VOLTAGE, CURRENT AND TEMPERATURE MEASUREMENT UNITS: GETTING STARTED & FULL DESIGN

Written for: **Product Health**

www.producthealth.org

By: **Renewable Energy Innovation**

Written by: **Matthew Little**

matt@re-innovation.co.uk

www.re-innovation.co.uk

Date: **6th May 2014**

Version: **0.0**

renewable energy innovation

electrical solutions for renewable energy systems

e: info@re-innovation.co.uk
w: www.re-innovation.co.uk

Hopkinson Gallery
21 Station Street
Nottingham
NG2 3AJ

Contents

Design overview

This document covers the full design and operation of a modular battery monitor.

It is also the 'getting started' guide for implementing a battery monitor system.

Batteries of all types are used for energy storage within renewable energy systems. They are a critical component within stand-alone (off-grid) power supply systems based upon renewable energy. Collecting data from the batteries within these systems helps with fault finding and preventative measures.

Useful battery parameters to measure are: Voltage, Current (in/out) and temperature.

This design is for a modular battery monitor unit which sends data back to a master unit, to be processed and stored.

This has been designed to be:

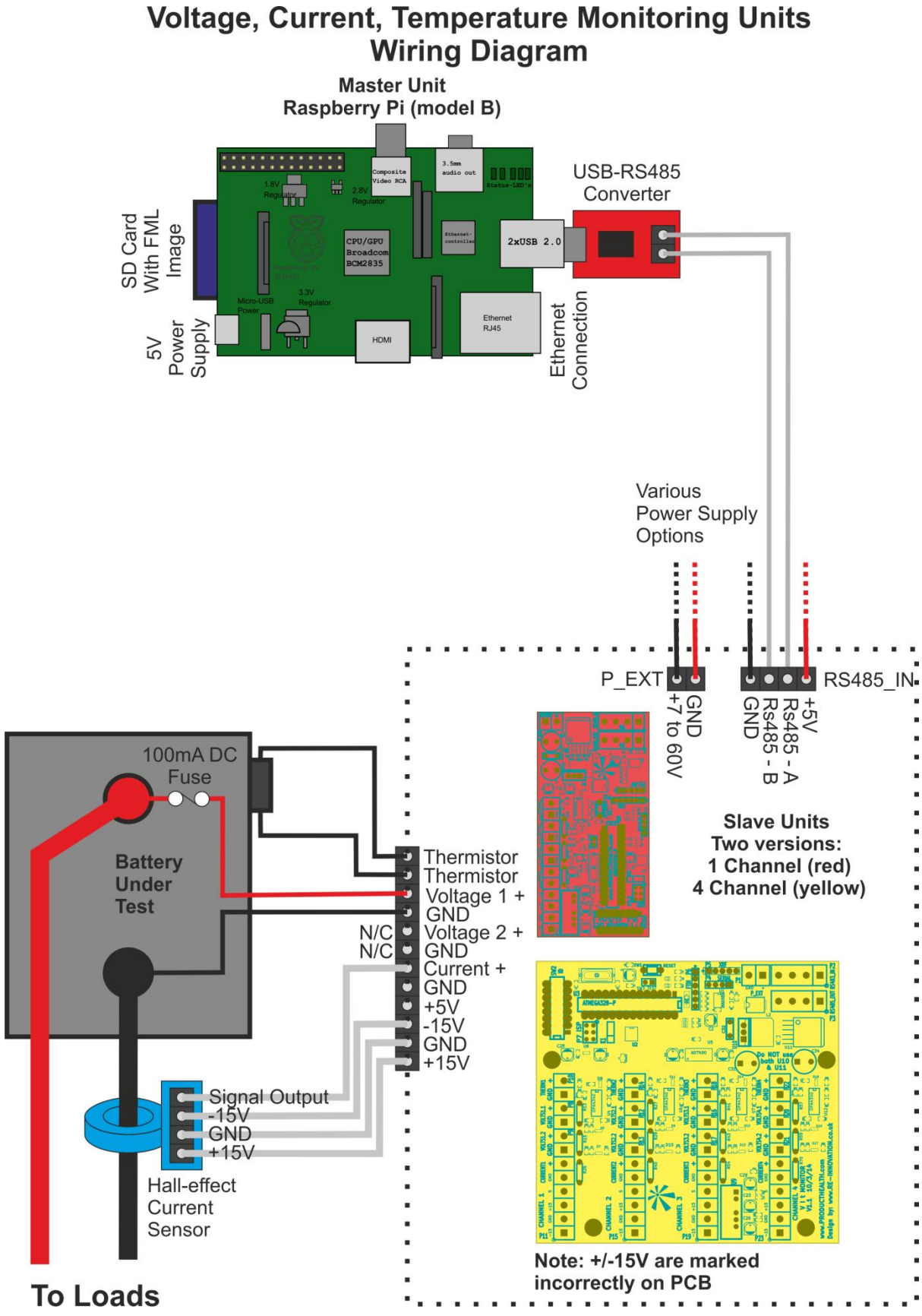
- Accurate
- Low cost
- Open-source
- Modular

There are two units within a basic system the master unit (which asks for data and stores/processed the data) and slave units (which measure the voltage, current and temperature data from the battery). Multiple slaves can be used within a system.

The Master unit is based upon the Raspberry Pi (www.raspberrypi.org) .

The Slave units are based upon the Arduino (www.arduino.cc).

The prototype was designed for Product Health (www.producthealth.org) by Renewable Energy Innovation (www.re-innovation.co.uk).



Wiring diagram

License

Please send any feedback to info@re-innovation.co.uk.

Please note: no responsibility is accepted for the accuracy of the information within this document. Use at your own risk.

More information

Please contact info@re-innovation.co.uk for more information on this project.

Specifications

- Voltage levels: 0-60V DC (up to 48V battery banks)
- Current levels: up to +/-50A maximum (with different ranges)
- Temperature: -10 to 100C.
- Resolution: >10 bit accuracy
- Sample rate: >20Hz for each sensor

Getting started guide

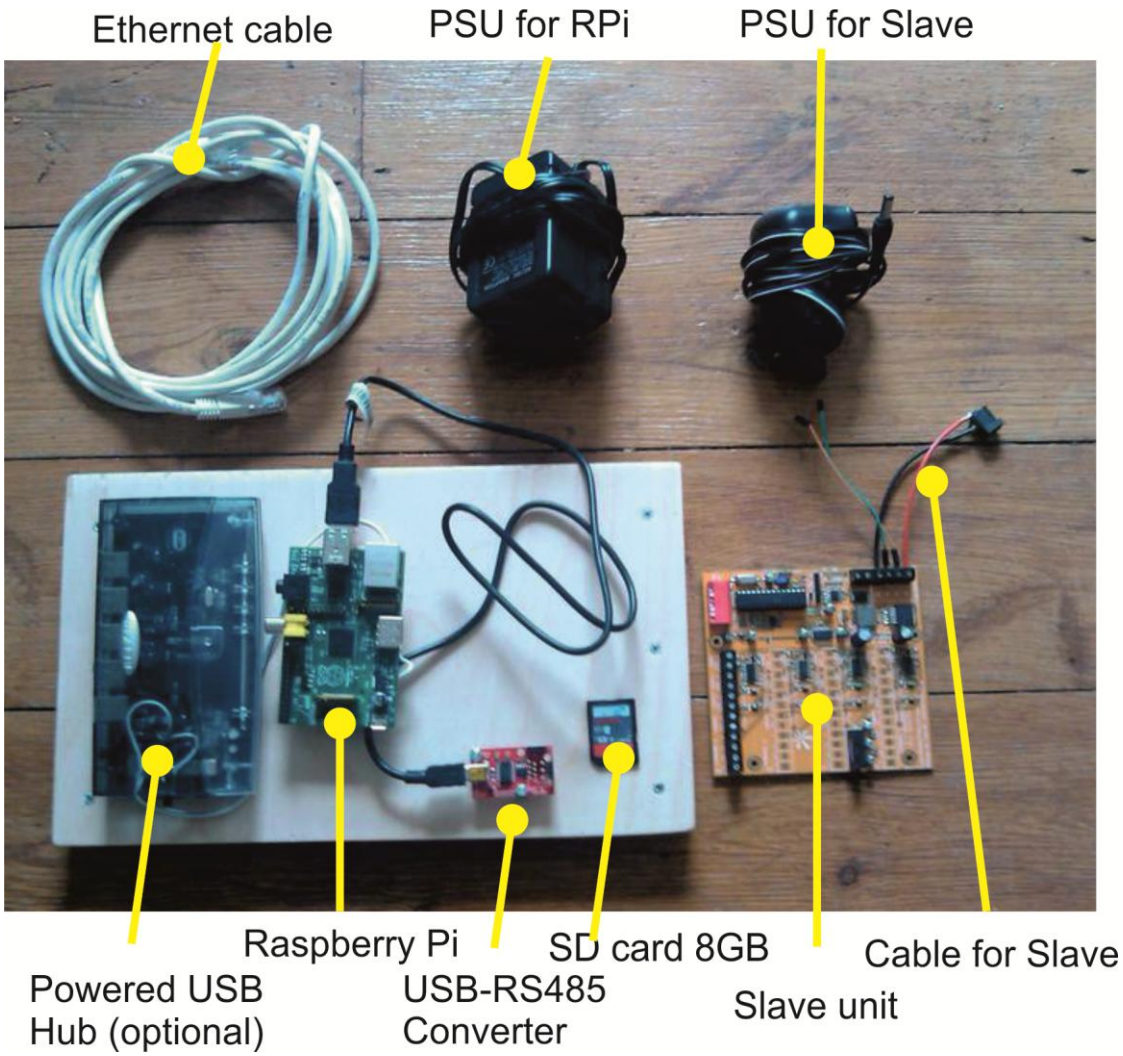
Basic information to get you going. Full design details are given later. Reading the whole of this document is recommended.

Note: Electrical knowledge is assumed here – if in doubt consult a qualified engineer.

Note: Batteries and high voltages are dangerous – Stay Safe.

Parts required:

- Master Unit – Raspberry Pi
- USB-RS485 converter
- Slave Units – 1 channel or 4 channel
- Cable
- 5V power supply



You will also need the sensors to monitor your system including:

- Thermistors
- Hall-effect current sensors
- Cable/fuses for monitoring voltages

Wiring

Depending upon the batteries to be measured, the following wiring diagram gives standard wiring for one battery.

It is recommended that the voltage measurements are fused with a 100mA in-line fuse.

Multiple slave units can be wired to the RS485 bus. Ensure each unit has a different device ID by adjusting the ID switches.

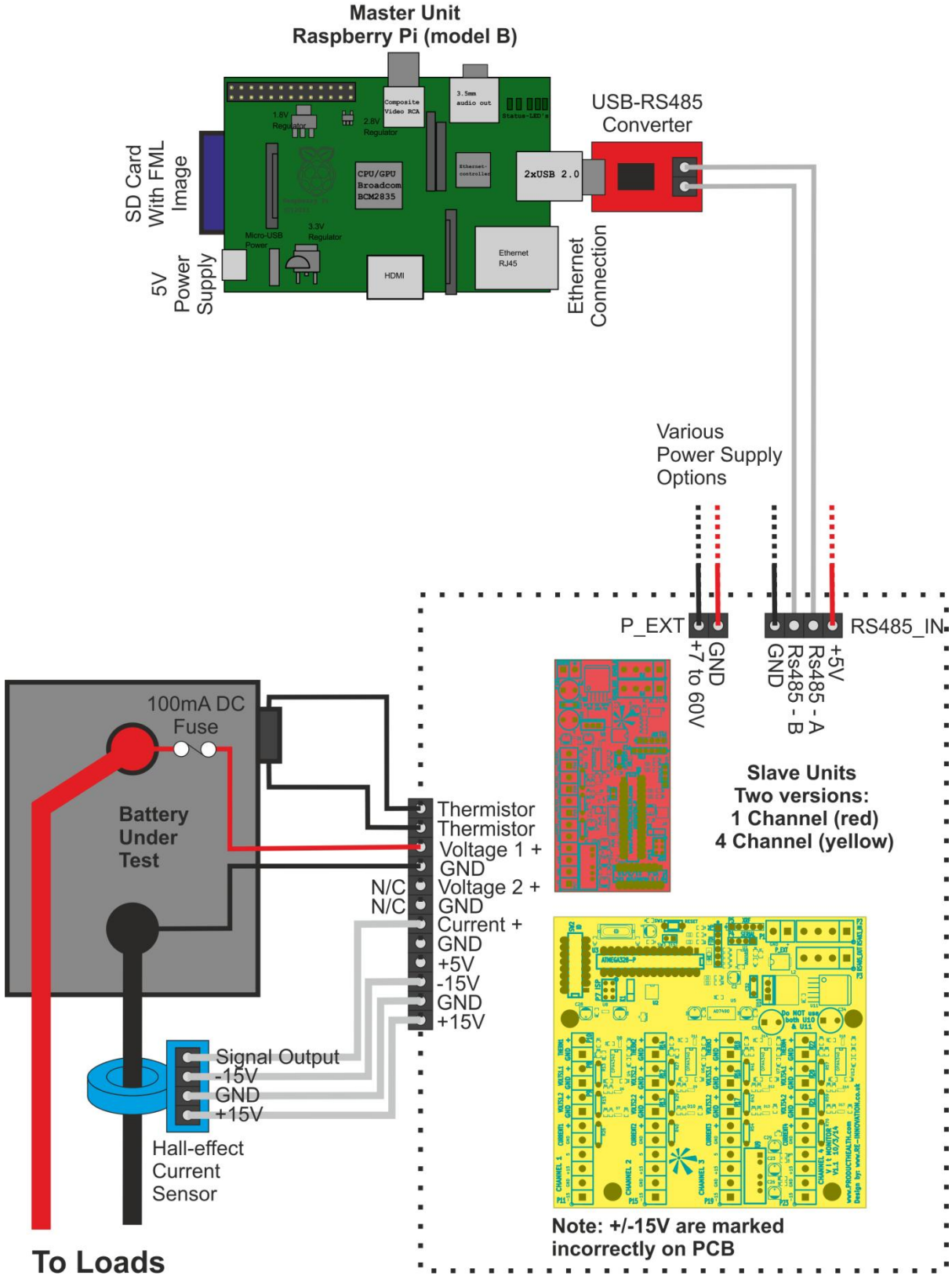
Power supply to the Slave units can either be a regulated 5V supply (ensure at least 200mA per a 1 channel slave unit or 750mA per 4 channel slave unit).

An external power supply can also be used (depending upon the power regulator chosen). This can range from 7VDC to 60V DC. It could also be powered from the battery under test, as long as this is not too high current draw for the system.

When powered, the 'Heartbeat' LED on the slave unit should flash approximately once per second.

The slave units can be tested by applying a jumper to the test pins. This will cause the device to stream data output. This can be viewed on the Raspberry Pi, or with a USB-Serial interface on a local computer.

Voltage, Current, Temperature Monitoring Units
Wiring Diagram



SD card image

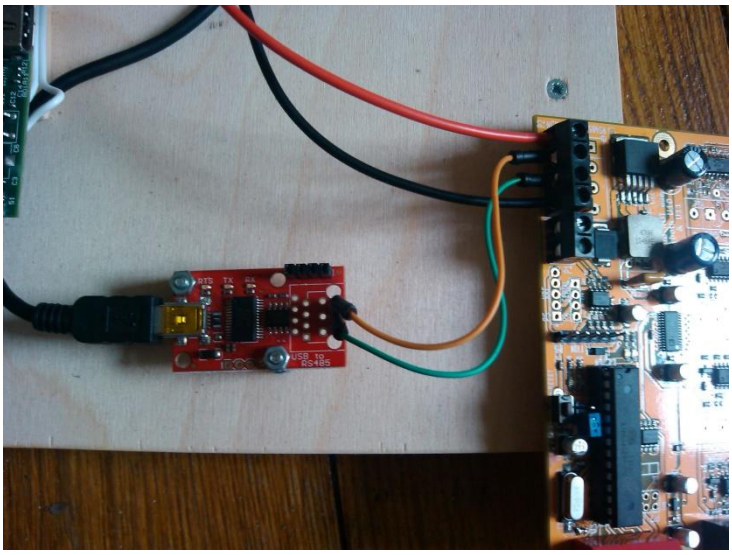
Download the most recent SD card image and place in the Raspberry Pi slot. At least 4GB SD card is required.

Raspberry Pi start up

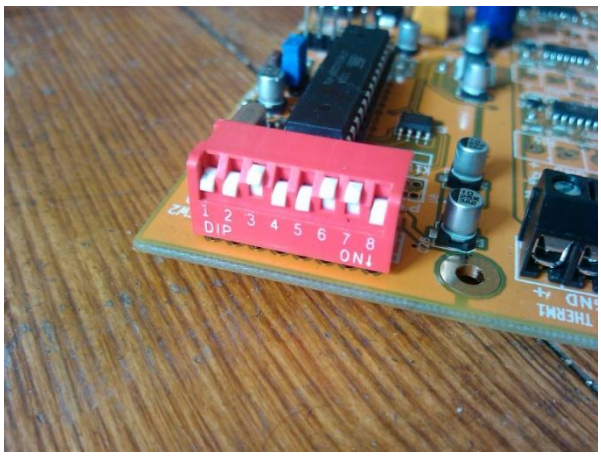
You will need to know:

- The IP address of the Raspberry Pi (use an IPscanner if required)
- The user name and password for the Raspberry Pi (Default for a Raspberry Pi is: 'pi' and 'raspberry').

Firstly, wire up the board RS485 A to A and B to B, power connections.



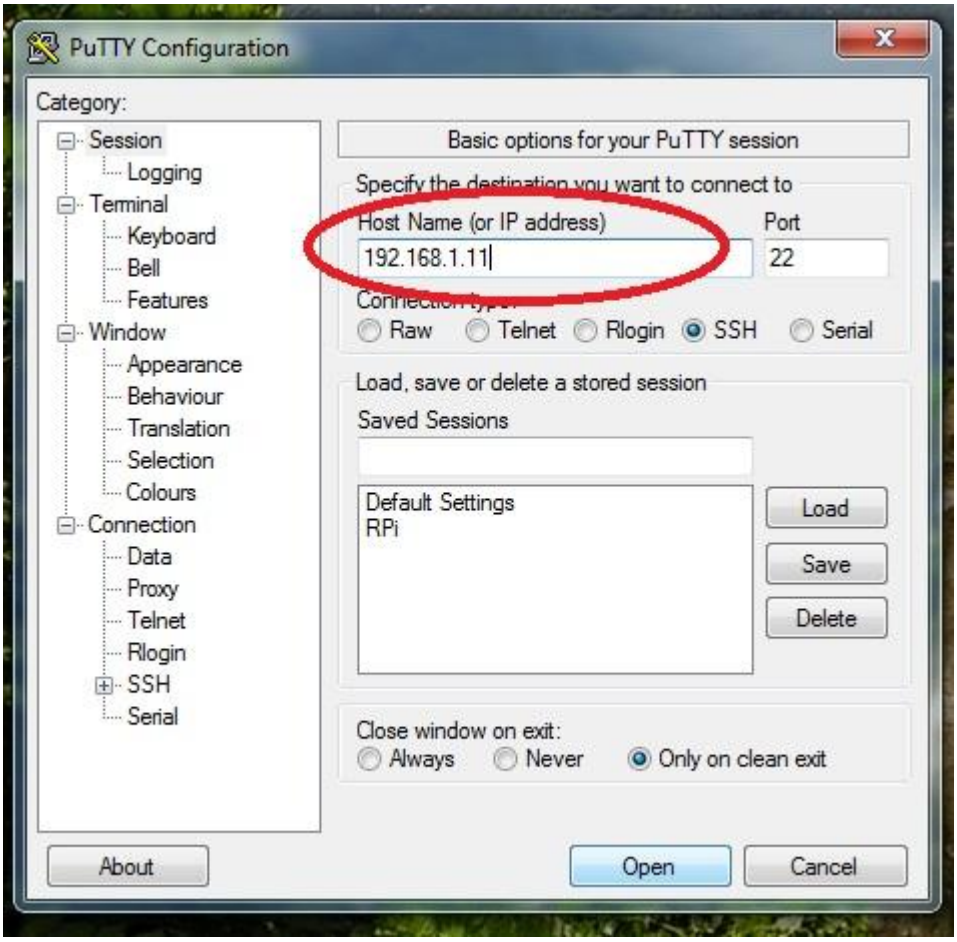
Set the Slave ID to 100. This is the default ID, but it can be changed as shown later.



Connect Raspberry Pi to router with Ethernet cable.

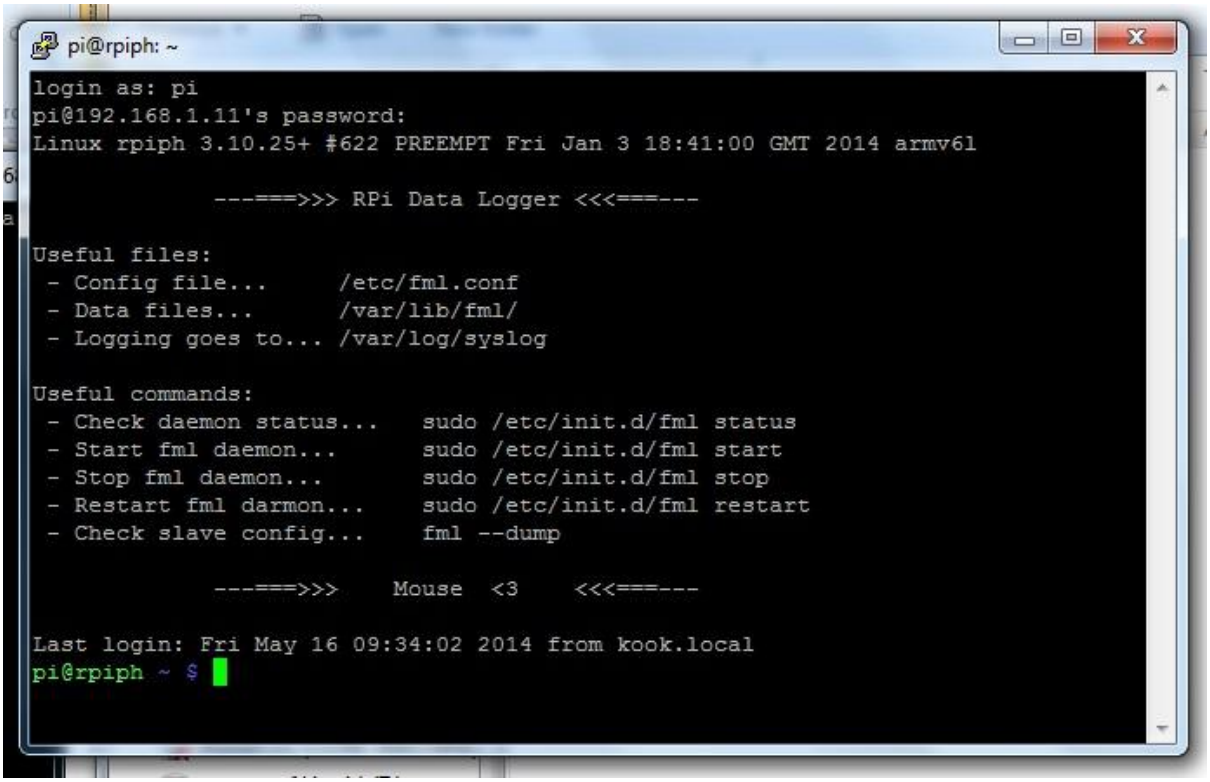
Power up both Master and Slave units (any order).

Start an SSH client (I use PuTTy: http://www.chiark.greenend.org.uk/~sgtatham/putty/)



Connect to the RPi using its IP address.

Use the username and password (RPi default is “pi” and “raspberry”). You will see this start screen, which has information about useful files.



The FML daemon will run automatically.

Checking data

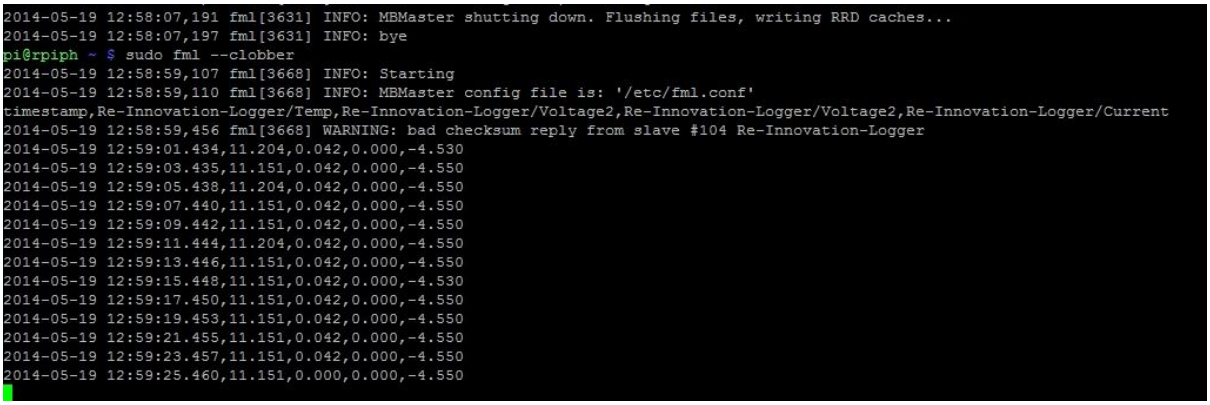
We need to check that the data is being received by the master unit from the slave correctly.

To do this, we need to stop the FML daemon and run in a terminal.

To run in the terminal, stop the daemon mode instance of fml, and run the fml program directly:

```
$ sudo /etc/init.d/fml stop
$ sudo fml
```

You will then see the data arriving:



You can check that the data looks OK here. This can be used each time the configuration ("/etc/fml.conf") is changed.

