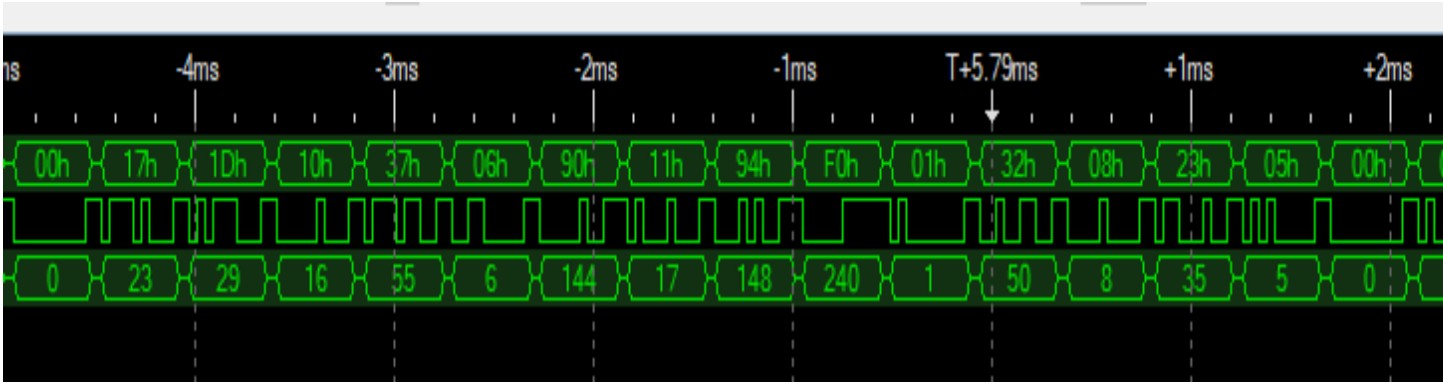


Hacking the Chinese Diesel Heater Communications Protocol

RAY JONES REV7, 15 SEP 2018.
RAY@MRJONES.ID.AU

INTRODUCTION

I recently installed a “Chinese diesel heater” into our caravan.

It was originally supplied with the basic Rotary Controller, which was then upgraded to an LCD Controller once I learnt the LCD Controller allows you to control mixture settings via a protected settings menu.
I wanted to tune the mixture as my heater would start screaming at full power, ala a real case of “full noise”!
It really was very noisy, and very objectional, 10’s of metres away. Screamed like a banshee.
Delicate crimping of the fuel line with pliers revealed this was probably due to too much fuel.

Whilst the LCD is cool n groovy and did indeed help me to tame the noise, it has the distinct annoyance that the display is always on. ALWAYS.
Firstly, this unnecessarily wastes power, especially when switched off, and could be a concern when 100% reliant upon solar recharge in bush camp settings.
Others have complained the LCD is simply too bright when shut eye time comes, especially as it remains lit even when the heater is off!

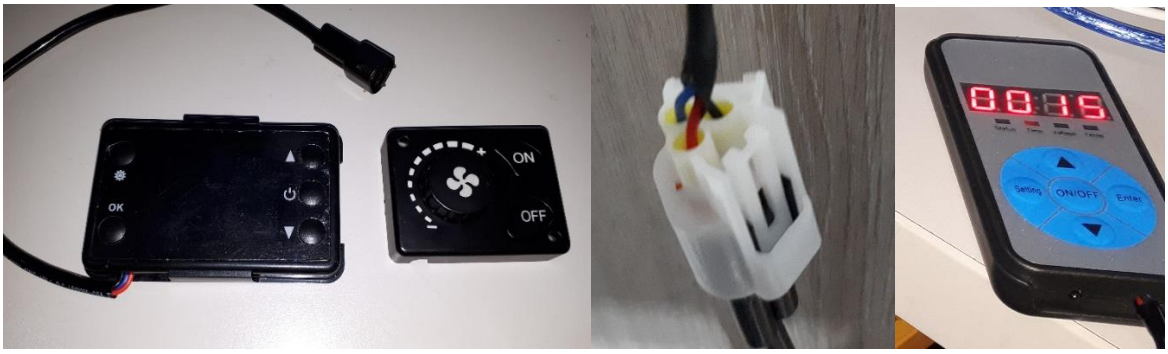
So, there is a mystical blue wire that connects between the heater itself and the controller.
If one can work out the serial protocol that exists upon that wire, one could roll their own controller implementation…

This has the distinct potential to improve the usability for “those that just have to fiddle”, lower power consumption, even develop friendly user interfaces that could be hosted via a smart phone app using Bluetooth!

HARDWARE UNDER INVESTIGATION

To set the scene of what is being discussed here, the units I have access to are the basic Rotary Controller, common to many ebay.au listings, as well as the LCD and LED variants I later bought.

The plug on my loom is a triangular 3 pin water proof connector – I note that the plug style may vary on other brands. *The plug is also not so easy to find*.
The relevance of this information may, or may not, apply to those other units.



The microcontroller device in the Rotary Controller is an STM8S003F3P6, in a TSSOP20 package.
The LCD Controller I suspect is in the same family, but it is in a LQFP32 package.
The manufacturer has ground the top off to hide the device ID, but the Tx/Rx pins do agree with the data sheet!
The LED controller has not been disassembled, but the same processor family is suspected.

POWER SUPPLY

5V is supplied via the red and black wires; red +5V, black 0V.
12V on the red wire will promptly destroy a controller.
The controllers do not include a voltage regulator or any decent protection – BE WARNED!

HALF-DUPLEX SERIAL DATA

Half duplex serial data travels over the blue wire.
The electronics to route the transmit and receive signals is quite unusual.
It uses a pair of BJT transistors to connect the Tx and Rx pins of the micro to the single blue wire, over which signalling can only happen one way at a time.

An NPN transistor exists on the Tx data path, a PNP transistor on the Rx data path.
A ‘TxENB’ gating signal determines which of these transistors is able to turn on, only allowing

the serial data signal to traverse that path.

'TxENB' is a GPIO signal and must be high for the micro to send data, with reference to the following schematic:

'TxENB' high:
T1 can conduct due to correct Base Emitter biasing. T2 is held off.
The signal from the micro's Tx output pin drives the blue wire.
The micro's Rx input pin is always high.

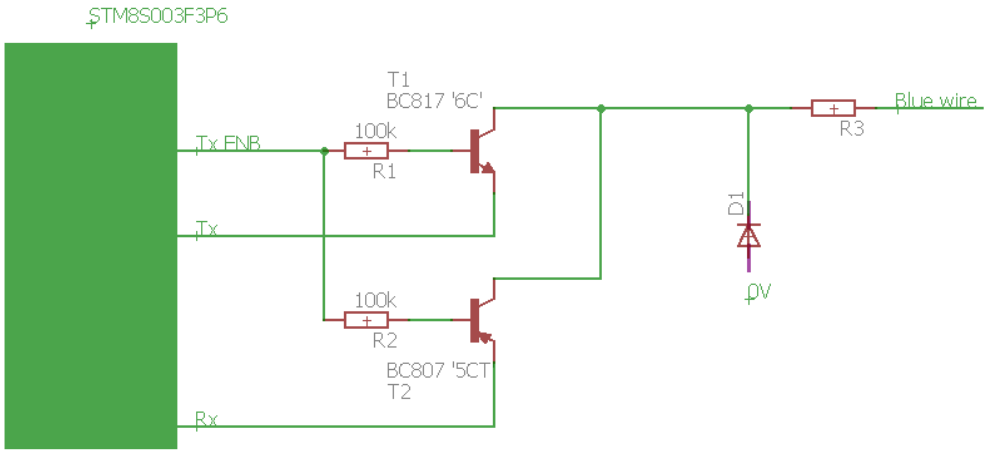
'TxENB' low:
'TxENB' is low for most of the time.
T1 is now held off, but T2 can now conduct as its Base Emitter biasing now allows it.
The micro's Rx input follows the blue wire.

