature and for controlling the external relay to control the mini refridgerator. The daughtercard will also contain several LEDs and a pushbutton to allow the temperature of each remote sensor and the status of the external relay to be monitored locally.

Four remote, that is connected to the daughtercard via wires, temperature sensors will be used. Two will be placed inside the fermentor itself to monitor the temperature of the beer more directly. Two additional temperature sensors will be placed inside the refrigerator but outside of the fermentor to monitor the ambient temperature inside the refrigerator. This sensor arrangement will allow both temperature regions to be monitored and correlated.

An external DC relay will be used to control the 120 volt AC (i.e. “wall power”) to the mini refrigerator. The relay will be connected to a header on the daughtercard and controlled by 5 volts DC as supplied from the Raspberry Pi.

Functional Description

Block Diagram



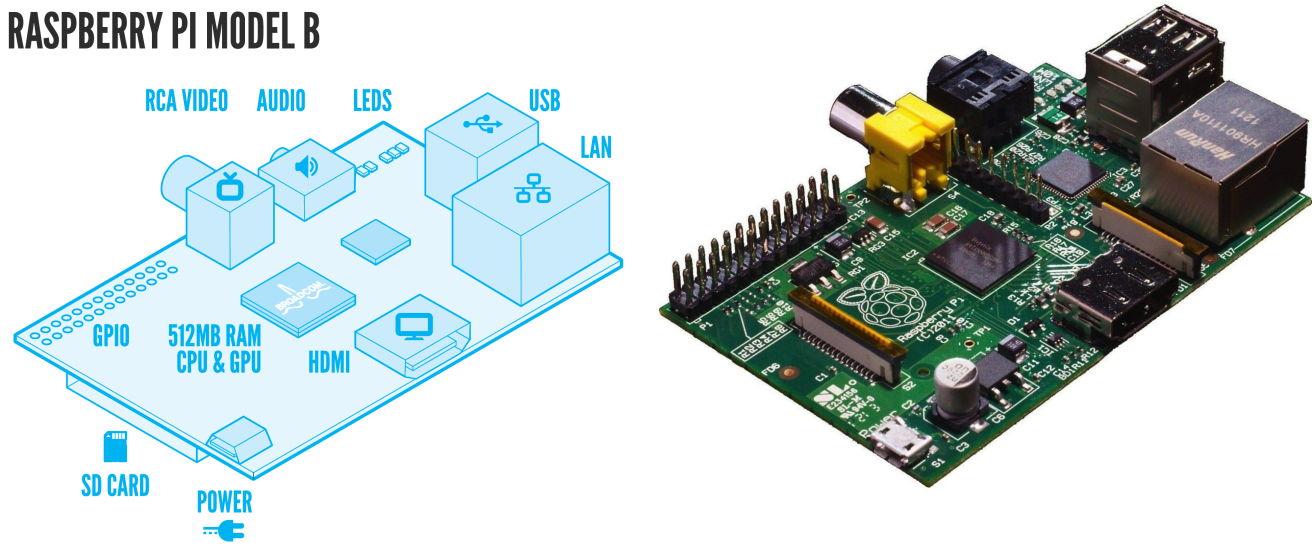
Functional Component Breakdown

- Raspberry Pi board
  - Broadcom BCM2835 SOC
    - ARM1176JZ-F CPU
    - 512 MB RAM
  - USB Wifi adapter
- daughtercard
  - 26 pin (2x13) header
  - remote temperature sensor headers
  - temperature sensor controller
  - 7 segment display
  - 7 segment display controller
  - relay header
  - pushbutton
  - LEDs
- DC controlled 120V AC relay

- remote web server

Materials

Raspberry Pi



The Raspberry Pi is a small, low cost, ARM-based computer which is powerful enough to run a standard desktop version of the Linux operating system. There is a large hobbyist community that actively provides support for all things related to the Raspberry Pi. The Raspberry Pi includes an ARM CPU, 512MB of RAM, 2 USB ports, 10/100 Ethernet, HDMI video output, component video output, analog audio output, and a 26 pin header.

The Raspberry Pi will run the Raspbian Linux distribution which is based off of the Debian Linux distribution and is customized to fully support the Raspberry Pi platform. Raspbian includes a large repository of precompiled software packages for the Raspberry Pi.

The Raspberry Pi will use the following peripherals:

- micro USB power adapter – MCM MWS5-0501000UC/M - External plugin 1A micro USB
  - The power adapter will supply power to the Raspberry Pi and all attached peripherals including the daughter card.
- SD card – SanDisk Extreme 16 GB SDHC Class 10 UHS-1 Flash Memory Card SDSDX-016G-X46
  - The SD card will be the primary nonvolatile storage for the Raspberry Pi and will hold the Raspbian Linux operating system along with all code needed by the system. A 2 GB or larger SD card is required by Raspbian, so this 16 GB card will have ample free space.
- USB Wifi Adapter – Edimax EW-7811Un 150 Mbps Wireless 11n Nano Size USB Adapter
  - The wifi adapter will allow the Raspberry Pi to connect to an existing Wifi network and send data to a web server on the network. The Edimax EW-7811Un contains a Realtek RTL8188CUS chip. This chipset is supported by the open source rtl8192cu driver, which is included by default with Raspbian Linux. The EW-7811Un is powered entirely via the on board USB ports. The EW-7811Un has been confirmed to work both by the Raspberry Pi community and by my own testing.

The Raspberry Pi includes a 26 pin (2 x 13 pin) header. The header includes connections for +5 V, +3.3 V, ground, and several GPIOs which operate at 3.3 V and can be configured to source or sink 2 mA to 16 mA. Several of the GPIOs are dual use can be used for I^2C, SPI, PWM, and a serial UART. Each GPIO can be configured to interrupt on a high level, low level, rising edge, falling edge, or any change.

For the power pins on the header, the maximum current draw is 50 mA for the 3.3 V pins and 300 mA for the 5 V pins.

Daughtercard

The daughtercard is a custom designed PCB. The PCB was designed using Altium Designer 10. I have access to this software at my workplace. I also have access to several coworkers who are very experienced users of Altium Designer. My workplace provided the necessary soldering and test equipment to assemble to perform electrical testing on the board.

The daughtercard was fabricated by OSH Park. OSH Park charges $5 per square inch for 3 copies of a 2 layer design. They advertise a 12 day typical turnaround time. OSH Park also fabricates 4 layer designs and charges $10 per inch for 3 copies.

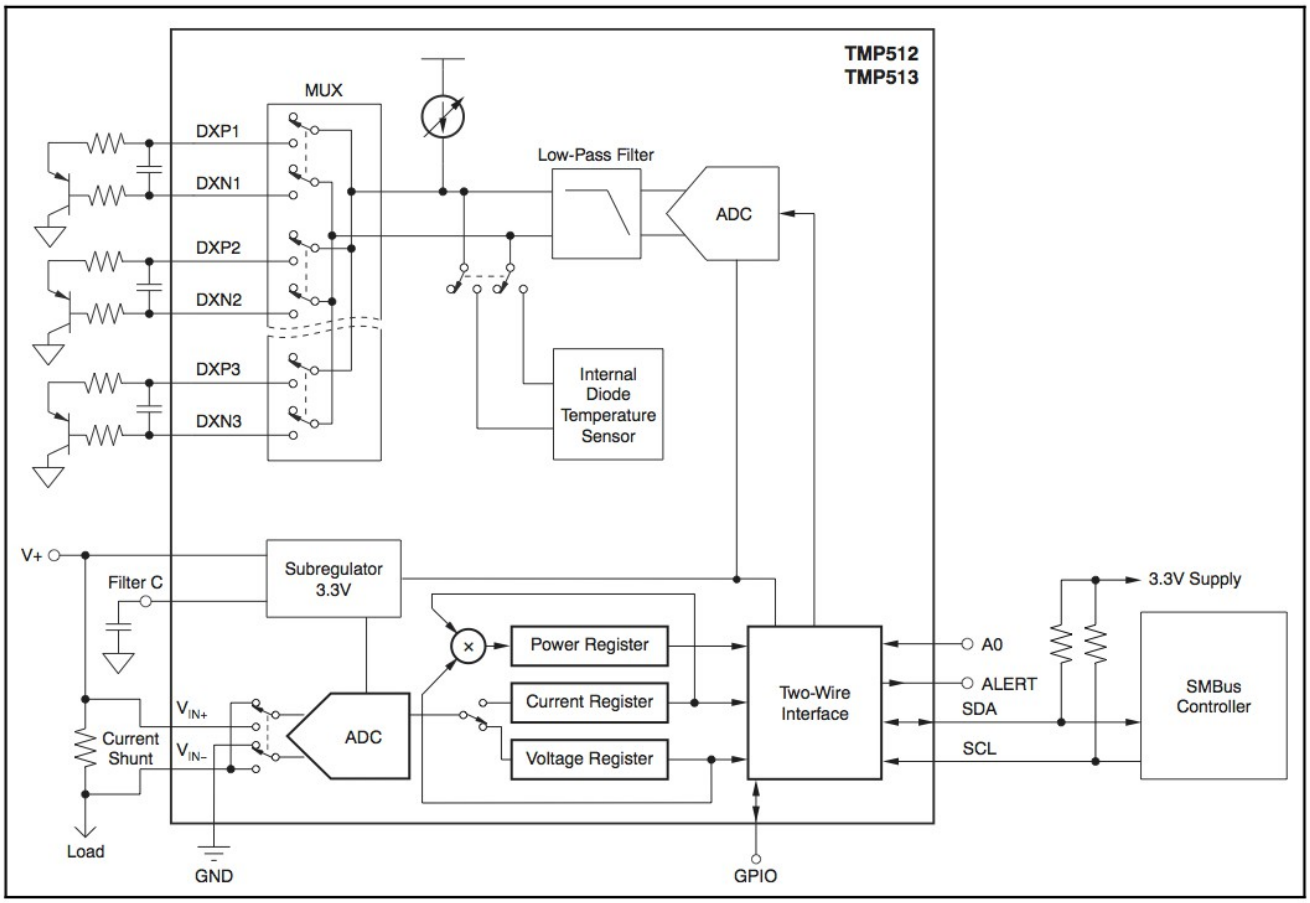
Temperature Sensor Controller

NOTE: I proposed using theTI LM95234 in my PDR however the LM95234 is only available in a QFN package that would be nearly impossible to hand solder. Therefore I selected a different part that is available in a package that is easier to hand solder.

The Texas Instruments TMP512 is used to control and interface to the remote temperature sensors. The TMP512 provides two remote temperature sensor channels, a local temperature sensor, and a current shunt monitor. The local temperature sensor and the current shunt monitor will not be used for the purposes of this project. The TMP512 stores temperature measurements in a 13-bit register where each bit represents 0.0625°C. This means that the TMP512 has a temperature measurement range of -256°C to 256°C, which is more than sufficient for the purposes of this project. The TMP512 uses remote diode-connected NPN or PNP type transistors to sense temperature. The TMP512 uses a 2 wire SMBus interface (SCL and SDA) for configuration and data. SMBus is compatible with the I^2C bus provided by the Raspberry Pi (see more details in the I^2C interface section). The TMP512 can be powered from a single +3V to +26V supply and draws a maximum of 1.4mA of current.