To stop the terminal mode execution, press control-C. If desired, re-start the daemon mode:

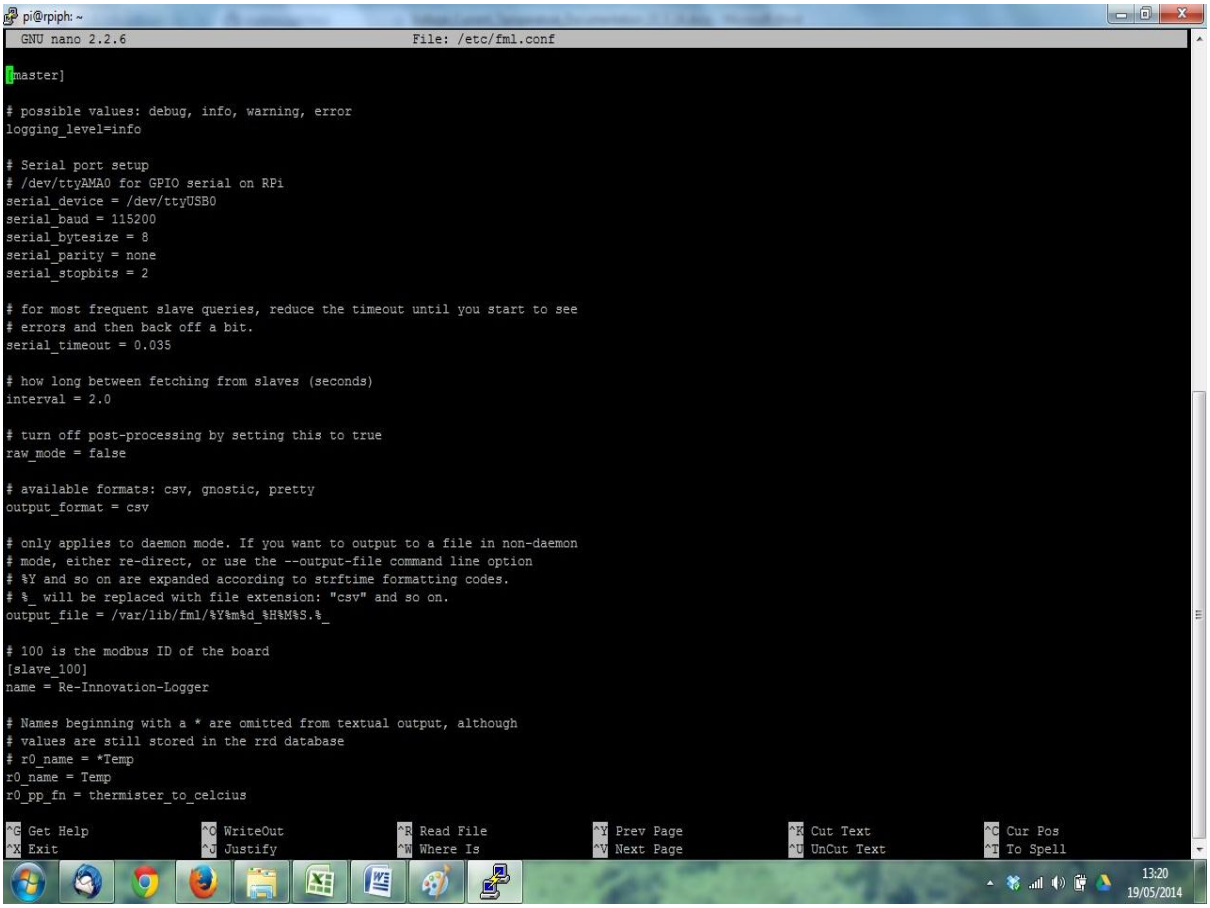
```
$ sudo /etc/init.d/fml start
```

Change the configuration file

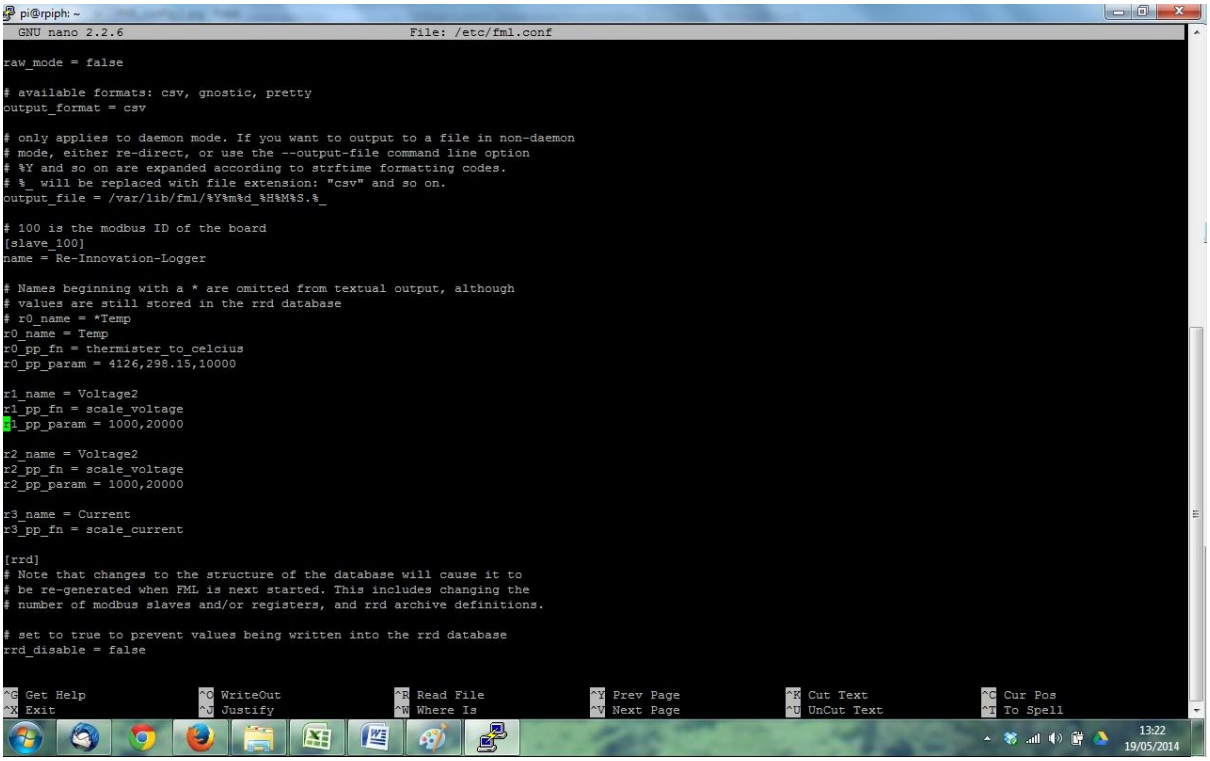
The configuration file can be edited by opening it in an editor program. The Raspberry Pi is installed with nano, so lets use that:

```
$ sudo nano /etc/fml.conf
```

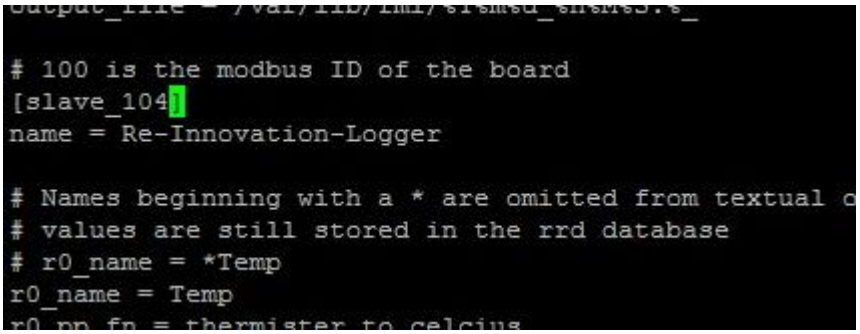
This will open the configuration file:



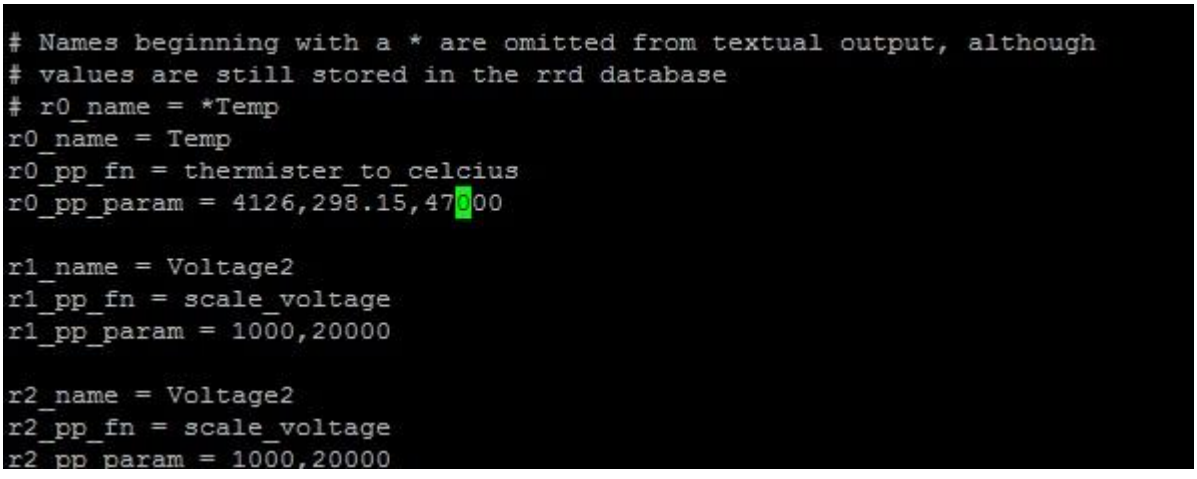
The .conf configuration file can be used to change the serial port settings, the slave ID(s), adding more slave units, change the temperature, voltage and current conversion parameters:



For example here we change the slave ID to 104:



And here we change the thermistor nominal value to 47000 ohms:



After changing the slave ID the round robin database will not function correctly, hence you will need to stop that database and re-start. This is done using the " –clobber" command to create a new database.

Each time we change the configuration file we must restart the Daemon, using:

```
sudo /etc/init.d/fml stop

sudo /etc/init.d/fml start
```

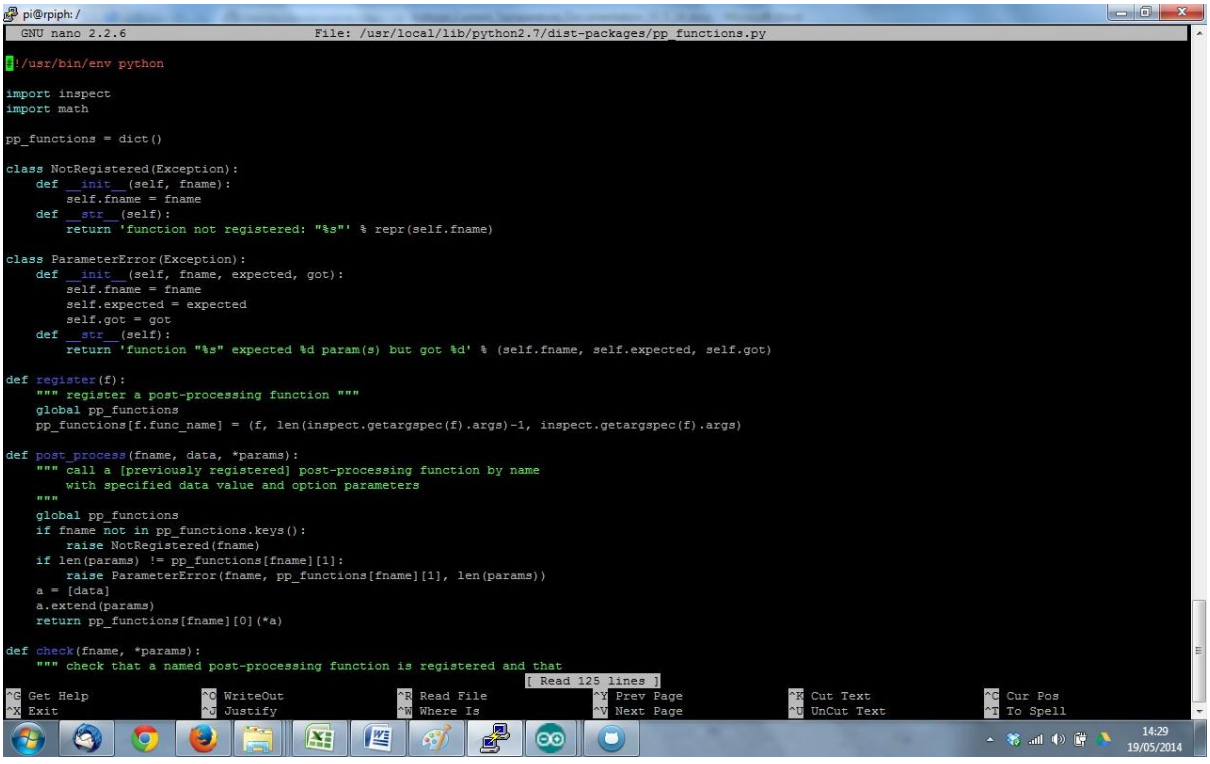
Changing a post processing function

The post processing functions convert the raw 12 bit value into the real-world value, such as temperature, current and voltage. You can write as many of these functions as you like, we have implemented thermistor to temperature, reading to current and reading to voltage.

To change the functions, we need to open the post-processing python script:

```
sudo nano /usr/local/lib/python2.7/dist-packages/pp_functions.py
```

This will open the post processing functions python script:



Change Thermistor post processing

Thermistor values are converted using a standard conversion function:

```
def thermister_to_celcius(data, B, T0, R0):
    # R = (V*R_Balance)/(3.3-V), where V = ((data*4.096)/2048).
    # temp in C = (1.0/(1.0/T0+(1.0/B)*log(R/R0))) - 273.15
    # R0, B, T0 are constants from the circuit / components
    # simplified formulae and generated code from mathomatic.  Assuming R_Balance is the same thing as R0
    # note: string representation of numeric input so convert to floats!
    data = float(data) # convert from string input
    B = float(B)       # convert from string input
    T0 = float(T0)     # convert from string input
    R0 = float(R0)     # convert from string input
    V = data/500.0
    R = V*R0/((33.0/10.0) - V)
    return (1.0/(1.0/T0+(1.0/B)*math.log(R/R0))) - 273.15
```

Change Voltage post processing

Voltage processing depends upon the potential divider used. The function is give here, but may be different for your application:

```
def scale_voltage(data, R1, R2):
    # Conversion from a data reading (16 bit number) into the voltage is:
    # Actual input voltage = ((R1+R2)/(R1))*((data*4.096)/2047)
    # Where R1 and R2 are the input resistor values which will vary with different ranges.
    # simplified by mathomatic
    data = float(data) # convert from string input
    R1 = float(R1)     # convert from string input
    R2 = float(R2)     # convert from string input
    return (data*5.0/4095.) * (R1+R2)/R1
```

Change current post processing

The current processing depends upon the hall-effect current sensor used. The function is given here, but may be different for your application:

```
def scale_current(data):
    # The current varies around a 2V midpoint (set by an accurate voltage reference).
    # If the value is >2V the it is input current, if it is <2V then it is output current.
    # The current transducer puts out 100mV per Amp. So conversion is:
    # Actual input current = (((data*4.096)/2047)-2)/0.1
    data = float(data) # convert from string input
    return (((data*5.0)/4095.)-1.25)/0.1
```

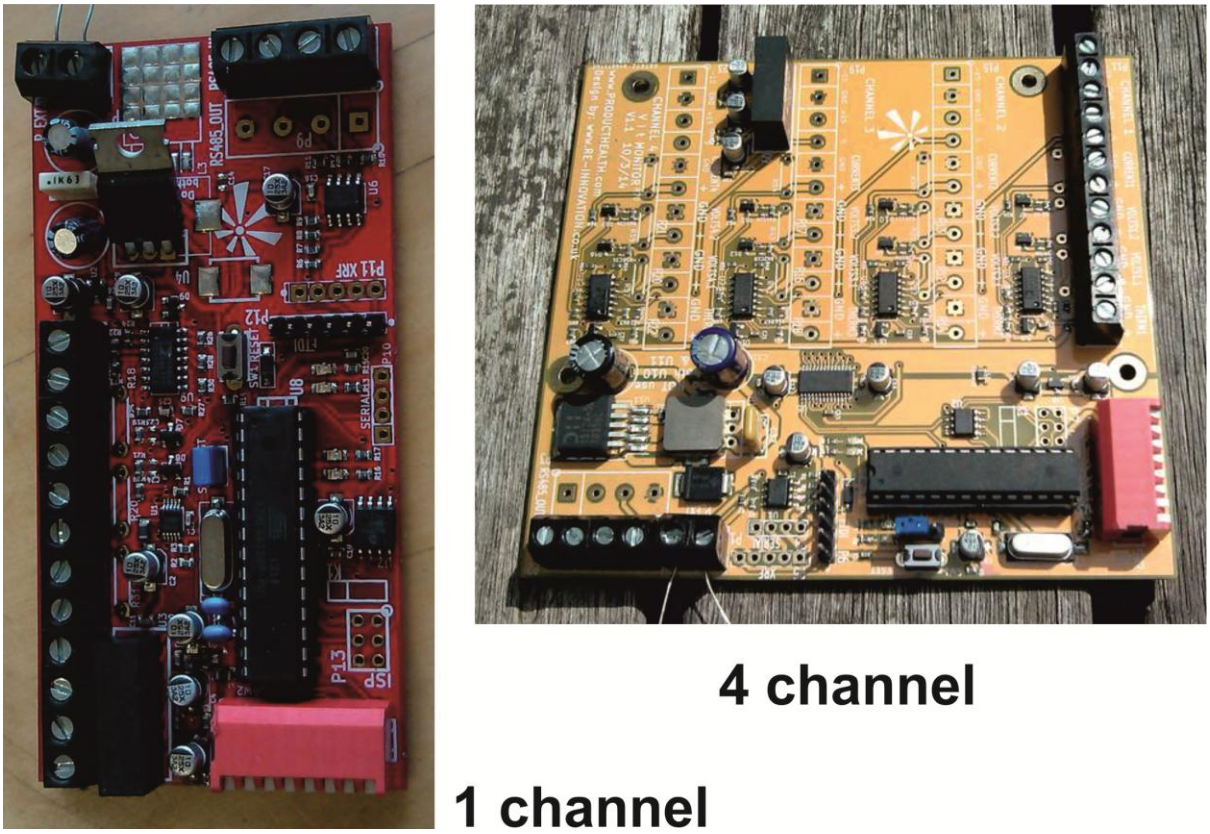
Slave unit - Hardware

Here the full technical design is explained along with design choices made.

This is based upon the ATMEGA329-PU microcontroller, using the Arduino bootloader and IDE for code development.

There are two versions of the slave unit – a unit to measure a single battery (called “1CHANNEL”) and a unit with 4 channels to measure multiple batteries (called “4CHANNEL”).

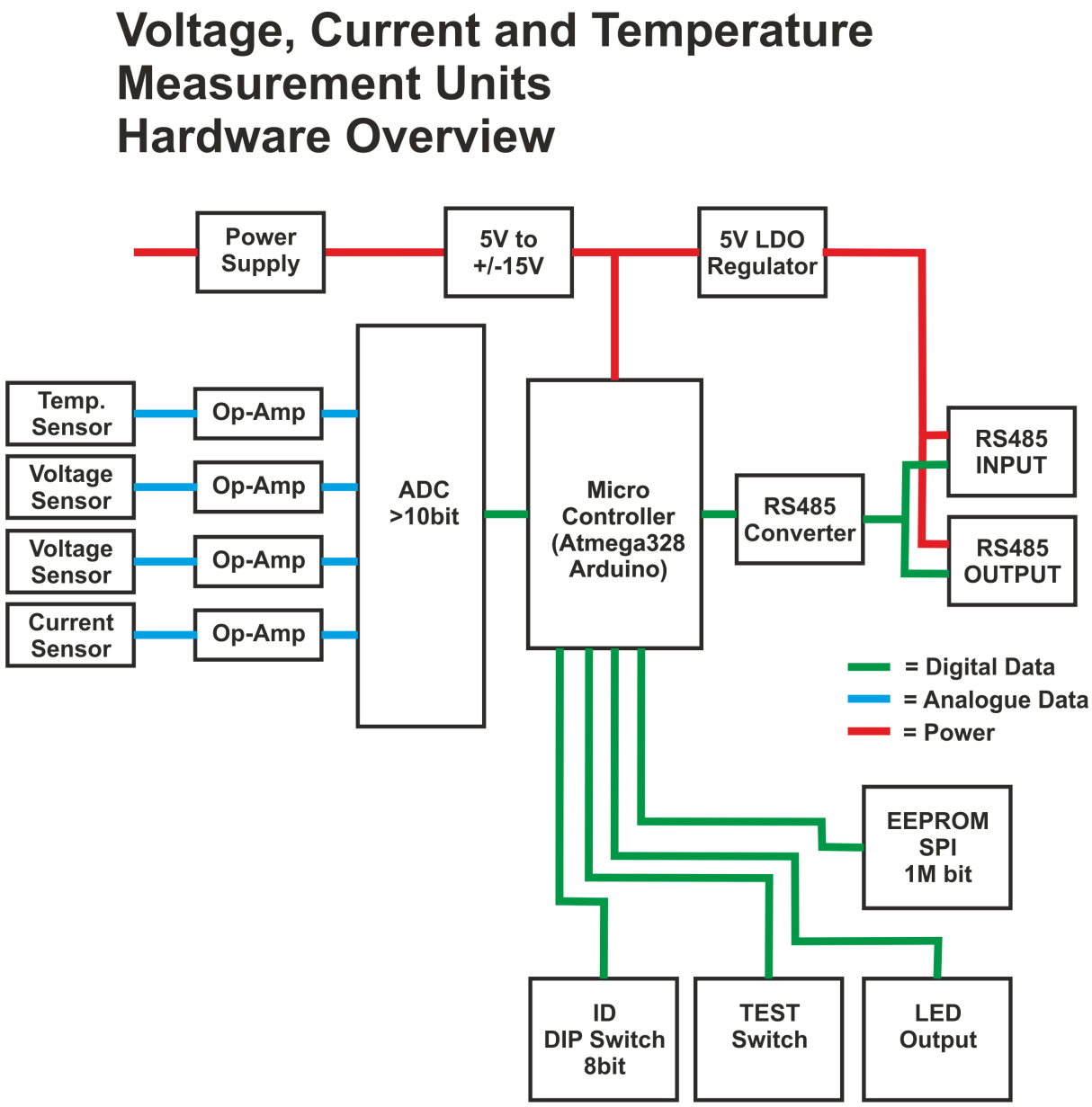
There are slight differences between the two designs, which will be noted when applicable.



The two different versions.

Slave hardware overview

This is a block diagram of the main parts on the slave units. A 1 channel device is shown here – the four channel device has (as you might have guessed) 4 channels, but the rest of the hardware is the same.



Microcontroller

These are all based upon the ATMEGA328-PU (http://www.atmel.com/Images/doc8161.pdf) which is an 8-bit microcontroller with 32Kbyte programmable flash memory, 2K SRAM and 1K EEPROM.

This has been programmed with the Arduino (www.arduino.cc) bootloader and the Arduino IDE (Integrated Development Environment) has been used for programming.

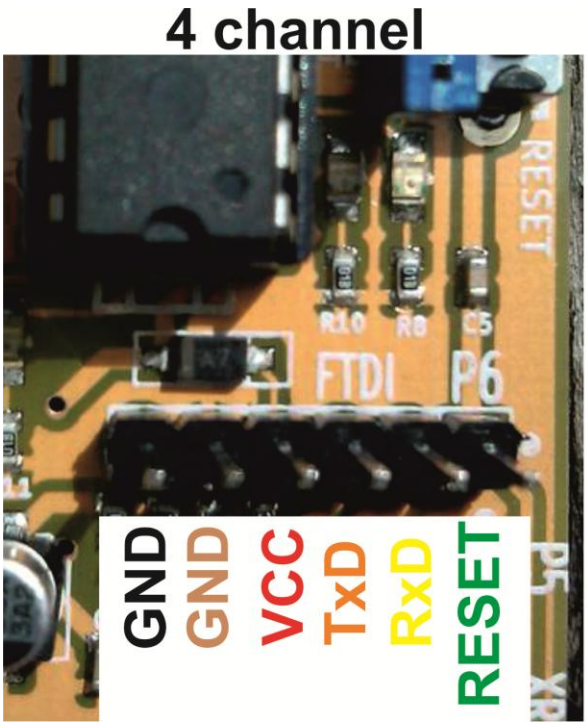
There are lots of examples, large user-base, fast to develop, potentially useful for open-source projects.

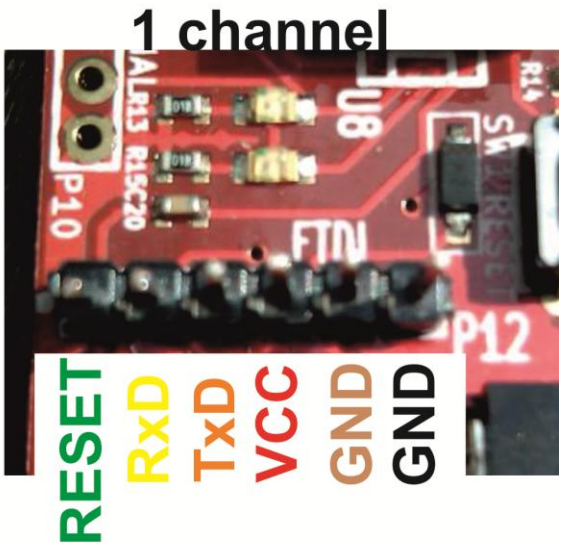
Clock speed is 16MHz. The maximum I2C is 100kHz (without changes to the Arduino IDE).

There may be issues with speed of data transfer, although it should be easily powerful enough for this application (as ADC is handled separately).

Programming of the slave unit is done via an FTDI programming cable (http://www.ftdichip.com/Products/Cables/USBTTLSerial.htm). There is a 6-pin 1x6 socket for this.

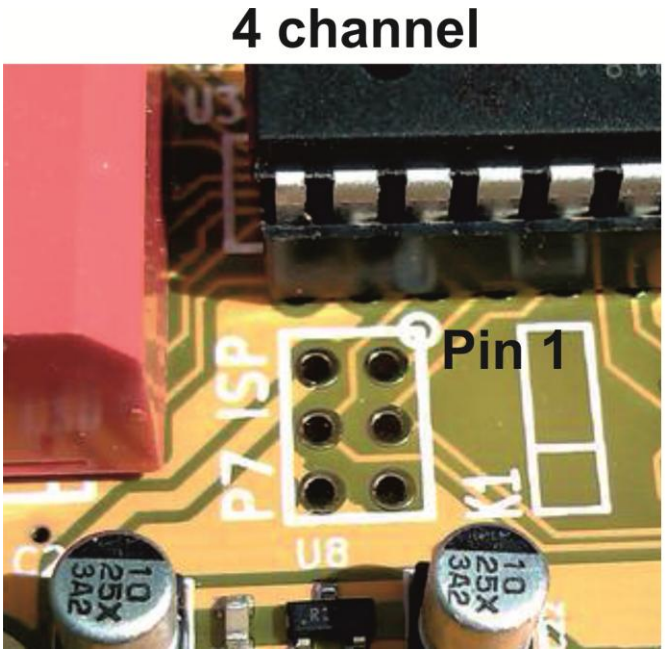
The pins are:

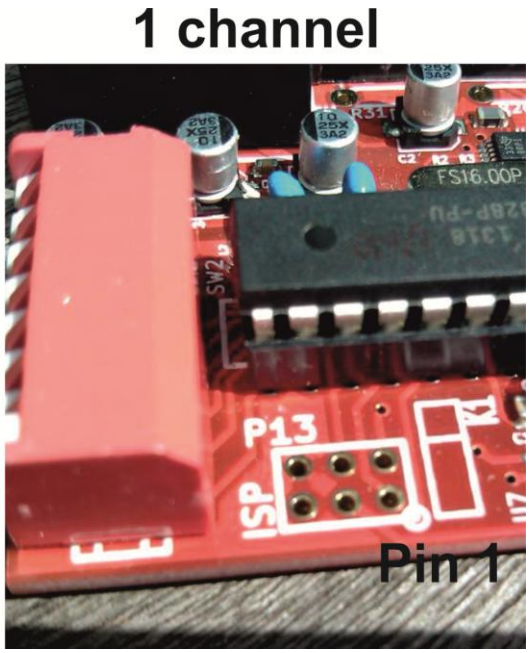




Programming can also be done via an ISP (In Series Programmer), which has a 6-pin 3x2 socket.

The pins are:





Sensors

The slave unit must measure three different parameters – current, voltage and temperature.

Current Sensor

Hall-effect sensors will be used as they are non-intrusive, fail safe and can be retro-fitted. Split core devices can be used for higher currents.

The output is a voltage which is proportional to the current flowing.

The majority of these devices require a +/-15V dual supply – hence we must supply that to the sensor (discussed in the power supply section).

The final choice of hall-effect sensor will depend upon the current range to be measured.

They are manufactured by a number of different companies including LEM and Honeywell.

0-3A: LEM HX 03-P/SP2

http://uk.rs-online.com/web/p/current-transducers/4358684/

0-50A: Honeywell CSCA0050A000B15B01

http://uk.rs-online.com/web/p/current-transducers/7660129/

Voltage Sensor

The specifications are that the unit must be able to measure DC voltage level, up to 60V DC (for 48V battery banks).