Note that when Tx data is sent, this does not appear back on the Rx data pin!

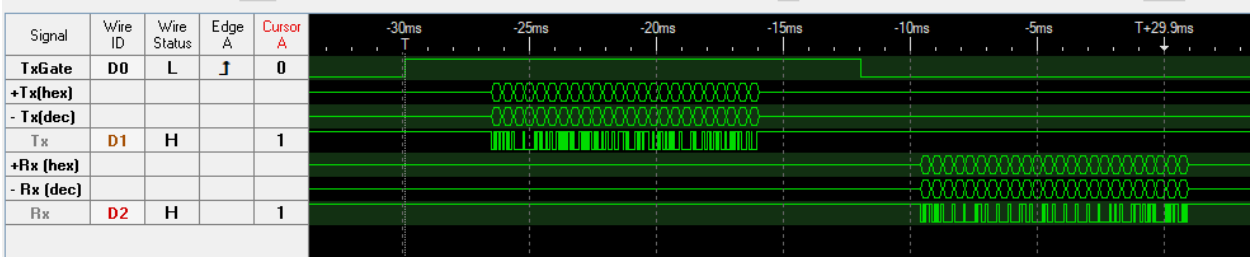
Unusual for sure, but it does model OK in LT-Spice.
It also obviously works in real life.

The part numbers BC817 & BC807 were found by Googling "6C" and "5CT" as the SMD body markings. These have been confirmed as correct due to an unfortunate incident requiring the transistors to be replaced in my LCD Controller ☹

The half-duplex serial circuit is as follows:

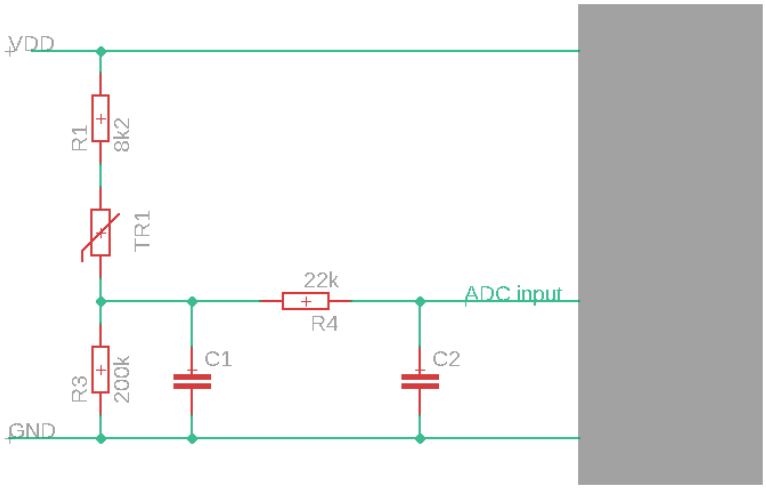


An example of the signal timing follows.
Note the controller sends data (Tx), then the heater responds about 5ms later (Rx).
Also note the gating signal on the top line that allows the clean split of Rx from Tx, on the micro's Rx input due the above circuitry.



TEMPERATURE SENSING

A small beaded thermistor sits near the edge of the PCB, acting as a temperature sensor.



Empirical testing reveals the following results:

Open circuit thermistor:	LCD shows -39°C
Shorted thermistor:	LCD shows 50°C

With the thermistor isolated, a resistance of 128kΩ was measured at 20°C, dropping with increased temperature.

Based upon the circuitry and the above test results, the thermistor is suspected to be a common glass beaded 100kΩ NTC type.

DATA PROTOCOL

The minimum bit period is 40us.
This is a somewhat oddball baud rate of 25,000 bits per second.
No big deal, just unusual.

The pulse pattern is conventional Asynchronous Serial type signalling with start and stop bits appearing to exist in the right places.

An Intronix Logicport logic analyser was configured to sample the Tx data line coming directly from the actual microcontroller.
An Asynchronous Serial interpreter was configured:

Baud Rate	25000
Logic Sense	positive
Structure	8N1, LSB first
Display Format	Hexadecimal

This revealed there were 24 bytes being sent in a burst, all neatly packed directly adjacent to each other, regularly repeating.

The repeat rate varies, dependent upon the controller:

- Rotary Controller: ~300ms
- LCD Controller: ~800ms
- LED Controller: ~300ms

Note that if the controller is not connected to the heater, the heater unit does not send any data.
i.e. all transmissions from the heater are in direct response to a controller transmission.

24 bytes is a fair bit of info, so I tried pressing the on/off buttons to see what happened.
The most obvious changes were the last two bytes always changed, which suggests a 16bit CRC lives there.
Closer inspection showed that certain bytes would change according to what was done.
Everything was clearly binary values, no ASCII data.

All very interesting, but the Rotary Controller was proving quite limited in exercising the link, so I swapped to the LCD Controller.
Now real progress was made.

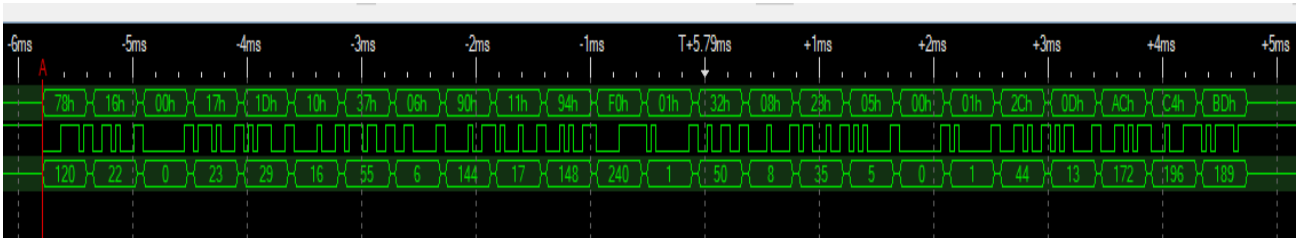
Pressing the up/down buttons to change the demanded temperature causes the 5th byte to change value up/down by 1.
Converting the hexadecimal value to decimal soon revealed the binary value was exactly the desired temperature, in Celsius, as you'd see on the display – Ah ha!
I eventually configured a second interpreter to read decimal values in parallel.

Resting a finger on the thermistor (temperature sensor) showed the byte 4's value change upwards!

Eventually by pushing this, doing that, going into the protected settings most of the mystical bytes started to reveal that they were simply conveying pretty much the all values presented in the LCD to the heater base unit on a continual basis.
But, not all locations are fully explained just yet.