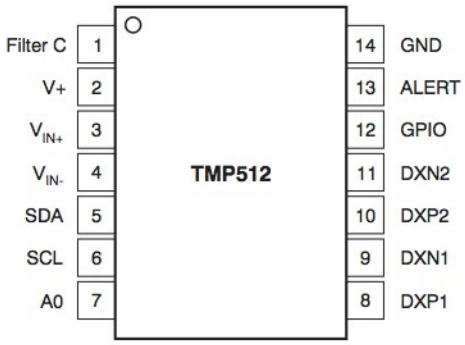
The TMP512 automatically cancels up to 3kΩ series line resistance from PCB traces and external wires. The TMP512 also has a 65kHz filter built in to each remote temperature sensing channel to reduce the effects of noise. The datasheet recommends a that suitable capacitor, between 100pF and 1nF, be placed differentially across DXP and DXN of each channel to further reduce the effects of any unwanted coupled signal noise. A 560pF capacitor will be used for each channel on the daughtercard based on this recommendation.

Note that the TMP512 datasheet shows several different methods in which the remote sensing transistors may be connected. The diagram below illustrates one of these methods however the method that will be used by this project is illustrated in the following section titled Remote Temperature Sensors. The purpose of the diagram below is only to show the overall architecture of the TMP512.



In order to support four remote temperature sensors, two TMP512's will be used. In order to connect both TMP512's to the same SMBus/I^{2}C bus, each device must be configured to have a separate slave address. The A0 pin was designed to provide this functionality. When the A0 pin is connected to ground the slave address of the device is set to 0x7C. When the A0 pin is connected to V_{DD} the slave address of the device is set to 0X7D. Appropriately setting the A0 pin will allow two TMP512's to be connected to the same bus.

D PACKAGE
SO-14
(TOP VIEW)

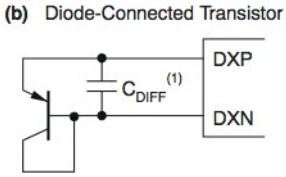


The TMP512 is available in two packages, a 14 pin SOIC package and a 16 pin QFN package. The 14 pin SOIC package will be used to facilitate hand soldering.

Remote Temperature Sensors

The TMP512 is designed to be used with either substrate transistors found in ASICs or with discrete PNP or NPN transistors. The TMP512 features ideality factor calibration for calibrating remote sensing diodes (diode-connected transistors) by comparing them to ideal diodes. Based on the default ideality factor calibration settings, the TMP512 datasheet recommends standard that 2N3904 NPN or 2N3906 PNP transistors be used as remote sensors. Based on this recommendation, the remote sensors used by this project are Fairchild 2N3906 PNP discrete transistors, in the TO-92 through-hole package. By using 2N3906 transistors, the default n-factor ideality calibration settings on the TMP512 will not need to be adjusted.

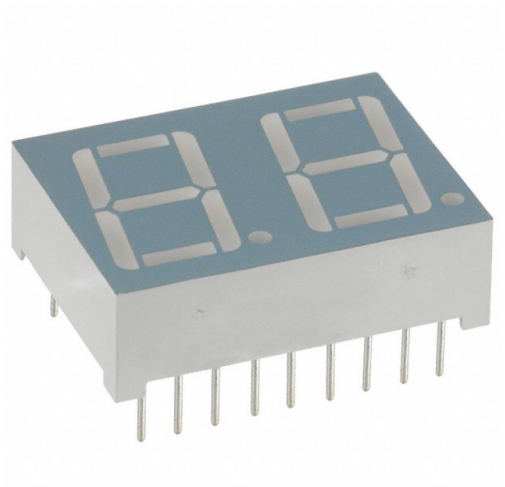
The 2N3906 is diode-connected such that the collector is connected to the base and the base emitter junction is used for temperature sensing, as shown below.

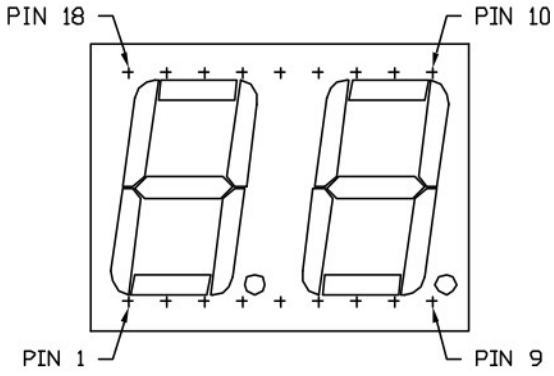


The TMP512 datasheet recommends adding a bypass (i.e. filtering) capacitor between the DXP and DXN inputs to minimize any noise effects. The recommended value of the filtering capacitor is 100pF to 1nF, therefore a 560pF capacitor will be used.

The two remote temperature sensors that will be placed into the fermentor (and therefore in direct contact with the beer) will be sealed in food-grade plastic tubing and sanitized before placement.

7 Segment Display



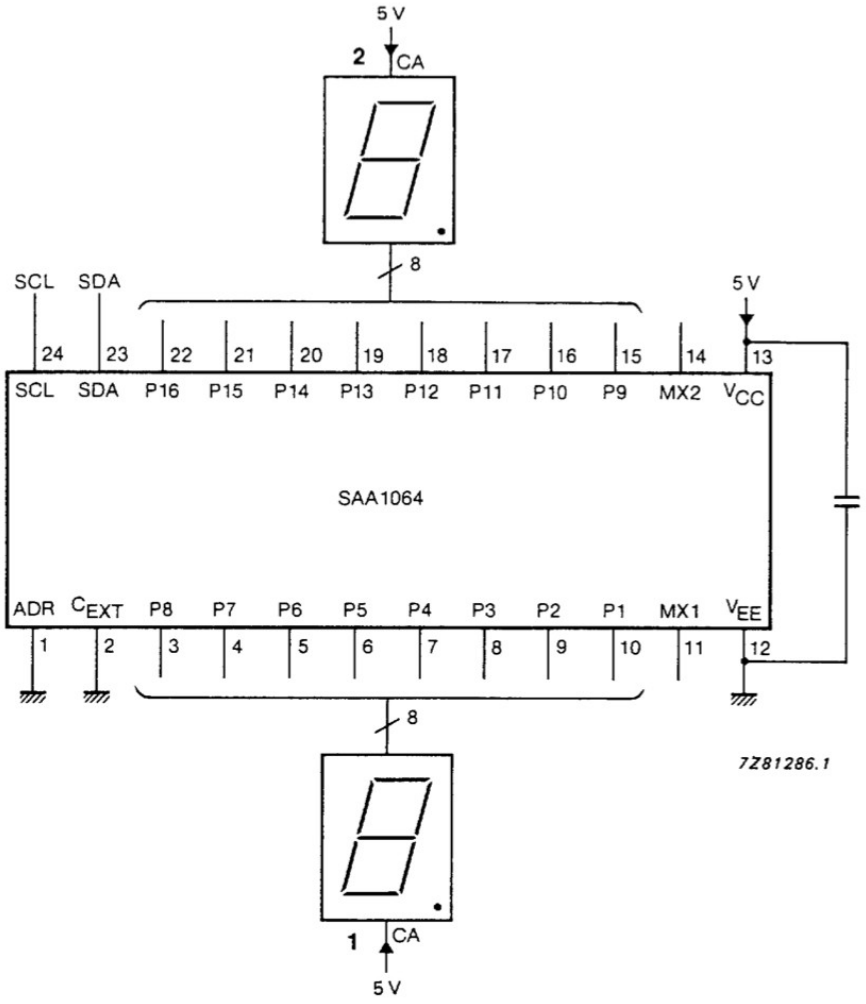


A 7 segment display will be used to display the temperature readings in degrees Fahrenheit (temperature readings are rounded to the nearest whole degree) when requested by the user. The Lumex Optoelectronics LDD-A516RI two digit 7 segment display will be used. The LDD-A516RI is a red common anode LED display that includes two digits and two decimal points for a total of 18 pins (including the two anodes). The LDD-A516RI requires a max input current of 30 mA, with a typical input current of 10 mA, and has a forward voltage

of 1.7 volts.

7 Segment Display Controller

In order to control the 7 segment display from the I²C bus a separate driver/controller chip is required. The NXP SAA1064 has been selected for this purpose. The SAA1064 is an I²C bus interface LED driver that is capable of driving up to four 7 segment digits. The SAA1064 requires an input voltage of 5V and features configurable current sink outputs that can each sink up to 21 mA.



The SAA1064 is available in a 24 pin SOIC package. The SAA1064 features an address pin that allows multiple

SAA1064's to be used on the same I²C bus. If the ADR pin is grounded then the I²C slave address is set to 0x38, which will be sufficient for the one SAA1064 in this design.

PINNING

SYMBOL	PIN	DESCRIPTION
ADR	1	I²C-Bus slave address input
C_{EXT}	2	external control
P8 to P1	3-10	segment output
MX1	11	multiplex output
V_{EE}	12	ground
V_{CC}	13	positive supply
MX2	14	multiplex output
P9 to P16	15-22	segment output
SDA	23	I²C-Bus serial data line
SCL	24	I²C-Bus serial clock line

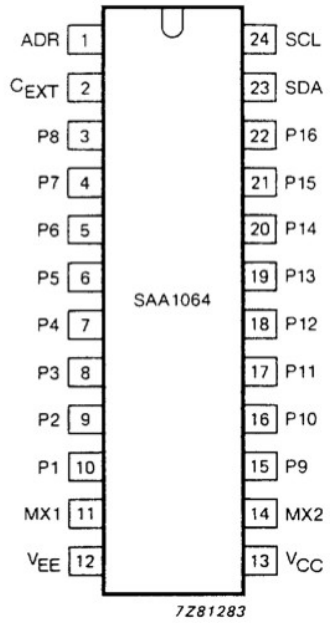
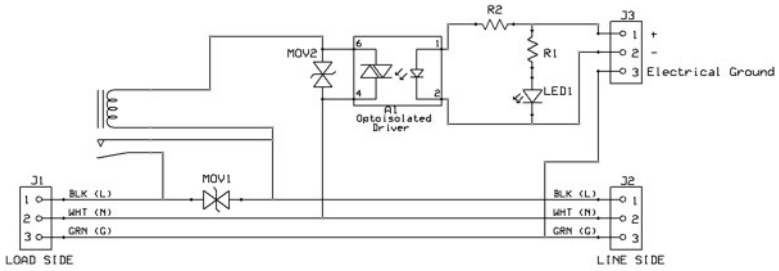


Fig.2 Pinning diagram.

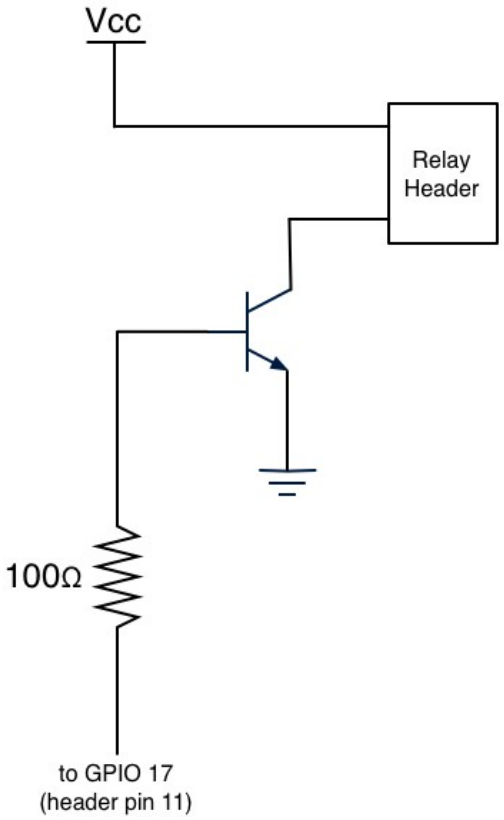
DC controlled 120V AC relay

The PowerSwitch Tail II is a DC voltage controlled 120 VAC relay. The PowerSwitch Tail II is intended to allow devices powered by 120 VAC wall power to be controlled by small DC voltages from a microcontroller

development board.