This can be easily done using a potential divider. Potential divider can be calculated to give the correct conversion range.

A 5.1V zener diode to protect against potential over voltage.

An operational amplifier is required as a buffer (unity-gain) and filter between the signal and the analogue to digital converter (ADC) input (discussed later).

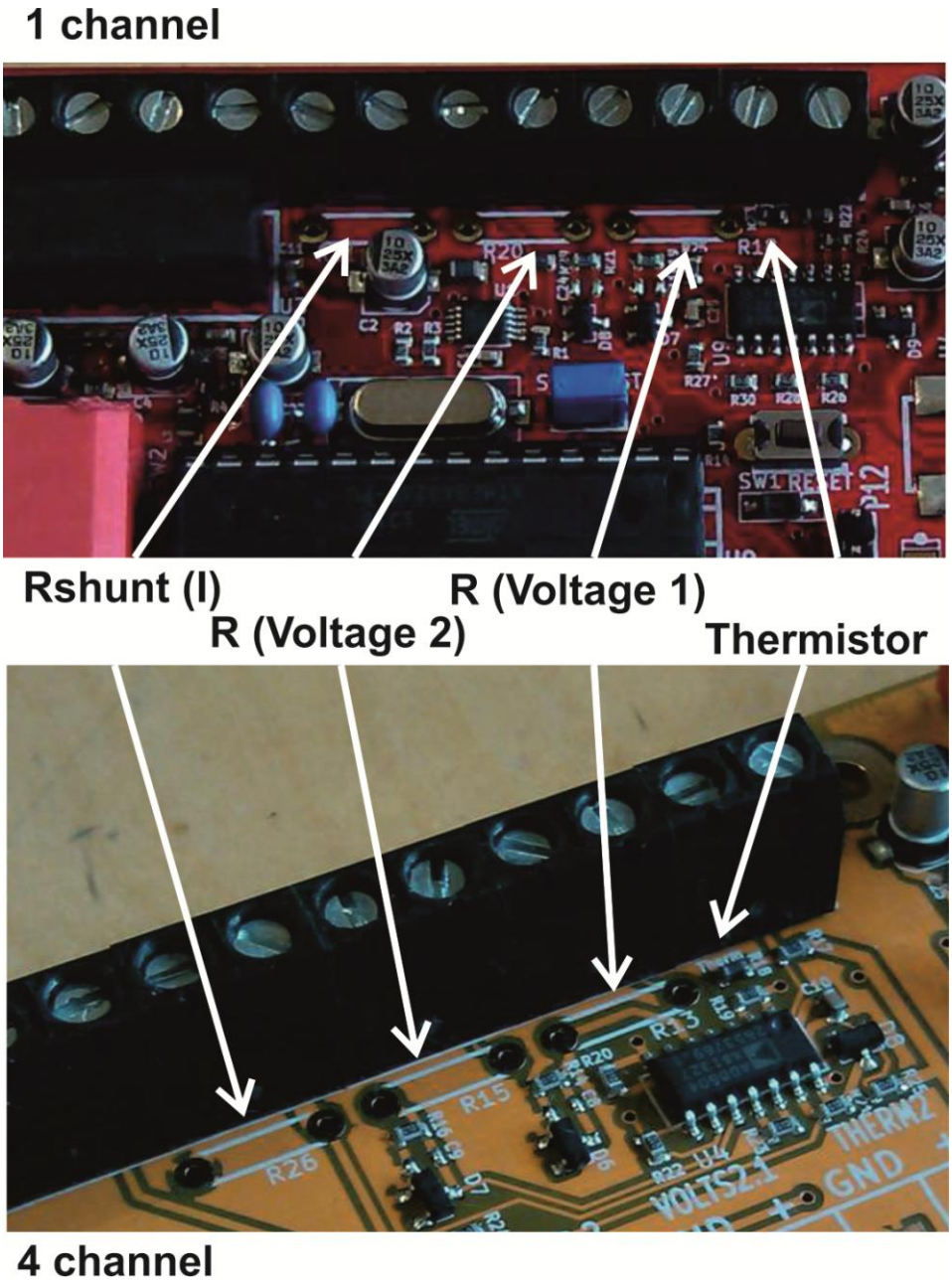
The maximum input to the ADC is 5V so we must calculate the divider circuit to ensure the output does not go above 5V for our input range.

A standard 10K resistor is used as the bottom resistor in the potential divider circuit.

For example:

To measure up to 60V DC:

Keeping within the 5V maximum input and using a 10k resistor as Ra, the potential divider will be comprised of Ra = 10k and Rb > 110k (where Vout = Vmax x(Ra /(Ra +Rb)), hence 5 = 60 x (10/(10+110))). It would be best to have some headroom, in case of any higher voltages, so using a value of Rb = 150k would be best, giving a voltage range of up to 80V.



Temperature Sensor

A thermistor is used as part of a potential divider circuit with an accurate and stable reference voltage.

- Low cost.
- Good accuracy (within 1-2C).

A 10K thermistor is used to record the temperature. The thermistor is put as half of a potential divider, with a 10k precision resistor and a precision voltage regulator to supply the potential divider.

The thermistor output must have the correct interpolation applied so that the temperature reading is accurate. There are standard Arduino libraries to perform this conversion.

The thermistor must be thermally bonded to the unit/device under test.

There is also a connection for a PCB based thermistor, which can be used to measure PCB or ambient temperature.

Operation Amplifier

An operational amplifier is used to buffer the signal from the transducer to the ADC. The ADC consumes current when performing a conversion. This can affect the signal (such as when a potential divider is used) hence an operational amplifier is used to supply the current required, while having very high input impedance for the signal.

Each channel will need a buffer op-amp, hence a quad op-amp package is used.

Must be good quality, linear, rail-to-rail, single supply units with very low off-set.

There are many op-amps available. The Analog Devices AD8604 (http://www.analog.com/static/imported-files/data_sheets/AD8601_8602_8604.pdf) was used in the prototype, but has a standard pin output so other can also be used.

Analogue to Digital Converter (ADC)

The analogue to digital converter is vital to the accuracy of this unit.

The ATMEGA328 has a 10-bit precision internal ADC which, at 5V Vref, gives a resolution of 5V/1024 levels = 4.88mV. This is OK, but not good enough for measuring small current variations from the hall effect sensors.

Each additional bit of precision improves the resolution and hence accuracy of the readings. The ADC must be >10bit precision, to give accurate results.

Differential inputs use two ADC channels to give better resolution and fewer problems with ground noise. But, this would mean using twice the number of ADC ICs, hence in these designs on single-ended measurements are used.

ADC for 1 Channel unit:

The ADS1015 (http://www.ti.com.cn/cn/lit/ds/symlink/ads1015.pdf) 4 channel 12-bit ADC is used for the 1 channel slave units. This is a relatively low cost unit.

This uses an I2C interface to communicate to the microcontroller.

The sample rate is 3.3k samples per second.

The resolution is actually 11-bits as 12-bits are only available when the device is used in differential input mode.

The I2C bus on the Arduino runs at 100kHz by default. The Arduino can run up to 400kHz but requires changes to wire.h.

Link: http://forum.arduino.cc/index.php/topic,16793.0.html

This change was implemented. The maximum I2C clock speed is 400kHz on the ATMega328.

This increased the data speed by around a factor of 2.

ADC for 4 Channel unit

For the 4 channel unit a higher specification ADC is used. This is much higher cost.

The AD7490 (http://www.analog.com/static/imported-files/data_sheets/AD7490.pdf) 16 channel 12-bit ADC is used in the 4 channel unit.

This uses an SPI interface to communicate to the microcontroller.

The sample rate is 1M samples per second, so much higher than the ADS1015.

Power Supplies

The logic and main components of both slave units runs at 5V DC. The hall-effect current transducers may require +/-15V. The thermistors and ADC require an accurate voltage reference. Hence there are three main power supplies on the slave units.

5V regulator

The input voltage will be in the range of 7-60V, depending upon the application. Ideally the unit should be powered by the battery bank under test, although power can be provided externally.

A good quality and stable linear regulator could be used, as long as the current draw is not too high (otherwise some type of switching regulator could be used).

Maximum current draw (approx):

- 20mA microcontroller,
- 20mA ADC and op-amps,
- 600mA for 3W DC/DC converter

Total: 640mA, so need a 1A regulator.

The slave units developed here can use either an LM7805 (http://www.fairchildsemi.com/ds/LM/LM7805.pdf) which is a 1A 5V linear regulator for input voltages up to 35V DC. This is low cost, but inefficient, especially for higher input voltages.

Or the slave units can use an AP1512/A (http://www.diodes.com/datasheets/AP1512_A.pdf) which is a 2A DC/DC converter IC. This requires a high speed diode and inductor, as shown in the circuit diagram. This would be much more efficient for higher voltages and can cope with 60V input.

The 5V power supply section has been designed to use either of the regulator mentioned, or a 5V supply can be fed directly, if available.

+/-15V regulator

NOTE: On this version the + and – 15V supplies are incorrectly marked – they should be reversed.

This is a small DC/DC converter, required for some versions of the hall-effect sensors.

Each sensor requires (up to) 45mA, so power required at 15V is 0.67W for each sensor. So we need a 2W or greater device, to power all three current sensors, or a separate supply could be used for each sensor.

Many different versions are available. They are relatively high cost. Here are some available versions:

- Murata NMA0515S
- TRACO TMA0515D
- XP Power IH0515S

The circuit board has been design to fit the standard SIL package which will fit with either the 1W or 3W versions.

Reference Voltage

An accurate voltage reference is required for the ADC and for the thermistors.

This is provided by the REF3025 (http://www.ti.com/general/docs/lit/getliterature.tsp?literatureNumber=sbvs032f) from Texas Instruments. This is a CMOS voltage reference.

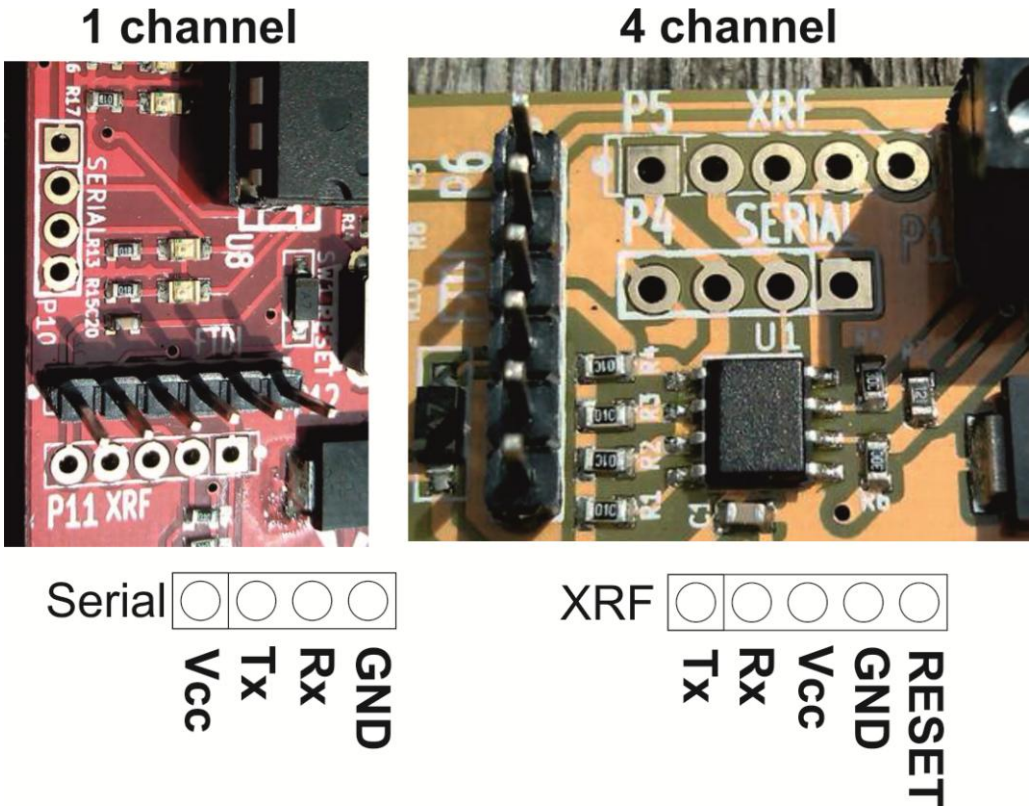
Data output

The output data from the Arduino will be TTL serial at 5V voltage levels. This will work for short range communications, but other outputs are required. One requirement was an RS485 compatible output.

Data is sent at 115200baud 8 bits No parity 1 stop bit (8N1).

Serial

The serial port has +5V, Tx, Rx 0V and can be used as a direct connection to a serial interface.



XRF

For wireless communications, which could be useful in some environments, a standard connection for the XRF low-cost wireless module (http://shop.ciseco.co.uk/xrf-wireless-rf-radio-uart-rs232-serial-data-module-xbee-shape-arduino-pic-etc/) is provided.

This unit runs at 3.3V so would need an XBBO breakout board and voltage regulator.

This can be used as a drop-in replacement for a serial connection, which may be useful in some applications. The connection diagram is above.

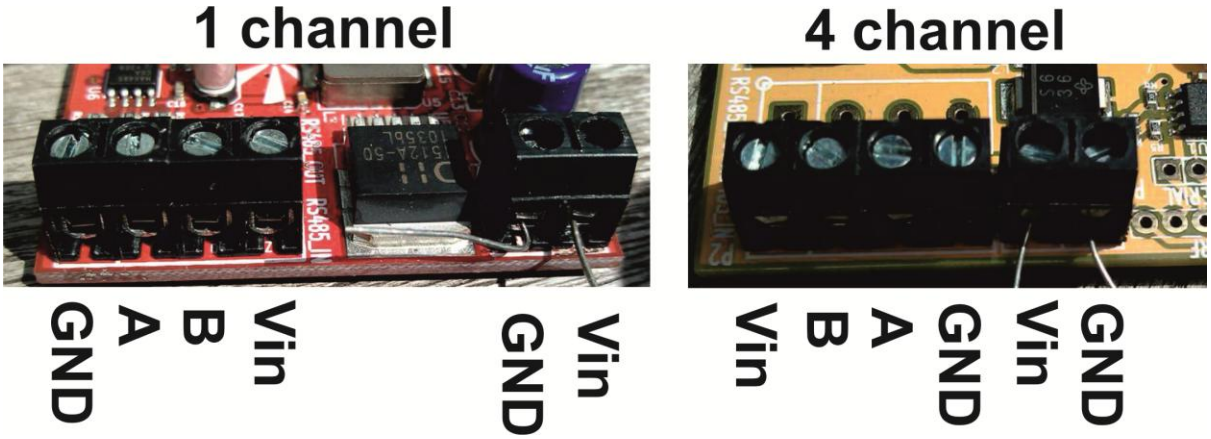
RS485 output

RS485 is an industry-standard for the signal voltage levels designed for high-noise industrial environments. This defines the physical layer (not the communications protocol (which uses MODBUS)). RS485 consists of three lines A, B and Ground. A will be an inverted version of B. Ground may not be required.

The output from the Arduino will be TTL serial. This must be converted into a correct RS485 signal. This is performed with an RS485 converter IC.

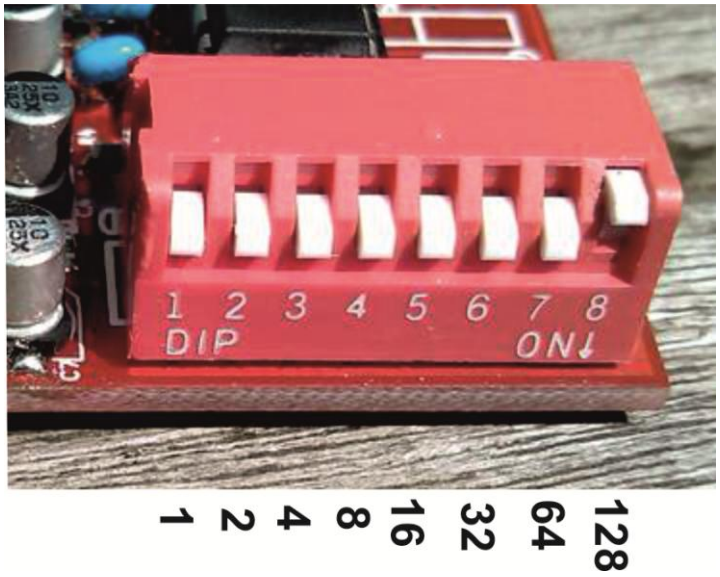
A half-duplex RS485 converter IC is used: the MAX485 (http://datasheets.maximintegrated.com/en/ds/MAX1487-MAX491.pdf) from Maxim.

This is a half-duplex device (so data can either flow in one direction or the other, never both at the same time). A transmit enable pin is required which is set whenever data is transmitted from the slave to the master. This is a digital pin from the ATMEGA328.



ID switch

This sets the ID for the board. This is an 8-way DIP switch to make it easily field programmable. This gives 256 different addresses. It uses 8 digital channels of the microcontroller.



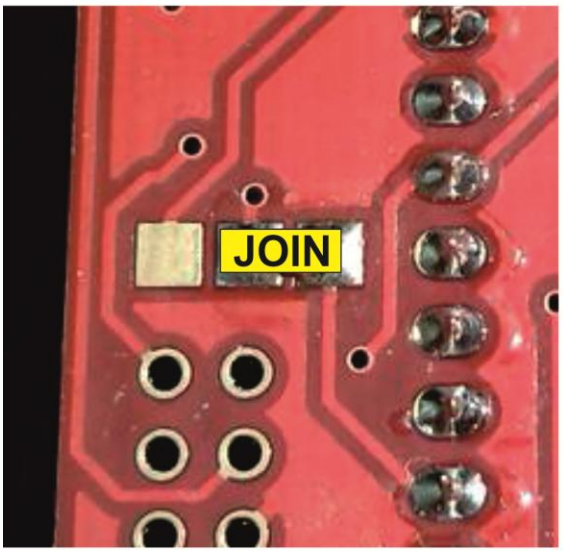
EEPROM

This has been added for future development. A 1M bit serial SPI EEPROM IC has been added to enable remote re-programming of the ATMEGA328.

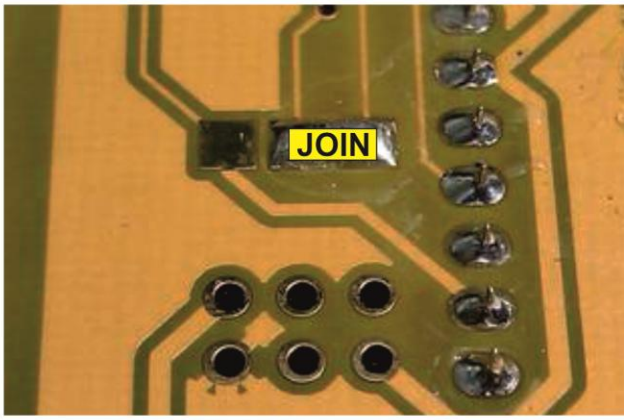
This has enough space so that two versions of the 32K byte code can be loaded. The ATMEGA328 can be programmed to read one version, while the other version can be altered via remote serial programming.

Due to the fact that the SPI is also used for the 4 channel salve unit a jumper pad is used to distinguish between the two versions. This is labelled K1 and should be connected as follows:

1 Channel



4 Channel



For SPI ADC

For I2C ADC

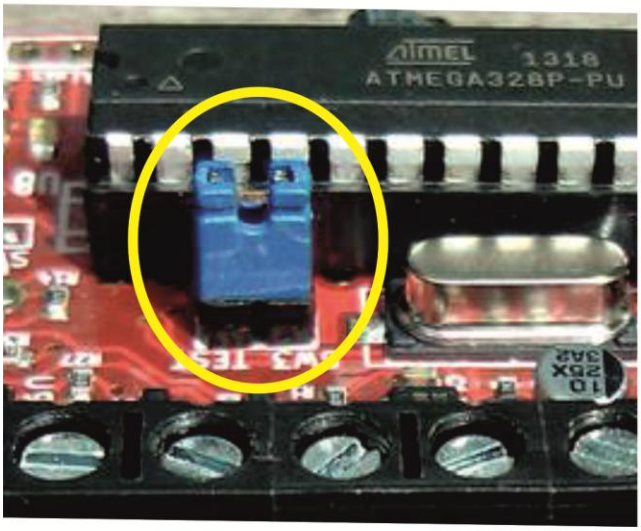
Note: This feature is not yet implemented and is for future reference.

Test Switch

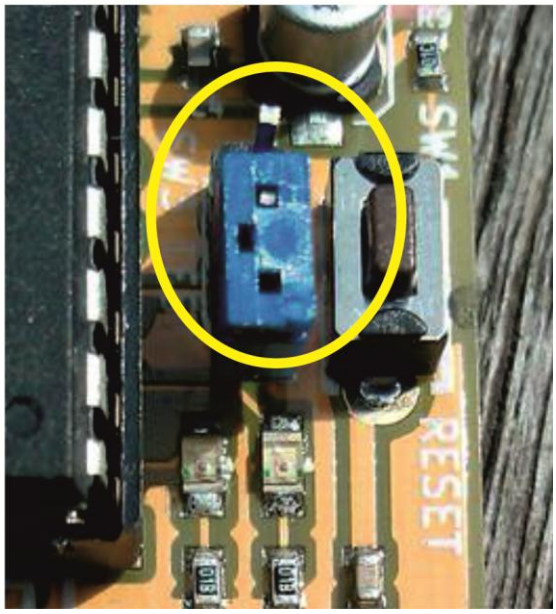
To put the slave unit into test mode a simple jumper pin is used. With a jumper applied then the data output will be written to the serial port.

Without a jumper pin then the slave unit will only respond when a MODBUS request is made.

1 Channel



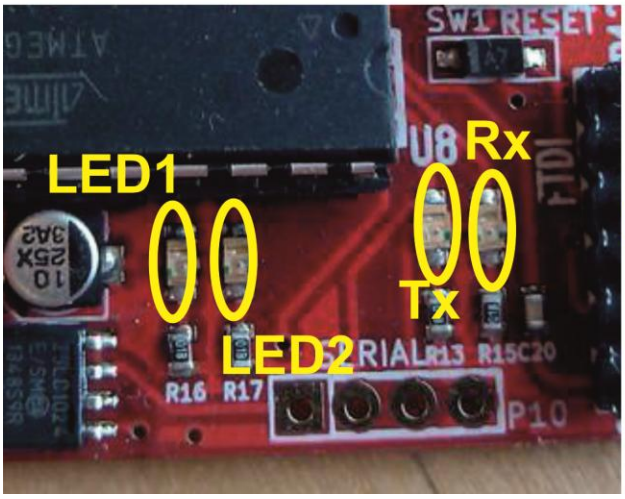
4 Channel



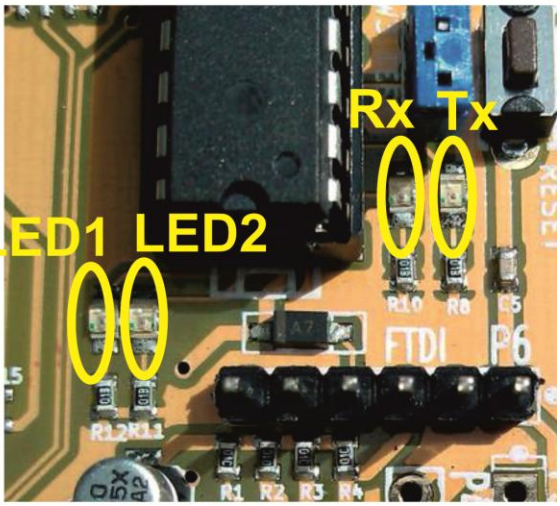
Output LEDs

There are four status indicators LEDs.

1 Channel



4 Channel



Tx and Rx both have LEDs, to show when they are active.

LED1 is a 'heartbeat' LED and shows that the micro-controller is running correctly. It flashes once per second.

LED2 shows when data is written to the serial port.

Upload Arduino code to slave unit

To upload the microcontroller code to the slave unit we can use the Arduino IDE.

Note: This assumes basic knowledge of using the Arduino IDE and uploading code using the Arduino bootloader.

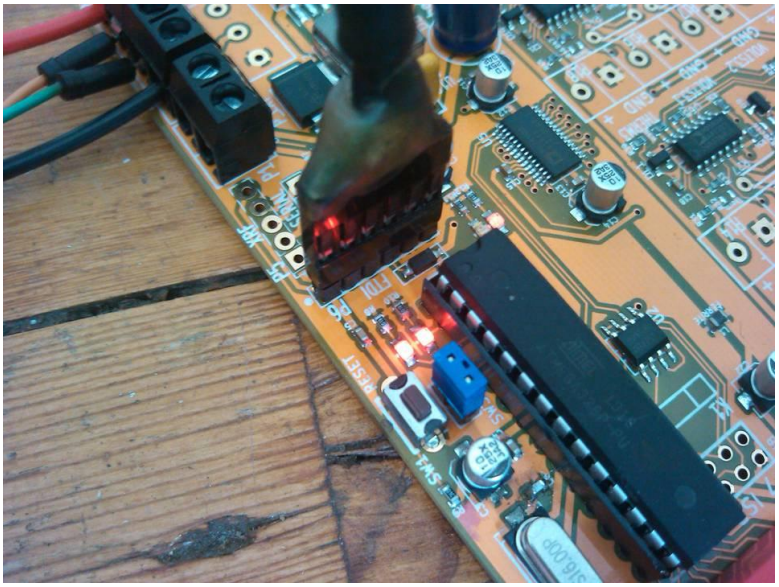
Download and install the latest version of the Arduino IDE (http://arduino.cc/).

Download the latest version of the software from the GitHub account (****to add***).

Download the simple-MODBUS library (http://code.google.com/p/simple-modbus/) and upload to your sketchbook.

Use an FTDI USB-Serial cable (code: **TTL-232R-3V3** available here: http://www.ftdichip.com/Products/Cables/USBTTLSerial.htm along with other places).