This is a typical capture from my Rotary Controller shortly after being powered up.
After a few seconds 7 LEDs are flashing about the knob – the blue wire goes nowhere, E-07!
Despite that, it still spews out a data stream like below, repeating about every 300ms.

It is always 24 bytes of binary data, only the contents change.
Whilst binary in general sucks as it is not self-describing, life is in some ways easier as you do not need to discover potentially hidden ASCII messages etc.



An Arduino sketch was later developed, and is provided later, which allows sniffing of the blue wire; collecting the two-way conversation taking place between the controller and heater.

The sketch is intended to be used in conjunction with a serial terminal application on a PC, logging the data received from the Arduino, over USB.

Putty has been used with success here to create log files which are later analysed, primarily using Excel.

DATA PROTOCOL BYTE DEFINITIONS

TRANSMIT PACKET

The following table reveals what has been determined for the 24 message bytes **coming from the controller, to the heater**.
Bytes I'm not 100% certain on are *italicised* in the Purpose column.

Index	Purpose		Known Values	Scaling
0	*Start of Frame?*		78h, 76h *(Rotary, LCD)*	-
1	*Data Size?*		16h (constant)	-
2	Command		00h / 05h (stop) / A0h (start)	-
3	Temp Sensor		*variable*, **0 in fixed mode**	1°C / digit
4	Desired Temperature		*Variable*	1°C / digit
5	Minimum Pump frequency		*Variable*	0.1Hz / digit ***
6	Maximum Pump frequency		*Variable*	0.1Hz / digit ***
7	Minimum fan speed	MSB	*16bit variable*	1RPM / digit ***
8		LSB		
9	Maximum fan speed	MSB	*16bit variable*	1RPM / digit ***
10		LSB		
11	Heater Operating Voltage		78h, F0h *(12.0V/24.0V)*	0.1V / digit ***
12	Fan speed sensor		01h, 02h *(FN-1, FN-2)*	-
13	Thermostat/Fixed mode		32h, CDh *(Thermostat, Fixed)*	**Byte 4 = 0 when fixed**
14	Lower temperature limit		08h (constant)	1°C / digit
15	Upper temperature limit		23h (constant)	1°C / digit
16	*Required Temperature Rise?*		05h (constant)	-
17	Manual pump (fuel prime)		00h, 5Ah *(Normal, Prime)*	-
18	*Maximum run time*	MSB	01h (constant)	1 sec / digit
19	*without ignition?*	LSB	2Ch (constant)	
20	*End of Frame?*	MSB	0Dh (constant)	-
21		LSB	ACh (constant)	-
22	CRC-16/MODBUS	MSB	*16bit variable*	-
23		LSB		

*** Tx Bytes[5..11] (Shaded orange) are ineffective from a Rotary Controller.

A more detailed discussion of the message bytes follows:

Tx Byte[0]

Start of Frame?
Always 78h from Rotary Controller.
Always 76h from LCD or LED Controller.

The value of this byte appears to have control over what parameters the heater will actually listen to, and also save into EEPROM.
The LCD/LED menus can tune the max min pump and fan speeds (mixture settings) and these also appear to be retained in the heater micro's internal EEPROM.
When you plug a Rotary controller in, its advertised max/min pump & fan settings are ignored, but the running limits match the max/min values that were set with an LCD/LED!

Likewise the heater ignores the operating voltage from a Rotary Controller, but honours it and faults out with voltage errors if set wrong from an LCD/LED.

Tx Byte[1]

Data size?
Always 16h => 22 decimal.
Adding 22 + 2 CRC bytes = 24 bytes…

Tx Byte[2]

Heater On/Off control
This byte is usually sent as 00h (do nothing / no change).
Others values are only sent at the instant the user desires to heater to start or stop:

- To start the heater, send A0h (for one transmission)
  – confirmation is relayed back in the Rx Data when "Run State" goes to 01h.
- To stop the heater, send 05h (for one transmission)
  – confirmation is relayed back in the Rx Data when "Run State" goes to 7, "Post glow".

Tx Byte[3]

Current actual temperature in degrees Celsius.
Changes if you heat/cool the bead thermistor on the PCB's edge (device that sticks out of the case's hole).
Each digit scales as 1°C, as read in decimal.

Tx Byte[4]

Desired temperature in degrees Celsius.
The streamed value exactly follows the LCD/LED's value after conversion to decimal.
Each digit scales as 1°C, as read in decimal.

Tx Byte[5]

Minimum Pump Frequency.
The minimum pump speed as set in the protected settings of the LCD/LED.
Each digit scales as 0.1Hz, as read in decimal => 10h = 16 decimal = 1.6Hz.
Combines with the minimum fan speed to define the low end fuel mixture.
**** Ineffective from the Rotary Controller.*

Tx Byte[6]

Maximum Pump Frequency.
The maximum pump speed as set in the protected settings of the LCD/LED.
Each digit scales as 0.1Hz, as read in decimal => 37h = 55 decimal = 5.5Hz.
Combines with the maximum fan speed to define the high end fuel mixture.
**** Ineffective from the Rotary Controller.*

Tx Bytes[7..8]

Minimum Fan Speed.
Combined as a 16bit value, Tx Byte[7] is the MSB.
The minimum fan speed as set in the protected settings of the LCD/LED.
Each digit scales as 1RPM as read in decimal => 0690h = 1680 decimal = 1680RPM.
Combines with the minimum pump frequency to define the low end fuel mixture.
**** Ineffective from the Rotary Controller.*

Tx Bytes[9..10]

Maximum Fan Speed.
Combined as a 16bit value, Tx Byte[9] is the MSB.
The maximum fan speed as set in the protected settings of the LCD/LED.
Each digit scales as 1RPM, as read in decimal => 1194h = 4500 decimal = 4500RPM.
Combines with the maximum pump frequency to define the high end fuel mixture.
**** Ineffective from the Rotary Controller.*

Tx Byte[11]

Heater Operating voltage.
Follows the operating voltage selection in the protected settings of the LCD/LED.
Each digit scales as 0.1V, as read in decimal.
Can be 78h or F0h (120 or 240 decimal => 12.0 or 24.0V)
If wrong from an LCD/LED, voltage faults will be generated, eg E-01 when set to 24V with a 12V power supply.
**** Ineffective from the Rotary Controller.*

Tx Byte[12]

Fan Speed Sensor count selection.
01h or 02h.
Follows the fan speed sensor selection in the protected settings of the LCD/LED. (SN-1 or SN-2).

Tx Byte[13]
Thermostatic or fixed power mode.
Toggling the operating mode on the controller changes this byte's value:
32h is Thermostatic mode.
CDh is Fixed power Hz mode. (one's complement of 0x32!)
When in Fixed power Hz mode, Tx Byte 4 always gets forced to zero (no temperature feedback).

Tx Byte[14]
Minimum allowed temperature setting.
Always 08h.
Matches the minimum desired temperature the controller can be set to in byte #5 (8°C).

Tx Byte[15]
Maximum allowed temperature setting.
Always 23h.
Matches the maximum desired temperature the controller can be set to in byte #5 (35°C).