The PowerSwitch Tail II switches up to 15 A ,120 VAC with a 3 mA, 3 VDC to 30 mA, 12 VDC control signal.

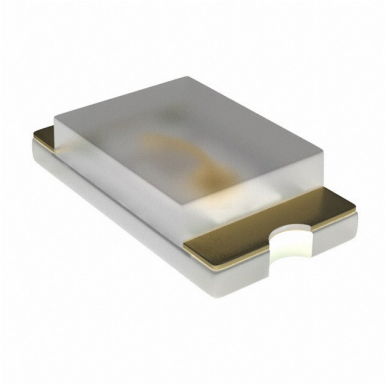


Although the PowerSwitch Tail II can be directly attached to the GPIO of a microcontroller board such as the Raspberry Pi, an isolation circuit (shown above) was developed to isolate the relay control voltage from the GPIO output. Given the design of both the PowerSwitch Tail II and the Raspberry Pi, this circuit is not strictly necessary however it is good practice to isolate relays from the signals that control them. The isolation circuit designed for this project uses a 2N3904 NPN transistor. The positive terminal of the relay header is connected to V_{CC} (V_{CC} = 5V) and the negative terminal is connected to the collector of the transistor. The emitter of the transistor is connected to ground and the base is connected to the controlling GPIO of the Raspberry Pi through a current-limiting 100Ω resistor. When the GPIO is low (i.e. off) the base is pulled low and the base-emitter voltages are equal, while the collector voltage is greater than the emitter voltage. This means that the transistor is “off”, that is, it is in cut-off mode and no current flows between the collector and emitter. When the GPIO is high (i.e. on) the base voltage is greater than the emitter voltage and the collector voltage is greater than the emitter voltage. This means that the transistor is “on”, that is, it is in forward active mode and current flows between the collector and emitter.

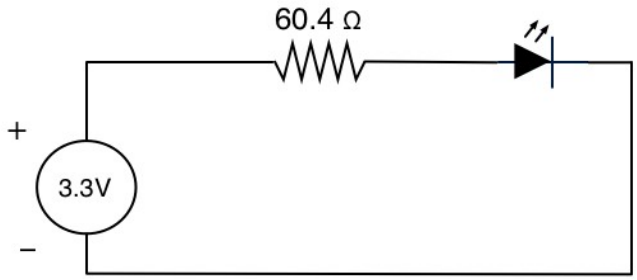
Pushbutton

The Tyco Electronics Connectivity FSM2JSMA off-momentary pushbutton switch will be connected to a GPIO input on the Raspberry Pi. It will used to provide input to the Raspberry Pi and cycle through each of the temperature sensors. Switch debouncing is performed in software using the RPIO library (see more details in the Software section).

LEDs



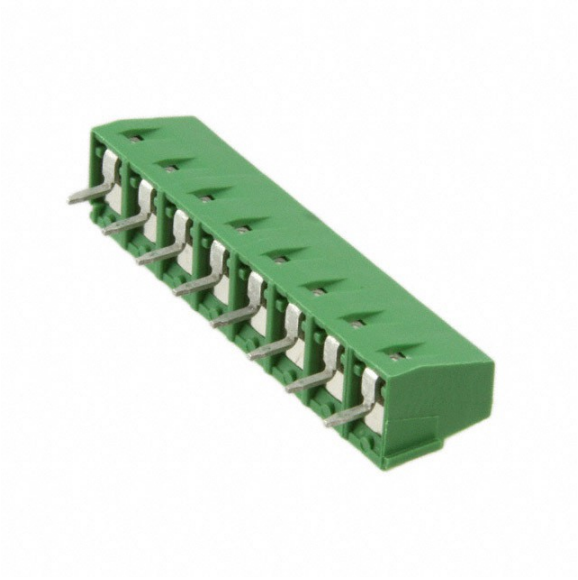
Five discrete LEDs will be used on the daughtercard. Four will be used to indicate which temperature sensor's value is currently being displayed on the 7 segment display. The fifth LED will be used to indicate if the external AC relay is on or off. The Lite-on Incorporated LTST-C171GKT green 0603 package LED will be used.



The LTST-C171GKT requires a typical operating current of 20 mA and has a 2.1 V forward voltage. The LEDs will be connected to 3.3 V GPIO outputs from the Raspberry Pi. Therefore each LED will need a 60.4 Ω (rounded up from 60 Ω) current limiting resistor in series.

Headers

The external AC relay will be connected to the daughtercard via wires attached to a two conductor screw terminal header. The Tyco Electronics Connectivity 282836-2 has been selected.



The four remote temperature sensors, each with two leads, will be connected via wires attached to an 8 conductor screw terminal header. The Tyco Electronics Connectivity 282836-8 has been selected.

Interface Descriptions

Raspberry Pi to Web Server

The Raspberry Pi will be connected to the web server via Wifi. The Raspberry Pi communicates with the web server using a custom protocol on top of TCP/IP. A daemon (written in Python) running on the Raspberry Pi will open a TCP socket connection to a daemon (also written in Python) on the web server. The daemon running on the Raspberry Pi will then send serialized (pickled in Python-speak) Python objects to the daemon on the web server. The daemon on the web server will receive each serialized object, deserialize (unpickle) it and then insert the sensor and relay status information that it contains into the database. The daemon running on the Raspberry Pi will send temperature readings from all four sensors and the status of the relay (i.e. on or off) to the daemon running on the web server once every second.

Raspberry Pi to Daughtercard

A 26 conductor ribbon cable will be used to physically connect the header on the Raspberry Pi to the header on the daughtercard. This connection includes 5 V, 3.3 V, ground, I^2C bus, and several GPIOs as described above.

The following pins are used on the daughtercard (if not specified below the pin is unused, see also schematic):

- Pin 2 – +5V
- Pin 3 – SDA, I^2C data
- Pin 4 – +5V
- Pin 5 – SCL, I^2C clock
- Pin 6 – ground
- Pin 9 – ground
- Pin 11 – GPIO 17, output to control relay header circuit
- Pin 12 – ground
- Pin 15 – GPIO 22, input for push button
- Pin 16 – GPIO 23, output for LED 1: relay status
- Pin 17 – 3.3V, power to push button switch
- Pin 18 – GPIO 24, output for LED 2: sensor 1 selected
- Pin 20 – ground
- Pin 22 – GPIO 25, output for LED 3: sensor 2 selected
- Pin 24 – GPIO 8, output for LED 4: sensor 3 selected
- Pin 25 – ground
- Pin 26 – GPIO 7, output for LED5: sensor 4 selected

SMBus/I^2C Bus

The SMBus used by the TMP512 was derived from the I^2C bus specification, however the SMBus voltage and timing requirements are more strictly defined than those for the I^2C bus. As long as the SMBus specifications are met it is entirely possible for SMBus and I^2C bus devices to coexist and interoperate on the same bus.

Since SMBus is a subset of I^2C bus specification, a SMBus will be compatible with I^2C bus capable devices. Therefore the two wire bus implemented on the daughtercard and controlled by the Raspberry Pi will conform with the SMBus specification. The electrical and important timing characteristics of the SMBus specification are as follows:

- V_{DD} – min 2.7 V, max 5.5 V
- V_{IL} (input low voltage) – max 0.8 V

- V_{IH} (input high voltage) – min 2.1 V, max V_{DD}
- I_{PULLUP} (pull-up resistor current) – min 100 µA, max 350 µA
- F_{SMB} (SMBus frequency) – min 10 KHz, max 100 KHz

For the purposes of this project, V_{DD} will be tied to 3.3 V so a pull up resistor of at least 9.5 kΩ is required order to not exceed the maximum current value of 350 µA specified by the SMBus specification.

Temperature Sensor Header to Temperature Sensor

The temperature sensor transistors will be physically connected via 26 gauge wire to the headers on the daughtercard.

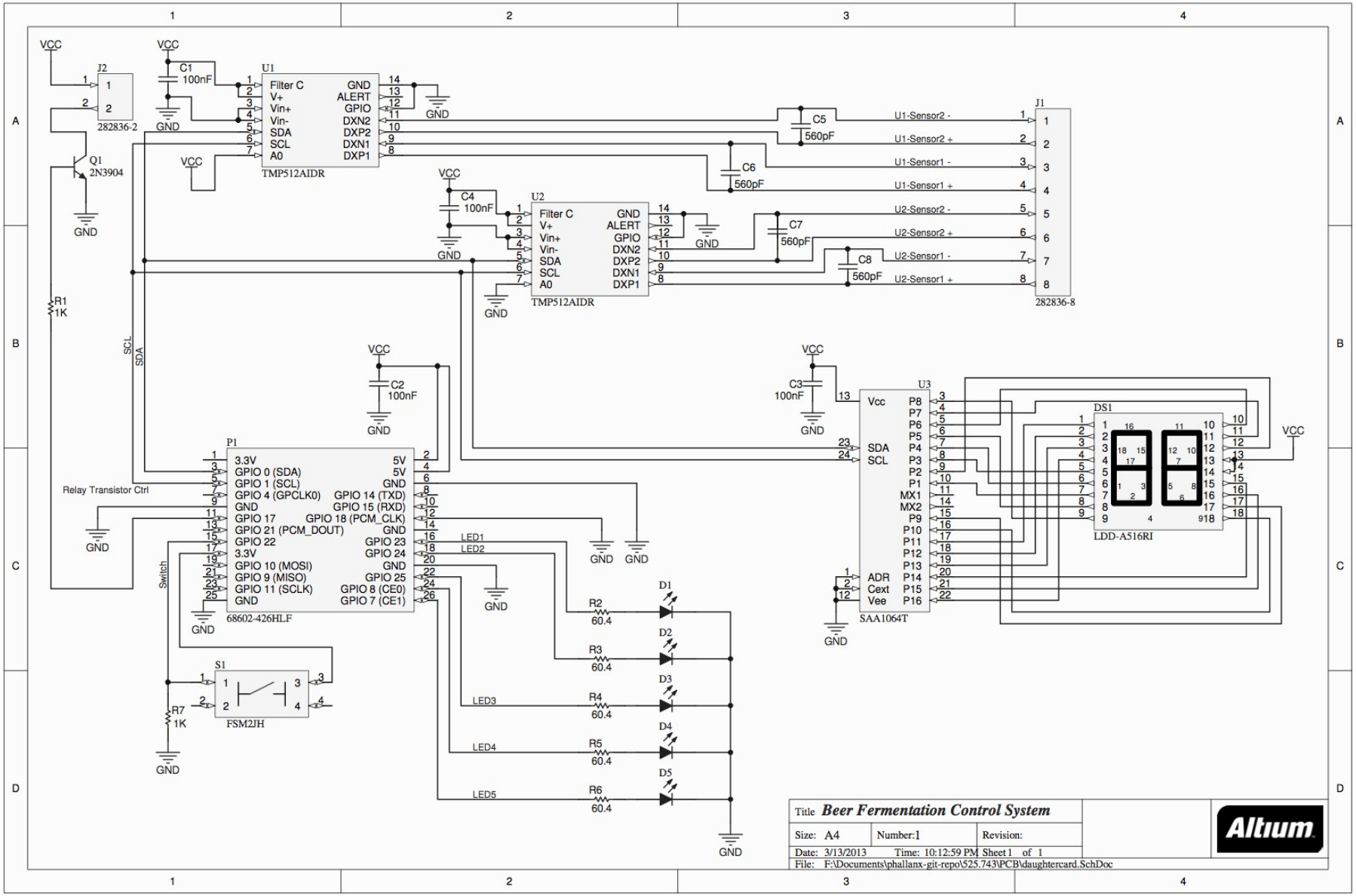
Relay Header to Relay

The relay will be physically connection via 12 gauge wire to the header on the daughtercard.