Plug in the FTDI cable:



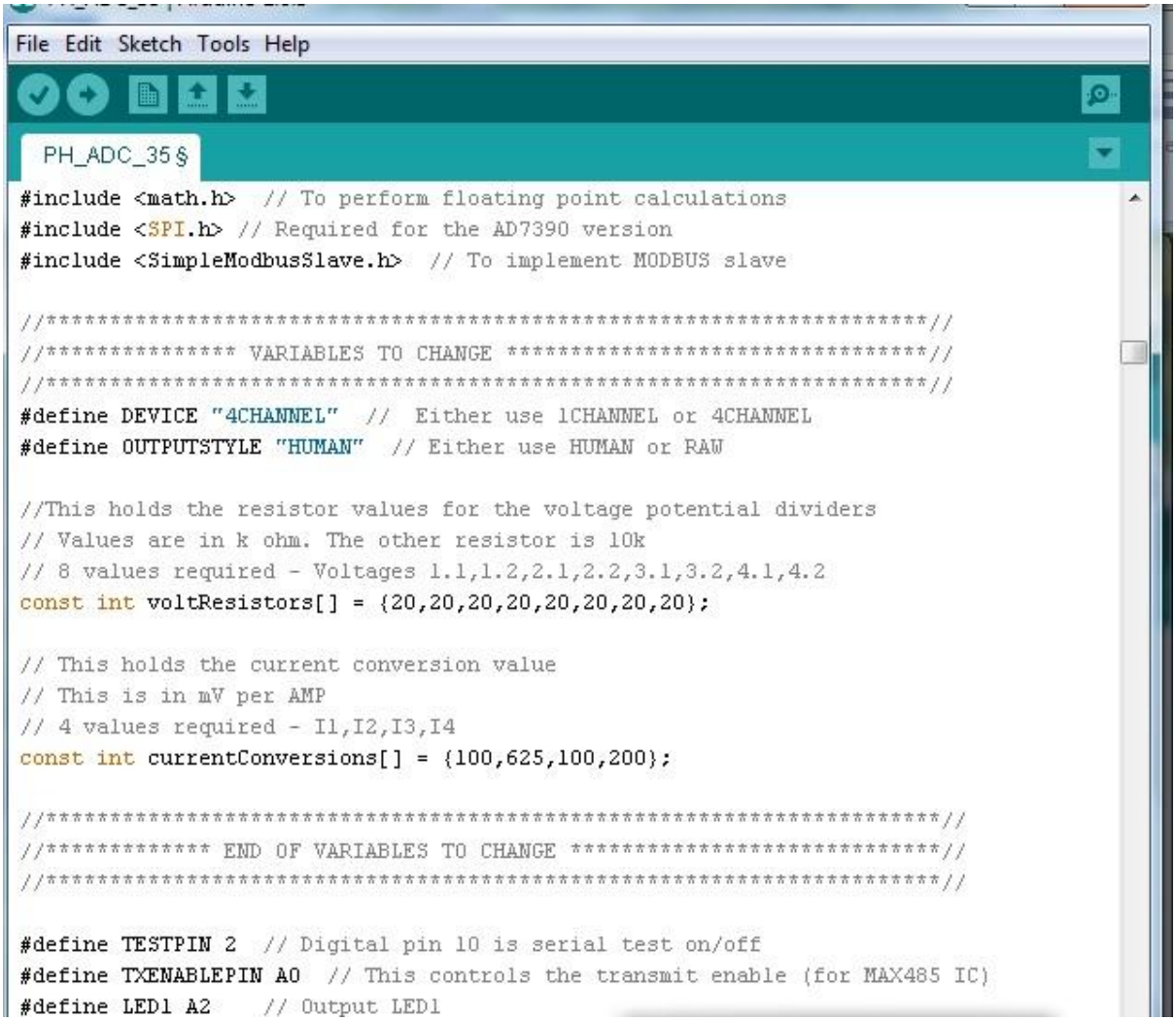
Upload code to the device, using the COM port for your FTDI cable.

There should be no errors.

Put the slave unit into test mode (using a jumper) and view the output on a serial monitor.

Within the Arduino code there is the option to set if the slave unit is a 1 channel or 4 channel device. You can also set if the test-mode output is HUMAN (converted into human readable format) or RAW (just shows the 12bit number (0-4095)).

You can also set the thermistor, voltage and current conversion factors. This only affects the human readable output.



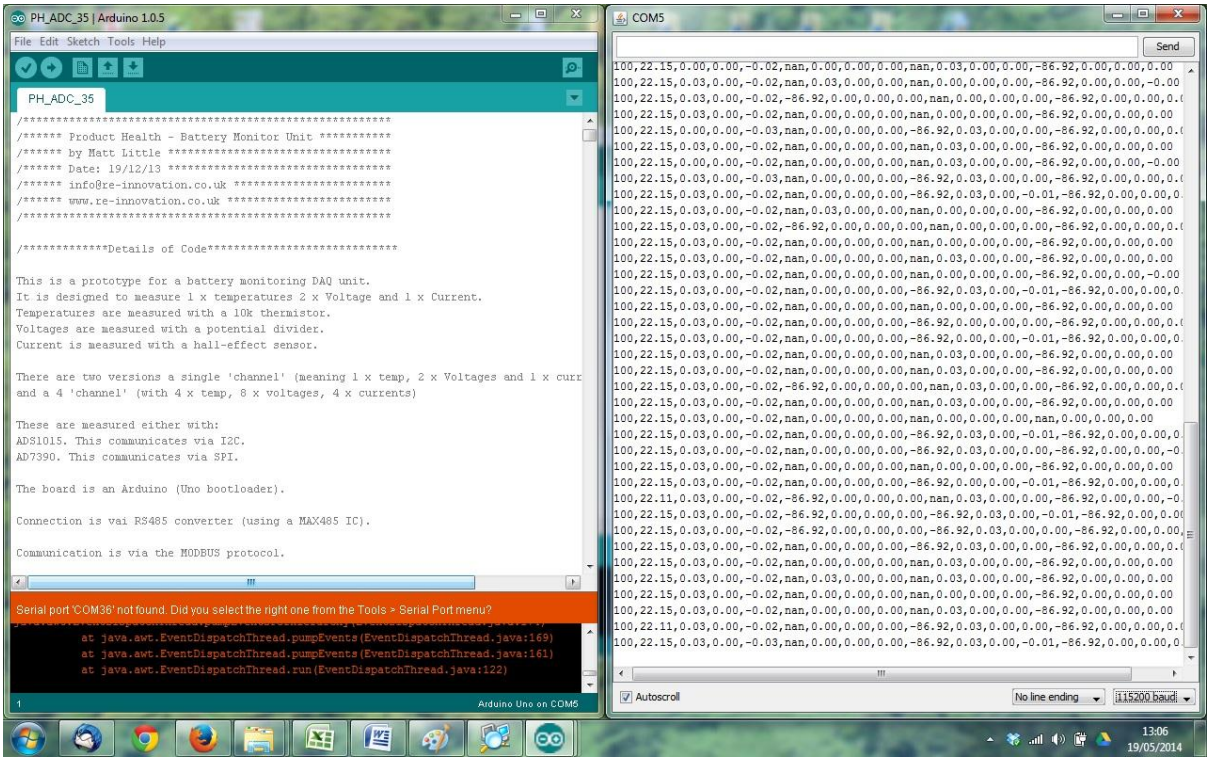
Test Mode for slave unit

When the test pin is set to low (a jumper placed on the pins), then the output on the serial line will be a raw data output.

The data will start with the ID number and the next numbers will be:

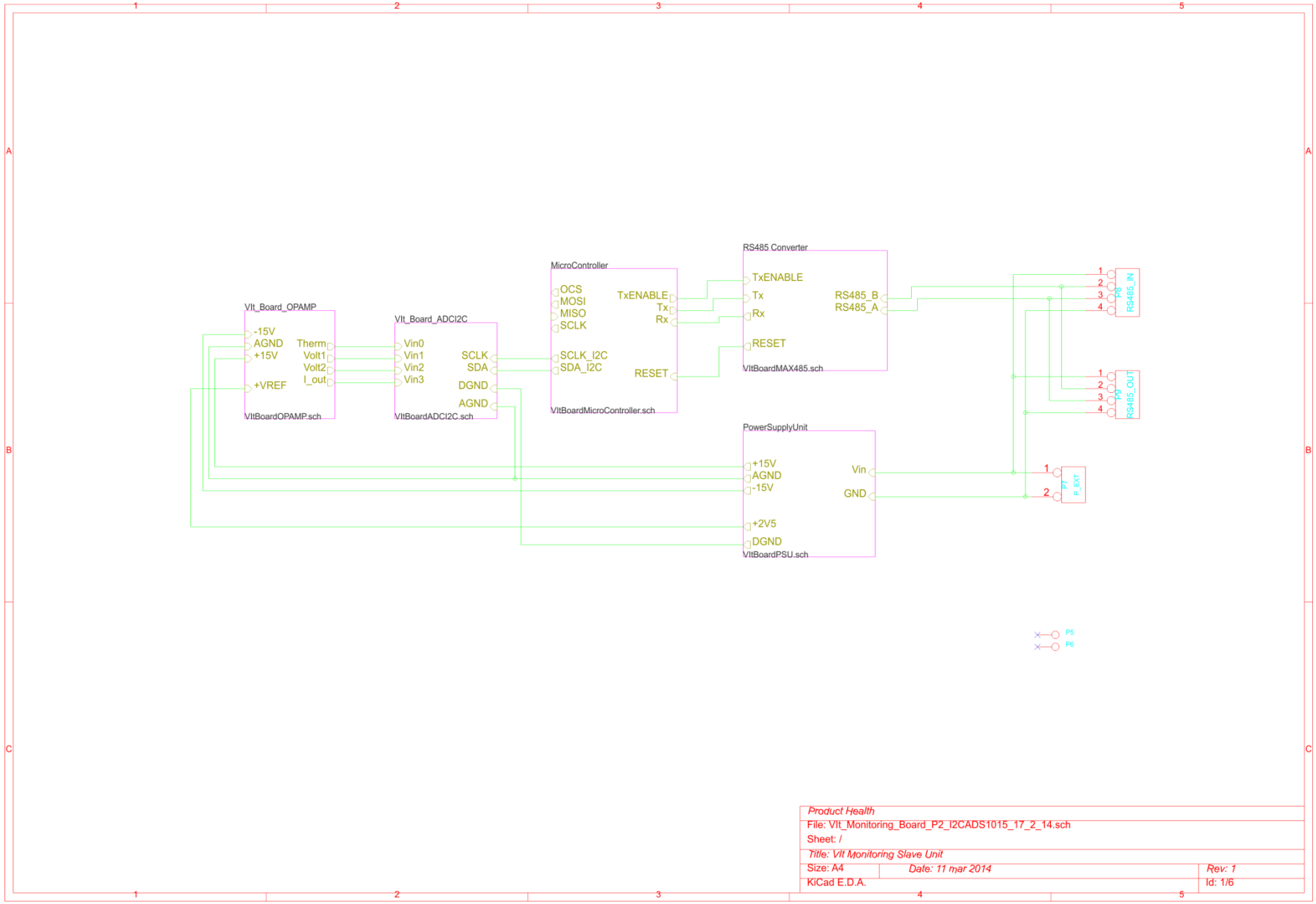
Temp1, Voltage 1.1, Voltage1.2, Current1 (for the 1 channel version)

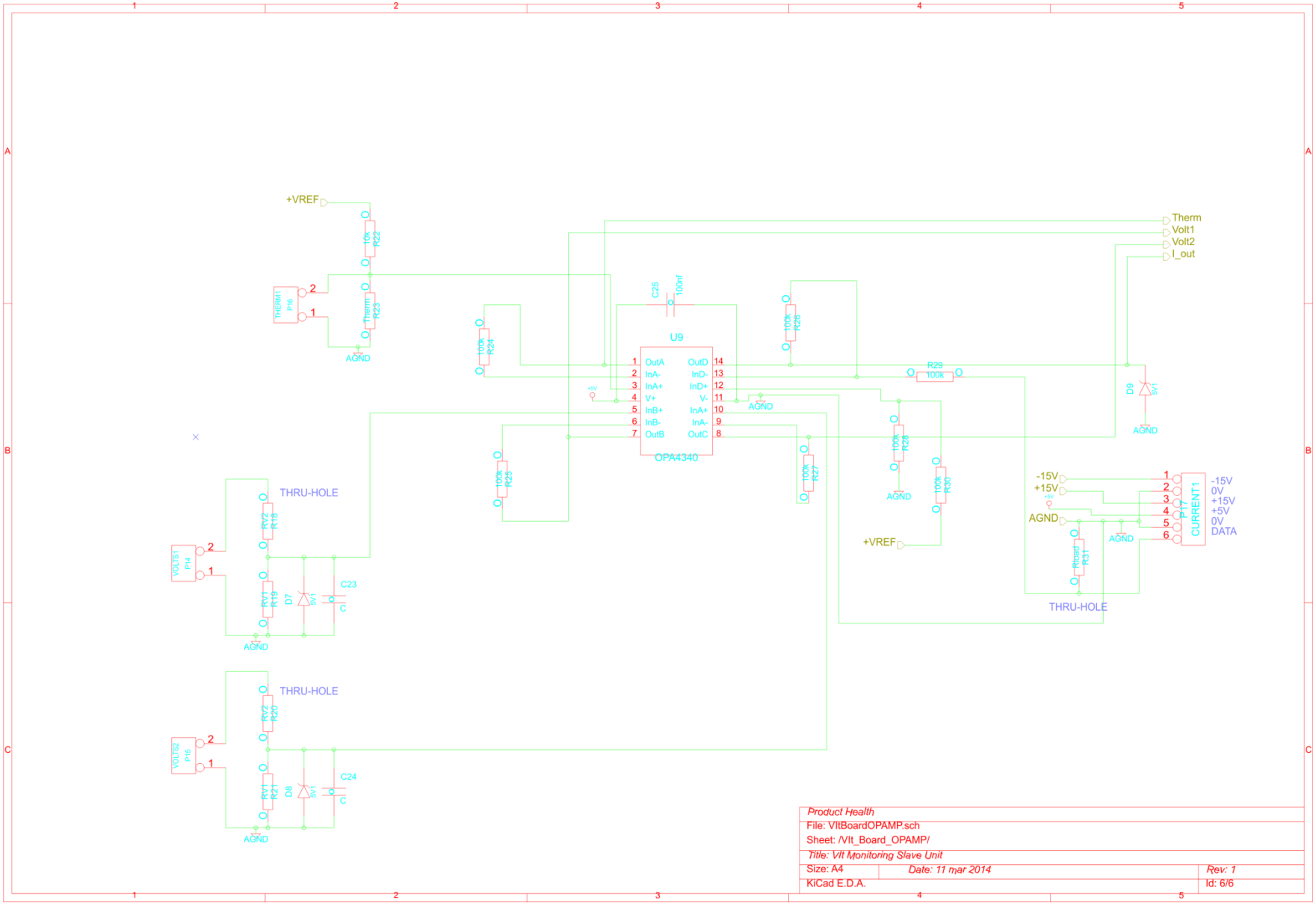
Or the same thing repeated 4 times for the 4 channel version.

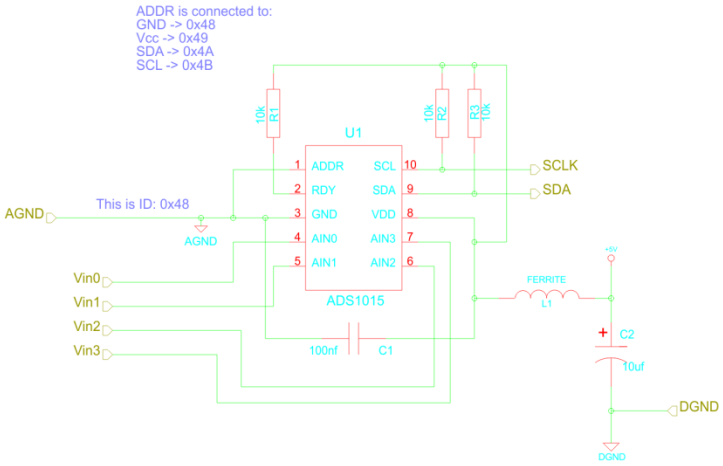


Slave unit 1 Channel circuit diagram

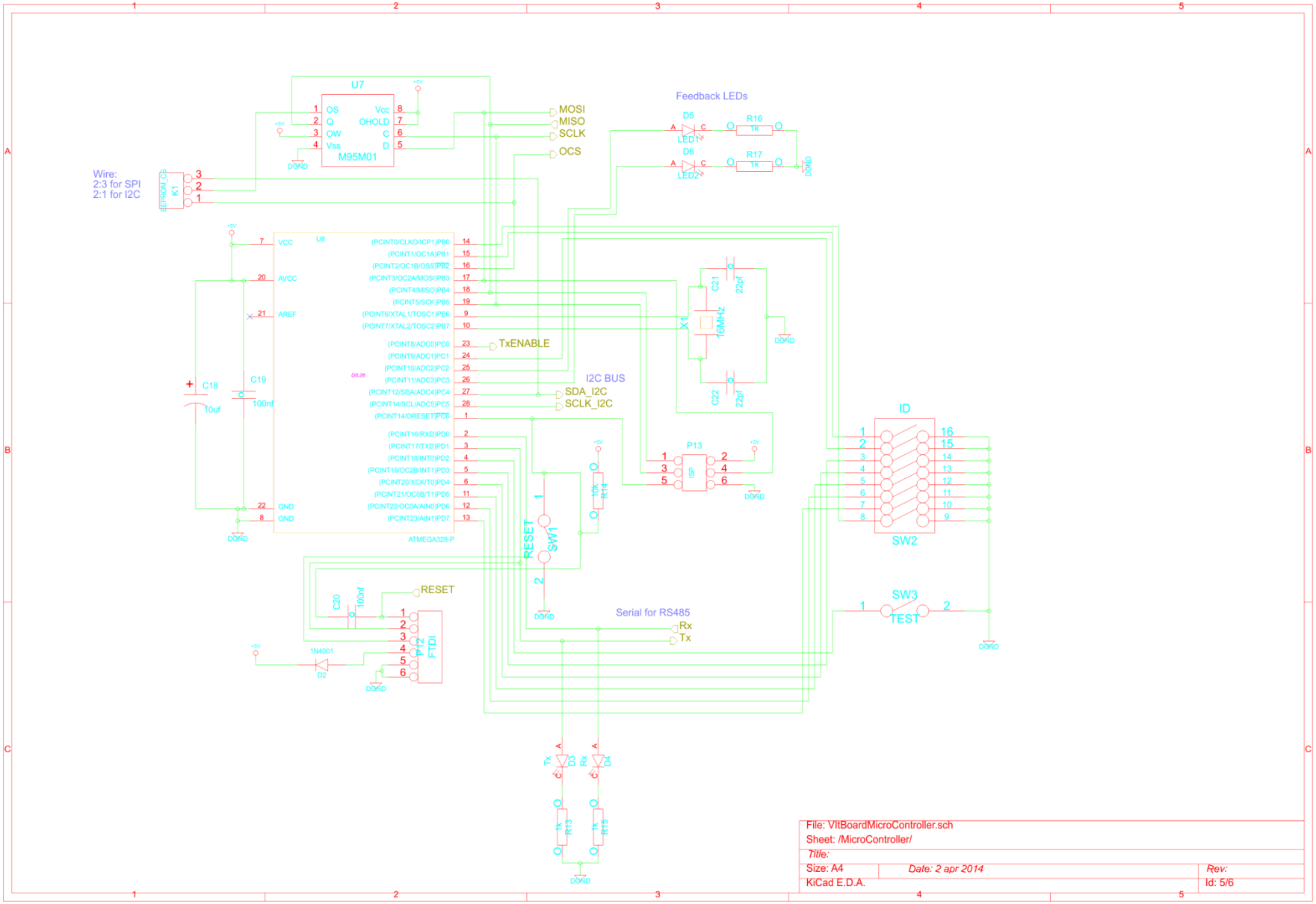
Designed in KiCAD (http://www.kicad-pcb.org/display/KICAD/KiCad+EDA+Software+Suite). Full designs available to download.

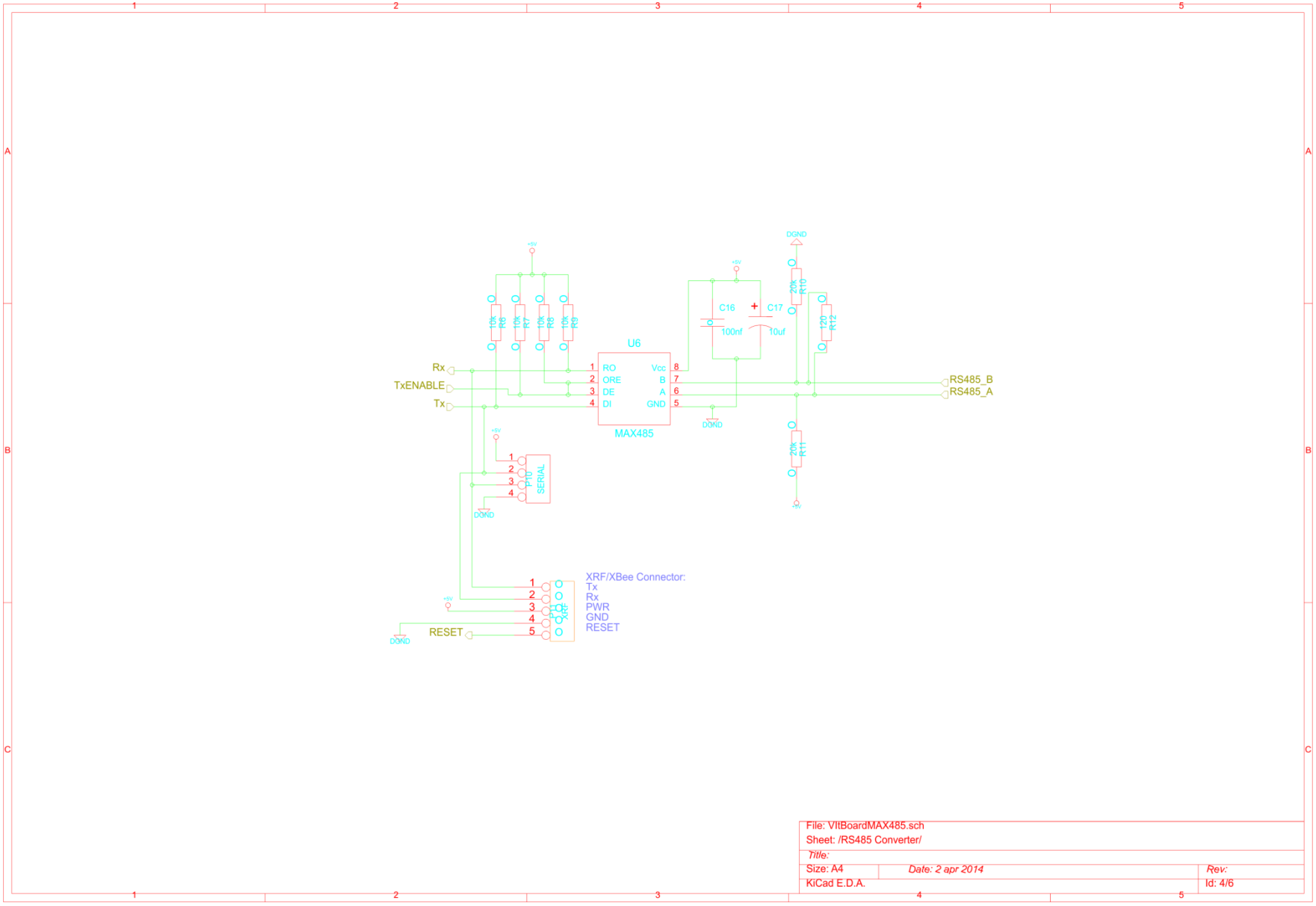

Vlt_Board_OPAMP
-15V
AGND
+15V
+VREF
Therm
Volt1
Volt2
I_out
VltBoardOPAMP.sch
Vlt_Board_ADCI2C
Vin0
Vin1
Vin2
Vin3
SCLK
SDA
DGND
AGND
VltBoardADCI2C.sch
MicroController
OCS
MOSI
MISO
SCLK
SCLK_I2C
SDA_I2C
TxENABLE
Tx
Rx
RESET
VltBoardMicroController.sch
RS485 Converter
TxENABLE
Tx
Rx
RESET
RS485_B
RS485_A
VltBoardMAX485.sch
PowerSupplyUnit
+15V
AGND
-15V
+2V5
DGND
Vin
GND
VltBoardPSU.sch
P8
RS485_IN
P9
RS485_OUT
P7
P_EXT
P5
P6
Product Health
File: Vlt_Monitoring_Board_P2_I2CADS1015_17_2_14.sch
Sheet: /
Title: Vlt Monitoring Slave Unit
Size: A4
Date: 11 mar 2014
Rev: 1
KiCad E.D.A.
Id: 1/6

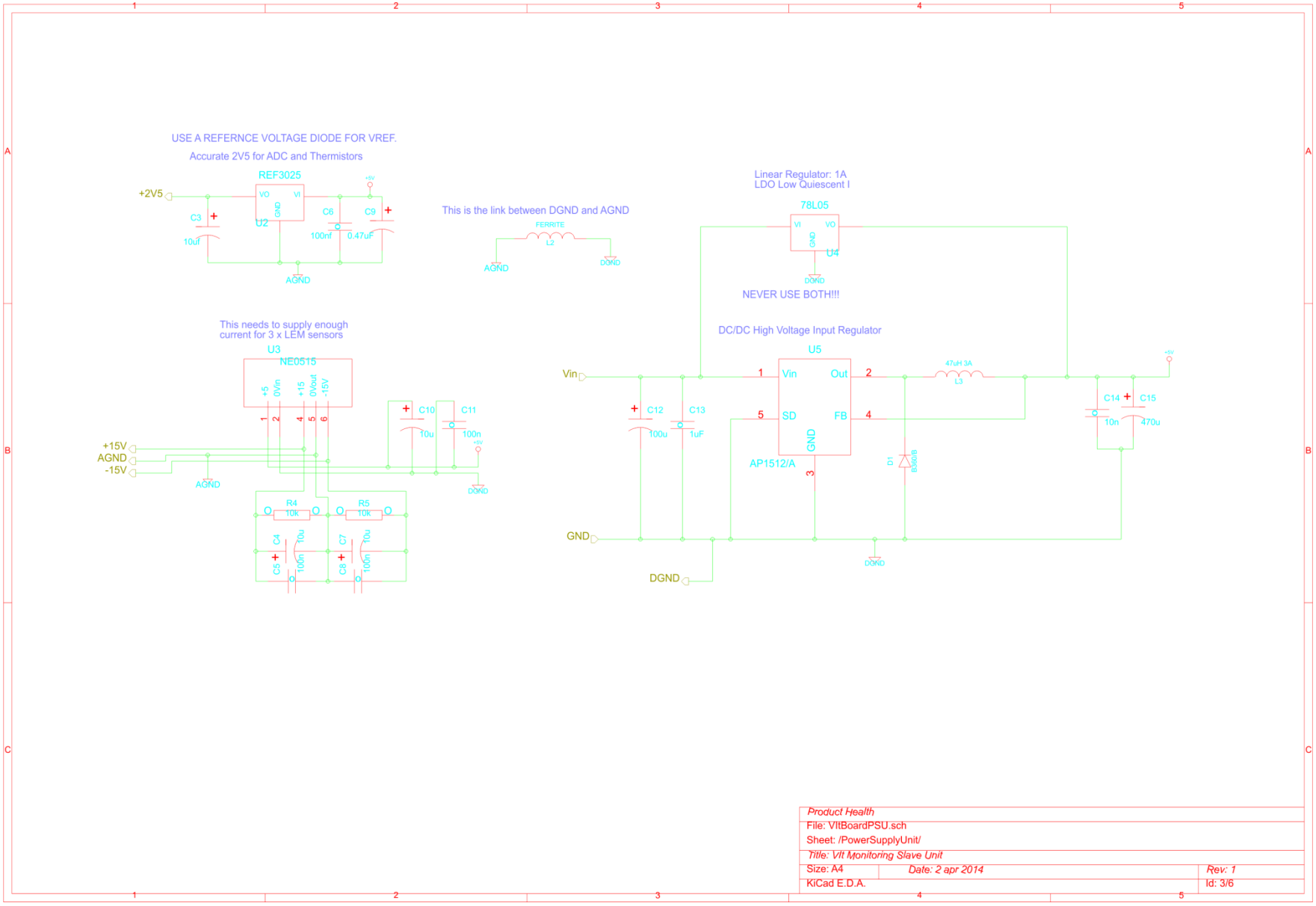

+VREF
10k
R22
THERM1
P16
Therm
R23
AGND
C25
100nf
U9
OutA
InA-
InA+
V+
InB+
InB-
OutB
OutD
InD-
InD+
V-
InA+
InA-
OutC
OPA4340
100k
R24
100k
R26
R29
100k
100k
R25
100k
R27
100k
R28
100k
R30
Therm
Volt1
Volt2
I_out
D9
5V1
AGND
-15V
+15V
AGND
CURRENT1
P17
-15V
0V
+15V
+5V
0V
DATA
Rload
R31
THRU-HOLE
VOLTS1
P14
RV2
R18
RV1
R19
D7
5V1
C23
C
AGND
THRU-HOLE
VOLTS2
P15
RV2
R20
RV1
R21
D8
5V1
C24
C
AGND
Product Health
File: VItBoardOPAMP.sch
Sheet: /Vlt_Board_OPAMP/
Title: Vlt Monitoring Slave Unit
Size: A4
Date: 11 mar 2014
Rev: 1
KiCad E.D.A.
Id: 6/6