Tx Byte[16]
Required Temperature Rise?
Always 05h.
The heater's "Run State" changes to 04h when the temperature of the heating chamber has increased by this amount since the pump was started, indicating fuel is burning OK.
The chamber temperature being reported via Rx Bytes[10..11].

Tx Byte[17]
Manual pump mode (priming).
00h in normal mode.
5Ah when in manual pump priming mode.

Tx Bytes[18..19]
Maximum Run Time without ignition?
Always 012Ch = 300 decimal => 5 minutes (300 seconds)?

Tx Bytes[20..21]
End of Transmit Frame?
Combined as a 16bit variable, MSB in Tx Byte[20].
Always 0DADh.

Tx Bytes[22..23]
CRC-16 / MODBUS checksum.
Calculated from the first 22 bytes.
Combined as a 16bit value, Tx Byte[22] is the MSB.

RECEIVE PACKET

The following table reveals what has been determined for the 24 message bytes **coming from the heater, into the controller**:

Index	Purpose		Known Values	Scaling
0	*Start of Frame?*		76h (constant)	-
1	*Data Size?*		16h (constant)	-
2	Run State		0, 1, 2, 3, 4, 5, 7, 8	-
3	On / Off		0, 1	-
4	Supply Voltage	MSB	*16bit Variable*	0.1V / digit
5		LSB		
6	Fan RPM	MSB	*16bit variable (within user limits)*	1RPM / digit
7		LSB		
8	Fan Voltage	MSB	*16bit variable*	0.1V / digit
9		LSB		
10	Heat exchanger temperature	MSB	*16bit variable*	1°C / digit
11		LSB		
12	Glow plug voltage	MSB	*16bit variable*	0.1V / digit
13		LSB		
14	Glow plug current	MSB	*16bit variable*	10mA / digit
15		LSB		
16	Pump frequency, actual		*Variable (within user limits)*	0.1Hz / digit
17	Error Code		*Variable*	-
18	Unknown		00h (constant)	-
19	Fixed mode Pump Frequency		*Variable*	0.1Hz / digit
20	*Run mode transition temp.*		64h (constant)	1°C / digit
21	*End of Frame?*		00h (constant)	-
22	CRC-16/MODBUS	MSB	*16bit variable*	-
23		LSB		

Rx Byte[0]

Start of Frame?

Always 0x76h.

Rx Byte[1]

Data size?

Always 16h => 22 decimal.

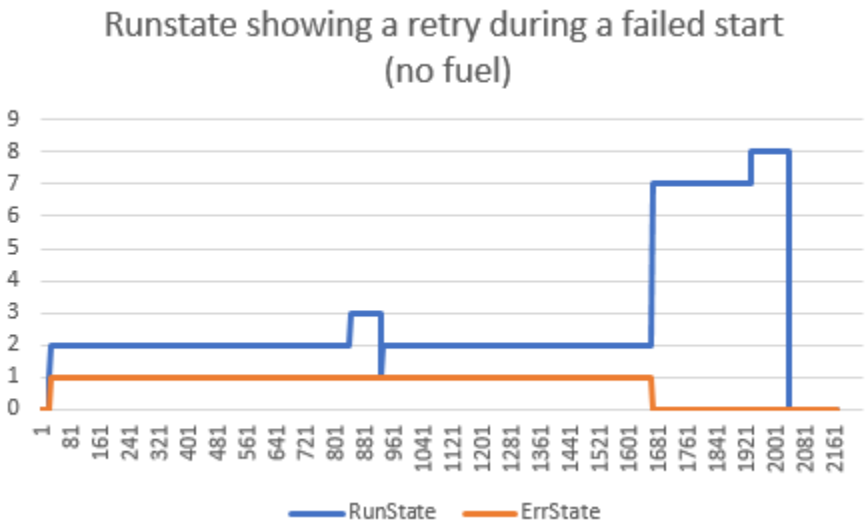
Adding 22 + 2 CRC bytes = 24 bytes…

Rx Byte[2]

Run State.

Conveys the state the heater is operating at:

0. Off / Standby
1. Start Acknowledge
2. Glow plug pre-heat
3. Failed ignition - pausing for retry.
4. Ignited – heating to full temp phase.
5. Running.
6. *Skipped – stop acknowledge?*
7. Stopping - Post run glow re-heat
8. Cooldown – returns to state 0 when body temp < 55°C



Rx Byte[3]

Error State.
00h if the heater is idle
01h if the heater is running normally (E-00).
02h+ - follows the value of Rx Byte[17] when error conditions occur.
Note the value here is always 1 more than the displayed error code.
eg: a simulated pump over current fault shows 5 here, the controller shows E-04.

Rx Bytes[4..5]

Measured supply voltage.
Combined as a 16bit variable, MSB in Rx Byte[4].
Reports voltage at heater as 0.1V / digit.
Based upon resistor scaling in heater, will show up to around 32V maximum.

Rx Bytes[6..7]

Fan RPM.
Combined as a 16bit variable, MSB in Rx Byte[6].
Reports fan speed as 1 RPM / digit.

Rx Bytes[8..9]

Fan Voltage
Combined as a 16bit variable, MSB in Rx Byte[8].
Reports average voltage across the fan motor as 0.1V / digit.
Note: when the heater is not running, this value tends to match the supply voltage value.
This must be due to the topology of the sensing circuit.

Rx Bytes[10..11]

Heat exchanger body temperature.
Combined as a 16bit variable, MSB in Rx Byte[10].
Reports temperature of heat exchanger body as 1°C / digit.

Rx Bytes[12..13]

Glow plug voltage.
Combined as a 16bit variable, MSB in Rx Byte[12].
Reports voltage across glow plug as 0.1V / digit.

Rx Bytes[14..15]

Glow plug current.
Combined as a 16bit variable, MSB in Rx Byte[14].
Reports current applied to glow plug as 10mA / digit.

Rx Byte[16]

Pump frequency, actual.
Reports current rate of fuel pump pulses.
Scaled as 0.1Hz / digit.

Rx Byte[17]

Stored Error Code from heater.
The value shown here is +1 over the displayed error on a controller.
eg 5 here shows E-04 on a controller.

0. No Error
1.
2. Voltage too low
3. Voltage too high
4. Ignition plug failure (no measured current flow?)
5. Pump Failure – over current
6. Too hot
7. Motor Failure
8. Serial connection lost
9. Fire is extinguished
10. Temperature sensor failure

Rx Byte[18]

Unknown.
Always 00h.

Rx Byte[19]

Fixed Mode Pump Frequency.
Changes to desired temperature appear to be integrated over about 8 seconds within the heater.
The final pump frequency changes according to the following formula:

$$Pfixed = Pmin + \left(\frac{Tdes - Tmin}{Tmax - Tmin} * (Pmax - Pmin) \right)$$

Where:
Pfixed = Rx Byte[19] *Pmin = Tx Byte[5]* *Pmax = Tx Byte[6]*
Tdes = Tx Byte[4] *Tmin = Tx Byte[14]* *Tmax = Tx Byte[15]*

The LCD/LED controller reads this byte to display the desired Hz value.
Desired Hz is not sent by the controller directly, it is calculated as shown above and fed back!

This value is scaled as 0.1Hz / digit.

Rx Byte[20]

Run mode transition temperature?.
Always 64h = 100.
The heater controller appears to usually transition from Run State 4 to 5 when the heat exchanger reaches 100 degrees...