Bill of Materials

Ref. Designator	Description	Manufacturer	Manufacturer Part No.	Supplier	Quantity	Quantity Extra	Unit Cost	Cost
R1, R7	0805 1K resistor	Rohm Semiconductor	MCR10ERTJ102	Digikey	2	2	$0.03	$0.13
R2, R3, R4, R5, R6	0805 60.4 resistor	Panasonic Electrical Components	ERJ-6ENF60R4V	Digikey	5	2	$0.10	$0.70
C1, C2, C3, C4	0805 100nF capacitor	TDK Corporation	C2012X7R1H104K085AA	Digikey	4	1	$0.07	$0.33
C5, C6, C7, C8	0805 560pF capacitor	Yageo	CC0805KRX7R9BB561	Digikey	4	1	$0.05	$0.26
D1, D2, D3, D4, D5	0805 green LED	Lite-on Incorporated	LTST-C171GKT	Digikey	5	1	$0.25	$1.49
Q1	through hole 3904 transistor	Fairchild Semiconductor	2N3904TFR	Digikey	1	0	$0.19	$0.19
DS1	2 digit common anode 7-segment display	Lumex Opto	LDD-A516RI	Digikey	1	0	$3.13	$3.13
S1	off momentary pushbutton switch	TE Connectivity	FSM2JSMA (2-1437565-9)	Digikey	1	0	$0.31	$0.31
U1, U2	I2C temperature sensor IC	Texas Instruments	TMP512AIDR	Digikey	2	0	$3.76	$7.52
U3	7-segment display driver	NXP Semiconductor	SAA1064T	Digikey	1	0	$2.88	$2.88
J1	8 position screw header	TE Connectivity	282836-8	Digikey	1	0	$2.51	$2.51
J2	2 position screw header	TE Connectivity	282836-2	Digikey	1	0	$0.56	$0.56
P1	26 pin (2x13) 0.100 mil spacing	FCI	68602-426HLF	Digikey	1	0	$0.73	$0.73
--	through hole 3906 transistor	Fairchild Semiconductor	2N3906TAR	Digikey	4	2	$0.20	$1.20
--	ribbon cable	Assmann WSW Components	H3BBS-2606G	Digikey	1	0	$1.54	$1.54
								$23.48
--	PowerSwitchTail II	PowerSwitch	Tail II	Sparkfun	1	0	$28.95	$28.95
--	Raspberry Pi			Newark	1	0	$35.00	$35.00
--	RPI Enclosure			Newark	1	0	$7.35	$7.35
--	Power Adapter Micro USB 1Amp			Newark	1	0	$6.99	$6.99
--	Edimax EW-7811Un Wireless USB Adapter			Amazon	1	0	$14.68	$14.68
--	SanDisk Extreme 16 GB SDHC Class 10 UHS-1 Flash Memory Card			Amazon	1	0	$18.44	$18.44
								$82.46
--	PCBs			OSHPark	3	0		$33.13
							Total:	**$168.02**

Schematic


Title Beer Fermentation Control System
Size: A4
Number:1
Revision:
Date: 3/13/2013
Time: 10:12:59 PM
Sheet 1 of 1
File: F:\Documents\phallanx-git-repo\525.743\PCB\daughtercard.SchDoc
Altium
TMP512AIDR
282836-2
282836-8
68602-426HLF
SAA1064T
LDD-A516RI
FSM2JH
2N3904
Relay Transistor Ctrl
Switch
SCL
SDA
LED1
LED2
LED3
LED4
LED5
VCC
GND

Software

Software Overview

The project software consists of three main parts, the control client daemon running on the Raspberry Pi, the server daemon running on the remote server, and the WSGI (Web Server Gateway Interface) web application that also runs on the remote server. All three pieces of software are written in Python (the web application also contains some client-side javascript). The control client daemon controls the devices on the daughtercard (the LEDs, the pushbutton, the seven segment display, the relay, and the temperature sensors) and sends temperature and relay status readings to the server daemon over a TCP socket once every second. The control client daemon uses the open source RPIO library to interface with the GPIO on the Raspberry Pi board. The client daemon also uses a collection of custom developed utilities to interface with the components on the daughtercard. These utilities are collectively referred to as "bfcslib". The server daemon waits to receive readings from the client and when it does, it inserts the reading values along with the current time into an SQLite database. The WSGI web application displays the sensor and relay status data that is stored in the SQLite database to a user using a web browser that is connected to the server.

Raspberry Pi Platform

The Raspberry Pi runs Raspbian, a customized version of the Debian Linux operating system specifically designed to run on the Raspberry Pi. The Raspberry Pi requires the following software to be installed:

- Python – version 2.7.3 is included by default in Raspbian
- RPIO – Python package for controlling the GPIO on the Raspberry Pi. The RPIO library is not included with Raspbian but was downloaded from the Python package index and installed manually.
- python-smbus – Python package for communicating with SMBus/I^2C devices. The python-smbus package is installed by default on Raspbian.

Server Platform

The remote web server runs Ubuntu Linux version 12.04 Server edition. The following software must be installed on the server:

- Python – version 2.7.2 is included by default in Ubuntu 12.04 Server
- Apache – open source web server
- mod_wsgi - Apache module to integrate WSGI applications into Apache
- Genshi – a Python package for XML and HTML templating

bfcslib

Bfcslib is a collection of utilities that were written to control the devices on the daughtercard. Most of these utilities started as a test program that were used to exercise and test a specific function of the daughtercard during initial testing.

- relay.py – contains the Relay object which encapsulates the relay status LED (GPIO 23) and GPIO 17 that is used to control the relay. The Relay object has the following methods:
  - on() - turns the relay and the relay status LED on
  - off() - turns the relay and the relay status LED of
- sensor_reading.py – contains the SensorReading object which holds a timestamp, four temperatures readings and a relay status reading. The SensorReading object represents a single reading of all four temperature sensors and the relay status at a single point in time. The SensorReading object is created by the client daemon, serialized (pickled), and then sent to the server daemon where it is deserialized (unpickled).
- sensor.py – contains the Sensor object which is used to logically group an LED object and a temperature reading function from the TMP512 object. Together these two items represent a sensor. The Senor

object has the following method:

  - get_temp() - calls the associated temperature sensor reading function and returns the temperature value

- seven_seg.py – contains the SevenSegmentDisplay object which abstracts away the details of communicating with the SAA1064 via I^2C to turn on and off segments of the seven segment display. The SevenSegmentDisplay object has the following methods:
  - segment_test() - used to exercise the segement test function of the SAA1064
  - read_status() - returns the status from the SAA1064
  - write_digit0(pattern) – writes the specified pattern to digit 0 on the display. The patterns are 1 byte bit fields. The necessary patterns to control each segment as well as to display the numeric digits 0-9 are created as global variables in seven_seg.py for convenience.
  - write_digit1(pattern) – writes the specified pattern to digit1 on the display.
  - clear() - turn off all segments on the display
  - display_number(number) – display the passed number on the seven segment display. This method calls write_digit0 and write_digit1.
- status_led.py – contains the LED object which abstracts the GPIO control need to use the LEDs on the daughtercard. The LED object uses the RPIO library and it's constructor takes the number of the GPIO associated with the desired LED. The LED object has the following methods:
  - on() - turns the LED on
  - off() - turns the LED off
  - toggle() - toggles the state of the LED (i.e. turns it off if it was already on or on if it was already off)
- tmp512.py – contains the TMP512 object which abstracts the details of communicating with the TMP512 chips via I^2C. The TMP512 object's constructor takes the I^2C address of the desired TMP512 chip as its only argument. The TMP512 constructor also disables the shunt power measurement functionality of the TMP512, sets the temperature measurement for continuous conversion (i.e. continuous readings) at a rate of one per second, and enables both remote temperature sensing channels. The TMP512 object contains the following methods:
  - _reverse16(value) – an internal method that takes the value argument and returns it in byte reversed order
  - read_reg(reg) – returns the value of the specified TMP512 register
  - write_reg(reg, value) – writes the value argument to the specified TMP512 register
  - reset() - resets the TMP512 chip
  - status() - returns the value of the status register
  - device_id() - returns the value of the device ID register
  - _convert_temp(raw_val) – an internal method that converts a raw 13 bit ADC register reading from the TMP512 into degree's Fahrenheit and returns the value
  - local_temp() - reads the local temperature of the TMP512 and returns the value in degrees Fahrenheit
  - temp_sensor1() - reads the remote channel 1 temperature register and returns the value in degrees Fahrenheit
  - temp_sensor2() - reads the remote channel 2 temperature register and returns the value in degrees Fahrenheit

rpi_client.py

The control client daemon runs on the Raspberry Pi and uses the code in bfcslib and RPIO to control the Raspberry Pi and daughtercard. The client daemon attempts to open a TCP socket connection to the host specified by the HOST global variable on the port specified by the PORT global variable, which defaults to 2243. After successfully creating the socket connection the client daemon sets up all of the sensor LEDs, both TMP512's and creates Sensor objects to encapsulate each sensor status LED with the appropriate TMP512 temperature reading function (either temp_senor1 or temp_sensor2). The client then creates a SevenSegmentDisplay object and a Relay object to control the seven segment display and relay respectively. Next the client, using the RPIO library, sets up GPIO 22 to be an input with internal pull down and configures an interrupt called button_isr to be trigger on the rising edge after a debounce timeout of 150ms. Finally the client begins waiting for interrupts using a separate thread that doesn't block execution of the main thread.

The button_isr function is triggered each time there is a rising edge on GPIO and serves as the interrupt service routine as its name suggests. The button_isr function turns off the LED corresponding to the currently selected sensor, iterates to the next sensor, and then turns on the new sensor's status LED. The button_isr iterates from sensor 1 to sensor 2 to sensor 3 to sensor 4 before starting again back at sensor 1.

The control client daemon then enters the main loop which continues to execute until the daemon is manually terminated. Inside the main loop the client records the current date and time. It then iterates over all four sensors and records their temperature readings. While iterating over the temperature sensors the currently selected (from the push button, defaults to 0 or sensor 1) index is checked against the index of the current temperature sensor. If there is a match then the temperature reading for that sensor is displayed on the seven segment display. A SensorReading object is created with the previously recorded date and time, and all four temperature readings.

In order to decide whether or not to turn on the relay the client daemon first sorts all of the temperature readings in descending order. It then checks, using a state variable, if the relay is currently on or off. If the relay is currently on the client looks at each temperature reading and compares it to the threshold temperature specified by the global variable THRESHOLD_TEMP. If all four temperature readings are lower than threshold temp the relay is turned off and its current state is recorded. If the relay I off the client daemon looks at the first two entries in the list of sorted temperature readings and compares them both to the THRESHOLD_TEMP global variable. If both readings are greater than the threshold temperature the relay is turned on and its current status is recorded. Note the value of THRESHOLD_TEMP must be set according to style of beer being brewed as discussed above. This very basic control algorithm should provide sufficient temperature control for beer brewing purposes. Finally, the status of the relay is recorded in the previously created SensorReading object. The SensorReading object is then pickled (serialized), using the cPickle package from the Python standard library, and sent to the server over the previously opened TCP socket. The control client then sleeps for approximately 0.9 seconds before executing the main loop again.

server.py

The server daemon runs on the remote web server and waits to receive pickled SensorReading objects from the control client daemon. The server daemon binds to the host specified in the HOST global variable on the port specified on the PORT global variable (default is 2243). Once a connection has been established the server daemon enters its main loop.

Inside the main loop, the server daemon blocks while waiting to receive data. When it receives data, the server daemon attempts to unpickle (i.e. deserialize) the data. If the server daemon is unable to unpickle the data it receives, that data is disguarded. If the server daemon is able to unpickle the data, it checks to see if the data is a SensorReading object. If the data is a SensorReading object the server daemon prints out the date and time information, all four temperature readings, and the relay status. The server daemon then opens a connection to the SQLite database specified by the DB_PATH global variable using the sqlite3 package from the Python standard library. The server daemon then executes an insert query to insert the data into the database. Finally, the server daemon closes the database connection and returns to the top of the main loop.

app.wsgi

The WSGI (Web Server Gateway Interface) web application runs on a web server and displays the data stored in the SQLite database to any user connected to the server via a web browser. The remote web server for this project is running the Apache web server and the mod_wsgi module on Linux, but other platforms and web servers could be supported with minimal additional effort. WSGI is an standard interface that allows Python

programs to be developed to handle request content generation for web servers. App.wsgi uses the open source Genshi templating engine (which is also written in Python). Genshi allows information that is dynamically generated by the WSGI application to be easily displayed as HTML.

The app.wsgi application first loads all of the necessary Genshi templates. The application then checks the request path and does several different things depending on that path. For the default "/" path, the application loads the index.html template, then opens a connection to the SQLite database specified by the DB_PATH global variable. The application executes a query which calculates the total number of readings as well as the average, minimum, and maximum temperature readings for each of the temperature sensors and groups this information by day so that the results of the query contain on row per day. The application then closes the database connection, passes this information to the Genshi template and renders it into HTML. Finally the application returns the HTML to the requestor.

For a request path that starts with "/day" and ends with "/all" (e.g. /day/20130505/all) the application loads the all.html template. It then parses the date out the request path. Next the application opens a connection to the SQLite database and executes a query that retrieves all of the raw sensor data for the day specified in the request path. The application then closes the database connection, passes the query data to the Genshi template, renders it to HTML, and returns the HTML.

For a request path that starts with "/day" (e.g. /day/20130430) the application loads the day.html template. It then parses the date out of the request path. Next the application opens a connection to the SQLite database and executes a query to retrieve the most recent 300 rows from the database for the day specified in the request path. The application then executes another query to retreive the most recent 60 rows from the database. The data from the second query is used to display a JavaScript chart showing temperature readings from the last minute. The application then closes the database connection, passes the query data to the Genshi template, renders it to HTML and returns the HTML.

The day.html template contains a JavaScript plot of the 60 most recent temperature readings for all four sensors. This plot is drawn using an open source Javascript library called flot. If the request path contains the date for the current date and both rpi_client.py and server.py are running, the plot will automatically update itself as new data is inserted into the SQLite database. The final request path in the WSGI application is "/chart_update" which the client-side Javascript in the day.html template will issue requests to containing the last timestamp of the last data point plotted. The WSGI application then executes a query on the SQLite database to fetch all rows greater than the passed timestamp argument for the current day. The results of this query are formatted into a string and returned as plain text to the client-side Javascript. The Javascript then parses the string and updates both the plot and the data table contained in the day.hml template.

Running the code

client.py – The control client daemon must be run on the Raspberry Pi board.

1. SSH into the Raspberry Pi.
2. Change to the code directory (e.g. `cd bfcs/code/`)
3. Run the daemon as root: `$ sudo python rpi_client.py`

server.py – The server daemon must be run on the remote web server.

1. SSH into the web server.
2. Change to the code directory (e.g. `cd bfcs/code/`)
3. Run the daemon: `$ python server.py`

app.wsgi – The WSGI application must be run on an Apache web server with mod_wsgi installed.

1. Edit your Apache configuration file and add the following to it (change /path/to/ to the appropriate path):

```
WSGIScriptAlias /bfcs /path/to/web/folder/app.wsgi
Alias /htdocs /path/to/htdocs
<Directory /path/to/htdocs>
         Order allow,deny
         Allow from all
</Directory>
```

2. Restart apache: `$ sudo service apache restart`
3. From a computer attached to the same network open a web browser and navigate to the web server's IP + “/bfcs”. For example, if the web server is located at 10.1.2.3 navigate to “http://10.1.2.3/bfcs” (without the quotes).

Code Listing

Full source code for the project appears on the following pages.

```
import sys

import RPIO

from status_led import LED

class Relay(object):

    def __init__(self):
        # Use the BCM addressing scheme
        RPIO.setmode(RPIO.BCM)

        # Setup output for relay
        RPIO.setup(17, RPIO.OUT)
        RPIO.output(17, False)

        # Instantiate the relay status LED (23)
        self.led = LED(23)

    def on(self):
        self.led.on()
        RPIO.output(17, True)

    def off(self):
        self.led.off()
        RPIO.output(17, False)

def main():

    import time

    relay = Relay()
    print 'Quick relay unit test'
    print 'Turning relay on...'
    relay.on()
    time.sleep(5)
    print 'Okay that\'s enough of that'
    relay.off()


if __name__ == '__main__':
    sys.exit(main())
```

```python
class SensorReading(object):

    def __init__(self, time, temp1, temp2, temp3, temp4, relay_status):
        self.time = time

        # Temperature readings
        self.temp1 = temp1
        self.temp2 = temp2
        self.temp3 = temp3
        self.temp4 = temp4

        # Relay status
        self.relay_status = relay_status
```

```python
class Sensor(object):
    """ Container abstraction for a 'sensor' """

    def __init__(self, led, temp_sensor_function):
        self.led = led
        self.temp_sensor_function = temp_sensor_function

    def get_temp(self):
        return round(self.temp_sensor_function())

    # When the "temp" attribute is accessed call the temp sensor function
    # which will return the desired temperature reading
    temp = property(get_temp)
```

```
import sys
import time

import smbus

SEGMENTS_OFF = 0x00
SEGMENT_G    = 0x01
SEGMENT_F    = 0x02
SEGMENT_E    = 0x04
SEGMENT_D    = 0x08
SEGMENT_C    = 0x10
SEGMENT_B    = 0x20
SEGMENT_A    = 0x40
SEGMENT_DOT  = 0x80

DIGIT_0 = SEGMENT_A | SEGMENT_B | SEGMENT_C | SEGMENT_D | SEGMENT_E | SEGMENT_F
DIGIT_1 = SEGMENT_B | SEGMENT_C
DIGIT_2 = SEGMENT_A | SEGMENT_B | SEGMENT_G | SEGMENT_E | SEGMENT_D
DIGIT_3 = SEGMENT_A | SEGMENT_B | SEGMENT_G | SEGMENT_C | SEGMENT_D
DIGIT_4 = SEGMENT_F | SEGMENT_G | SEGMENT_B | SEGMENT_C
DIGIT_5 = SEGMENT_A | SEGMENT_F | SEGMENT_G | SEGMENT_C | SEGMENT_D
DIGIT_6 = SEGMENT_A | SEGMENT_F | SEGMENT_G | SEGMENT_E | SEGMENT_D | SEGMENT_C
DIGIT_7 = SEGMENT_A | SEGMENT_B | SEGMENT_C
DIGIT_8 = SEGMENT_A | SEGMENT_B | SEGMENT_C | SEGMENT_D | SEGMENT_E | SEGMENT_F | SEGMENT_G
DIGIT_9 = SEGMENT_A | SEGMENT_B | SEGMENT_C | SEGMENT_D | SEGMENT_F | SEGMENT_G

class SevenSegmentDisplay(object):

    digit_map = {'0': DIGIT_0,
                 '1': DIGIT_1,
                 '2': DIGIT_2,
                 '3': DIGIT_3,
                 '4': DIGIT_4,
                 '5': DIGIT_5,
                 '6': DIGIT_6,
                 '7': DIGIT_7,
                 '8': DIGIT_8,
                 '9': DIGIT_9}

    def __init__(self):
        self.bus = smbus.SMBus(1)
        self.address = 0x38
        # 2 digits at 9ma sink
        self.bus.write_byte_data(self.address, 0x00, 0x35)

    def segment_test(self):
        print 'Testing segments...'
        self.bus.write_byte_data(self.address, 0x00, 0x3E)
        time.sleep(3)
        self.bus.write_byte_data(self.address, 0x00, 0x35)
        print 'Test complete'

    def read_status(self):
        return bus.read_byte(address)

    def write_digit0(self, pattern):
        self.bus.write_byte_data(self.address, 0x03, pattern)

    def write_digit1(self, pattern):
        self.bus.write_byte_data(self.address, 0x01, pattern)

    def clear(self):
        self.write_digit0(SEGMENTS_OFF)
        self.write_digit1(SEGMENTS_OFF)

    def display_number(self, number):
        str_num = '%02d' % number
        self.write_digit1(self.digit_map[str_num[1]])
        self.write_digit0(self.digit_map[str_num[0]])



def main():

    display = SevenSegmentDisplay()

    # segment test
    display.segment_test()
```

```
    # blink segment G x20
    for ctr in xrange(20):
        display.clear()
        time.sleep(0.1)
        SEGMENT_G
        display.write_digit1(SEGMENT_G)
        display.write_digit0(SEGMENT_G)
        time.sleep(0.1)

    # outer snake clockwise, both
    segs = [SEGMENT_F, SEGMENT_A, SEGMENT_B, SEGMENT_C,
            SEGMENT_D, SEGMENT_E]

    for xtr in xrange(2):
        for segment in segs:
            display.clear()
            display.write_digit1(segment)
            display.write_digit0(segment)
            time.sleep(0.2)

    # outer snake counter clockwise, both
    segs = [SEGMENT_E, SEGMENT_D, SEGMENT_C, SEGMENT_B,
            SEGMENT_A, SEGMENT_F]

    for xtr in xrange(2):
        for segment in segs:
            display.clear()
            display.write_digit1(segment)
            display.write_digit0(segment)
            time.sleep(0.2)

    # figure 8 both (clockwise start)
    segs = [SEGMENT_G, SEGMENT_F, SEGMENT_A, SEGMENT_B, SEGMENT_G,
            SEGMENT_E, SEGMENT_D, SEGMENT_C]

    for xtr in xrange(3):
        for segment in segs:
            display.clear()
            display.write_digit1(segment)
            display.write_digit0(segment)
            time.sleep(0.1)

    # figure 8 both (counter clockwise start)
    segs = [SEGMENT_G, SEGMENT_E, SEGMENT_D, SEGMENT_C, SEGMENT_G,
            SEGMENT_F, SEGMENT_A, SEGMENT_B]

    for xtr in xrange(3):
        for segment in segs:
            display.clear()
            display.write_digit1(segment)
            display.write_digit0(segment)
            time.sleep(0.1)

    # count down
    while True:
        for num in xrange(99, -1, -1):
            display.clear()
            display.display_number(num)
            time.sleep(0.3)

if __name__ == '__main__':
    sys.exit(main())
```

```python
import sys

import RPIO

class LED(object):

    def __init__(self, gpio_num):
        self.gpio_num = gpio_num
        self.state = False

        # Use the BCM addressing scheme
        RPIO.setmode(RPIO.BCM)

        # Configure GPIO as output and turn off
        RPIO.setup(self.gpio_num, RPIO.OUT)
        RPIO.output(self.gpio_num, False)

    def on(self):
        self.state = True
        RPIO.output(self.gpio_num, True)

    def off(self):
        self.state = False
        RPIO.output(self.gpio_num, False)

    def toggle(self):
        self.state = not self.state
        RPIO.output(self.gpio_num, self.state)

def main():

    import time

    led1 = LED(23)
    led2 = LED(7)
    print 'Quick status LED unit test'
    print 'Turning on LED1'
    led1.on()
    time.sleep(1)
    print 'Turning on LED2'
    led2.on()
    time.sleep(1)
    print 'Turning off LED1'
    led1.off()
    for ctr in xrange(0, 10):
        time.sleep(1)
        led1.toggle()
        led2.toggle()
        print 'Toggle iteration %d' % (ctr + 1)

if __name__ == '__main__':
    sys.exit(main())
```

```python
import sys
import time

import smbus

class TMP512(object):

    def __init__(self, address):
        self.bus = smbus.SMBus(1)
        self.address = address

        # Shunt Measurement Configuration
        #   - turn off shunt/bus voltage measurement
        self.write_reg(0x00, 0x3998)

        # Temperature Measurement Configuration
        #   - continous conversion at 1 conversion/second
        #   - enable remote 1, remote 2 and local
        self.write_reg(0x01, 0xBE00)

    def _reverse16(self, value):
        """ A quick hack method for reversing byte order for a 16 bit value """
        str_val = '%04X' % value
        new_val = str_val[2:4] + str_val[0:2]
        return int(new_val, 16)

    def read_reg(self, reg):
        raw = self.bus.read_word_data(self.address, reg)
        return self._reverse16(raw)

    def write_reg(self, reg, data):
        raw = self._reverse16(data)
        self.bus.write_word_data(self.address, reg, raw)

    def reset(self):
        self.write_reg(0x00, 0xB998)
        time.sleep(1)

    def status(self):
        return self.read_reg(0x02)

    def device_id(self):
        return '0x%04X' % self.read_reg(0x1F)

    def _convert_temp(self, raw_val):
        # Raw temp is a 13 bit number, the units are 0.0625 C
        temp = (raw_val >> 3) * 0.0625
        # Convert C to F
        return (temp * 1.8) + 32

    def local_temp(self):
        return self._convert_temp(self.read_reg(0x08))

    def temp_sensor1(self):
        return self._convert_temp(self.read_reg(0x09))

    def temp_sensor2(self):
        return self._convert_temp(self.read_reg(0x0A))


def main():
    import time

    tmp512_1 = TMP512(0x5d)
    tmp512_2 = TMP512(0x5c)

    print 'TMP512-1 status: 0x%04X' % tmp512_1.status()
    print 'TMP512-2 status: 0x%04X' % tmp512_2.status()

    print 'TMP512-1 local temp is %d' % tmp512_1.local_temp()
    print 'TMP512-2 local temp is %d' % tmp512_2.local_temp()

    print 'TMP512-1 remote 1 temp is %d' % tmp512_1.temp_sensor1()
    print 'TMP512-1 remote 2 temp is %d' % tmp512_1.temp_sensor2()
    print 'TMP512-2 remote 1 temp is %d' % tmp512_2.temp_sensor1()
    print 'TMP512-2 remote 2 temp is %d' % tmp512_2.temp_sensor2()

    print '---'
    time.sleep(1)
```

```
    print 'TMP512-1 remote 1 temp is %d' % tmp512_1.temp_sensor1()
    print 'TMP512-1 remote 2 temp is %d' % tmp512_1.temp_sensor2()
    print 'TMP512-2 remote 1 temp is %d' % tmp512_2.temp_sensor1()
    print 'TMP512-2 remote 2 temp is %d' % tmp512_2.temp_sensor2()

    print '---'
    time.sleep(1)

    print 'TMP512-1 remote 1 temp is %d' % tmp512_1.temp_sensor1()
    print 'TMP512-1 remote 2 temp is %d' % tmp512_1.temp_sensor2()
    print 'TMP512-2 remote 1 temp is %d' % tmp512_2.temp_sensor1()
    print 'TMP512-2 remote 2 temp is %d' % tmp512_2.temp_sensor2()


if __name__ == '__main__':
    sys.exit(main())
```

```
import cPickle
import datetime
import itertools
import os
import socket
import sys
import time

import RPIO

from bfcslib.relay import Relay
from bfcslib.sensor import Sensor
from bfcslib.sensor_reading import SensorReading
from bfcslib.seven_seg import SevenSegmentDisplay
from bfcslib.status_led import LED
from bfcslib.tmp512 import TMP512

# Global variables
DEBUG = False

NUM_SENSORS = 4
sensor_iterator = itertools.cycle(range(0, NUM_SENSORS))
CURRENT_IDX = sensor_iterator.next()
SENSORS = None

THRESHOLD_TEMP = 82

HOST = '192.168.1.1'
PORT = 2243
REMOTE_DATA_INTERVAL = 1 # in seconds

def button_isr(gpio_id, val):
    global CURRENT_IDX
    global SENSORS

    # Turn off previous sensor indicator LED
    SENSORS[CURRENT_IDX].led.off()

    # Update current index
    # print 'CURRENT_IDX: %s' % CURRENT_IDX
    CURRENT_IDX = sensor_iterator.next()

    # Turn on new sensor indicator LED
    SENSORS[CURRENT_IDX].led.on()

def main():
    global CURRENT_IDX
    global SENSORS
    global DEBUG

    # Make sure the script is being run as root before continuing
    if os.geteuid() != 0:
        print 'ERROR: You must run the client as root!'
        return

    # Disable "channel already in use" warnings
    RPIO.setwarnings(False)

    print 'Beer Fermentation Control System - RPI Client'

    # Check for the debug command line argument
    if len(sys.argv) > 1:
        if sys.argv[1] == '-d':
            DEBUG = True

    # Attempt to connect to the server, make 5 attempts
    print '\n\nAttempting to connect to server...',
    sys.stdout.flush()

    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    tries = 5
    while tries > 0:
        try:
            s.connect((HOST, PORT))
            break
        except socket.error:
            tries -= 1
            if tries == 0:
                print 'FAILED'
```

```
            print 'Unable to connect to %s:%s check your connection and ' \
                  'try again.' % (HOST, PORT)
            return
        time.sleep(1)

    # Connection was successful
    print 'SUCCESS'
    print 'Connected to %s:%s' % (HOST, PORT)

    # Setup the sensor status LEDs
    sensor1_led = LED(24)
    sensor2_led = LED(25)
    sensor3_led = LED(8)
    sensor4_led = LED(7)

    # Setup both TMP512's
    tmp512_1 = TMP512(0x5d)
    tmp512_2 = TMP512(0x5c)

    # Create the sensor objects to encapsulate a status LED and temperature
    # sensor channel
    sensor1 = Sensor(sensor1_led, tmp512_1.temp_sensor2)
    sensor2 = Sensor(sensor2_led, tmp512_1.temp_sensor1)
    sensor3 = Sensor(sensor3_led, tmp512_2.temp_sensor2)
    sensor4 = Sensor(sensor4_led, tmp512_2.temp_sensor1)

    # Create the global list of all the sensor objects
    SENSORS = [sensor1, sensor2, sensor3, sensor4]

    # Turn on initial sensor LED
    SENSORS[CURRENT_IDX].led.on()

    # Setup 7 seg display
    ss_display = SevenSegmentDisplay()

    # Setup relay
    relay = Relay()
    relay_state = False

    # Setup pushbutton
    # Use the BCM addressing scheme, set GPIO 22 as input
    RPIO.setmode(RPIO.BCM)
    RPIO.setup(22, RPIO.IN)

    # Setup switch interrupt with debounce
    RPIO.add_interrupt_callback(22, button_isr, edge='rising',
                                pull_up_down=RPIO.PUD_DOWN, threaded_callback=True,
                                debounce_timeout_ms=150)

    # Wait for interrupts in a non-blocking fashion
    RPIO.wait_for_interrupts(threaded=True)

    # Main loop
    interval_counter = 0
    while True:

        # Reset variables
        temps = []
        if interval_counter == REMOTE_DATA_INTERVAL:
            interval_counter = 0

        interval_counter += 1

        # Grab the current date/time
        now = datetime.datetime.now()

        # Iterate over all temperature sensors and read their temp
        for idx in xrange(NUM_SENSORS):

            # Get the current temperature reading
            temp = SENSORS[idx].temp
            temps.append(temp)

            # Display the currently selected temp sensor value
            if idx == CURRENT_IDX:
                ss_display.display_number(temp)

        # Bit 'o debug information
        if DEBUG:
```

```
            print now.strftime('%m/%d/%Y %H:%M:%S')
            for idx in xrange(NUM_SENSORS):
                print '\t Sensor %d: %d' % (idx + 1, temps[idx])

        # Create the SensorReading object before we sort the temps
        reading = SensorReading(now, temps[0], temps[1], temps[2],
                                temps[3], False)

        # Sort all the temperature readings for easier manipulation
        temps.sort(reverse=True)

        if relay_state:
            # Relay is on, decide if we need to turn it off
            if temps[0] < THRESHOLD_TEMP and \
               temps[1] < THRESHOLD_TEMP and \
               temps[2] < THRESHOLD_TEMP and \
               temps[3] < THRESHOLD_TEMP:
                relay_state = False
                relay.off()
        else:
            # Relay is off, decide if we need to turn it on
            if temps[0] > THRESHOLD_TEMP and temps[1] > THRESHOLD_TEMP:
                relay_state = True
                relay.on()

        # Update the relay status after deciding to turn it on or off
        reading.relay_status = relay_state

        # Send a reading to the server once every minute
        if interval_counter == REMOTE_DATA_INTERVAL:
            if DEBUG:
                print 'Sending reading to server...',
                sys.stdout.flush()
            try:
                s.sendall(cPickle.dumps(reading))
                if DEBUG:
                    print 'DONE'
            except:
                if DEBUG:
                    print 'ERROR'

        # Read once every second (approximated to account for processing
        # latencies)
        time.sleep(0.9)

    s.close()

if __name__ == '__main__':
    sys.exit(main())
```

```
import calendar
import cPickle
import datetime
import socket
import sqlite3
import sys

from bfcslib.sensor_reading import SensorReading

DB_PATH = '/home/dsorber/bfcs/beer_temps.db'
HOST = '192.168.1.1'
PORT = 2243

def main():

    # Create socket and wait for connections
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind((HOST, PORT))
    s.listen(1)
    conn, addr = s.accept()
    print 'Connected by', addr

    # Main loop
    while True:

        # Block and wait to receive data
        data = conn.recv(65536)

        # Try to unpickle any data received
        try:
            obj = cPickle.loads(data)
        except:
            print 'I got some data that I couldn\'t unpickle...'
            print '-' * 100
            print data
            print '-' * 100
            print '\n\n'
            continue

        # Is the unpickled object a SensorReading object?
        if isinstance(obj, SensorReading):
            print obj.time.strftime('%m/%d/%Y %H:%M:%S')
            print calendar.timegm(obj.time.timetuple())
            print '\tTemp 1: %s' % obj.temp1
            print '\tTemp 2: %s' % obj.temp2
            print '\tTemp 3: %s' % obj.temp3
            print '\tTemp 4: %s' % obj.temp4
            print '\tRelay is %s' % (obj.relay_status and 'on' or 'off')

            # Insert the reading data into a database
            print 'Inserting data into the database...'
            sys.stdout.flush()

            db_conn = sqlite3.connect(DB_PATH)
            db = db_conn.cursor()
            try:
                db.execute("INSERT INTO temp_data VALUES (%s, %s, %s, %s, %s, %s)" %
                           (calendar.timegm(obj.time.timetuple()), obj.temp1,
                            obj.temp2, obj.temp3, obj.temp4,
                            int(obj.relay_status and '1' or '0')))
                db_conn.commit()
            except:
                print 'FAILED'
                db_conn.close()
                continue

            db_conn.close()
            print 'SUCCESS'

    conn.close()


if __name__ == '__main__':
    sys.exit(main())
```

```
import cgi
import datetime
import os
import sqlite3

from genshi.template import TemplateLoader
from genshi.template.base import Context

BASE_PATH = '/home/dsorber/bfcs'
DB_PATH = os.path.join(BASE_PATH, 'beer_temps.db')
TEMPLATE_PATH = os.path.join(BASE_PATH, 'templates')

# It accepts two arguments:
# environ points to a dictionary containing CGI like environment variables
# which is filled by the server for each received request from the client
# start_response is a callback function supplied by the server
# which will be used to send the HTTP status and headers to the server

def application(environ, start_response):

    loader = TemplateLoader(TEMPLATE_PATH, auto_reload=True)

    if not environ['PATH_INFO'] or environ['PATH_INFO'] == '/':
        # Load the index template
        template = loader.load('index.html')

        # Create the response args
        response_args = {'title': 'Beer Fermentation Control System'}

        # Open up the connection to the database, execute the query,
        # grab the results, then close the DB connection.
        db_conn = sqlite3.connect(DB_PATH)
        db = db_conn.cursor()
        db.execute("select strftime('%m/%d/%Y', date(ts, 'unixepoch')), "
                   "       count(ts), avg(temp1), max(temp1), min(temp1),"
                   "       avg(temp2), max(temp2), min(temp2), avg(temp3),"
                   "       max(temp3), min(temp3), avg(temp4), max(temp4),"
                   "       min(temp4)"
                   "from temp_data "
                   "group by date(ts, 'unixepoch')")
        response_args['results'] = db.fetchall()
        db_conn.close()

        # Generate the stream and then render it to HTML
        stream = template.generate(**response_args)
        response_body = stream.render('html', doctype='html', encoding='utf-8')

        # Set the appropriate response headers
        response_headers = [('Content-Type', 'text/html; charset=utf-8'),
                            ('Content-Length', str(len(response_body)))]

    elif environ['PATH_INFO'].startswith('/day/') and environ['PATH_INFO'].endswith('/all'):
        # Load the all template
        template = loader.load('all.html')

        # Parse the date out of the URL (TODO: validate all this)
        day = environ['PATH_INFO'][5:7]
        month = environ['PATH_INFO'][7:9]
        year = environ['PATH_INFO'][9:13]
        date = '%s/%s/%s' % (month, day, year)

        # Create the response args
        response_args = {'title': 'Sensor Readings for %s' % date}

        # Open up the connection to the database, execute the query
        # (limit to today and the past 5 minutes),
        # grab the results, then close the DB connection.
        db_conn = sqlite3.connect(DB_PATH)
        db = db_conn.cursor()
        db.execute("select strftime('%%H:%%M:%%S', ts, 'unixepoch'), "
                   "       temp1, temp2, temp3, temp4, relay_status "
                   "from temp_data "
                   "where date(ts, 'unixepoch') is date('%s-%s-%s')"
                   "order by ts desc" % (year, month, day))
        response_args['results'] = db.fetchall()

        db_conn.close()

        # Generate the stream and then render it to HTML
```

```
        stream = template.generate(**response_args)
        response_body = stream.render('html', doctype='html', encoding='utf-8')

        # Set the appropriate response headers
        response_headers = [('Content-Type', 'text/html; charset=utf-8'),
                            ('Content-Length', str(len(response_body)))]

    elif environ['PATH_INFO'].startswith('/day/'):
        # Load the day template
        template = loader.load('day.html')

        # Parse the date out of the URL (TODO: validate all this)
        day = environ['PATH_INFO'][5:7]
        month = environ['PATH_INFO'][7:9]
        year = environ['PATH_INFO'][9:13]
        date = '%s/%s/%s' % (month, day, year)

        # Create the response args
        response_args = {'title': 'Sensor Readings for %s' % date,
                         'url_date': environ['PATH_INFO'][5:13]}

        # Open up the connection to the database, execute the query
        # (limit to today and the past 5 minutes),
        # grab the results, then close the DB connection.
        db_conn = sqlite3.connect(DB_PATH)
        db = db_conn.cursor()
        db.execute("select strftime('%%H:%%M:%%S', ts, 'unixepoch'), "
                   "       temp1, temp2, temp3, temp4, relay_status "
                   "from temp_data "
                   "where date(ts, 'unixepoch') is date('%s-%s-%s')"
                   "order by ts desc limit 300" % (year, month, day))
        response_args['results'] = db.fetchall()

        # Chart query
        #
        db.execute("select strftime('%%H:%%M:%%S', ts, 'unixepoch'), ts, temp1, "
                   "       temp2, temp3, temp4 from ( "
                   "select ts, temp1, temp2, temp3, temp4 "
                   "from temp_data "
                   "where date(ts, 'unixepoch') is date('%s-%s-%s')"
                   "order by ts desc limit 60) order by ts"
                   % (year, month, day))
        chart_results = db.fetchall()

        # Process chart query results
        sensor1_data = []
        sensor2_data = []
        sensor3_data = []
        sensor4_data = []
        for row in chart_results:
            sensor1_data.append([int(row[1]) * 1000, row[2]])
            sensor2_data.append([int(row[1]) * 1000, row[3]])
            sensor3_data.append([int(row[1]) * 1000, row[4]])
            sensor4_data.append([int(row[1]) * 1000, row[5]])
        response_args['sensor1_data'] = str(sensor1_data)
        response_args['sensor2_data'] = str(sensor2_data)
        response_args['sensor3_data'] = str(sensor3_data)
        response_args['sensor4_data'] = str(sensor4_data)

        # This is a bit of hack to get around a javascript issue in
        # the template
        # response_args['chart_results'] = chart_results[:-1]
        # response_args['chart_last_row'] = chart_results[-1]

        db_conn.close()

        # Generate the stream and then render it to HTML
        stream = template.generate(**response_args)
        response_body = stream.render('html', doctype='html', encoding='utf-8')

        # Set the appropriate response headers
        response_headers = [('Content-Type', 'text/html; charset=utf-8'),
                            ('Content-Length', str(len(response_body)))]

    elif environ['PATH_INFO'] == ('/chart_update'):
        # Update the self updating chart
        response_body = ''

        if environ['QUERY_STRING']:
```

```
        # Parse the query string to get the last timestamp from the client
        get_args = cgi.parse_qs(environ['QUERY_STRING'])
        last_ts = int(get_args['last_ts'][0])
        date_from_ts = datetime.datetime.fromtimestamp(last_ts)

        # Reconstruct portions of the date from the ts
        year = date_from_ts.year
        month = '%02d' % date_from_ts.month
        day = '%02d' % date_from_ts.day

        # Grab any entries from the supplied date that are greater than
        # the last timestamp sent from the client javascript
        db_conn = sqlite3.connect(DB_PATH)
        db = db_conn.cursor()
        db.execute("select strftime('%%H:%%M:%%S', ts, 'unixepoch'), ts, "
                   "       temp1, temp2, temp3, temp4, relay_status "
                   "from temp_data "
                   "where date(ts, 'unixepoch') is date('%s-%s-%s') and "
                   "      ts > %s "
                   "order by ts asc limit 60" % (year, month, day, last_ts))
        db_rows = db.fetchall()
        db_conn.close()

        # Iterate over the rows fetched by the query and pretty them up
        # in preparation of being sent back to the client javascript for
        # display
        result_rows = []
        for row in db_rows:
            result_rows.append('%s %s %s %s %s %s %s' %
                               (row[0], (int(row[1]) * 1000),
                               row[2], row[3], row[4], row[5],
                               int(row[6]) and 'on' or 'off'))

        response_body = ','.join(result_rows)
        response_body = response_body.encode('utf-8')


    response_headers = [('Content-Type', 'text/plain; charset=utf-8'),
                        ('Content-Length', str(len(response_body)))]

else:
    # User supplied an invalid URL
    response_body = 'Invalid URL "%s"' % environ['PATH_INFO']
    response_headers = [('Content-Type', 'text/plain'),
                        ('Content-Length', str(len(response_body)))]

# HTTP response code and message
status = '200 OK'

# Send the status and response headers to the server
start_response(status, response_headers)

# Return the response body. (Notice it is wrapped in a list although it
# could be any iterable.)
return [response_body]
```

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:py="http://genshi.edgewall.org/">
<head>
<title>Beer Fermenation Control System</title>
<style type="text/css">
    body            {font-family: sans-serif;
                     font-size: 16px}
    h3              {font-size: 24px;}
    table           {border: 1px solid #000;
                     border-spacing: 0px;
                     margin-left: auto;
                     margin-right: auto;}
    table a:link    {color: #000;
                     text-decoration: none;}
    table a:visited {color: #000;
                     text-decoration: none;}
    table a:hover   {color: #000;
                     text-decoration: underline;}
    tr.even         {background-color: #FFF;}
    tr.odd          {background-color: #FFC;}
    th              {border: 1px solid #000;
                     padding: 10px;
                     border-spacing: 2px;}
    td              {border-top: 1px solid #000;
                     padding: 2px 5px;}
    .center         {text-align: center;}
    .sensor1        {background-color: #FC0;}
    .sensor2        {background-color: #C3F;}
    .sensor3        {background-color: #0C0;}
    .sensor4        {background-color: #F60;}
    .avg            {background-color: #AAA;}
    .max            {background-color: #CCC;}
    .min            {background-color: #EEE;}
</style>
</head>
<body>
<h1>${title}</h1>
<h3>Sensor Reading Summary</h3>
<table>
    <tr>
        <th colspan="2"> </th>
        <th colspan="3" class="sensor1">Sensor 1</th>
        <th colspan="3" class="sensor2">Sensor 2</th>
        <th colspan="3" class="sensor3">Sensor 3</th>
        <th colspan="3" class="sensor4">Sensor 4</th>
    </tr>
    <tr>
        <th>Date</th>
        <th>Readings</th>
        <th class="avg">Avg</th>
        <th class="max">Max</th>
        <th class="min">Min</th>
        <th class="avg">Avg</th>
        <th class="max">Max</th>
        <th class="min">Min</th>
        <th class="avg">Avg</th>
        <th class="max">Max</th>
        <th class="min">Min</th>
        <th class="avg">Avg</th>
        <th class="max">Max</th>
        <th class="min">Min</th>
    </tr>
    <tr py:for="ctr, row in enumerate(results)" class="${ctr % 2 and 'odd' or 'even'}">
        <td><a href="bfcs/day/${row[0][3:5] + row[0][0:2] + row[0][6:10]}">${row[0]}</a></td>
        <td class="center">${row[1]}</td>
        <td class="center">${round(float(row[2]), 3)}</td>
        <td class="center">${row[3]}</td>
        <td class="center">${row[4]}</td>

        <td class="center">${round(float(row[5]), 3)}</td>
        <td class="center">${row[6]}</td>
        <td class="center">${row[7]}</td>

        <td class="center">${round(float(row[8]), 3)}</td>
        <td class="center">${row[9]}</td>
        <td class="center">${row[10]}</td>

        <td class="center">${round(float(row[11]), 3)}</td>
        <td class="center">${row[12]}</td>
```

```
        <td class="center">${row[13]}</td>
    </tr>
</table>

</body>
</html>
```

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:py="http://genshi.edgewall.org/">
<head>
    <title>Beer Fermenation Control System</title>
    <style type="text/css">
        body            {font-family: sans-serif;
                         font-size: 16px}
        h3              {font-size: 24px;}
        table           {border: 1px solid #000;
                         border-spacing: 0px;
                         margin-left: auto;
                         margin-right: auto;}
        table a:link    {color: #000;
                         text-decoration: none;}
        table a:visited {color: #000;
                         text-decoration: none;}
        table a:hover   {color: #000;
                         text-decoration: underline;}
        tr.even         {background-color: #FFF;}
        tr.odd          {background-color: #FFC;}
        th              {border: 1px solid #000;
                         padding: 10px;
                         border-spacing: 2px;}
        td              {border-top: 1px solid #000;
                         padding: 2px 5px;}
        .center         {text-align: center;}
        .sensor1        {background-color: #FC0;}
        .sensor2        {background-color: #C3F;}
        .sensor3        {background-color: #0C0;}
        .sensor4        {background-color: #F60;}
        .avg            {background-color: #AAA;}
        .max            {background-color: #CCC;}
        .min            {background-color: #EEE;}
        .relay_on       {font-weight: bold;
                         color: #0C0;}
        #chart_div      {margin-left: auto;
                         margin-right: auto;}
    </style>
</head>
<body>
    <h1>${title}</h1>
    <h3>Table of All Readings</h3>
    <table>
        <tr>
            <th>Time</th>
            <th class="sensor1">Sensor 1</th>
            <th class="sensor2">Sensor 2</th>
            <th class="sensor3">Sensor 3</th>
            <th class="sensor4">Sensor 4</th>
            <th class="max">Relay Status</th>
        </tr>
        <tr py:for="ctr, row in enumerate(results)" class="${ctr % 2 and 'odd' or 'even'}">
            <td>${row[0]}</td>
            <td class="center">${row[1]}</td>
            <td class="center">${row[2]}</td>
            <td class="center">${row[3]}</td>
            <td class="center">${row[4]}</td>
            <td class="center ${row[5] == 1 and 'relay_on' or ''}">${int(row[5]) and 'on' or 'off'}</td>
        </tr>
    </table>
    <br />
    <a href="/bfcs">Back</a>
</body>
</html>
```

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:py="http://genshi.edgewall.org/">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>Beer Fermenation Control System</title>
    <style type="text/css">
        body            {font-family: sans-serif;
                         font-size: 16px}
        h3              {font-size: 24px;}
        table           {border: 1px solid #000;
                         border-spacing: 0px;
                         margin-left: auto;
                         margin-right: auto;}
        table a:link    {color: #000;
                         text-decoration: none;}
        table a:visited {color: #000;
                         text-decoration: none;}
        table a:hover   {color: #000;
                         text-decoration: underline;}
        tr.even         {background-color: #FFF;}
        tr.odd          {background-color: #FFC;}
        th              {border: 1px solid #000;
                         padding: 10px;
                         border-spacing: 2px;}
        td              {border-top: 1px solid #000;
                         padding: 2px 5px;}
        .center         {text-align: center;}
        .sensor1        {background-color: #FC0;}
        .sensor2        {background-color: #C3F;}
        .sensor3        {background-color: #0C0;}
        .sensor4        {background-color: #F60;}
        .avg            {background-color: #AAA;}
        .max            {background-color: #CCC;}
        .min            {background-color: #EEE;}
        .relay_on       {font-weight: bold;
                         color: #0C0;}
        #chart_div      {margin-left: auto;
                         margin-right: auto;}
    </style>
    <!--[if lte IE 8]><script language="javascript" type="text/javascript" src="http://127.0.0.1/htdocs/flot/
excanvas.min.js"></script><![endif]-->
    <script type="text/javascript" src="http://127.0.0.1/htdocs/flot/jquery.min.js"></script>
    <script type="text/javascript" src="http://127.0.0.1/htdocs/flot/jquery.flot.min.js"></script>
    <script type="text/javascript" src="http://127.0.0.1/htdocs/flot/jquery.flot.time.min.js"></script>

    <script type="text/javascript">
        $(function() {
            var ctr = 0;

            // Assemble initial sensor data to plot from the server
            var sensor1_data = ${sensor1_data};
            var sensor2_data = ${sensor2_data};
            var sensor3_data = ${sensor3_data};
            var sensor4_data = ${sensor4_data};

            // Plot the sensor data
            plot = $.plot("#chart_div",
                [
                 {label: "Sensor 1", data: sensor1_data, color: "#FC0"},
                 {label: "Sensor 2", data: sensor2_data, color: "#C3F"},
                 {label: "Sensor 3", data: sensor3_data, color: "#0C0"},
                 {label: "Sensor 4", data: sensor4_data, color: "#F60"}
                ],
                {
                    series: {
                        lines: {show: true},
                        points: {show: true}
                    },
                    grid: {
                        hoverable: true,
                        clickable: true
                    },
                    xaxis: {
                        mode: "time",
                        ticks: 12
                    },
                    legend: {
                        show: true,
                        noColumns: 4
```

```
        }

});

// Function that actually "draws" (inserts, really) the tooltip used below
function showTooltip(x, y, contents) {
    $('<div id="tooltip">' + contents + "</div>").css({
        position: "absolute",
        display: "none",
        top: y + 5,
        left: x + 5,
        border: "1px solid #ccc",
        padding: "2px",
        "background-color": "#eee",
        opacity: 0.80
    }).appendTo("body").fadeIn(200);
}

// Show a handy tooltip when a data point is clicked on the plot
$("#chart_div").bind("plotclick", function (event, pos, item) {
    $("#tooltip").remove();
    if (item) {
        var date = new Date(item.datapoint[0]);
        var str_ts = date.getHours() + ":" + date.getMinutes() + ":" + date.getSeconds();

        showTooltip(item.pageX, item.pageY,
                    item.series.label + " at " + str_ts + " = " + item.datapoint[1]);
    }
});

// When the page loads we need to know the latest timestamp so we can keep updating from there
// Store the value in a hidden dive on the page to make retrieval simple
$("#last_ts").text(sensor1_data[59][0]);

var ret = [];
function get_new_values(timestamp) {
    // Perform an AJAX get request to get the new data values to display
    $.get("/bfcs/chart_update", {last_ts: timestamp / 1000}, function(data) {
        ret = []

        if (data != '') {
            // Parse the returned string into an array
            rows = data.split(",")
            //<![CDATA[
            for (i = 0; i < rows.length; i++) {
            // ]]>
                ret.push(rows[i].split(" "))
            }

            // Grab the last row and set the new starting timestamp
            // alert(ret);
            $("#last_ts").text(ret[ret.length - 1][1]);
        }
    });
    return ret;
}

function update() {
    // Get the new values to plot and show in the readings table
    var ts = $("#last_ts").text();
    var rows = get_new_values(parseInt(ts));

    //<![CDATA[
    if (typeof rows !== 'undefined' && rows.length > 0) { // stuff like this is why js is stupid
    // ]]>

        sensor1_data.splice(0, rows.length);
        sensor2_data.splice(0, rows.length);
        sensor3_data.splice(0, rows.length);
        sensor4_data.splice(0, rows.length);

        //<![CDATA[
        for (i = 0; i < rows.length; i++) {
        // ]]>
            var row = rows[i];
            // Update the arrays containing the raw sensor data for plotting
            sensor1_data.push([row[1], row[2]]);
            sensor2_data.push([row[1], row[3]]);
            sensor3_data.push([row[1], row[4]]);
```

```
                        sensor4_data.push([row[1], row[5]]);

                        // Remove the last row from readings table
                        $("#readings tr:last").remove();

                        // Toggle ctr to keep track of the appropriate row CSS class
                        var tr_class;
                        if (ctr == 0) {
                            tr_class = "odd";
                        } else {
                            tr_class = "even";
                        }
                        ctr ^= 1;

                        // If the relay is on, set the appropriate CSS class
                        var relay_class = '';
                        if (row[6] == "on") {
                            relay_class = "relay_on";
                        }

                        // Insert new rows after the table header in readings table
                        $('<tr class="' + tr_class + '">' +
                            '<td>' + row[0] + '</td>' +
                            '<td class="center">' + row[2] + '</td>' +
                            '<td class="center">' + row[3] + '</td>' +
                            '<td class="center">' + row[4] + '</td>' +
                            '<td class="center">' + row[5] + '</td>' +
                            '<td class="center ' + relay_class + '">' + row[6] + '</td>' +
                          '</tr>').insertBefore($("#readings tr:eq(1)"));
                    }

                    // Assemble new plot data into format needed to plot
                    var new_data = [
                        {label: "Sensor 1", data: sensor1_data, color: "#FC0"},
                        {label: "Sensor 2", data: sensor2_data, color: "#C3F"},
                        {label: "Sensor 3", data: sensor3_data, color: "#0C0"},
                        {label: "Sensor 4", data: sensor4_data, color: "#F60"}
                        ];

                    // Plot the new data, update the grid, then redraw
                    plot.setData(new_data);
                    plot.setupGrid()
                    plot.draw();
                }

                // Re-call this function every second so that updates happen automagically
                setTimeout(update, 1000);
            }
            update();

        });
    </script>
</head>
<body>
    <h1>${title}</h1>
    <h3>Graph of Last 60 Readings</h3>
    <div id="chart_div" style="width: 900px; height: 500px;"></div>
    <div id="last_ts" style="display:none;"></div>
    <h3>Table of Last 300 Readings (5 mins)</h3>
    <table id="readings">
        <tr>
            <th>Time</th>
            <th class="sensor1">Sensor 1</th>
            <th class="sensor2">Sensor 2</th>
            <th class="sensor3">Sensor 3</th>
            <th class="sensor4">Sensor 4</th>
            <th class="max">Relay Status</th>
        </tr>
        <tr py:for="ctr, row in enumerate(results)" class="${ctr % 2 and 'odd' or 'even'}">
            <td>${row[0]}</td>
            <td class="center">${row[1]}</td>
            <td class="center">${row[2]}</td>
            <td class="center">${row[3]}</td>
            <td class="center">${row[4]}</td>
            <td class="center ${row[5] == 1 and 'relay_on' or ''}">${int(row[5]) and 'on' or 'off'}</td>
        </tr>
    </table>
    <br />
    <a href="/bfcs">Back</a> -- <a href="./${url_date}/all">All</a>
```

```
</body>
</html>
```

Development Plan

- Design daughtercard using Altium Designer. Then have daughtercard fabricated. Estimated turnaround time for PCB fab is ~2 weeks. (1 week design, 2 weeks fab)
- Develop server daemon and sample Raspberry Pi daemon, while PCB is being fabricated. Also begin developing sample test programs using Raspberry Pi and a breadboard. (2 weeks during PCB fab)
- Assemble daughtercard and perform electrical testing. (1 week)
- Develop test programs to exercise single elements of the daughtercard: (1 week)
    - Read and debounce pushbutton
    - Activate relay using GPIO from Raspberry Pi
    - Illuminate each standalone LED
    - Display digits on the 7 segment display
    - Read temperature values from all four remote sensors
- Integrate all test programs together with the sample Raspberry Pi daemon to create the final version of the software. (1 week)
- Assemble and test the entire system. (1 week)

References

1. SAA1064 4-digit LED-driver with I2C-Bus interface Data Sheet
2. 2N3906 / MMBT3906 / PZT3906 PNP General Purpose Amplifier Data Sheet
3. Broadcom BCM2835 ARM Peripherals
4. LTST-C171GKT Data Sheet
5. LDD-A516RI Rev C Drawing
6. PowerSwitch Tail II Product Insert August 2012
7. Temperature and Power Supply SystemMonitors TMP512/3 - SBOS491A
8. System Management Bus (SMBus) Specification Version 2.0
9. Maxim Integrated Application Note 476: Comparing the I^2C Bus to the SMBus