Product Health		
File: VItBoardADCI2C.sch		
Sheet: /VIt_Board_ADCI2C/		
Title: VIt Monitoring Slave Unit		
Size: A4	*Date: 11 mar 2014*	*Rev: 1*
KiCad E.D.A.		Id: 2/6


U7
M95M01
MOSI
MISO
SCLK
OCS
Feedback LEDs
D5
LED1
R16
1k
D6
LED2
R17
1k
Wire:
2:3 for SPI
2:1 for I2C
EEPROM_CS
K1
U8
ATMEGA328-P
TxENABLE
I2C BUS
SDA_I2C
SCLK_I2C
C18
10uf
C19
100nf
C21
22pf
C22
22pf
X1
16MHz
P13
ISP
ID
SW2
RESET
SW1
R14
10k
C20
100nf
P12
FTDI
1N4001
D2
Serial for RS485
Rx
Tx
SW3
TEST
D3
D4
R13
1k
R15
1k
File: VltBoardMicroController.sch
Sheet: /MicroController/
Title:
Size: A4
Date: 2 apr 2014
Rev:
KiCad E.D.A.
Id: 5/6


Rx
TxENABLE
Tx
RESET
U6
MAX485
RO
ORE
DE
DI
Vcc
B
A
GND
C16
100nf
C17
10uf
R6
R7
R8
R9
R10
R11
R12
10k
20k
120
P10
SERIAL
P11
XRF
XRF/XBee Connector:
Tx
Rx
PWR
GND
RESET
RS485_B
RS485_A
+5V
DGND
File: VltBoardMAX485.sch
Sheet: /RS485 Converter/
Title:
Size: A4
Date: 2 apr 2014
Rev:
KiCad E.D.A.
Id: 4/6

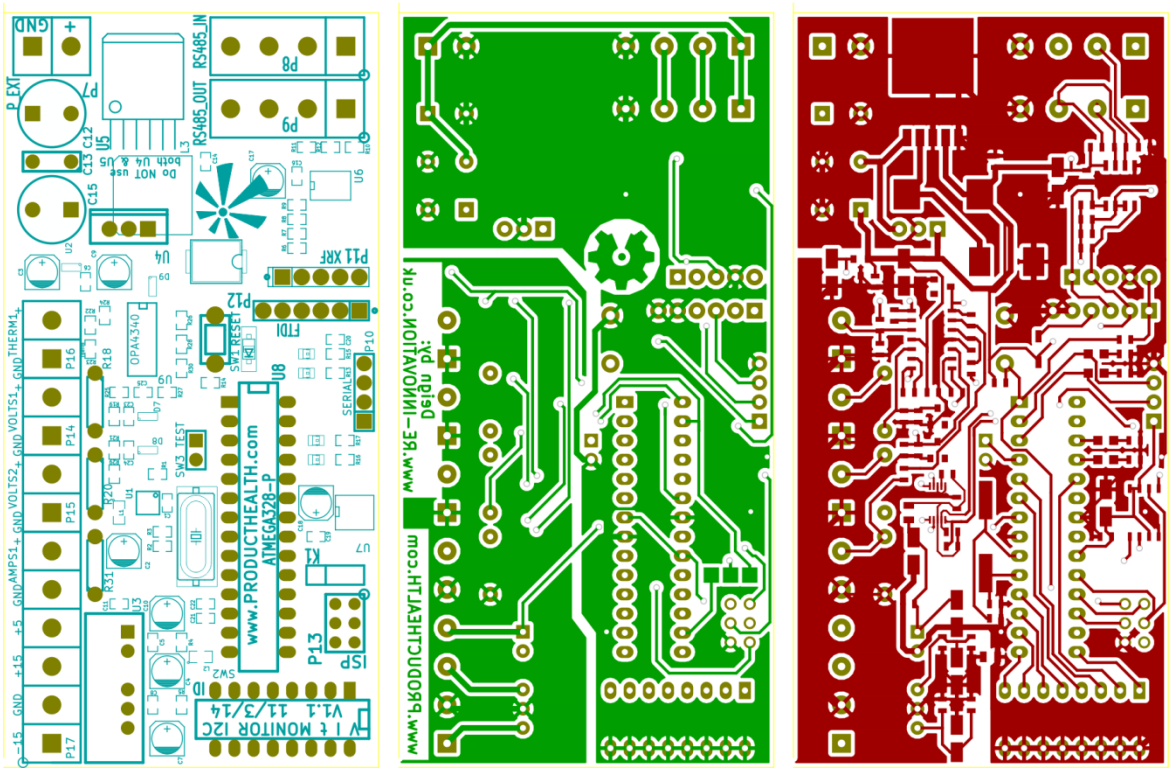


Product Health		
File: VItBoardPSU.sch		
Sheet: /PowerSupplyUnit/		
Title: VIt Monitoring Slave Unit		
Size: A4	*Date: 2 apr 2014*	*Rev: 1*
KiCad E.D.A.		Id: 3/6

PCB for the 1 Channel Slave Unit

Designed in KiCAD (http://www.kicad-pcb.org/display/KICAD/KiCad+EDA+Software+Suite). Full designs available to download.

Showing the silk screen, the bottom copper and the top copper.

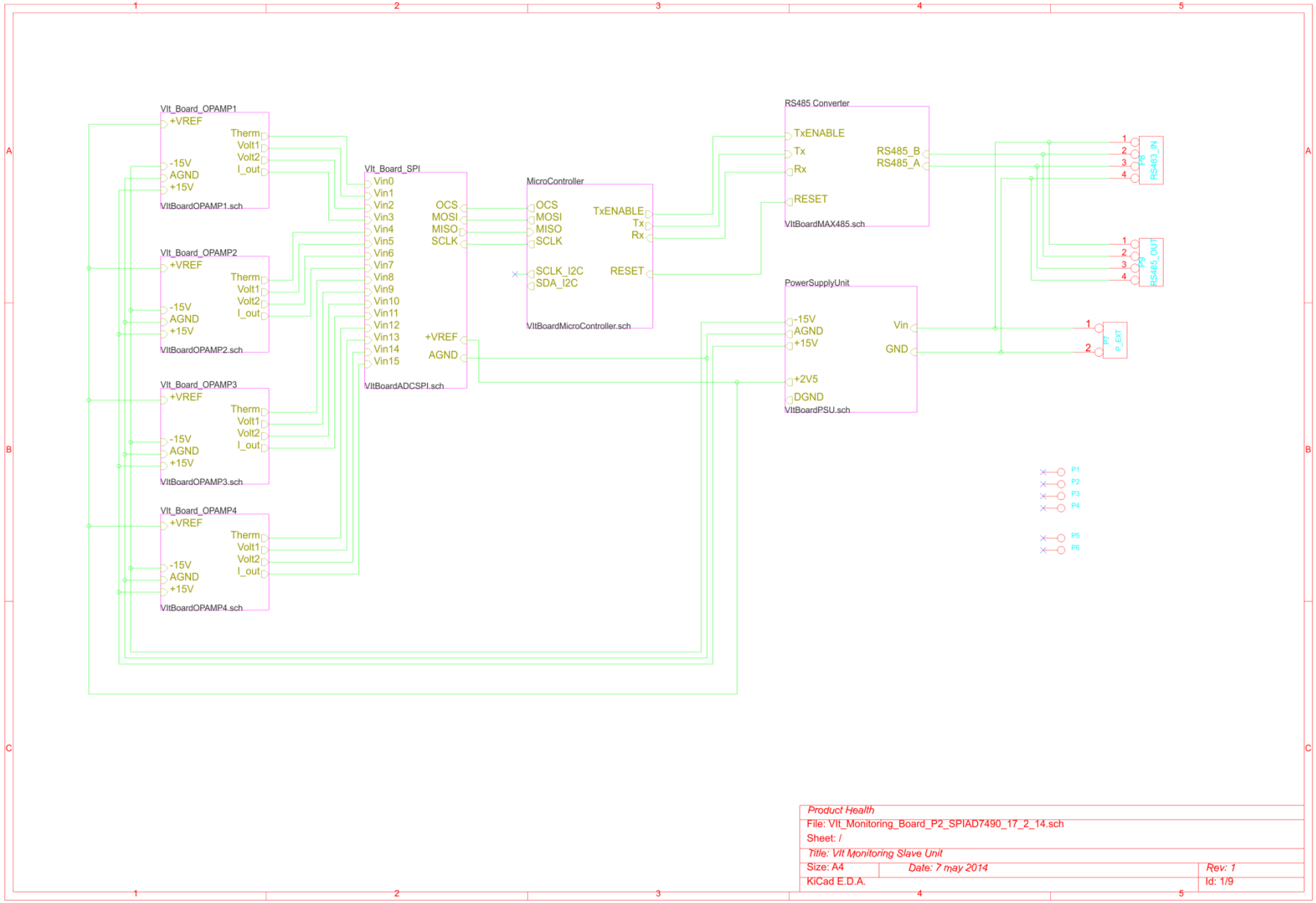


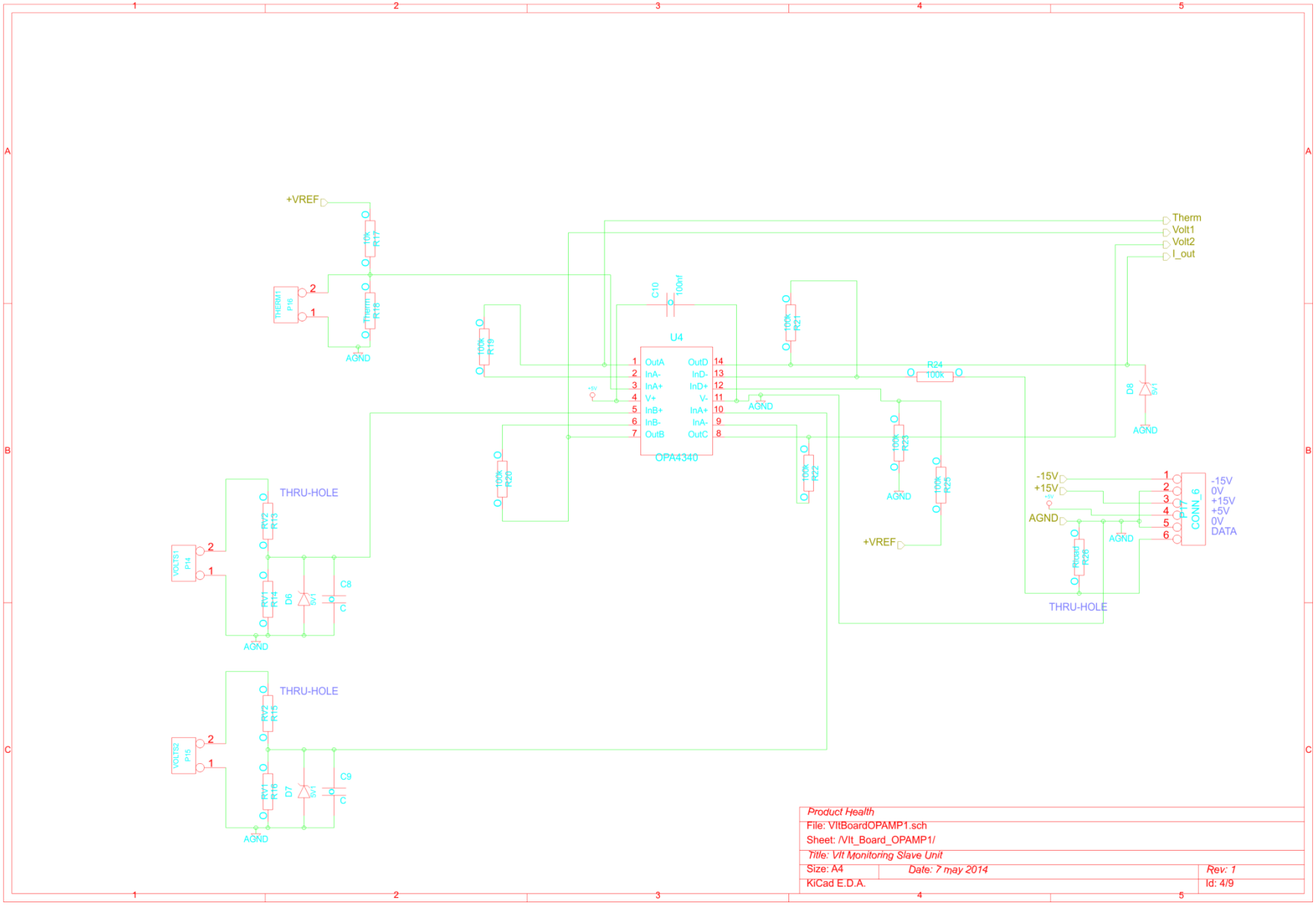
Parts List for 1 Channel Slave Unit

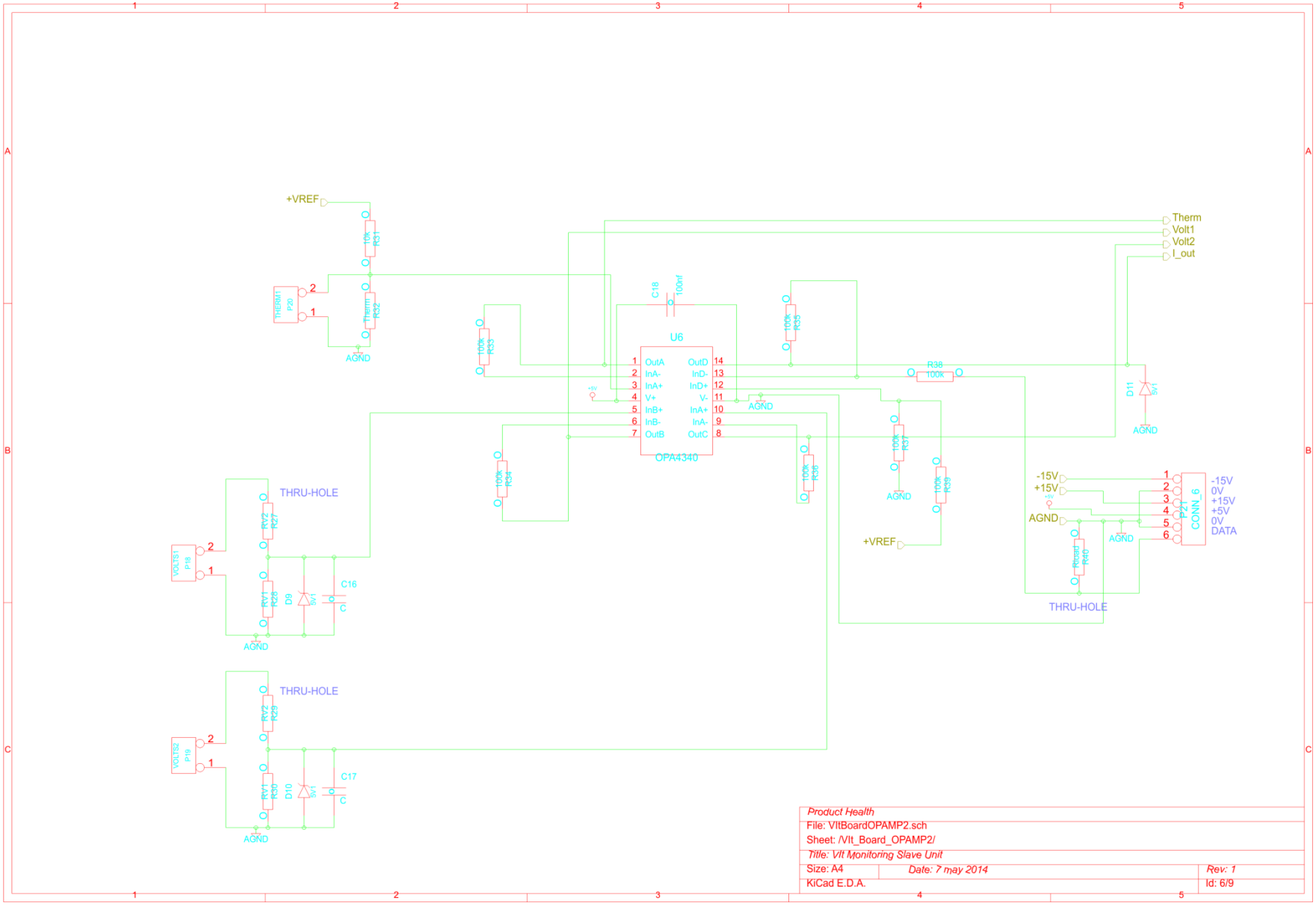
C8	100n		P12	FTDI		SW2	ID
C9	0.47uF		P13	ISP		SW3	TEST
C10	10u		P14	VOLTS1		U1	ADS1015
C11	100n		P15	VOLTS2		U2	REF3025
C12	100u		P16	THERM1		U3	NE0515
C13	1uF		P17	CURRENT1		U4	78L05
C14	10n		R1	10k		U5	AP1512/A
C15	470u		R2	10k		U6	MAX485
C16	100nf		R3	10k		U7	M95M01
C17	10uf		R4	10k		U8	ATMEGA328-P
C18	10uf		R5	10k		U9	AD8604
C19	100nf		R6	10k		X1	16MHz
C20	100nf		R7	10k			
C21	22pf		R8	10k			
C22	22pf		R9	10k			
C23	C		R10	20k			
C24	C		R11	20k			
C25	100nf		R12	120			
D1	B360/B		R13	1k			
D2	1N4001		R14	10k			
D3	Tx		R15	1k			
D4	Rx		R16	1k			
D5	LED1		R17	1k			
D6	LED2		R18	RV2			
D7	5V1		R19	RV1			
D8	5V1		R20	RV2			
D9	5V1		R21	RV1			
K1	EEPROM_CS		R22	10k			
L1	FERRITE		R23	Therm			
L2	FERRITE		R24	100k			
L3	47uH 3A		R25	100k			

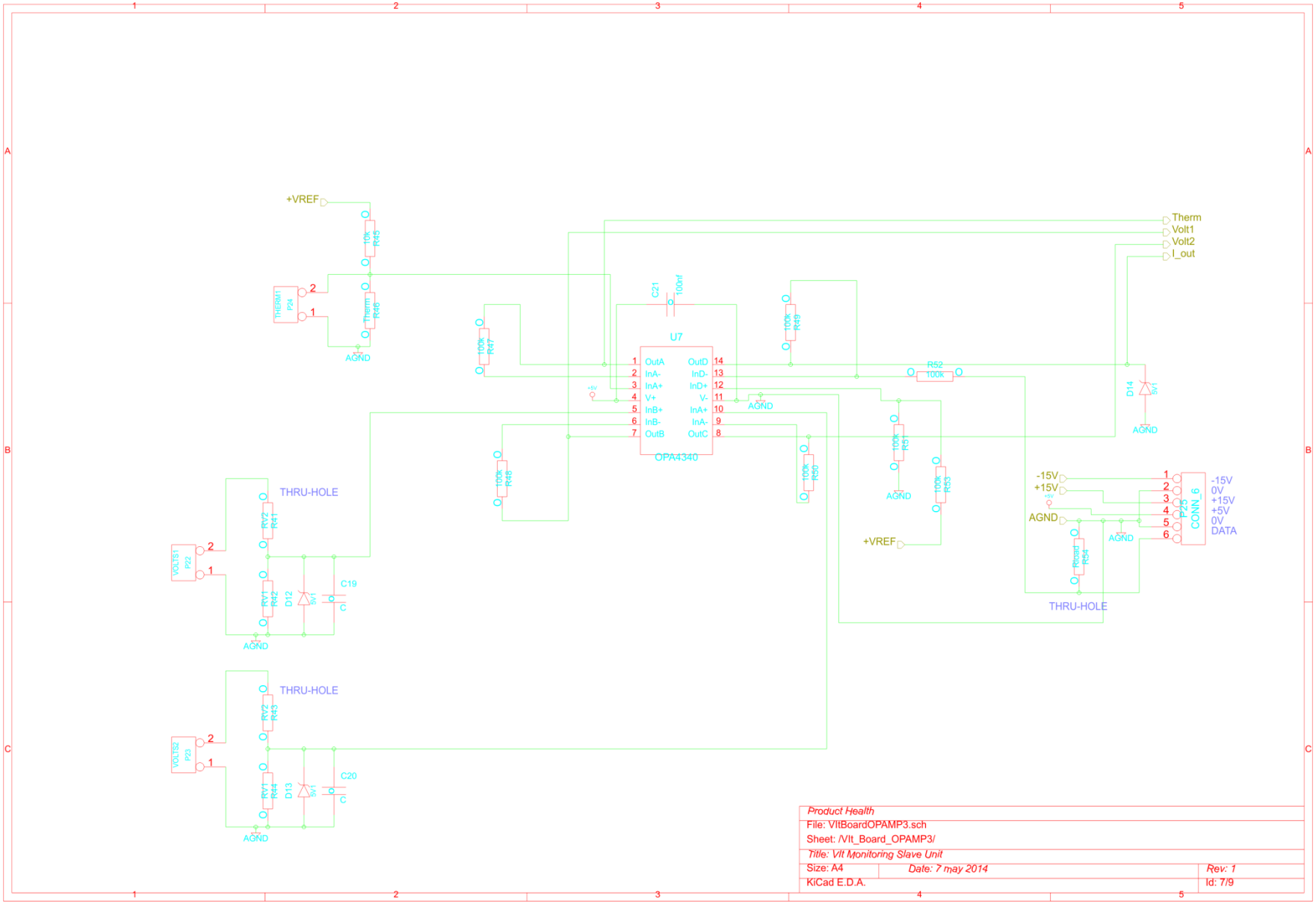
Schematic diagram for 4 Channel Slave Unit

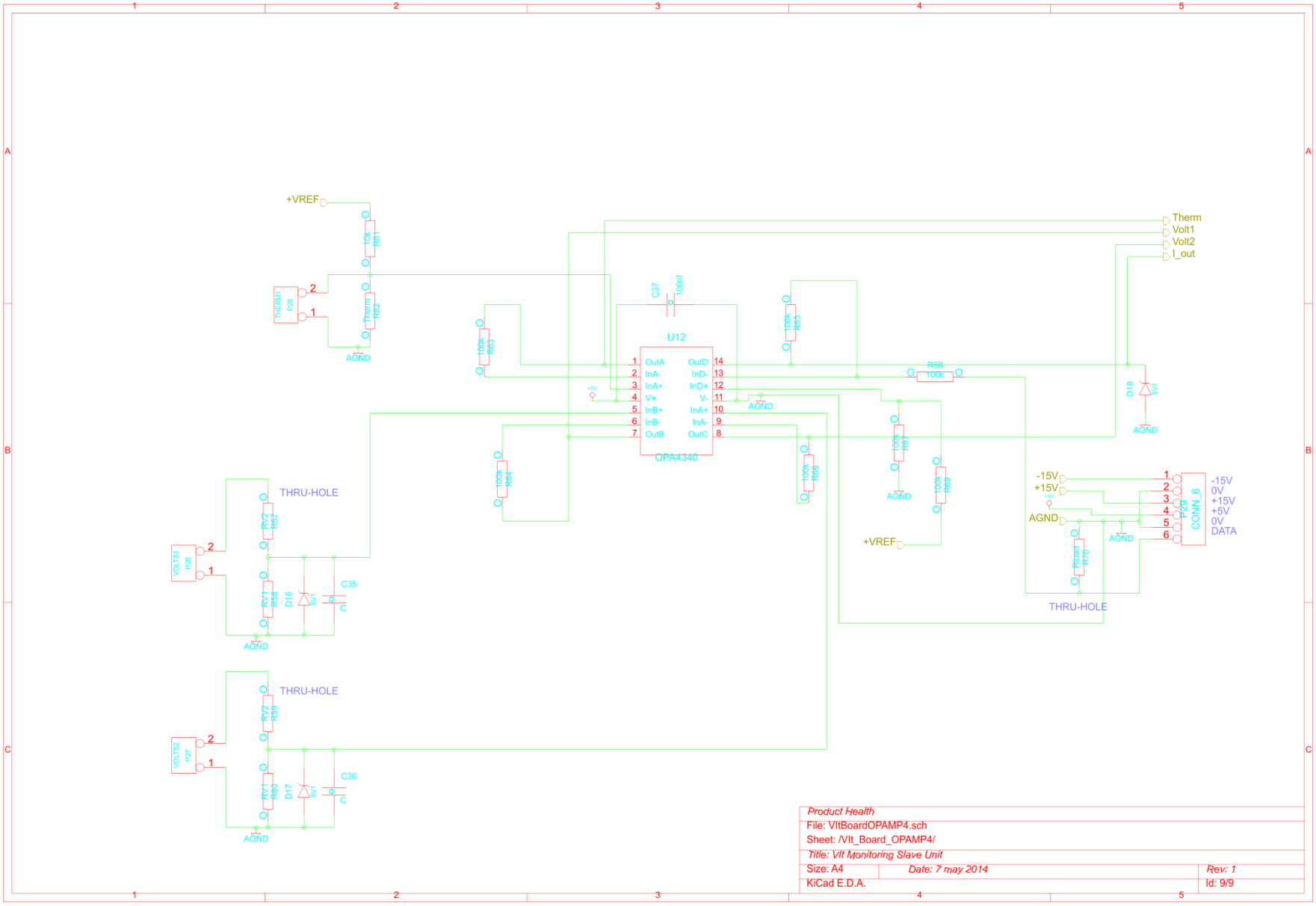
Designed in KiCAD (http://www.kicad-pcb.org/display/KICAD/KiCad+EDA+Software+Suite). Full designs available to download.