Rx Byte[21]

Unknown
Always 00h.

Rx Bytes[22..23]

CRC-16 / MODBUS checksum.
Combined as a 16bit variable, MSB in Rx Byte[22].

CRC-16 DECODING

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Power On	78	16	00	16	23	10	37	06	90	11	94	F0	01	32	08	23	05	00	01	2C	0D	AC	DC	ED
10 secs	78	16	00	17	23	10	37	06	90	11	94	F0	01	32	08	23	05	00	01	2C	0D	AC	20	D0
Off	78	16	05	17	23	10	37	06	90	11	94	F0	01	32	08	23	05	00	01	2C	0D	AC	75	1C
Long Off	78	16	05	17	23	10	37	06	90	11	94	F0	01	32	08	23	05	5A	01	2C	0D	AC	78	44
Long On	78	16	05	00	23	10	37	06	90	11	94	F0	01	CD	08	23	05	00	01	2C	0D	AC	36	28
Min temp	78	16	05	00	08	10	37	06	90	11	94	F0	01	CD	08	23	05	5A	01	2C	0D	AC	D7	09
Max temp	78	16	05	00	23	10	37	06	90	11	94	F0	01	CD	08	23	05	5A	01	2C	0D	AC	3B	70
Long on	78	16	05	18	23	10	37	06	90	11	94	F0	01	32	08	23	05	5A	01	2C	0D	AC	0D	13
Off	78	16	05	00	23	10	37	06	90	11	94	F0	01	CD	08	23	05	00	01	2C	0D	AC	36	28

The above table shows some of the initial byte sequences that were captured from a Rotary Controller.

The shaded cells indicate the values that were different to the previous line.
Note the last 2 bytes always change value if another value changed, this is classic CRC behavior.

Being a 16bit CRC, there are many and varied implementations out there.
Fortunately the first online CRC calculator I encountered delivered pay dirt;
https://crccalc.com/

After laboriously entering **the first 22** bytes (as hex values), then pressing “Calc CRC-16” on the website, the results from 23 CRC-16 algorithms were presented.
Scanning the list, way down near the bottom an exact match for the last 2 bytes was found against CRC-16/MODBUS.

Trying all the other captured value sets in the table revealed the correct result every time.
CRC-16/MODBUS it is.

A tidy table-driven approach, using C code, to generate CRC-16/MODBUS checksums was found here: http://www.modbustools.com/modbus_crc16.html

The source code is replicated for convenience on the next page and has been tested to work on the tabled data above using Visual Studio 2017.

SAMPLE CODE TO GENERATE CRC-16/MODBUS CHECKSUMS

Original source: http://www.modbustools.com/modbus_crc16.html

```
WORD CRC16 (const BYTE *nData, WORD wLength)
{
static const WORD wCRCTable[] = {
0X0000, 0XC0C1, 0XC181, 0X0140, 0XC301, 0X03C0, 0X0280, 0XC241,
0XC601, 0X06C0, 0X0780, 0XC741, 0X0500, 0XC5C1, 0XC481, 0X0440,
0XCC01, 0X0CC0, 0X0D80, 0XCD41, 0X0F00, 0XCFC1, 0XCE81, 0X0E40,
0X0A00, 0XCAC1, 0XCB81, 0X0B40, 0XC901, 0X09C0, 0X0880, 0XC841,
0XD801, 0X18C0, 0X1980, 0XD941, 0X1B00, 0XDBC1, 0XDA81, 0X1A40,
0X1E00, 0XDEC1, 0XDF81, 0X1F40, 0XDD01, 0X1DC0, 0X1C80, 0XDC41,
0X1400, 0XD4C1, 0XD581, 0X1540, 0XD701, 0X17C0, 0X1680, 0XD641,
0XD201, 0X12C0, 0X1380, 0XD341, 0X1100, 0XD1C1, 0XD081, 0X1040,
0XF001, 0X30C0, 0X3180, 0XF141, 0X3300, 0XF3C1, 0XF281, 0X3240,
0X3600, 0XF6C1, 0XF781, 0X3740, 0XF501, 0X35C0, 0X3480, 0XF441,
0X3C00, 0XFCC1, 0XFD81, 0X3D40, 0XFF01, 0X3FC0, 0X3E80, 0XFE41,
0XFA01, 0X3AC0, 0X3B80, 0XFB41, 0X3900, 0XF9C1, 0XF881, 0X3840,
0X2800, 0XE8C1, 0XE981, 0X2940, 0XEB01, 0X2BC0, 0X2A80, 0XEA41,
0XEE01, 0X2EC0, 0X2F80, 0XEF41, 0X2D00, 0XEDC1, 0XEC81, 0X2C40,
0XE401, 0X24C0, 0X2580, 0XE541, 0X2700, 0XE7C1, 0XE681, 0X2640,
0X2200, 0XE2C1, 0XE381, 0X2340, 0XE101, 0X21C0, 0X2080, 0XE041,
0XA001, 0X60C0, 0X6180, 0XA141, 0X6300, 0XA3C1, 0XA281, 0X6240,
0X6600, 0XA6C1, 0XA781, 0X6740, 0XA501, 0X65C0, 0X6480, 0XA441,
0X6C00, 0XACC1, 0XAD81, 0X6D40, 0XAF01, 0X6FC0, 0X6E80, 0XAE41,
0XAA01, 0X6AC0, 0X6B80, 0XAB41, 0X6900, 0XA9C1, 0XA881, 0X6840,
0X7800, 0XB8C1, 0XB981, 0X7940, 0XBB01, 0X7BC0, 0X7A80, 0XBA41,
0XBE01, 0X7EC0, 0X7F80, 0XBF41, 0X7D00, 0XBDC1, 0XBC81, 0X7C40,
0XB401, 0X74C0, 0X7580, 0XB541, 0X7700, 0XB7C1, 0XB681, 0X7640,
0X7200, 0XB2C1, 0XB381, 0X7340, 0XB101, 0X71C0, 0X7080, 0XB041,
0X5000, 0X90C1, 0X9181, 0X5140, 0X9301, 0X53C0, 0X5280, 0X9241,
0X9601, 0X56C0, 0X5780, 0X9741, 0X5500, 0X95C1, 0X9481, 0X5440,
0X9C01, 0X5CC0, 0X5D80, 0X9D41, 0X5F00, 0X9FC1, 0X9E81, 0X5E40,
0X5A00, 0X9AC1, 0X9B81, 0X5B40, 0X9901, 0X59C0, 0X5880, 0X9841,
0X8801, 0X48C0, 0X4980, 0X8941, 0X4B00, 0X8BC1, 0X8A81, 0X4A40,
0X4E00, 0X8EC1, 0X8F81, 0X4F40, 0X8D01, 0X4DC0, 0X4C80, 0X8C41,
0X4400, 0X84C1, 0X8581, 0X4540, 0X8701, 0X47C0, 0X4680, 0X8641,
0X8201, 0X42C0, 0X4380, 0X8341, 0X4100, 0X81C1, 0X8081, 0X4040 };

BYTE nTemp;
WORD wCRCWord = 0xFFFF;

   while (wLength--)
   {
      nTemp = *nData++ ^ wCRCWord;
      wCRCWord >>= 8;
      wCRCWord ^= wCRCTable[nTemp];
   }
   return wCRCWord;

}
```

ARDUINO SKETCH TO READ BLUE WIRE

The following source code allows an Arduino that hosts more than one serial port to capture data from the blue wire, and relay that into a PC for logging over USB:

```
/*

  Chinese Heater Half Duplex Serial Data Collection Tool

  Connects to the blue wire of a Chinese heater, which is the half duplex serial link.
  Receives data from serial port 1.

  Terminology: Tx is to the heater unit, Rx is from the heater unit.

  The binary data is received from the line.
  If it has been > 100ms since the last activity this indicates a new frame
  sequence is starting, synchronise as such then count off the next 48 bytes
  storing them in the Data array.

  The "outer loop" then detects the count of 48 and packages the data to be sent
  over Serial to the USB attached PC.

  Typical data frame timing on the blue wire is:

  __Tx_Rx_____________________________Tx_Rx_____________________________Tx_Rx___________

  Rx to next Tx delay is always > 100ms and is paced by the controller.
  The delay before seeing Rx data after Tx is usually much less than 10ms.
  **The heater only ever sends Rx data in response to a data frame from the controller**

  Resultant data is tagged and sent out on serial port 0 (the default debug port),
  along with a timestamp for relative timing.

  This example works only with boards with more than one serial port like Arduino
  Mega, Due, Zero etc.

  The circuit:
  - Blue wire connected to Serial 1 Rx input - preferably with series 680ohm resistor.
  - Serial logging software on Serial port 0 via USB link

  created 24 Aug 2018 by Ray Jones

  modified 25 Aug by Ray Jones
    - simplifed to read 48 bytes, synchronised by observing a long pause
      between characters. The heater only sends when prompted.
      No longer need to discrimate which packet of data would be present.

  This example code is in the public domain.
*/


unsigned long lasttime;   // used to calculate inter character delay

void setup()
{
  // initialize listening serial port
  // 25000 baud, Tx and Rx channels of Chinese heater comms interface:
  // Tx/Rx data to/from heater, special baud rate for Chinese heater controllers
  Serial1.begin(25000);

  // initialise serial monitor on serial port 0
  Serial.begin(115200);

  // prepare for detecting a long delay
  lasttime = millis();
}
```

```
void loop()
{

  static byte Data[48];
  static bool RxActive = false;
  static int count = 0;

  // read from port 1, the "Tx Data" (to heater), send to the serial monitor:

  if (Serial1.available()) {

    // calc elapsed time since last rx'd byte to detect start of frame sequence
    unsigned long timenow = millis();
    unsigned long diff = timenow - lasttime;
    lasttime = timenow;

    if(diff > 100) {       // this indicates the start of a new frame sequence
      RxActive = true;
    }

    int inByte = Serial1.read(); // read hex byte

    if(RxActive) {
      Data[count++] = inByte;
      if(count == 48) {
        RxActive = false;
      }
    }
  }

  if(count == 48) {  // filled both frames - dump

    count = 0;
    char str[16];

    sprintf(str, "%08d  ", lasttime);
    Serial.print(str);                // print timestamp

    for(int i=0; i<48; i++) {

      if(i == 0)
        Serial.print("Tx ");          // insert Tx marker on first pass

      if(i == 24)
        Serial.print("Rx ");          // insert Rx marker after first 24 bytes

      sprintf(str, "%02X ", Data[i]);  // make 2 dig hex values
      Serial.print(str);              // and print

    }

    Serial.println();                 // newline and done

  }  // count == 48

}  // loop
```

SAMPLED DATA RUNS

The following image is from the first captured data set, using the Arduino sketch.
The charts are drawn solely from the data frames sent by the heater.
The data from the controller is not shown. The only data from the controller that actually changes whilst running is the sensed temperature and momentary commands to start and stop the heater.

The charts clearly show the operation, especially over the full duration of a run.

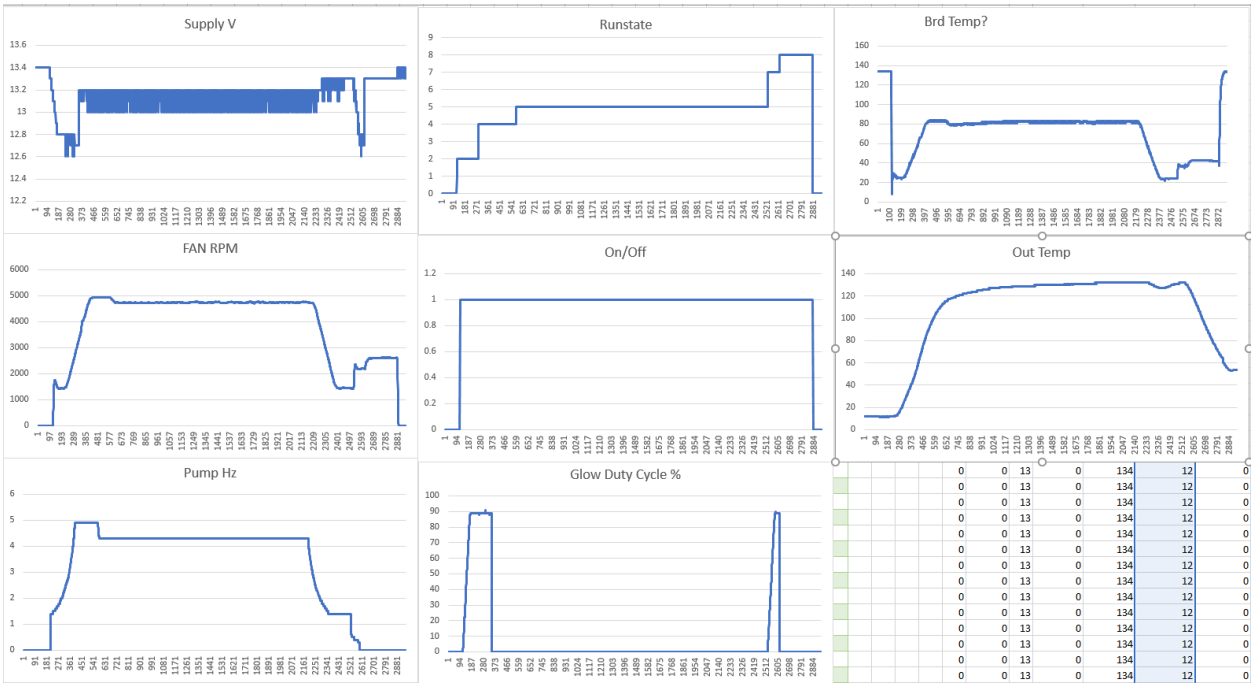
The dramatic drop in fan speed and pump frequency about 80% across are when the set temperature was exceeded by 1°C.

Shortly thereafter the unit was asked to shutdown.
You can see the glow pin being reheated for a short while, the pump stopping, then a cooldown phase and finally shutdown when the heat exchanger drops below 55°C.

One notable exception to the charts is the glow pin current, that rockets up to 10A.

Also noteworthy is the supply voltage drooping according to the current draw. I may need to beef my supply cables up!