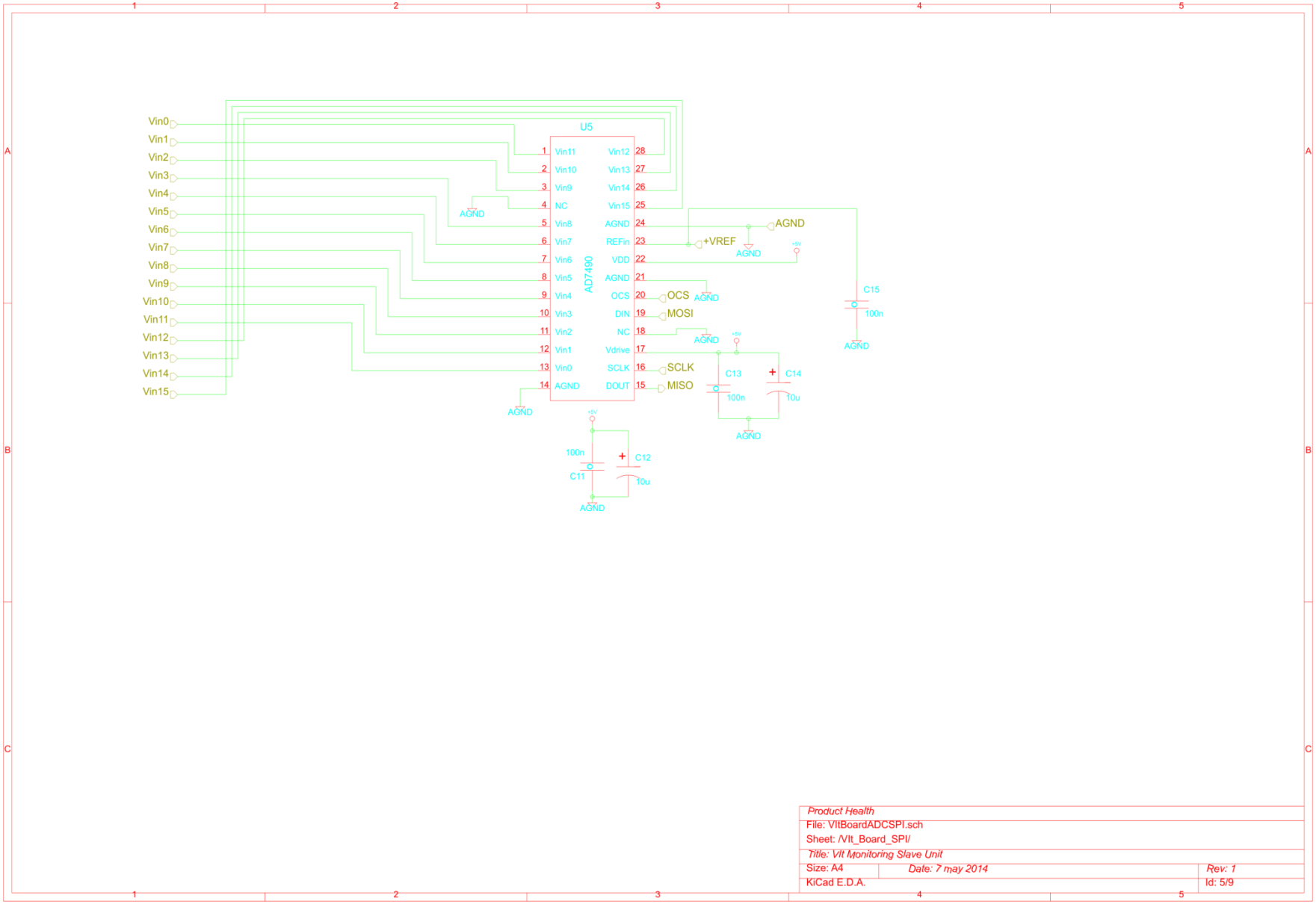

Vlt_Board_OPAMP1
+VREF
-15V
AGND
+15V
Therm
Volt1
Volt2
I_out
VltBoardOPAMP1.sch
Vlt_Board_OPAMP2
VltBoardOPAMP2.sch
Vlt_Board_OPAMP3
VltBoardOPAMP3.sch
Vlt_Board_OPAMP4
VltBoardOPAMP4.sch
Vlt_Board_SPI
Vin0
Vin1
Vin2
Vin3
Vin4
Vin5
Vin6
Vin7
Vin8
Vin9
Vin10
Vin11
Vin12
Vin13
Vin14
Vin15
OCS
MOSI
MISO
SCLK
+VREF
AGND
VltBoardADCSPI.sch
MicroController
SCLK_I2C
SDA_I2C
TxENABLE
Tx
Rx
RESET
VltBoardMicroController.sch
RS485 Converter
RS485_B
RS485_A
VltBoardMAX485.sch
PowerSupplyUnit
Vin
GND
+2V5
DGND
VltBoardPSU.sch
P1
P2
P3
P4
P5
P6
P7
P_EXT
RS485_IN
RS485_OUT
Product Health
File: Vlt_Monitoring_Board_P2_SPIAD7490_17_2_14.sch
Sheet: /
Title: Vlt Monitoring Slave Unit
Size: A4
Date: 7 may 2014
Rev: 1
KiCad E.D.A.
Id: 1/9

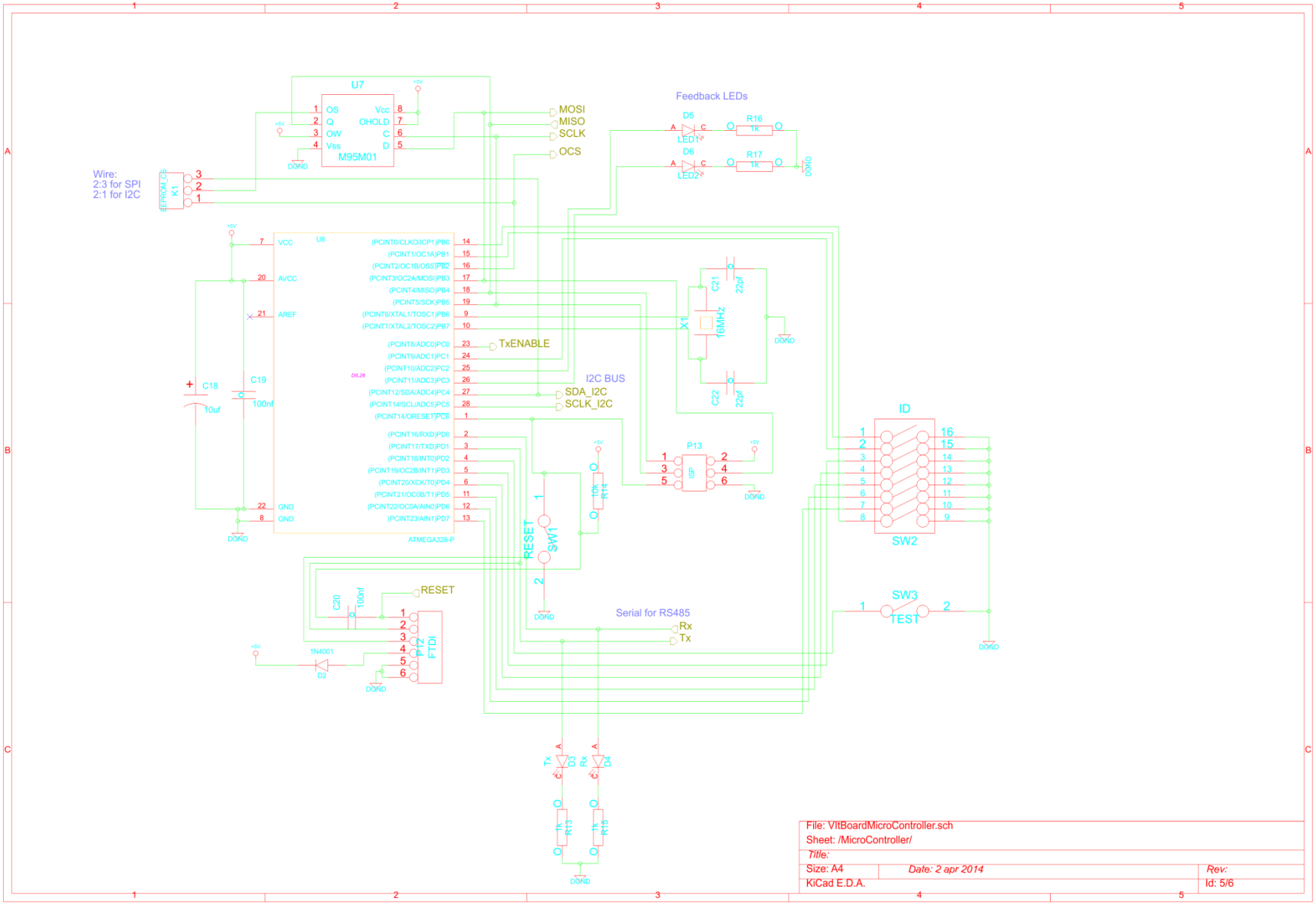

+VREF
10k
R17
THERM1
P16
Therm
R18
AGND
C10
100nf
U4
OutA
InA-
InA+
V+
InB+
InB-
OutB
OutD
InD-
InD+
V-
InA+
InA-
OutC
OPA4340
100k
R19
100k
R21
R24
100k
100k
R20
100k
R22
100k
R23
100k
R25
+5V
AGND
Therm
Volt1
Volt2
I_out
D8
5V1
-15V
+15V
AGND
Rload
R26
CONN_6
P17
-15V
0V
+15V
+5V
0V
DATA
THRU-HOLE
VOLTS1
P14
RV2
R13
RV1
R14
D6
5V1
C8
C
THRU-HOLE
VOLTS2
P15
RV2
R15
RV1
R16
D7
5V1
C9
C
Product Health
File: VItBoardOPAMP1.sch
Sheet: /VIt_Board_OPAMP1/
Title: VIt Monitoring Slave Unit
Size: A4
Date: 7 may 2014
Rev: 1
KiCad E.D.A.
Id: 4/9

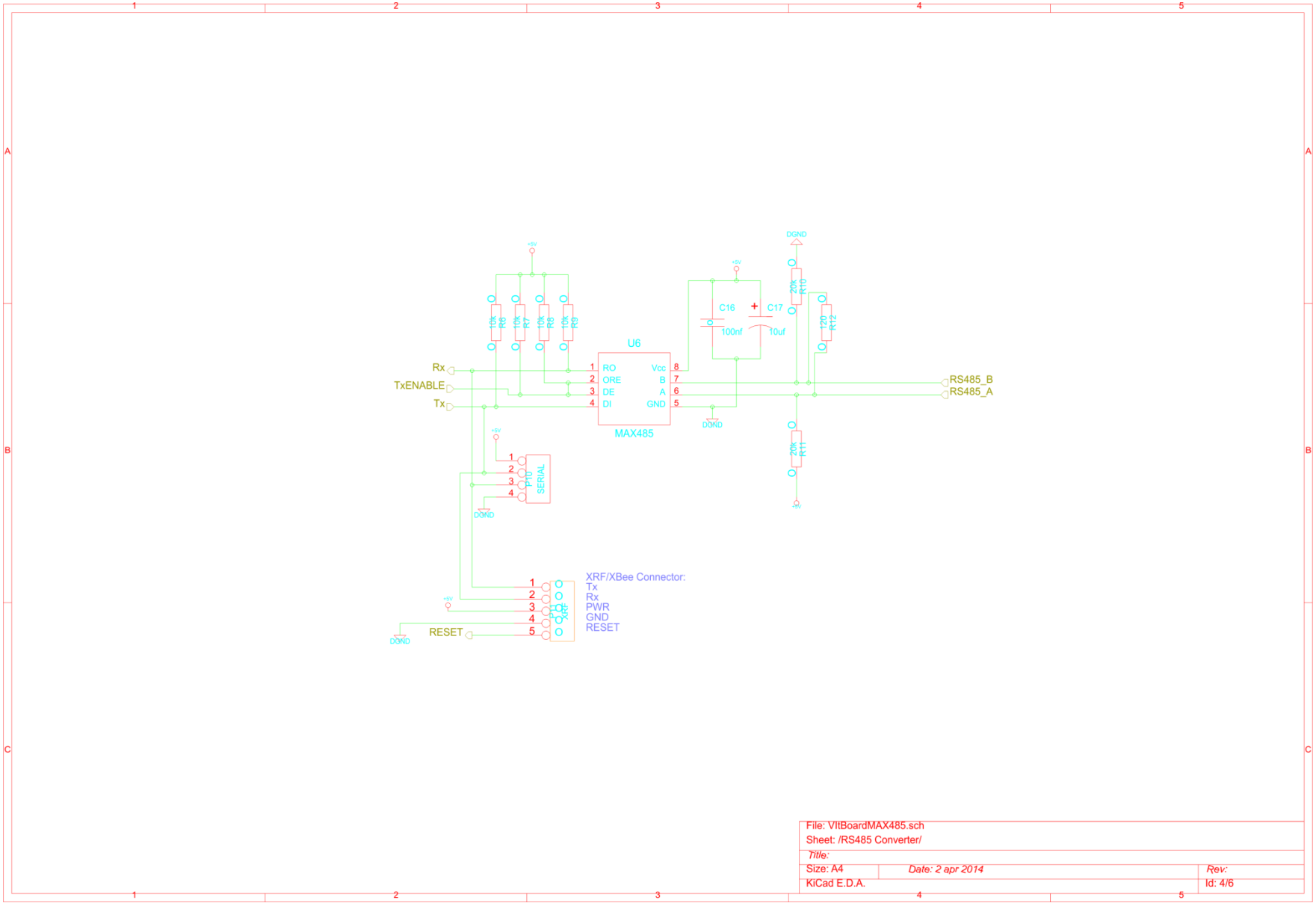

+VREF
10k
R31
THERM1
P20
Therm
R32
AGND
C18
100nf
U6
OPA4340
100k
R33
100k
R34
100k
R35
100k
R36
R38
100k
100k
R37
100k
R39
AGND
+VREF
Therm
Volt1
Volt2
I_out
D11
5V1
AGND
-15V
+15V
AGND
AGND
Rload
R40
CONN_6
P21
-15V
0V
+15V
+5V
0V
DATA
THRU-HOLE
VOLTS1
P18
RV2
R27
RV1
R25
D9
5V1
C16
C
AGND
THRU-HOLE
VOLTS2
P19
RV2
R29
RV1
R30
D10
5V1
C17
C
AGND
THRU-HOLE
Product Health
File: VItBoardOPAMP2.sch
Sheet: /Vlt_Board_OPAMP2/
Title: Vlt Monitoring Slave Unit
Size: A4
Date: 7 may 2014
Rev: 1
KiCad E.D.A.
Id: 6/9

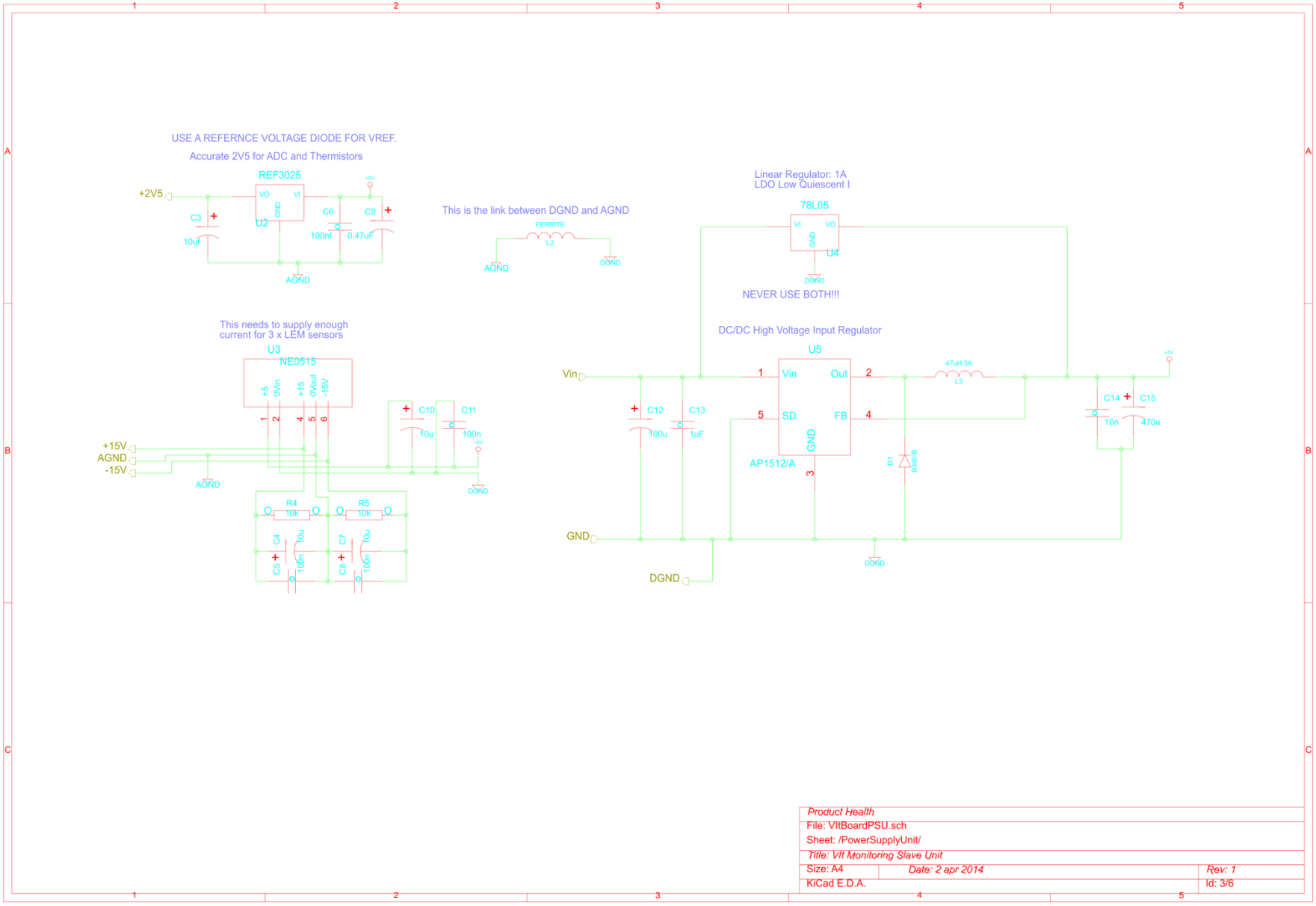

+VREF
10k
R45
THERM1
P24
Therm
R46
AGND
C21
100nf
U7
OPA4340
100k
R47
100k
R49
R52
100k
Therm
Volt1
Volt2
I_out
D14
5V1
AGND
100k
R48
100k
R50
100k
R51
100k
R53
+VREF
THRU-HOLE
RV2
R41
VOLTS1
P22
RV1
R42
D12
5V1
C19
C
AGND
THRU-HOLE
RV2
R43
VOLTS2
P23
RV1
R44
D13
5V1
C20
C
AGND
-15V
+15V
AGND
Rload
R54
AGND
P25
CONN_6
-15V
0V
+15V
+5V
0V
DATA
THRU-HOLE
Product Health
File: VItBoardOPAMP3.sch
Sheet: /VIt_Board_OPAMP3/
Title: VIt Monitoring Slave Unit
Size: A4
Date: 7 may 2014
Rev: 1
KiCad E.D.A.
Id: 7/9


+VREF
10k
R61
THERM1
P28
Therm
R62
AGND
C37
100nf
U12
OutA
InA-
InA+
V+
InB+
InB-
OutB
OutD
InD-
InD+
V-
InA+
InA-
OutC
OPA4340
100k
R63
100k
R64
100k
R65
100k
R66
R68
100k
100k
R67
100k
R69
+5V
AGND
Therm
Volt1
Volt2
I_out
D18
5V1
AGND
-15V
+15V
+5V
AGND
CONN_6
P29
-15V
0V
+15V
+5V
0V
DATA
AGND
Rload
R70
THRU-HOLE
+VREF
THRU-HOLE
VOLTS1
P26
RV2
R57
RV1
R58
D16
5V1
C35
C
AGND
THRU-HOLE
VOLTS2
P27
RV2
R59
RV1
R60
D17
5V1
C36
C
AGND
Product Health
File: VItBoardOPAMP4.sch
Sheet: /VIt_Board_OPAMP4/
Title: VIt Monitoring Slave Unit
Size: A4
Date: 7 may 2014
Rev: 1
KiCad E.D.A.
Id: 9/9


Vin0
Vin1
Vin2
Vin3
Vin4
Vin5
Vin6
Vin7
Vin8
Vin9
Vin10
Vin11
Vin12
Vin13
Vin14
Vin15
U5
AD7490
Vin11
Vin10
Vin9
NC
Vin8
Vin7
Vin6
Vin5
Vin4
Vin3
Vin2
Vin1
Vin0
AGND
Vin12
Vin13
Vin14
Vin15
AGND
REFin
VDD
AGND
OCS
DIN
NC
Vdrive
SCLK
DOUT
+VREF
OCS
MOSI
SCLK
MISO
C15
100n
C13
100n
C14
10u
C12
10u
C11
100n
+5V
Product Health
File: VItBoardADCSPI.sch
Sheet: /VIt_Board_SPI/
Title: VIt Monitoring Slave Unit
Size: A4
Date: 7 may 2014
Rev: 1
KiCad E.D.A.
Id: 5/9


U7
M95M01
MOSI
MISO
SCLK
OCS
Feedback LEDs
D5
LED1
R16
1k
D6
LED2
R17
1k
Wire:
2:3 for SPI
2:1 for I2C
EEPROM_CS
K1
U8
ATMEGA328-P
TxENABLE
I2C BUS
SDA_I2C
SCLK_I2C
C18
10uf
C19
100nf
C21
22pf
C22
22pf
X1
16MHz
P13
ISP
ID
SW2
RESET
SW1
R14
10k
C20
100nf
P12
FTDI
1N4001
D2
Serial for RS485
Rx
Tx
SW3
TEST
D3
D4
R13
1k
R15
1k
File: VltBoardMicroController.sch
Sheet: /MicroController/
Title:
Size: A4
Date: 2 apr 2014
Rev:
KiCad E.D.A.
Id: 5/6


Rx
TxENABLE
Tx
RESET
U6
MAX485
RO
ORE
DE
DI
Vcc
B
A
GND
C16
100nf
C17
10uf
R6
R7
R8
R9
R10
R11
R12
10k
20k
120
RS485_B
RS485_A
P10
SERIAL
P11
XRF/XBee Connector:
Tx
Rx
PWR
GND
RESET
DGND
+5V
File: VltBoardMAX485.sch
Sheet: /RS485 Converter/
Title:
Size: A4
Date: 2 apr 2014
Rev:
KiCad E.D.A.
Id: 4/6

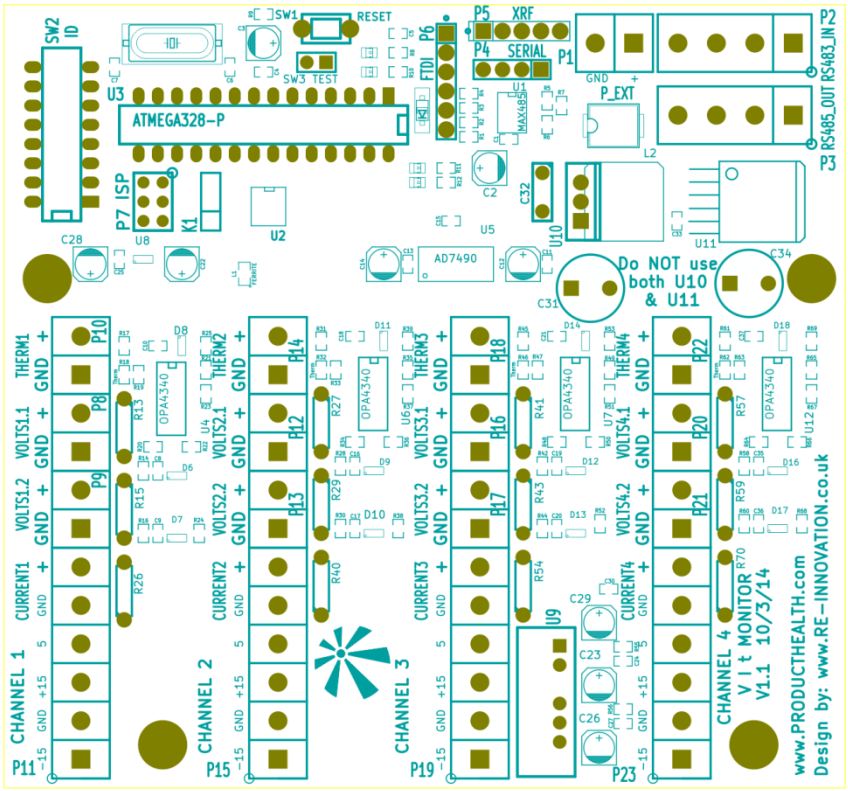


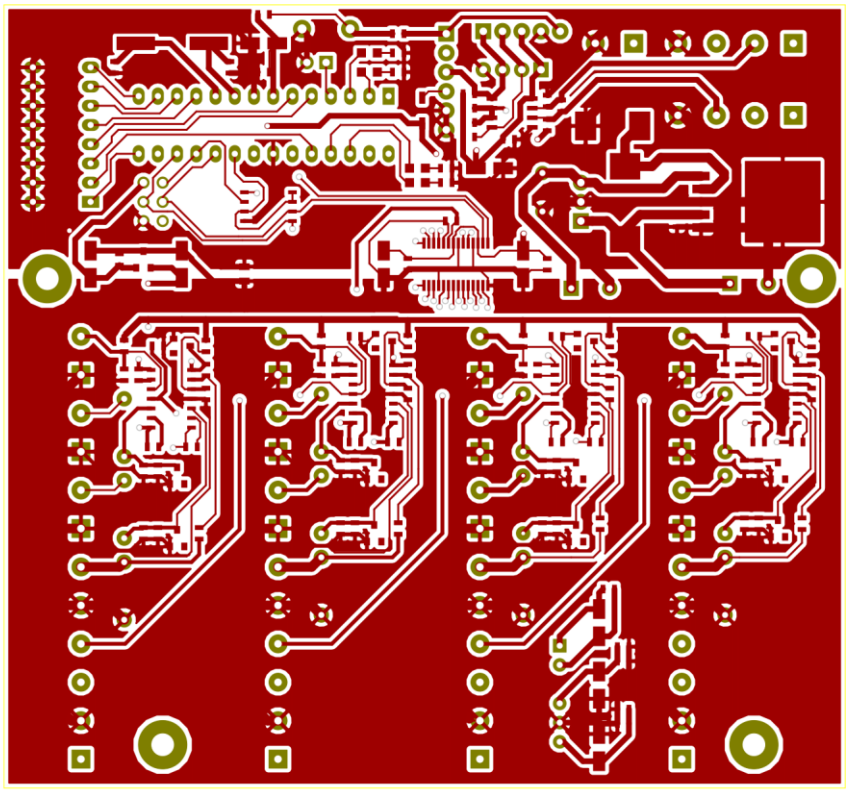
Product Health		
File: VItBoardPSU.sch		
Sheet: /PowerSupplyUnit/		
Title: VIt Monitoring Slave Unit		
Size: A4	*Date: 2 apr 2014*	*Rev: 1*
KiCad E.D.A.		Id: 3/6

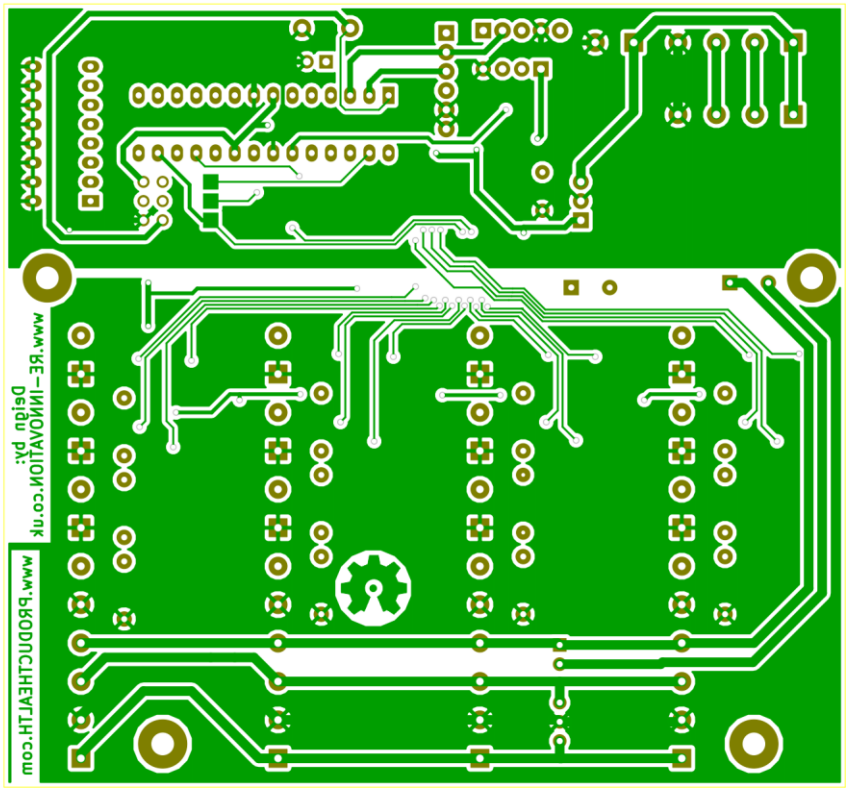
PCB layout for 4 Channel Slave Unit

Designed in KiCAD (http://www.kicad-pcb.org/display/KICAD/KiCad+EDA+Software+Suite). Full designs available to download.

Showing the silk screen, the bottom copper and the top copper.






www.RE-INNOVATION.co.uk
Design by:
www.PRODUCTENGLISH.com

Parts List for 4 Channel Slave Unit

Ref	Value		Ref	Value		Ref	Value		Ref	Value
C1	100nf		D8	5V1		R2	10k		R46	Therm
C2	10uf		D9	5V1		R3	10k		R47	100k
C3	10uf		D10	5V1		R4	10k		R48	100k
C4	100nf		D11	5V1		R5	20k		R49	100k
C5	100nf		D12	5V1		R6	20k		R50	100k
C6	22pf		D13	5V1		R7	120		R51	100k
C7	22pf		D14	5V1		R8	1k		R52	100k
C8	C		D15	B360/B		R9	10k		R53	100k
C9	C		D16	5V1		R10	1k		R54	Rload
C10	100nf		D17	5V1		R11	1k		R55	10k
C11	100n		D18	5V1		R12	1k		R56	10k
C12	10u		K1	EEPROM_CS		R13	RV2		R57	RV2
C13	100n		L1	FERRITE		R14	RV1		R58	RV1
C14	10u		L2	47uH 3A		R15	RV2		R59	RV2
C15	100n		P1	CONN_1		R16	RV1		R60	RV1
C16	C		P2	CONN_1		R17	10k		R61	10k
C17	C		P3	CONN_1		R18	Therm		R62	Therm
C18	100nf		P4	CONN_1		R19	100k		R63	100k
C19	C		P5	CONN_1		R20	100k		R64	100k
C20	C		P6	CONN_1		R21	100k		R65	100k
C21	100nf		P7	P_EXT		R22	100k		R66	100k
C22	10uf		P8	RS483_IN		R23	100k		R67	100k
C23	10u		P9	RS485_OUT		R24	100k		R68	100k
C24	100n		P10	SERIAL		R25	100k		R69	100k
C25	100nf		P11	XRF		R26	Rload		R70	Rload
C26	10u		P12	FTDI		R27	RV2		SW1	RESET
C27	100n		P13	ISP		R28	RV1		SW2	ID
C28	0.47uF		P14	VOLTS1		R29	RV2		SW3	TEST
C29	10u		P15	VOLTS2		R30	RV1		U1	MAX485
C30	100n		P16	THERM1		R31	10k		U2	M95M01
C31	100u		P17	CONN_6		R32	Therm		U3	ATMEGA328-P
C32	1uF		P18	VOLTS1		R33	100k		U4	AD8604
C33	10n		P19	VOLTS2		R34	100k		U5	AD7490
C34	470u		P20	THERM1		R35	100k		U6	AD8604
C35	C		P21	CONN_6		R36	100k		U7	AD8604
C36	C		P22	VOLTS1		R37	100k		U8	REF3025
C37	100nf		P23	VOLTS2		R38	100k		U9	NE0515
D1	1N4001		P24	THERM1		R39	100k		U10	78L05
D2	Tx		P25	CONN_6		R40	Rload		U11	AP1512/A
D3	Rx		P26	VOLTS1		R41	RV2		U12	AD8604
D4	LED1		P27	VOLTS2		R42	RV1		X1	16MHz
D5	LED2		P28	THERM1		R43	RV2			
D6	5V1		P29	CONN_6		R44	RV1			
D7	5V1		R1	10k		R45	10k			

Slave unit - Software

The software must be robust and simple but perform a number of tasks:

- Measure all the analogue channels (via the I2C ADC)
- Keep a running average of that data (with an adjustable sample period)
- Respond to MODBUS requests for data

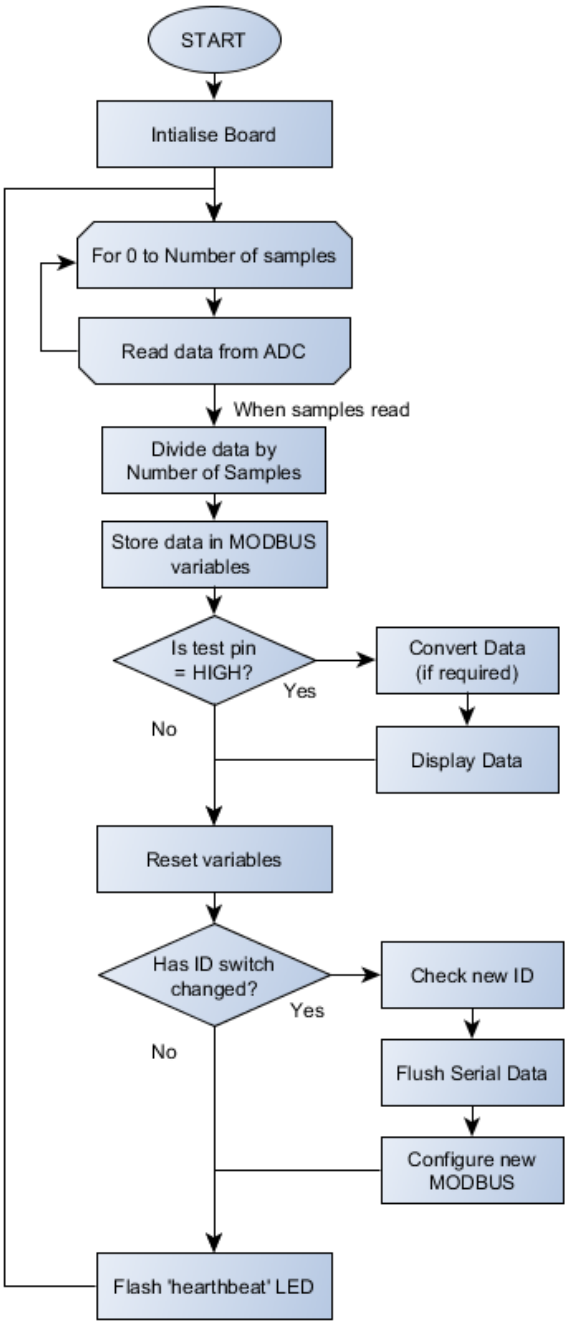
It could also perform some conversion, although this requires the conversion factors to be hard-programmed onto the slave unit. It is best that these conversions are performed on the master unit, but having a conversion

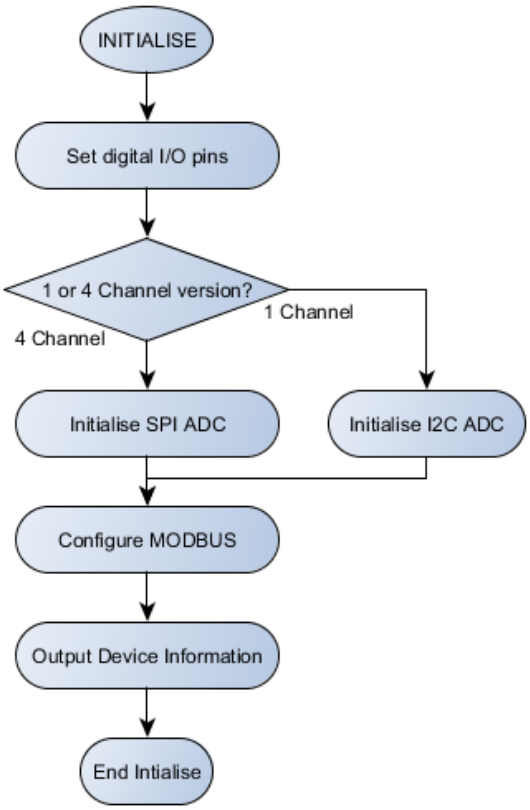
- Convert temperature readings into degrees C (this could be done off-unit)
- Convert voltage readings into V (this could be done off-unit)
- Convert current readings into I (this could be done off-unit)

The software for the Arduino is the same for both 1 Channel and 4 Channel versions. A Definition at the beginning of the program is used to change the version.

Software flow diagram

This shows the basic structure of the Arduino software:





Communications Protocol

This unit will use MODBUS (http://en.wikipedia.org/wiki/Modbus) as the communications prototcol.

MODBUS is a standard industrial serial communications protocol. Initially developed for PLCs. It uses the Master/Slave technique, with up to 247 slaves to each master. This VIt unit is a MODBUS slave, responding to the master, which is the Master unit (a Raspberry Pi).

Each slave has an ID from 1 to 247. This will be the first byte the master sends. A slave will only respond when its unique ID is written.

The master will then send a function code. This tells the slave what to do:

Function Code	Action	Table Name
01 (01 hex)	Read	Discrete Output Coils
05 (05 hex)	Write single	Discrete Output Coil
15 (0F hex)	Write multiple	Discrete Output Coils
02 (02 hex)	Read	Discrete Input Contacts
04 (04 hex)	Read	Analog Input Registers
03 (03 hex)	Read	Analog Output Holding Registers
06 (06 hex)	Write single	Analog Output Holding Register
16 (10 hex)	Write multiple	Analog Output Holding Registers

At the end of every MODBUS message there is a Cyclic Redundancy Check (CRC) which are two bytes used for error detection. The CRC value is calculated by performing logical operations on the data which creates a unique 2 byte code, added to the message.

Data is stored within four different tables, which relate to output 'coils' (on/off display units), input 'contacts' (switches/sensors), Analog Input Registers (read-only analog sensors), Analog outputs holding registers (analog output units).

This monitoring board will only be an Analog Input Register, which is a read-only unit (we never need to send analog data from the master to the slave. Hence we only need to partially implement the MODBUS protocol).

Typical example:

> *This command is requesting the content of analog input register # 30009*
> *from the slave device with address 17.*
>
> `11 04 0008 0001 B298`
>
> *11: The Slave Address (17 = 11 hex)*
> *04: The Function Code (read Analog Input Registers)*
> *0008: The Data Address of the first register requested. (30009-30001 = 8)*
> *0001: The total number of registers requested. (read 1 register)*
> *B298: The CRC (cyclic redundancy check) for error checking.*

Response

`11 04 02 000A F8F4`

`11`: The Slave Address (17 = `11` hex)
`04`: The Function Code (read Analog Input Registers)
`02`: The number of data bytes to follow (1 registers x 2 bytes each = 2 bytes)
`000A`: The contents of register 30009
`F8F4`: The CRC (cyclic redundancy check).

Guide available here:

- http://modbus.org/docs/PI_MBUS_300.pdf
- http://jamod.sourceforge.net/kbase/protocol.html
- http://www.simplymodbus.ca/faq.htm

The MODBUS communications protocol is implemented on the microcontroller. A standard library is used to implement MODBUS-compatible communications on the Arduino. There are a number of MODBUS libraries around. This unit uses the simple-modbus library:

- http://code.google.com/p/simple-modbus/

MODBUS specifications

For the 1 channel device the data to store is:

Temperature 1, Voltage 1.1, Voltage 1.2, Current 1

For the 4 channel device the data to store is:

Temperature 1, Voltage 1.1, Voltage 1.2, Current 1
Temperature 2, Voltage 2.1, Voltage 2.2, Current 2
Temperature 3, Voltage 3.1, Voltage 3.2, Current 3
Temperature 4, Voltage 4.1, Voltage 4.2, Current 4

The data will be stored in different registers by the Arduino. Each 'set' of temperature, voltage and current will be individually readable. This can improve data transfer rates if there are fewer parameters being checked.

This will be a 12 bit resolution number but to give expandability we store this within an 'unsigned long int' which is 4 bytes or 16 bit each. Each data value is 16 bits = 2 bytes = 4 hex values.

The data storage registers are:

Register (hex)	Temperature	Voltage 1	Voltage 2	Current	Total Hex
30000	0xXXXX	0xXXXX	0xXXXX	0xXXXX	16 hex
30010	0xXXXX	0xXXXX	0xXXXX	0xXXXX	16 hex
30020	0xXXXX	0xXXXX	0xXXXX	0xXXXX	16 hex
30030	0xXXXX	0xXXXX	0xXXXX	0xXXXX	16 hex

Send the MODBUS 'ask for analogue data' command:

ID 03 0000 0004 ????

In the form device ID, 03: return analog register data, from register 0100, return 4 bytes of data and ???? is the CRC. Data sent for each 'channel' is 16 pieces of 4-bit hex data. So each returned data set is 64 bits or 8 bytes.

Total hex data to return when the MODBUS command "ask for data" is sent are:

ID 03 0100 XXXX XXXX XXXX XXXX ????

(Where ID is the device ID, XXXX is data and ???? is the CRC). Data sent back for each 'channel' is 28 pieces of 4-bit hex data. So each returned data set is 112 bits or 14 bytes.

So to get one channels data we need to exchange 64+112 bits = 176 bits.

We will use 115200 bits/sec data rate. This means we can read 654 channels per second, or one channel 654 times per second.

More info about CRC details here:

http://www.control.com/thread/1026204601

http://www.modbus.pl/node/23

http://www.lammertbies.nl/comm/info/crc-calculation.html

This last link has an on-line CRC calculator. We are using CRC-16 MODBUS and the data is sent with the last two bytes first.

But the CRC is dealt with by the simple-MODBUS code.

An example request for data:

- Return data from channel 1: 02 03 0000 0004 443a
- Return data from channel 2: 02 03 0004 0004 05fb
- Return data from channel 3: 02 03 0008 0004 c5f8

ADC readings

1 Channel Unit

The 1 channel slave unit uses an I2C interface. This has been implemented using an Adafruit Arduino Library (http://www.adafruit.com/products/1083).

The gain is set to x1 and the IC is initialised. Data readings are taken using the ads.readADC_singleEnded() command.

4 Channel Unit

The 4 channel slave unit uses an SPI interface. Data transfer uses the SPI.transfer() commands. The SPI chip select (CS) pin is set low to enable a write to the IC.

Data is transferred in 16 bit transfers. The SPI.transfer() command just sends an 8 bit number, so two transfers must be made.

To initialise the control register must be written to. These are:

ID	WRITE	SEQ	ADD3	ADD2	ADD1	ADD0	PM1	PM0	SHADOW	WEAK	RANGE	CODING
Data	1	0	0	0	0	0	1	1	1	0	0	1

Please see the datasheet for more information about these parameters.

Then the Shadow register must be written to. This is a list of the channels to read in sequence. We want to read all 16 channels, so we must send binary "1111 1111 1111 1111".

Now the SPI ADC is initialised and will return the data in sequence whenever data is clocked into it.

To return data then 16 bits of data are clock in (in this case binary "0000 0000 0000 0000") and, at the same time, the data is clocked out. The data clocked out is in the format:

Channel number (4 bits): Data (12 bits)

This must be formatted and stored, as shown in the Arduino code.

This must be done for all 16 channels.

Thermistor conversion

The thermistor conversion is performed by a library taken from the Arduino website (http://playground.arduino.cc/ComponentLib/Thermistor2).

The Steinhart-Hart Thermistor Equation is used to convert the resistance reading into a temperature.

The voltage reading must first be converted into a resistance reading. This is performed using an accurate voltage reference of 2.5V in the following manner:

```
2.5V (Vref) -------10k ----------|------------Thermistor ---------GND
                                 Measured Voltage
```

The ADC reading is converted into a voltage then the resistance of the thermistor is calculated by reversing the potential divider equation.

```
R = (V*R_Balance)/(2.5-V);
```

The resistance value is converted into the temperature with the Steinhart-Hart Thermistor Equation

```
T=1.0f/(1.0f/T0+(1.0f/B)*log(R/R0));
```

Where T0, B and R0 are all parameters from the thermistor datasheet. In this case they are: B = 4126.0, T0 = 298.15 and R0 = 10000.

Current conversion

The output from the hall-effect current sensor is a voltage proportional to the current flowing (sometimes a shunt resistor is required, which has space on the circuit board, if required).

This voltage is positive or negative, depending upon the direction of current flow.

As the ADC cannot read negative voltages, this voltage is level shifted up to 1.25V +/- the current reading. The 1.25V is taken from the 2.5V reference voltage, divided by a 100K-100K potential divider.

The output voltage of the hall-effect current sensor is usually stated as a maximum voltage at the rated current.

For example a 25A current sensor will put out 0.625V at 25A. So each amp will be 0.025V or 25mV.

To convert into a real value we use the conversion table at the top of the Arduino code. This holds the four conversion values for the 4 current sensors (only the first one used in the 1 channel slave unit). The numbers are the mV per amp conversion.

```
// This holds the current conversion value
// This is in mV per AMP
// 4 values required - I1,I2,I3,I4
const int currentConversions[] = {100,625,100,200};
```

This example shows 100mV/A, 625mV/A, 100mV/A and 200mV per amp.

The higher the mV/amp value the more accurate the measurement, but the full scale must be appropriate to the current being measured.

The maximum deflection to be measured is 1.25V or 1250mV, so the maximum output at full current must not be greater than this.

Voltage conversion

Voltage conversion is performed using a simple potential divider. This is protected by a 5.1V zener diode, but the maximum output voltage of the potential divider must not exceed 5V. The balance resistors for all the potential divider circuits are 10K ohm. The calculations are performed using the voltResistors array values, which gives the input resistors for all 8 voltage channels (only the first two are used in the 1 channel version).

```
//This holds the resistor values for the voltage potential dividers
// Values are in k ohm. The other resistor is 10k
// 8 values required - Voltages 1.1,1.2,2.1,2.2,3.1,3.2,4.1,4.2
const int voltResistors[] = {20,20,20,20,20,20,20,20};
```

In this example all the input resistors are 20k ohm. This means that the voltage range is 0 to 5 x ((20+10)/10) = 0 to 15V DC. In reality there should b a higher range than that, even if measuring a ‘12’ volt battery unit.

Master unit - Hardware

Raspberry Pi

The master unit does not include any bespoke circuit boards. It is based upon a Raspberry Pi (http://www.raspberrypi.org/) small, low-cost, single board computer module.

A Raspberry Pi has been specified here as it is:

- Low cost
- Single board
- Low power consumption
- No excess hardware required

But any computer running Linux could be used as the master unit.

USB-RS485 converter

If RS485 communications layer is used then a USB to RS485 converter is required to communicate with the Slave units.

To convert the RS485 signals to be sent/received by the Raspberry Pi a converter is required. This plugs into a spare USB socket.

The BOB-09822 USB to RS485 converter from Sparkfun Electronics (https://www.sparkfun.com/products/9822) is used here. Other adaptors are available and can be used.

Master Unit – Software

We have written and provided some simple functions and routines to get you up and running with the VIt montiroing system. This is named the Fast MODBUS library (FML).

This is available as a github repository and that will include the latest changes, so please check there for any updates.

The software for the master unit is available on Github here:

https://github.com/matthewg42/fml

The 'How To' documentation for the software is available here:

https://github.com/matthewg42/fml/blob/master/doc/howto.md

Note: This is designed as a starting point for future work and implementation within larger data-gathering systems.

How to prepare your OS

1. Download the Raspian image file from the Raspberry Pi website http://downloads.raspberrypi.org/raspbian_latest
2. Unzip the image and write it to an SD card. A 4 GB card is ample in size. Using a class 10 or faster card is recommended.
3. Boot the Raspberry Pi and use the raspi-config program to enlarge the file system, set the hostname as desired and reboot.

How to download using git and build the installer

This procedure may be run on the Raspberry Pi itself, or another computer. Make sure you have the following packages / tools installed: git, make.

Clone the git repo https://github.com/matthewg42/fml.git

i.e. use the command

```
$ git clone https://github.com/matthewg42/fml.git
```

How to: make installer tarball from github repo

Change into the fml root directory and run make. i.e.

```
$ cd fml
$ make
```

A file named fml-{version}.tar.gz should be generated. This contains everything needed to install FML on a raspberry pi with a fresh Raspian installation.

How to: use an installer tarball

Synopsis: copy the tarball to a raspberry pi running Raspian, untar it, change into the directory it creates and run the install.sh with root permissions:

```
$ tar zxf fml-1.0.tar.gz
$ cd fml-1.0
$ sudo ./install.sh
```

How to: check if FML is running

Run this command:

```
$ fml --list
```

This will list any FML processes which are running, or print a message saying no FML processes could be found.

How to: start & stop FML service

The installer creates an init script for fml, which may be used to start and stop the fml as a service, and check if the service has been started:

```
$ sudo /etc/init.d/fml start
$ sudo /etc/init.d/fml status
$ sudo /etc/init.d/fml stop
```

How to: edit the configuration file

The configuration file is "/etc/fml.conf". Use your favourite editor (e.g. nano, vim) to edit this file. The file is formatted in "ini" style.

Change serial port settings

The following options can be set in the "[master]" section of the configuration file: serial_device (e.g. /dev/ttyUSB0), serial_baud, serial_bytesize, serial_parity (none, odd, even), serial_stopbits, serial_timeout

Change how frequently FML polls slave devices

Change the value of "interval" in the "[master]" section of the config file.

Add a new modbus slave

First create a new section in the config file where the section name is "[slave_{ID}]".

For each register create a key / value pair in the new section with the key set to "r{register-id}_name" and the value being some short description of the contents of the register.

Example

Slave ID (numerical modbus address): 240, has two temperature sensors with register IDs 4 and 7. The raw values from these registers is to be used, so no post-processing function is needed. Create the following section in the config file:

```
[slave_240]
r4_name = Temp1
r7_name = Temp1
```

Note

Any change to the slave configuration will result in the re-creation of the RRD database, destroying historical data.

Adding a new post-processing function

Post-processing function are defined in the pp_functions.py python module which is found here:

```
/usr/local/lib/python2.7/dist-packages/pp_functions.py
```

The file is owned by root, so to edit it, you will need to use sudo (with the editor of your choice, e.g. nano, vim, emacs etc):

```
sudo nano /usr/local/lib/python2.7/dist-packages/pp_functions.py
```

To add a new post-processing function, first define a function call which does the necessary manipulation, e.g.:

```
def my_post_processing_function(input1, input2)
  return (input1+input2)/2.0
```

...then call the register function. The best place to do this is where the register function is also called for existing post-processing functions - just before the **main** dection of the pp_functions.py file:

```
register(thermister_to_celcius)         # these three
register(scale_voltage)                 # lines already exist
register(scale_current)                 # in pp_functions.py
register(my_post_processing_function)   # Add this line for your
function
```

Adding a register with a post-processing function

Assuming an appropriate post-processing function is already implemented, it may be called by adding a new key in the slave section of the config file named "r{register-id}_pp_fn". The value for this key is the name of the post-processing function to be used.

Some post-processing functions require additional per-register parameters. These may be specified using a key named "r{register-id}_pp_param" with parameters passed as a comma separated list of values.

Example

To specify the post-processing function "thermister_to_celcius" for register 0 with parameters 4126, 298.15, 10000, the register should defined in the appropriate slave section of the config file as follows:

```
r0_name = Temp1
r0_pp_fn = thermister_to_celcius
r0_pp_param = 4126,298.15,10000
```

How to: run in a terminal and see data coming through fml

To run in the terminal, stop the daemon mode instance of fml, and run the fml program directly:

```
$ sudo /etc/init.d/fml stop
$ sudo fml
```

To stop the terminal mode execution, press control-C. If desired, re-start the daemon mode:

```
$ sudo /etc/init.d/fml start
```

How to: add a graph to the web server

Updating what the web server shows is still pretty manual. Future development may make this easier. For now you have to do the following:

1. Edit /usr/local/bin/fml_update_graphs.sh to generate new graphs using rrdtool.
2. Edit /var/lib/fml/www/index.html to add the graph to the web page.

Appendices

Arduino code

```
/*********************************************************
/****** Product Health - Battery Monitor Unit ***********
/****** by Matt Little **********************************
/****** Date: 19/12/13 **********************************
/****** info@re-innovation.co.uk ************************
/****** www.re-innovation.co.uk *************************
/*********************************************************

/*************Details of Code*****************************

This is a prototype for a battery monitoring DAQ unit.
It is designed to measure 1 x temperatures 2 x Voltage and 1 x Current.
Temperatures are measured with a 10k thermistor.
Voltages are measured with a potential divider.
Current is measured with a hall-effect sensor.

There are two versions a single 'channel' (meaning 1 x temp, 2 x Voltages
and 1 x current)
and a 4 'channel' (with 4 x temp, 8 x voltages, 4 x currents)

These are measured either with:
ADS1015. This communicates via I2C.
AD7390. This communicates via SPI.

The board is an Arduino (Uno bootloader).

Connection is vai RS485 converter (using a MAX485 IC).

Communication is via the MODBUS protocol.

There is an on-board EEPROM for on-the-fly reprogramming.

/************ Wiring Details ****************************

The circuit schematic should be available at www.re-innovation.co.uk

****** ADD FULL PIN WIRING HERE ********

Arduino ID  Actual Pin    Function
D0          2             Rx
D1          3             Tx
D2          4             TEST switch
D3          5             ID3
D4          6             ID4
D5          11            ID5
D6          12            ID6
D7          13            ID7
D8          14            ID8
D9          15            ID1
D10         16            ~CS for EEPROM in I2C and ADC in SPI
D11         17            MOSI
D12         18            MISO
D13         19            SCLK

A0          23            TxEnable for RS485
A1          24            ID2
A2          25            LED1
```

```
A3          26            LED2
A4          27            SDA in I2C or ~CS for EEPROM in SPI
A5          28            SCLCK in I2C


//*****Code Examples Used:**************

//****** ADS1015 Code *****************
Adafurit ADS1015 break out board used
http://learn.adafruit.com/adafruit-4-channel-adc-breakouts/programming
The ADS1015 measures +/- 4.096V (differential). Full scale is 4.096V.

//******* AD7390 Code *****************
The ADC requires a control string to be written to the device.
This is 12 bits long. At the same time it will clock back the
16 bit result from the previous conversion.
The 12 Control bits are:
WRITE :SEQ  :ADD3  :ADD2  :ADD1  :ADD0  :PM1  :PM0  :SHADOW  :WEAK  :RANGE
:CODING
11    :10   :9     :8     :7     :6     :5    :4    :3       :2     :1
:0
Which are set to:
1     :0    :0     :0     :0     :0     :1    :1    :1       :0     :0
:1
See the datasheet for full expalination of these.

Wiring for AD7390 in order for good PCB layout this is non-standard:
Data  ADC Input Line
T1.1  11
V1.1  10
V1.2  9
I1.1  8
T2.1  7
V2.1  6
V2.2  5
I2.1  4
T3.1  3
V3.1  2
V3.2  1
I3.1  0
T4.1  12
V4.1  13
V4.2  14
I4.1  15

//*********Thermistor code**************
From here:
http://www.arduino.cc/playground/ComponentLib/Thermistor2

//**********MODBUS Code*****************From here:
http://code.google.com/p/simple-modbus/

  Updates:
  10/4/14   Changed ID for new ID switch layout - Matt Little
  10/4/14   Added SPI/I2C choice - Matt Little
  10/4/14   Added SPI code for reading AD7390 - Matt Little
  10/4/14   Flash LED 'heatbeat' - Matt Little
  6/5/14    Sorted out SPI reading code - Matt Little
  6/5/14    Sample rates for different versions - Matt Little
  7/5/14    Sorting out conversion - Matt Little
```

```
  7/5/14    Adding LEDs to show when MODBUS data received/sent - Matt
Little


  To Do:
  EEPROM read/write - show it works.
  LEDs to show MODBUS data I/O - Need to change ModbusSlave library
  Check MODBUS data


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

#include <Wire.h>
#include <Adafruit_ADS1015.h>  // For the ADS1015 ADC
#include <math.h>  // To perform floating point calculations
#include <SPI.h> // Required for the AD7390 version
#include <SimpleModbusSlave.h>  // To implement MODBUS slave

//*****************************************************************//
//*************** VARIABLES TO CHANGE *******************************//
//*****************************************************************//
#define DEVICE "4CHANNEL"  //  Either use 1CHANNEL or 4CHANNEL
#define OUTPUTSTYLE "HUMAN"  // Either use HUMAN or RAW

//This holds the resistor values for the voltage potential dividers
// Values are in k ohm. The other resistor is 10k
// 8 values required - Voltages 1.1,1.2,2.1,2.2,3.1,3.2,4.1,4.2
const int voltResistors[] = {20,20,20,20,20,20,20,20};

// This holds the current conversion value
// This is in mV per AMP
// 4 values required - I1,I2,I3,I4
const int currentConversions[] = {100,625,100,200};

//*****************************************************************//
//************** END OF VARIABLES TO CHANGE ***********************//
//*****************************************************************//

#define TESTPIN 2  // Digital pin 10 is serial test on/off
#define TXENABLEPIN A0  // This controls the transmit enable (for MAX485
IC)
#define LED1 A2    // Output LED1
#define LED2 A3    // Output LED2
#define CSADCSPI 10      // Chip select pin for the SPI ADC
#define CSEEPROMSPI A4    // Chip select pin for the EEPROM with SPI ADC
#define CSEEPROMI2C 10   // Chip select pin for the EEPROM with I2C ADC

#define NUMSAMPLESI2C 20  // This is the number of samples to average over
I2C (slow) version
#define NUMSAMPLESSPI 200  // This is the number of samples to average over
I2C (slow) version

unsigned int NUMSAMPLES = 0;  // Filled with data from above

// ADC info:
// I2C Version
// Adafruit_ADS1115 ads;  /* Use this for the 16-bit version */
Adafruit_ADS1015 ads1(0x48);     /* Use this for the 12-bit version */
// SPI Version
// Does not need anything - not using a library
```

```
//Thermistor info:
// enumarating 3 major temperature scales
enum {
  T_KELVIN=0,
  T_CELSIUS,
  T_FAHRENHEIT
};
// manufacturer data for episco k164 10k thermistor
// simply delete this if you don't need it
// or use this idea to define your own thermistors
//#define EPISCO_K164_10k 4300.0f,298.15f,10000.0f  // B,T0,R0
#define GT_Thermistor_10k 4126.0f,298.15f,10000.0f  // B,T0,R0


//****** MODBUS INFO ***************
// Using the enum instruction allows for an easy method for adding and
// removing registers. Doing it this way saves you #defining the size
// of your slaves register array each time you want to add more registers
// and at a glimpse informs you of your slaves register layout.

//////////////// registers of your slave ///////////////////
enum
{
  // just add or remove registers and your good to go...
  // The first register starts at address 0
  TEMP1,
  VOLTS11,
  VOLTS12,
  AMPS1,
  TEMP2,
  VOLTS21,
  VOLTS22,
  AMPS2,
  TEMP3,
  VOLTS31,
  VOLTS32,
  AMPS3,
  TEMP4,
  VOLTS41,
  VOLTS42,
  AMPS4,
  HOLDING_REGS_SIZE // leave this one
  // total number of registers for function 3 and 16 share the same
register array
  // i.e. the same address space
};

unsigned int holdingRegs[HOLDING_REGS_SIZE]; // function 3 and 16 register
array
////////////////////////////////////////////////////////////

int ID = 0x00;    // This holdes the unit ID
int newID = 0x00; // This holds the ID to test each time - only change ID
if newID is different

long oldMillis = 0;  // For the LED flashing

float Temp1 = 0;  // This holds the converted value of temperature
float Temp2 = 0;  // This holds the converted value of temperature
float Temp3 = 0;  // This holds the converted value of temperature
float Temp4 = 0;  // This holds the converted value of temperature
```

```
float Volts11 = 0;  // This holds the converted value of Voltage
float Volts12 = 0;  // This holds the converted value of Voltage
float Volts21 = 0;  // This holds the converted value of Voltage
float Volts22 = 0;  // This holds the converted value of Voltage
float Volts31 = 0;  // This holds the converted value of Voltage
float Volts32 = 0;  // This holds the converted value of Voltage
float Volts41 = 0;  // This holds the converted value of Voltage
float Volts42 = 0;  // This holds the converted value of Voltage
float Amps1 = 0;  // This holds the converted value of Current
float Amps2 = 0;  // This holds the converted value of Current
float Amps3 = 0;  // This holds the converted value of Current
float Amps4 = 0;  // This holds the converted value of Current

//int16_t adc0, adc1, adc2, adc3;  // These hold the raw ADC values

unsigned long int tempAve1,voltsAve11,voltsAve12,ampsAve1;
unsigned long int tempAve2,voltsAve21,voltsAve22,ampsAve2;
unsigned long int tempAve3,voltsAve31,voltsAve32,ampsAve3;
unsigned long int tempAve4,voltsAve41,voltsAve42,ampsAve4;

unsigned long int data1,data2;  // Hold result from ADC

int IDarray[] = {8,7,6,5,4,3,A1,9};  // This holds the configuration for
the ID switch connections

void setup(void)
{
  // Initialise the digital I/O
  pinMode(TESTPIN, INPUT_PULLUP);
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);

  //An 8 bit switch is used as the device ID
  for(int i=0;i<8;i++)
  {
    pinMode(IDarray[i], INPUT_PULLUP);
  }

  //Sort out the ID from from the associated ID array
  ID=0;  // Reset the ID
  ID = readID();

  // Set up the ADC
  if(DEVICE=="1CHANNEL")
  {
    //Set up sample rate
    NUMSAMPLES = NUMSAMPLESI2C;
    // This uses the I2C for ADC
    pinMode(CSEEPROMI2C, OUTPUT);
    // The ADC input range (or gain) can be changed via the following
    // functions, but be careful never to exceed VDD +0.3V max, or to
    // exceed the upper and lower limits if you adjust the input range!
    // Setting these values incorrectly may destroy your ADC!
    //
ADS1015  ADS1115
    //                                                                -----
--  -------
    // ads.setGain(GAIN_TWOTHIRDS);  // 2/3x gain +/- 6.144V  1 bit = 3mV
0.1875mV (default)
    ads1.setGain(GAIN_ONE);        // 1x gain   +/- 4.096V  1 bit = 2mV
0.125mV
```

```
    // ads.setGain(GAIN_TWO);        // 2x gain   +/- 2.048V  1 bit = 1mV
0.0625mV
    // ads.setGain(GAIN_FOUR);       // 4x gain   +/- 1.024V  1 bit = 0.5mV
0.03125mV
    // ads.setGain(GAIN_EIGHT);      // 8x gain   +/- 0.512V  1 bit =
0.25mV   0.015625mV
    // ads.setGain(GAIN_SIXTEEN);    // 16x gain  +/- 0.256V  1 bit =
0.125mV  0.0078125mV
    ads1.begin();

  }
  else if(DEVICE=="4CHANNEL")
  {
    //Set up sample rate
    NUMSAMPLES = NUMSAMPLESSPI;

    // This uses the SPI ADC
    pinMode(CSADCSPI, OUTPUT);
    pinMode(CSEEPROMSPI, OUTPUT);

    SPI.begin();
    // Data clocked on falling edge.
    // If the first bit sent is 1 then
    // this is the CONTROL REGISTER

    digitalWrite(CSADCSPI,LOW);
    //  send in the address and value via SPI:
    SPI.transfer(B10000011);
    SPI.transfer(B10010000);
    // take the SS pin high to de-select the chip:
    digitalWrite(CSADCSPI,HIGH);

    //Then send the SHADOW REGISTER
    // Set the shadow register - all HIGH to read all channels
    digitalWrite(CSADCSPI,LOW);
    //  send in the address and value via SPI:
    SPI.transfer(B11111111);
    SPI.transfer(B11111111);
    // take the SS pin high to de-select the chip:
    digitalWrite(CSADCSPI,HIGH);
  }
  else
  {
    // Catch in case no DEVICE specified
    Serial.println("NO DEVICE");
  }

 // Set up the MODBUS slave device
    /* parameters(HardwareSerial* SerialPort,
                long baudrate,
          unsigned char byteFormat,
                unsigned char ID,
                unsigned char transmit enable pin,
                unsigned int holding registers size,
                unsigned int* holding register array)
  */

  /* Valid modbus byte formats are:
     SERIAL_8N2: 1 start bit, 8 data bits, 2 stop bits
     SERIAL_8E1: 1 start bit, 8 data bits, 1 Even parity bit, 1 stop bit
     SERIAL_8O1: 1 start bit, 8 data bits, 1 Odd parity bit, 1 stop bit
```

```
     You can obviously use SERIAL_8N1 but this does not adhere to the
     Modbus specifications. That said, I have tested the SERIAL_8N1 option
     on various commercial masters and slaves that were suppose to adhere
     to this specification and was always able to communicate... Go figure.

     These byte formats are already defined in the Arduino global name
space.
  */
  modbus_configure(&Serial, 115200, SERIAL_8N2, ID, TXENABLEPIN,
HOLDING_REGS_SIZE, holdingRegs);

  // If we are in test mode then output data on the serial port
  // Otherwise dont
  if(digitalRead(TESTPIN)==LOW)
  {
    Serial.println("Product Health");
    Serial.println("MODBUS Battery Monitor");
    Serial.println("Version 1.0");
    Serial.println("Matt Little (matt@re-innovation.co.uk)");
    if(DEVICE=="1CHANNEL")
    {
      Serial.println("1 Channel Device");
      Serial.println("I2C ADC Range: +/- 4.096V (Gain = 1x)");
    }
    else if(DEVICE=="4CHANNEL")
    {
      Serial.println("4 Channel Device");
      Serial.println("SPI ADC Range: ???");
    }
    Serial.print("Device ID: ");
    Serial.println(ID);
  }

}

void loop(void)
{
  // Take readings depending upon the device used (1CHANNEL or 4CHANNEL)

  for(int j=0;j<NUMSAMPLES;j++)
  {

    if(DEVICE=="1CHANNEL")
    {
      // READ I2C ADC
      tempAve1 += ads1.readADC_SingleEnded(0);
      voltsAve11 += ads1.readADC_SingleEnded(1);
      voltsAve12 += ads1.readADC_SingleEnded(2);
      ampsAve1 += ads1.readADC_SingleEnded(3);
    }
    else if(DEVICE=="4CHANNEL")
    {
      // READ SPI ADC
      // Here we can ask for data and see what we get back.
      // take the SS pin low to select the chip:
      for(int y=0;y<16;y++)
      {

        // Data clocked on falling edge.
        // Data transferred to output
```

```
        // First 4 bits are the address bits
        // Next 12 bits are data
        digitalWrite(CSADCSPI,LOW);
        //  send in the address and value via SPI:
        data1 = SPI.transfer(B00000000);
        data2 = SPI.transfer(B00000000);
        // take the SS pin high to de-select the chip:
        digitalWrite(CSADCSPI,HIGH);

//        Serial.print("Channel ");
//        // Want to get the first 4 binary digits
//        Serial.print(data1>>4);
//        Serial.print(": ");

        // Data is in the format:
        // Data1 =  ID ID ID ID D11 D10 D9 D8
        // Data2 =  D7 D6 D5 D4 D3  D2  D1 D0

        // So to get the data out we need to AND data1 with a mask
        // Then left shift 8 bits
        // Then add data2

        //Serial.println(data1,BIN);
        data1 = data1 & B00001111;
        //Serial.println(data1,BIN);
        data1 = data1<<8;
        //Serial.println(data1,BIN);
        data1 = data1+data2;

        //Serial.println(data1);

        switch(y)
        {
          case 0:
            ampsAve3 += data1;
          break;
          case 1:
            voltsAve32 += data1;
          break;
          case 2:
            voltsAve31 += data1;
          break;
          case 3:
            tempAve3 += data1;
          break;
          case 4:
            ampsAve2 += data1;
          break;
          case 5:
            voltsAve22 += data1;
          break;
          case 6:
            voltsAve21 += data1;
          break;
          case 7:
            tempAve2 += data1;
          break;
          case 8:
            ampsAve1 += data1;
          break;
          case 9:
```

```
            voltsAve12 += data1;
          break;
          case 10:
            voltsAve11 += data1;
          break;
          case 11:
            tempAve1 += data1;
          break;
          case 12:
            tempAve4 += data1;
          break;
          case 13:
            voltsAve41 += data1;
          break;
          case 14:
            voltsAve42 += data1;
          break;
          case 15:
            ampsAve4 += data1;
          break;
        }

        data1 = 0;
        data2 = 0;
      }
    }
    // Sort out the MODBUS information
     // modbus_update() is the only method used in loop(). It returns the
total error
    // count since the slave started. You don't have to use it but it's
useful
    // for fault finding by the modbus master.
    modbus_update();
    // MODBUS Commands (sent from Master:
    // Get Data from channel 1: 0x 02 03 0000 0004 443a
    // Get Data from channel 2: 0x 02 03 0004 0004 05fb
    // Get Data from channel 3: 0x 02 03 0008 0004 c5f8
    // Get Data from all channels: 0x 02 03 0000 000c 45fc
    //
    /* Note:
       The use of the enum instruction is not needed. You could set a
maximum allowable
       size for holdinRegs[] by defining HOLDING_REGS_SIZE using a constant
and then access
       holdingRegs[] by "Index" addressing.
       I.e.
       holdingRegs[0] = analogRead(A0);
       analogWrite(LED, holdingRegs[1]/4);
    */
  }

  // Here we divide through to get the average:
  if(DEVICE=="1CHANNEL"||DEVICE=="4CHANNEL")
  {
    tempAve1 = tempAve1/NUMSAMPLES;
    voltsAve11 = voltsAve11/NUMSAMPLES;
    voltsAve12 = voltsAve12/NUMSAMPLES;
    ampsAve1 = ampsAve1/NUMSAMPLES;

    // Store data for MODBUS
```

```
    holdingRegs[TEMP1] = tempAve1;     // update data to be read by the
master
    holdingRegs[VOLTS11] = voltsAve11; // update data to be read by the
master
    holdingRegs[VOLTS12] = voltsAve12; // update data to be read by the
master
    holdingRegs[AMPS1] = ampsAve1;     // update data to be read by the
master
  }
  if(DEVICE=="4CHANNEL")
  {
    tempAve2 = tempAve2/NUMSAMPLES;
    voltsAve21 = voltsAve21/NUMSAMPLES;
    voltsAve22 = voltsAve22/NUMSAMPLES;
    ampsAve2 = ampsAve2/NUMSAMPLES;
    tempAve3 = tempAve3/NUMSAMPLES;
    voltsAve31 = voltsAve31/NUMSAMPLES;
    voltsAve32 = voltsAve32/NUMSAMPLES;
    ampsAve3 = ampsAve3/NUMSAMPLES;
    tempAve4 = tempAve4/NUMSAMPLES;
    voltsAve41 = voltsAve41/NUMSAMPLES;
    voltsAve42 = voltsAve42/NUMSAMPLES;
    ampsAve4 = ampsAve4/NUMSAMPLES;

    // Store data for MODBUS
    holdingRegs[TEMP2] = tempAve2;     // update data to be read by the
master
    holdingRegs[VOLTS21] = voltsAve21; // update data to be read by the
master
    holdingRegs[VOLTS22] = voltsAve22; // update data to be read by the
master
    holdingRegs[AMPS2] = ampsAve2;     // update data to be read by the
master
    holdingRegs[TEMP3] = tempAve3;     // update data to be read by the
master
    holdingRegs[VOLTS31] = voltsAve31; // update data to be read by the
master
    holdingRegs[VOLTS32] = voltsAve32; // update data to be read by the
master
    holdingRegs[AMPS3] = ampsAve3;     // update data to be read by the
master
    holdingRegs[TEMP4] = tempAve4;     // update data to be read by the
master
    holdingRegs[VOLTS41] = voltsAve41; // update data to be read by the
master
    holdingRegs[VOLTS42] = voltsAve42; // update data to be read by the
master
    holdingRegs[AMPS4] = ampsAve4;     // update data to be read by the
master
  }

  //******Check TEST MODE *****
  // Test mode puts the data out on the serial line
  // This means no MODBUS master is required
  // Used only for testing that data is correct and in correct range
  // If TESTPIN is LOW then enter test mode

  if(digitalRead(TESTPIN)==LOW)
  {

    // Print the data to the serial port
```

```
    digitalWrite(TXENABLEPIN, HIGH);  // Need to enable transmission
    digitalWrite(LED2,HIGH);  // Switch ON the LED
    if(OUTPUTSTYLE=="HUMAN")
    {
      // HUMAN READABLE
      Serial.print(ID);
      Serial.print(',');

Serial.print(Temperature(tempAve1,T_CELSIUS,GT_Thermistor_10k,10000.0f));
      Serial.print(',');
      Serial.print(Voltage(voltsAve11,0));
      Serial.print(',');
      Serial.print(Voltage(voltsAve12,1));
      Serial.print(',');
      Serial.print(Current(ampsAve1,0));
      // Only want to show the next values if 4 CHANNEL device
      if(DEVICE=="4CHANNEL")
      {
        Serial.print(',');

Serial.print(Temperature(tempAve2,T_CELSIUS,GT_Thermistor_10k,10000.0f));
        Serial.print(',');
        Serial.print(Voltage(voltsAve21,2));
        Serial.print(',');
        Serial.print(Voltage(voltsAve22,3));
        Serial.print(',');
        Serial.print(Current(ampsAve2,1));
        Serial.print(',');

Serial.print(Temperature(tempAve3,T_CELSIUS,GT_Thermistor_10k,10000.0f));
        Serial.print(',');
        Serial.print(Voltage(voltsAve31,4));
        Serial.print(',');
        Serial.print(Voltage(voltsAve32,5));
        Serial.print(',');
        Serial.print(Current(ampsAve3,2));
        Serial.print(',');

Serial.print(Temperature(tempAve4,T_CELSIUS,GT_Thermistor_10k,10000.0f));
        Serial.print(',');
        Serial.print(Voltage(voltsAve41,6));
        Serial.print(',');
        Serial.print(Voltage(voltsAve42,7));
        Serial.print(',');
        Serial.print(Current(ampsAve4,3));
      }
      Serial.println();  // End the line with a CR
    }
    else
    {
      // RAW DATA
      Serial.print(ID);
      Serial.print(',');
      Serial.print(tempAve1);
      Serial.print(',');
      Serial.print(voltsAve11);
      Serial.print(',');
      Serial.print(voltsAve12);
      Serial.print(',');
      Serial.print(ampsAve1);
```

```
       // Only want to show the next values if 4 CHANNEL device
      if(DEVICE=="4CHANNEL")
      {
        Serial.print(',');
        Serial.print(tempAve2);
        Serial.print(',');
        Serial.print(voltsAve21);
        Serial.print(',');
        Serial.print(voltsAve22);
        Serial.print(',');
        Serial.print(ampsAve2);
        Serial.print(',');
        Serial.print(tempAve3);
        Serial.print(',');
        Serial.print(voltsAve31);
        Serial.print(',');
        Serial.print(voltsAve32);
        Serial.print(',');
        Serial.print(ampsAve3);
        Serial.print(',');
        Serial.print(tempAve4);
        Serial.print(',');
        Serial.print(voltsAve41);
        Serial.print(',');
        Serial.print(voltsAve42);
        Serial.print(',');
        Serial.print(ampsAve4);
      }
      Serial.println();  // End the line with a CR
    }
    digitalWrite(TXENABLEPIN, LOW); // Need to disable transmission
    digitalWrite(LED2,LOW);  // Switch ON the LED
  }

  // RESET DATA
  // Here we reset all the averages, ready for the next set
  tempAve1 = 0;
  voltsAve11 = 0;
  voltsAve12 = 0;
  ampsAve1 = 0;
  tempAve2 = 0;
  voltsAve31 = 0;
  voltsAve22 = 0;
  ampsAve2 = 0;
  tempAve3 = 0;
  voltsAve31 = 0;
  voltsAve32 = 0;
  ampsAve3 = 0;
  tempAve4 = 0;
  voltsAve41 = 0;
  voltsAve42 = 0;
  ampsAve4 = 0;

  // Check the device ID digital pins.
  // Have they changed?
  checkID();

  // Flash the LED to show its all working
  flashLED();

}
```

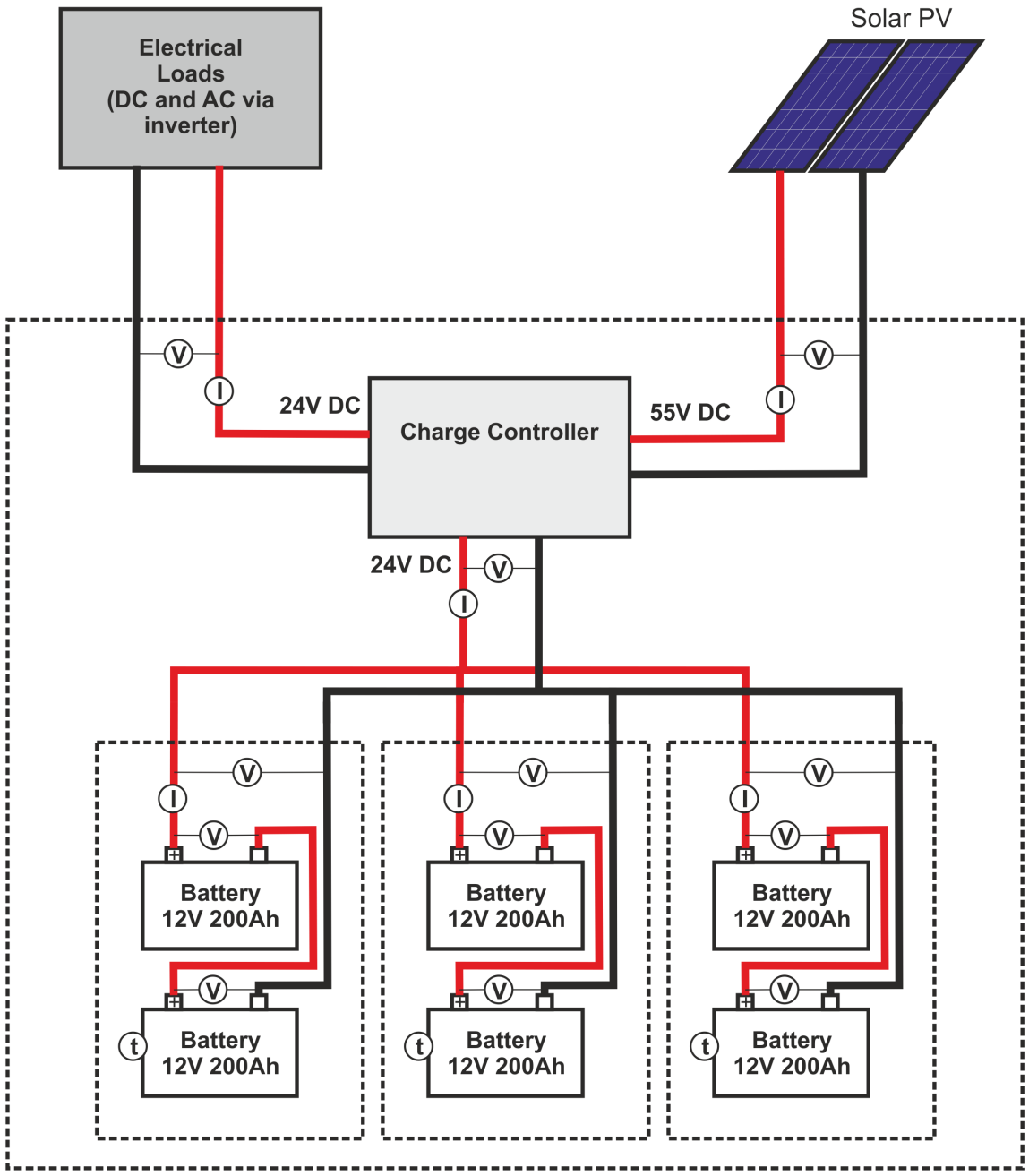
```
// Voltage function outputs float , the actual voltage
// The input is the ADC value and the channel number
// Each channel relates to a different load resistor value
// This is defined at the start

float Voltage(int16_t Input, int channel)
{
  float Vin,Vout;

  // This design has:
  // +Vinput ----- Rbk ---- Vmeasured ----- 10k ----- Gnd

  // So we calculate voltage:
  if(DEVICE=="1CHANNEL")
  {
    Vin = float(Input)*4.096/2048;
  }
  else if(DEVICE=="4CHANNEL")
  {
    Vin = float(Input)*5.0/4095;
  }

  Vout = (Vin*(voltResistors[channel]+1))/1;

  return Vout;
}

// Current function outputs float, the actual current
// The input is the ADC value and a channel value so each can be different
//

float Current(int16_t Input, int channel)
{
  float Vin,Iout;

  // This design uses a Hall Effect Sensor
  // Output voltage is 100mV/Amp

  // So we calculate voltage:
  if(DEVICE=="1CHANNEL")
  {
    Vin = float(Input)*4.096/2048;
  }
  else if(DEVICE=="4CHANNEL")
  {
    Vin = float(Input)*5.0/4095;
  }

  Iout = ((Vin-1.25)*1000)/currentConversions[channel];

  return Iout;
}

// Temperature function outputs float , the actual
// temperature
// Temperature function inputs
// 1.Input - The data to convert (as an int16_t)
// 2.OuputUnit - output in celsius, kelvin or fahrenheit
// 3.Thermistor B parameter - found in datasheet
// 4.Manufacturer T0 parameter - found in datasheet (kelvin)
```

```
// 5. Manufacturer R0 parameter - found in datasheet (ohms)
// 6. Your balance resistor resistance in ohms

float Temperature(int16_t Input,int OutputUnit,float B,float T0,float
R0,float R_Balance)
{
  float R,T,V;

  // This design has:
  // +2.5V ----- 10k precsision ---- Vmeasured ----- Thermistor ----- Gnd
  // The 2.5V is from an accurate reference.
  // The 10k precision is 0.1% tolerance

  if(DEVICE == "1CHANNEL")
  {
    // The conversion within the ADC is FSD = 4.096V
    // The 2048 comes from the output value with single ended values
    // So we calculate voltage:
    V = float(Input)*4.096/2048;
  }
  else if(DEVICE == "4CHANNEL")
  {
     // The conversion within the ADC is FSD = 2 x 2.5V = 5.00V
    // So we calculate voltage:
    V = float(Input)*5.0/4095;
  }
  // Which is used to calculate resistance:
  R = (V*R_Balance)/(2.5-V);

  T=1.0f/(1.0f/T0+(1.0f/B)*log(R/R0));

  switch(OutputUnit)
  {
    case T_CELSIUS :
      T-=273.15f;
    break;
    case T_FAHRENHEIT :
      T=9.0f*(T-273.15f)/5.0f+32.0f;
    break;
    default:
    break;
  };

  return T;
}

// This function check of the ID switch
// It restarts the MODBUS if the ID has changed

void checkID()
{
  //Sort out the ID from the switch value
  newID=0;  // Reset the ID
  newID = readID();
  if(newID!=ID)
  {
    Serial.flush();  // Stop and flush out all the serial data
    Serial.end();

    ID = newID;  // Update for the new ID
    //Serial.println("ID NOT correct - Restarting MODBUS!!!");
```

```
    modbus_configure(&Serial, 115200, SERIAL_8N2, ID, TXENABLEPIN,
HOLDING_REGS_SIZE, holdingRegs);

  }
}

// This reads the digital pins set up for the ID
// Made into a routine as this is used in different parts of the code
int readID()
{
  int tempID= 0x00;

  for(int i=0;i<8;i++)
  {
    // Check each dgital line
    if(digitalRead(IDarray[i])==HIGH)
    {
      tempID+=(0b00000001<<(i));
    }
  }
  return(tempID);
}

// This flashes the LED based upon the timer millis()
void flashLED()
{
  if(millis()>(oldMillis+1000))
  {
    digitalWrite(LED1,HIGH);
  }
  if(millis()>(oldMillis+1200))
  {
    digitalWrite(LED1,LOW);
    oldMillis=millis();
  }
}
```

Typical application diagram

Typical Battery Monitoring System



Ⓥ = Voltage Measurement
ⓣ = Temperature Measurement
Ⓘ = Current Measurement