Note that some of the annotations here have since been proven wrong, eg Brd Temp and Glow Duty cycle, they are actually fan volatge and glow plug voltage.



OBSERVED FAULT BEHAVIOUR

To learn more, certain deliberate errors were placed upon the heater to see how it reacted.

WRONG OPERATING VOLTAGE

Using the LCD/LED protected settings menu (passcode = 1688), the system was set to 24V, with a 12V battery system.

When set to run, the controller soon reported an E-01 fault, and the LED in the heater entered a “flash once” mode. – ala “low voltage”.

The protocol reported the value 02h in both Rx Byte[3] & Rx Byte[17].

Interestingly, my Rotary Controller also demands a 24V system, but a no fault is ever thrown.

The distinct difference is Tx Byte[0] is sent as 78h from the Rotary Controller, but 76h from the LCD/LED.
It appears code Tx Byte[0] = 78h causes the operating voltage byte to be ignored (and others).

PUMP OVERCURRENT

The heater circuit includes a simple current sense on the pump’s current draw, a transistor that turns on if the current is too high.

If the collector of that transistor is deliberately shorted simulating an overcurrent, the heater instantly faults out with E-04 “pump failure”.
Rx Bytes [3] & [17] both report the value 05h.

PUMP NOT CONNECTED

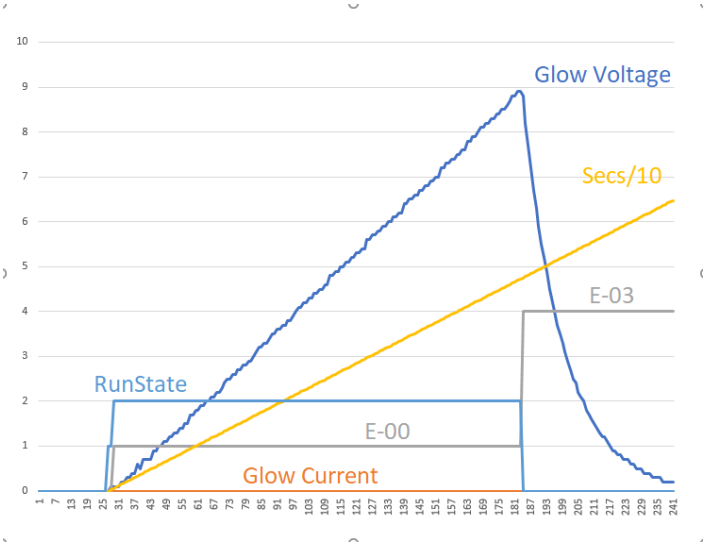
The heater circuit includes monitoring of the voltage after 12V has passed though the pump. If the pump is not connected, this voltage will never approach 12V when the pump is not driven.

The heater instantly faults out with E-04 “pump failure” when a start is attempted without the pump attached.
Rx Bytes [3] & [17] both report the value 05h.

GLOW PLUG NOT CONNECTED

If the glow plug is not connected, or open circuit, the heater will attempt to start in the usual manner, gently increasing the voltage across the plug until ~9V is reached.
If no glow plug current is detected when the voltage reaches the maximum, the heater faults out with E-03.
Rx Bytes [3] & [17] both report the value 05h.

- Orange trace is the glow plug current, as sensed by the heater.
- Blue trace is glow plug voltage.
- Pale blue trace is the RunState of the heater (Rx Byte[2])
- Grey trace is the ErrState of the heater (Rx Byte[3])
- Yellow trace indicates elapsed time in seconds divided by 10 (for scaling).

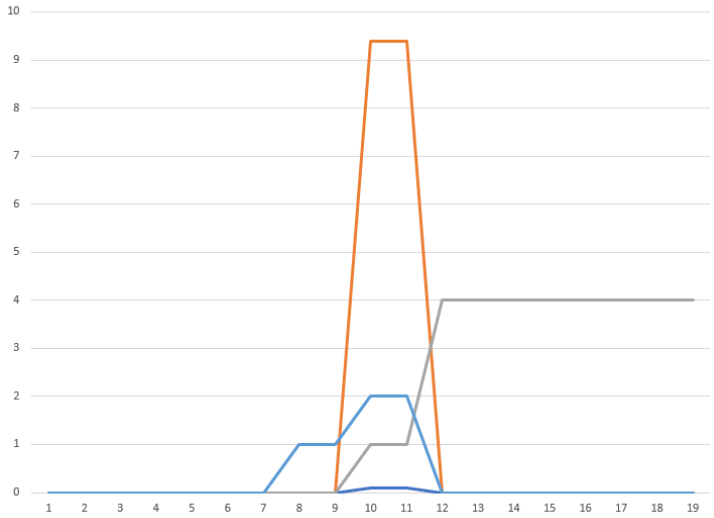


GLOW PLUG SHORTED

If the glow plug is shorted, the heater will instantly fault when a start is attempt with E-03.
Rx Bytes [3] & [17] both report the value 05h.

The glow plug current immediately rises to the 10A limit, and minimal glow plug voltage exists:

- Orange trace is the glow plug current, as sensed by the heater.
- Pale blue trace is the RunState of the heater (Rx Byte[2])
- Grey trace is the ErrState of the heater (Rx Byte[3])
- Blue trace is glow plug voltage.



OTHER OBSERVATIONS

TRANSITION TO RUN STATE 5 – NORMAL RUNNING

When the heater exchanger exceeds 100°C, the heater seems to consistently switch to Run State 5 (Rx Byte[2]).

This matches the value reported by the heater in Rx Byte[20]…

TRANSITION TO SHUTDOWN – RUN STATE 0

The heater consistently shuts down when the heater exchanger has cooled below 55°C.

GLOW PLUG

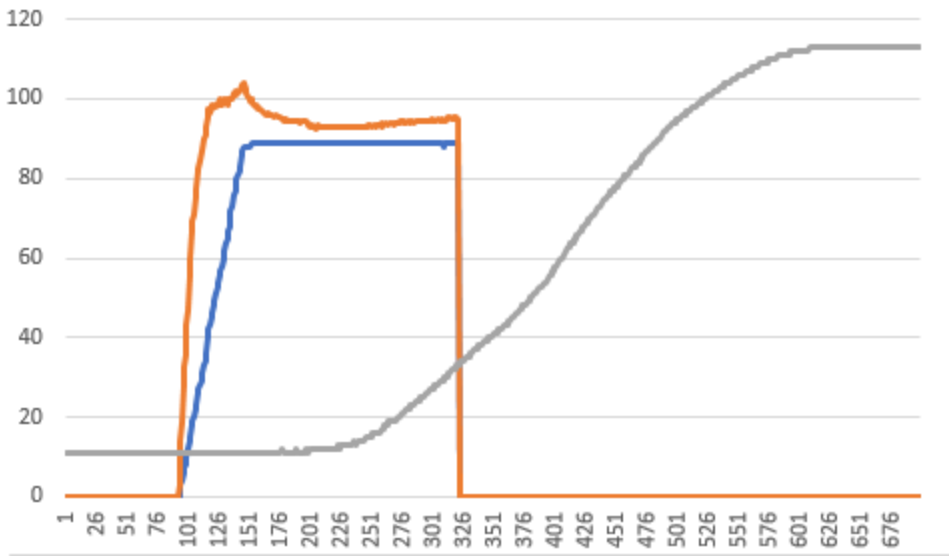
It appears the heater primarily controls the current going to the glow plug, at 10amps.
When cold, the current rises to 10A quite quickly, but the voltage slow climbs to ~9V.

This generally agrees with the resistance increasing with temperature, needing more voltage to get the same current.

The glow plug is driven by a buck regulator in the heater, which receives a 40kHz PWM signal from the microcontroller.

The following image shows a typical start:

- Orange trace is the glow plug current, as sensed by the heater.
- Blue trace is glow plug voltage, as sensed by the heater.
- Grey trace is the heat exchanger's body temperature, as sensed by the heater.



SYSTEM VOLTAGE

The heater PCB can safely operate with a 24V or 12V battery system, just needing an LCD or LED controller to define the correct operating voltage to avoid over or under voltage errors.

The 5V for the logic is created via a buck regulator, as is the glow plug.

The fan is driven by a PWM signal meaning a 24V supply can run a 12V fan given appropriate reduction on PWM duty cycle. The required fan speed is regulated using feedback from a hall effect sensor driven by magnets embedded in the room air intake impeller.

The pump being a momentary impulse action could possibly run with 24V OK, but I note 24V or 12V pumps can be purchased.

The resistor scaling on the ADC inputs will cope with 32V before saturating.

Finally, the PCB is labelled 24V against the input power pins.

THERMOSTATIC OPERATION

The LCD Controller can be switched to/from thermostatic mode by holding down the “Settings” and “Up” arrow buttons.
The LCD shows x°C for thermostat mode, “x.x Hz” for pump frequency mode.

The LED seven segment controller likewise can be switched to/from thermostatic mode by holding down the “Setting” and “Up” arrow buttons.

The Rotary Controller can be switched to/from thermostatic mode by holding down the ON key for around 8 seconds, it does not matter if the heater is running or not.
When running, the knob lights red for thermostatic mode, blue for fixed pump frequency “Hz mode”.

When in thermostatic mode, the heater runs at maximum fan and pump speeds until the sensed temperature is 1°C higher than the set value.
Note that the Rotary Controller only sets the target temperature in 3°C increments Tx Byte[4].
The heater then backs off to the minimum pump and fan speeds until the temperature falls 1°C below the set temperature.

This cycle repeats, going from maximum to minimum heater operation, spanning the 2°C window.

HEATER PROGRAMMING

The LCD or LED seven segment Controllers can change the max/min fan and pump speeds.
i.e. the top end and bottom end fuel mixtures.

If an LCD or LED Controller is replaced by a Rotary Controller, the heater then operates within the limits that were defined by the LCD/LED controller, not the values advertised by the Rotary Controller in the data protocol!
Therefore it appears the heater unit stores these limits in non-volatile EEPROM memory within the microcontroller, and these can only be changed by using an LCD/LED Controller.
You do not need to enter the settings menu for this action to take place.

Likewise, the 12/24V setting is stored.
My Rotary Controller advertises 24V in the protocol, but does not cause low voltage faults.
However if I set 24V with the LCD, the heater soon faults out with an Undervoltage Fault – E-01.

In terms of the protocol, the only difference appears that Tx Byte[0] is 76h for the LCD Controller, and 78h for the Rotary Controller.
78h appears to inhibit these aspects of the protocol!

PUMP PRIMING

If Tx Byte[17] is set to the value 5Ah, the pump will run for the purposes for fuel priming.

From the Rotary Controller, priming is initiated by holding the OFF button for a few seconds (with the heater off).

From the LCD/LED Controller, priming is initiated by holding both the “OK/Enter” and “Down” buttons until HA-1 appears on the LCD.
The pump can then be turned on by pressing “Up”, and off by pressing “Down”.
The relay within the heater should also be heard clicking as it supplies and removes power to the pump, fan and glow plug circuits.

THE "PICCOLO"

This entire quest began soon after I first installed my heater.

Once up to full power or should I say, "full noise", a horrible resonant whistling was coming from within the heater.
This was very objectional, far worse than the dull low frequency roar immediately prior.
The wife was not happy, nor was I.

Fiddling about, it was found that crimping the fuel line would cause the heater to stop whistling, yet still run OK.

I sought knowledge from Dr Google and found that LCD Controllers can be used to control the air fuel mixture – aha!

As I only had a Rotary Controller, I bought one off eBay so I could try backing off the maximum fuel rate.

Soon after I had the LCD running the protected menu settings were used to wind back the max pump frequency.

A test run soon confirmed the piccolo was now tamed.

At some later stage I replaced the Rotary Controller, and it no longer whistled???

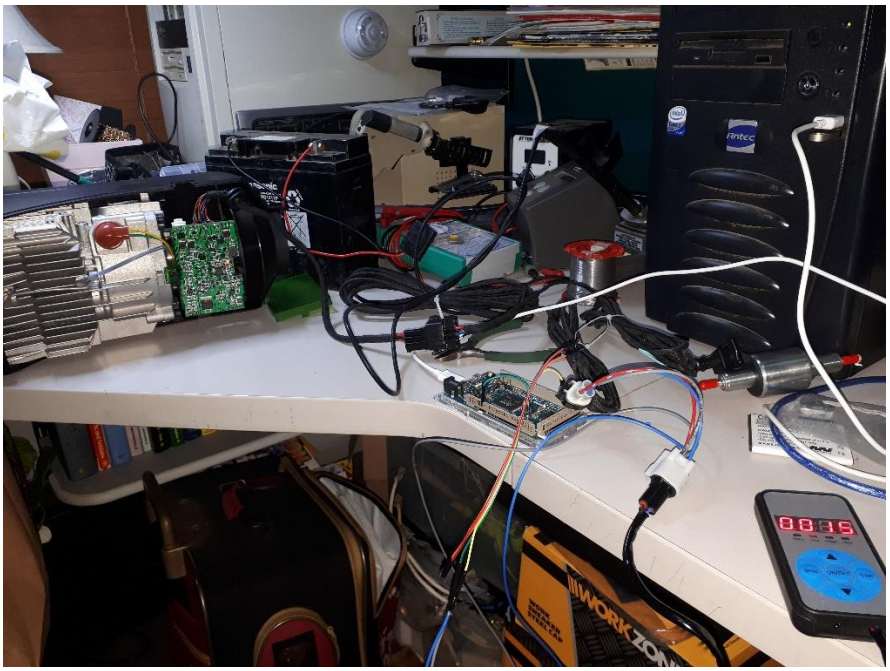
WTF?

The rest is now history and documented in this document.
It is the result of me wanting to properly understand how these things go about their business.
I have learnt a LOT, and I wish to share that knowledge back onto the Internet.

I also would not mind getting an android app working, running the heater via a Bluetooth interface.

Regards,

Ray Jones



A 5kW heater on the “Autopsy Table”.
Not dead though, shiny and new, destined for the shed!
It was an opportunity to test some things like what happens with no fuel, pump disconnected/shorted. Glow plug disconnected/shorted etc.

You can see my Arduino Due patched into the blue wire via a breakout cable I made up with some triangular plug/sockets from eBay.
Data is captured using Putty on a PC, later analysed in Excel: