UNIVERSITY OF EXETER

Design of an Arduino-based GSM Datalogger

CENTRE FOR ENERGY AND THE ENVIRONMENT

Software Document 40

January 2019

Report Name:	Design of an Arduino-based GSM Datalogger
Author(s):	T.A. Mitchell
Report Number:	Software Document 40
Publication Date:	January 2019

CENTRE FOR ENERGY AND THE ENVIRONMENT

UNIVERSITY OF EXETER,
HOPE HALL,
PRINCE OF WALES ROAD,
EXETER,
EX4 4PL

T: 01392 724143
W: www.exeter.ac.uk/cee
E: t.a.mitchel@exeter.ac.uk

Rev. No.	*Comments*	*Approved By*	*Date*
0	Final Draft	-	-

Contents

Management Summary 5
1. Introduction 6
2. Hardware 6
3. Software 7
3.1 Header 7
3.1.1 Libraries 8
3.1.2 Definitions and Global Variable Declaration 8
3.2 Setup 8
3.3 Main Loop 9
3.4 Other Functions 9
3.4.1 Date Time 9
3.4.2 EEPROMToFile 10
3.4.3 EEPROMToGSM 10
3.4.4 EEPROMReadUint16 10
3.4.5 EEPROMReadUint32 10
3.4.6 EEPROMWriteUint8 10
3.4.7 EEPROMWriteUint16 10
3.4.8 EEPROMWriteUint32 10
3.4.9 FileToEEPROM 10
3.4.10 Find Next 10
3.4.11 FTPSend 10
3.4.12 FTPTest 11
3.4.13 GSMRead 11
3.4.14 Interval Check 11
3.4.15 ReadButton 11
3.4.16 ReadCharFromFile 11
3.4.17 ReadGSMResponse 11
3.4.18 ReadNumberFromFile 11
3.4.19 ReadPulseInputs 11
3.4.20 ResetGSM 11
3.4.21 UpdateGSMSigStr 11
3.4.22 WriteReadsToEEPROM 12
3.5 Recommendations for Future Improvements 12
4. User Guide 12
APPENDIX 1 Component Listing 15

APPENDIX 2 Pin Usage 16
APPENDIX 3 Main Code Listing 16
APPENDIX 4 EEPROM_Write code 41
APPENDIX 5 GSM code 42
APPENDIX 6 Code Version History 43
APPENDIX 7 EEPROM Map 44
APPENDIX 8 Typical AT Commands 46
APPENDIX 9 GSM Account Details 47
APPENDIX 10 Example Configuration File 48
APPENDIX 11 Pulse Count Inputs 49

Management Summary

This report documents the design and construction of a low-cost datalogger that transmits logged data to a fileserver over a mobile telephone network. It is a development of an earlier MODBUS datalogger based around an Arduino Nano microcontroller, to which a mobile telephone modem module has been added.

The design allows the logging of up to four pulse inputs instead of MODBUS data. Development of a datalogger that combines the MODBUS and mobile communication functionality would be possible, but owing to memory limitations of the microcontroller would either require optimisation of the current code or replacement of the Arduino Nano with the closely compatible Arduino Mega, which has more memory for both code and data.

The design offers potential for very low cost implementation of datalogging with remote retrieval capability. Proprietary dataloggers most typically connect to wireless internet networks. Security settings on networks such as those found in offices and schools, and the complexity of these networks, can complicate the use of such devices. The Centre's existing solution for remote data retrieval is a multi-channel datalogger costing about £500. GSM modules are available from about £7; the parts cost of a complete datalogger is about £65.

1. Introduction

This report documents the design and construction of a low-cost datalogger that transmits data to a fileserver. It has been developed from the MODBUS datalogger described in Software Document 39 [1]. A Global System for Mobile communications (GSM) communications module has been added, which allows communications over a "2G" mobile phone network. Both designs are based around the Arduino Nano microcontroller.

The prototype was designed for a specific installation, and other modifications from the previous design include the logging of up to four pulse inputs instead of MODBUS data, provision of a DIN rail mounted enclosure and replacement of the liquid crystal display (LCD) with a more compact organic LED (OLED) display. Code relating to MODBUS has been commented out to allow relatively easy reinstatement. However, the 31.5 kB of flash memory available on the Arduino Nano for the compiled program is almost fully utilised, as are the 1 kB EEPROM used for data storage and the 2 kB SRAM used for declared variables. Development of a MODBUS GSM datalogger would either require significant optimisation of the current code, or (perhaps simpler) replacement of the Arduino Nano with the closely compatible Arduino Mega, which expands the available flash memory to 248kB, EEPROM to 4 kB and SRAM to 8 kB.

The design offers potential for very low cost implementation of datalogging with remote retrieval capability. Proprietary dataloggers most typically connect to wireless internet networks [2]. Security settings on networks such as those found in offices and schools, and the complexity of these networks (e.g. multiple subnets), can complicate the use of such devices. The Centre's existing solution for remote data retrieval is the Controlord Gigalog S [3] with the optional GSM modem module fitted. This 16-channel datalogger retails at about £500 and although low cost compared to other comparable devices such as products by Datataker and Campbell Scientific, still represents a significant outlay—particularly for applications where monitoring locations are dispersed and sensors cannot be wired back to a central point. GSM modules compatible with Transistor-transistor logic (TTL) level serial communications (such as that available on the Arduino's digital pins) are available from about £7; the parts cost of a complete datalogger is about £65.

2. Hardware

The design uses the following components (further details are provided in Appendix 1): -

1. *Arduino Nano*: The *Arduino* is an open-source design and the product used is a clone. It uses the CH340G USB driver chip, which differs from an original Arduino and requires a suitable driver to be installed[1].
2. *Real-Time Clock (RTC)*: Whilst the *Arduino* provides timer functions, for a datalogger a reliable real-time clock that will maintain timekeeping even if the device is reset or loses power is required. The RTC circuit incorporates a CR1220 backup battery and connects to the I2C port on the *Arduino*[2].
3. *SD Card interface:* The limited non-volatile EEPROM memory on the Arduino is insufficient to store a long term archive of logged data. The SD card interface enables an SD card to be used for storage, the card can subsequently be removed and the file copied to a PC. Configuration settings for the logger are read from a file on the card. The card interface connects to the SPI interface on the *Arduino*.

[1] CHG_SEP_2017.zip, available from rebrand.ly/b35d (last accessed 26/2/2018).
[2] A further unused pin provides a square wave reference. The device also contains 56 bytes of flash memory storage, which is not used.

4. *OLED Screen*: A display screen is useful both for debugging during development, and also to provide confirmation messages and information to the user. During normal datalogging operation, the GSM signal strength, current date and time and value of a selected pulse input are displayed. The OLED screen used is a monochrome 128 x 64 pixel display and connects to the I2C port on the *Arduino*. Most function libraries for this type of display allow pixel-level control, but utilise over one-half of the available RAM on the Arduino Nano as a graphics buffer. A text-only function library has been utilised that has far more modest memory requirements.
5. *GSM Module*: The SIM800L EVB V.2.0 module utilised is based around the SIMcom SIM800L surface mount GSM module[3]. It provides peripheral circuitry, connections to power, TTL serial data and reset functions, a micro SIM card holder and U.FL antennae connector.
6. *LM2596 Voltage Converters*: The GSM module requires a well-regulated and stable power supply of 4.6 to 5.2 V able to supply a peak current of 2 A. Whilst peak current draw is transitory and occurs under certain conditions only, if the power supply cannot meet the requirement, operation will not be reliable. Power to the GSM module is provided via an LM2596-based voltage converter adjusted to provide a voltage of 4.85 V, from an in-line power supply rated 9 V DC, 2.5A. The Arduino Nano was initially powered directly from the 9 V supply, but operation was found to be unstable and at one point the OLED overheated. A second LM2596-based voltage converter was added to supply 9 volts to the Arduino.
7. *Pushbutton*: A single pushbutton has been provided to allow user-acknowledgment of prompts (e.g. to set the clock and start logging), and to select which of the pulse inputs is displayed on screen. The Arduino can be reset using the on-board reset button accessible through a hole in the case.
8. *Indicator LED*: A single bi-colour LED is provided to indicate error conditions (red), awaiting user acknowledgment (yellow) and to indicate that a pulse has been received at the input currently displayed on screen (green).

Connections to each component are described in Appendix 2.

The pulse inputs were originally wired between the digital input pin and ground, with the input configured to use the internal pullup resistor. This registered contact closures, but did not work with open collector transistorised meter output circuits. To resolve this, the inputs were re-configuredas floating, with additional terminals providing 9 V (Arduino power supply) and 5 V (Arduino voltage) allowing more flexible configuration of the inputs, as described in Appendix 11.

3. Software

The software was developed in the Windows *Arduino* programming environment, which provides a rudimentary library manager, compile-time debugging messages and facilities for uploading code to the *Arduino* and receiving diagnostic output written by the *Arduino* to the serial (USB) port during program execution.

Each section of the program is described below. A program listing is provided in Appendix 3. A version history is given in Appendix 6.

3.1 Header

This section includes references to libraries and global variable declarations

[3] The "L" variant has FM radio functionality (although the evaluation board ("EVB") implementation does not break out any audio connections, precluding voice calls or the FM radio). A "C" variant is available that provides Bluetooth in place of FM radio.

3.1.1 Libraries

Publicly available libraries were used to accomplish "low level" tasks without reinventing the wheel:

- *EEPROM.h* Standard Arduino library for accessing the EEPROM memory.
- *SPI.h* Standard Arduino library for communications on the SPI port (used by the SD card);
- *SD.h* Standard Arduino library for reading and writing from and to an SD card;
- *Wire.h* Standard Arduino library for communications on the I2C port (used by the RTC and OLED display);
- *SSD1306Ascii.h* [4] Contributed library for text-only use of the OLED;
- *SSD1306AsciiWire.h* [4] Contributed library adding support to *SSD1306Ascii.h* for displays equipped with an SPI interface;
- *HCRTC.h* [5] Contributed library for reading and setting the RTC;
- *SoftwareSerial.h* Standard Arduino library for r serial communications using any digital pins (standard Serial communications are only available on pins D0 and D1).

Documentation of some libraries is poor and compatibility with pre-built modules (which are frequently supplied with no documentation) is to some extent a matter of trial-and-error. Several alternative libraries were tested for both the RTC and OLED before a suitable library was identified.

3.1.2 Definitions and Global Variable Declaration

One way of specifying what is effectively a global constant is to use the #define construct. This simply specifies a *name* and a string that the compiler replaces any instances of *name* with at compile time. Given the very limited data memory on the *Arduino*, this is advantageous as the information is them held in program memory rather than data memory.

Global variables are declared, i.e. those which may need to be accessed during both setup and the main program loop. These include digital pins used by the various components, timeouts and array sizing, also the addresses of key information held in the EEPROM memory.

The OLED, RTC and software serial libraries are also initialised.

3.2 Setup

setup() is a standard *Arduino* function that runs once following the processing of any header information. For the datalogger, it includes the following:

1. Declaration of local variables only used within the setup function, mainly to parse data read from the configuration file and to set the RTC.
2. Further initialise the OLED (setting its pixel dimensions and communications address), SD card and RTC interfaces, initialise serial communications to the GSM and initialise LED and button pins.
3. Attempt to read the configuration file *gsmlog.cfg* from the SD card. Return an error if the card cannot be read. If the file is missing, create an example file for the user to edit and do not proceed further. This example file is composed from information held in EEPROM. The first, general, part of the file is taken from an image written to EEPROM using the program *EEPROM_Write_V1.ino* (this simply copies an incomplete *gsmlog.cfg* file from the SD card to the EEPROM, and is listed in Appendix 4). The second part of the file, containing the commands required for GSM operation, is written using values held in EEPROM that were previously read from a complete *gsmlog.cfg* file.

4. Read parameters from the configuration file into variables or EEPROM. These include an option to set the clock, datalogger parameters (logging rate, file format, number of pulse inputs, pulse input weightings and names, file upload frequency, starting readings, and option to start on keypress). Further parameters specify all of the commands used by the GSM modem to upload data. Note error checking is minimal; this could cause unexpected behaviour if numerical values in the file are out of range. All of the GSM modem commands and the last saved meter readings are stored in EEPROM[4]; the former so they can be used as the starting readings following a power outage, and the latter because lengthy text data would not fit in available SRAM memory.
5. Checks are performed that the data read from the *gsmlog.cfg* file will fit in the EEPROM memory. If not, an error message is displayed ("EEPROM Overflow!")
6. If the option to set the clock was specified in the configuration file, a prompt is displayed on the LCD and the clock is reset when a button is pressed.
7. If the option to start on the press of the button was specified in the configuration file, a prompt is displayed on the LCD and the program resumes when the button is pressed (this option is intended to allow the datalogger to be synchronised with the meter.
8. If the clock set or start on button press options were specified, modify the configuration file so these options are not implemented on subsequent resets.
9. Display an information message confirming the baud rate, parity, stop bit, logging rate and new file creation options.
10. Initialise timers used to read the RTC.

3.3 Main Loop

loop() is a standard *Arduino* function that runs continuously as a loop once the setup() function has been completed. For the datalogger, it includes:

1. Only run any code in the loop if no errors occurred during setup (i.e. a *gsmlog.cfg* file was successfully read).
2. Read the RTC. Note if the seconds or minutes have incremented.
3. If the time has incremented by at least the logging interval, read the RTC, determine the data filename, and write data to the file.
4. If the hour or day has incremented, or previous transmission was unsuccessful, transmit the previous datafile to the FTP server. Update the meter reads held on EEPROM which are restored after a power outage[5].
5. Read the current GSM signal strength, read the pulse inputs and pushbutton state and update the information on the OLED[6].

3.4 Other Functions

These are functions that are called from the setup() or loop() functions.

3.4.1 Date Time

By default, when new files are written to the SD card, the datestamp is set to 1/1/2000 01:00. Modified files retain their previous datestamp. To set the datestamp correctly, this function is called

[4] Appendix 7 lists the data held at each EEPROM address

[5] The readings are only stored at the file upload interval, as the EEPROM is limited to a life of about 100,000 write cycles.

[6] Note: all of these actions apart from reading the signal strength are also performed periodically during potentially long operations such as uploading data to the FTP server, to ensure the display is kept up to date and input pulses are not missed.

to set the datestamp to the current value of the RTC. Note: if the configuration file is altered by the program, the datestamp is not updated so it still indicates when the file was last edited by the user.

3.4.2 EEPROMToFile

Copies the number of bytes specified by the second parameter starting at the address stated in the first parameter to the currently open datafile on the SD card (referred to by the object *DataFile*)

3.4.3 EEPROMToGSM

Copies the number of bytes specified by the second parameter starting at the address stated in the first parameter to the GSM modem. A third parameter, if true, sends carriage return (ASCII 13) plus linefeed (ASCII 10) characters after the data have been sent.

3.4.4 EEPROMReadUint16

Reads and returns a 16-bit unsigned integer ("unsigned int" datatype) from the specified EEPROM address and the subsequent address.

3.4.5 EEPROMReadUint32

Reads and returns a 32-bit unsigned integer ("unsigned long" datatype) from the specified EEPROM address and three subsequent addresses.

3.4.6 EEPROMWriteUint8

Writes an 8-bit integer ("byte" datatype) specified in the second parameter to the EEPROM address specified in the first parameter. Creates an error condition if the address extends outside of the available EEPROM space.

3.4.7 EEPROMWriteUint16

Writes an 16-bit integer ("unsigned int" datatype) specified in the second parameter to EEPROM, starting at the address specified in the first parameter. Creates an error condition if the address extends outside of the available EEPROM space.

3.4.8 EEPROMWriteUint32

Writes a 32-bit integer ("unsigned long" datatype) specified in the second parameter to EEPROM, starting at the address specified in the first parameter. Creates an error condition if the address extends outside of the available EEPROM space.

3.4.9 FileToEEPROM

Copies the number of bytes specified by the second parameter from the currently open datafile on the SD card (referred to by the object *DataFile*) to EEPROM, starting at the address stated in the first parameter. Creates an error condition if the address extends outside of the available EEPROM space.

3.4.10 Find Next

Accepts a reference to an open file and the ASCII code of a character to search for. Returns the next instance of this character forward from the current position in the file. Used to skip over comments when reading the configuration file.

3.4.11 FTPSend

Creates a new file on the FTP server, requests the maximum length of a data packet and copies bytes from the file on the SD card to the FTP server until the entire file has been transmitted. Closes the active FTP session.

3.4.12 FTPTest

Obtains current signal strength. If signal strength exceeds 10%, send GSM test commands to the GSM modem and await a valid response each time. If any response is invalid, the GSM modem is reset by holding the reset pin low for 1 s; after a 30 s delay the signal strength is requested, if greater than 10% command echo is disabled and GSM initialisation commands are sent.

If the GSM test or initialisation commands above are successful, FTP initialisation commands are sent in preparation for the transmission of data. Further information on GSM modem commands is included in Appendix 8.

3.4.13 GSMRead

Read a single byte response from the GSM modem with the specified response timeout (in ms).

3.4.14 Interval Check

Accepts a time datum and interval, compares to the current timer value (milliseconds since board booted up, as an unsigned long value), and returns 1 if the interval has been exceeded or zero otherwise. Accounts for the case where the timer has exceeded 65,535 and restarted counting from zero.

3.4.15 ReadButton

Checks a minimum interval (500 ms) has elapsed since a button press last triggered actions, this stops multiple actions due to the button being held down longer than it takes to increment around the loop() function once. If this condition is met, return 1 indicating the button was pressed.

3.4.16 ReadCharFromFile

Read a single character from the open file on the SD card to a temporary string variable.

3.4.17 ReadGSMResponse

Read the specified number of characters from the GSM modem (using the GSMRead function) and compare them to data in EEPROM starting at the specified address. GSM responses start with carriage return (ASCII 13) plus linefeed (ASCII 10) and these are read in addition to the specified number of characters. If the response matches the EEPROM data, 1 is returned.

3.4.18 ReadNumberFromFile

Reads a numerical value from the open file on the SD card into a string array. The maximum number of digits to read is specified.

3.4.19 ReadPulseInputs

Checks the state of each of the pulse inputs and increments the meter reading for the input if state has changed to low and the debounce period has been exceeded. Flashes the LED green if the input being checked is that currently displayed on the OLED. Call ReadButton and update the text on the OLED.

3.4.20 ResetGSM

Reset the GSM by setting the GSM module's Reset pin low for 1 second; wait 30 s to re-negotiate communications to the network.

3.4.21 UpdateGSMSigStr

Request the signal strength from the GSM modem and read the response. Valid responses are 0-31; these are multiplied by 3 to give an approximate percentage scale, the value of which is returned by the function. Other values are forced to 0.

3.4.22 WriteReadsToEEPROM

Writes the current value of each input to EEPROM.

3.5 Recommendations for Future Improvements

Issues not fully resolved or implemented include:

1. A lack of exhaustive error checking (for invalid values entered by the user) when reading the configuration file. This could significantly complicate reading data from the file and would only be possible with an Arduino with more flash memory.
2. No checking for a full SD card. Again this would only be possible using a more low level and complicated SD card library.

4. User Guide

1. *Connect the datalogger to the pulse outputs of meters to be monitored*: Connect each meter to the +9V terminal, and to one of the inputs via a 470Ω resistor; also connect the input to the ground(⏚) terminal via another 470Ω resistor. This will present a voltage of 9 V to the pulse output switch, and current of 10 mA through the switch. See Appendix 11 for further information.
2. *Insert a SIM card*: Obtain a SIM card from an operator offering GSM (2G) communication (at the time of writing all except "3" in Great Britain. Register the SIM card and insert it into the GSM module.
3. *Prepare FTP server*: Obtain server space on an FTP server offering plain FTP (not SFTP). The University of Exeter FTP officially disallows plain FTP and it was blocked for a while, but is now working again. Create a new folder to receive the data (transmission will fail if the filename to be used already exists on the server, so it is recommended that each datalogger uploads to its own folder).
4. *Connect GSM aerial*: Connect the aerial to the SMA connector. NB: Do not turn the datalogger on without an aerial connected as the GSM module can be damaged[7].
5. *Arrange power supply to datalogger*: Connect the 2.1 mm DC power socket. to the 9 volt, 2.5 amp DC power supply.
6. *Prepare a gsmlog.cfg* file to configure the datalogger (an example file is listed in Appendix 10):

   The first line in the file is `Update Time (N/Y)`: Setting to `Y` enables the real-time clock on the datalogger to be set when the logger is started. Set to `N` (also the default for any other character) if the clock is already correct. Note: the clock is only set the first time the file is read (when the datalogger is powered on or reset); the file is then modified so that the clock is not set if the datalogger is subsequently reset or power is lost and restored.

   The second line is `New Time (YYMMDDHHMMSS)`: This is the date and time that will be set if the option to set the clock is specified, starting with the two last digits of the year, then two digits indicating the month and so on for day, hour, minute and seconds. Set it a little while into the future, you will be prompted to press a button to set the clock to this value when the datalogger is started. If the clock is not being set, this line must still be included in the file, but the value is ignored.

[7] If the Arduino is powered via the USB connector, all components except the GSM modem are powered, and an aerial is not necessary. This is useful for testing and development.

The next line is `Wait for keypress to start (N/Y):`, set to `N` to start immediately, or to `Y` to wait for a keypress (e.g. to wait for a meter reading to match that set later in the configuration file).

The next line is `Max Pulse Input No (1/2/3/4):`. Set the highest number input to be used, to avoid logging superfluous data from unused inputs.

For each pulse input, there is then a line `Pulse Weight # (1-65535 units per pulse):`. Enter 1 to increment the reading by 1 for each contact closure, or by more than 1 to increment in larger steps (e.g. if a meter issues a pulse every 5 litres or 2 kWh). Since the current reading is stored as a long integer, it is not possible to require multiple pulses per unit of consumption, such scaling must be done manually.

For each pulse input, there is then a line `Pulse Input 1 Name (1 to 8 characters):`. Enter a name of 1 to 8 characters which will identify the input in the data files.

The next parameter is **`Pulse Debounce (1-255ms):`**. Specify the minimum time for which the state must change to log a pulse (in milliseconds). Note: very short pulses may be missed due to the size of the main code loop, tests should be carried out when commissioning the logger.

The next parameter is **`Log Rate (1-65535 s):`**. Specify how often the meter reading is to be recorded to file. It is also the retry interval for FTP transmission. Typical rates might be 60, 300 or 1800 seconds.

The next parameter is **`Upload Rate (H/D):`**. This is the frequency at which files are uploaded to the FTP server, either `H` for hourly or `D` for daily. It is also the frequency at which meter readings are saved to EEPROM for restoration after a power outage. It is recommended that the Log Rate is set to a significantly smaller interval to allow the FTP operation to retry several times if it is unsuccessful.

The next parameter is **`Multi Column Format (N/Y):`**. This sets the datafile format. If `N` is specified, there is one meter reading per line, preceded by the input name and the date and time (in the form DD-MM-YY HH:MM:SS). This format is most suitable for reading into database system (it is compatible with the TEAM energy management system's Satchwell Schneider BMS importer). If `Y` is specified, a header line lists the input names, there is then one line per logging interval with the date and time (in the form DD/MM/YY HH:MM:SS) followed by the meter reading(s). This is most suited to import into a spreadsheet. Both formats are comma delimited.

The parameter is `Specify Starting Readings (N/Y):`. Set to Y to specify specific starting meter readings, e.g. when initially synchronising the datalogger to the meters. Note: the readings are only set the first time the file is read (when the datalogger is powered on or reset); the file is then modified so that the readings are not reset if the datalogger is subsequently reset or power is lost and restored.

There follows one line (`Meter Reading Input 1 (0-999999):`) for each input in use, specifying each starting reading. These must always be present, but are ignored if `Specify`

`Starting Readings` is set to `N`. If when logging the reading exceeds 999,999, it resets to zero.

There are then additional lines specifying each of the GSM modem commands, as follows:

- Commands to retrieve current signal strength
- Communications test commands (if any of these fail to return the specified response, the modem is reset)
- GSM initialisation commands (e.g. the Access Point (APN) information) for the mobile phone network
- FTP initialisation commands (e.g. the FTP server login details and file path)
- A series of commands to open a file on the FTP server, write data and close the connection.

Further details of these commands are provided in Appendix 8.

7. *Switch the datalogger on:* Insert the SD card prepared in step 6 and provide power as in step 5. The datalogger is initialised as follows:
    a. *The SD card is checked.* If the SD card cannot be read, an error message `SD Card Error` is displayed and the red LED lights. Datalogging is not possible. Check that the card is formatted as FAT16 or FAT32.
    b. *The gsmlog.cfg file is read.* If the file is not found but an example file could be written, the error message `Config file not found, created` is displayed. If the file is not found and an example file could not be written, the error message `Config file not found, failed` is displayed. If the file is present but could not be read, the error message `Config file won't open` is displayed. In any of these cases the red LED lights and datalogging is not possible.
    c. If the option to set the clock was specified in the *gsmlog.cfg* file, the message `Press OK to set clock` is displayed along with a yellow LED. Wait until the current time is that specified in the file, then press the OK button momentarily. A confirmation message is shown briefly: `Clock set to` *`DD/MM/YY HH:MM:SS`* (showing the date and time).
    d. If the option to require a keypress to start was specified in the gsmlog.cfg file, the message `Press OK to start` is displayed along with a yellow LED. Wait until the meter reading matches that specified in the file, then press the OK button momentarily.
8. The datalogger enters its normal running mode. The display shows the following:

`GSM Signal 54%`	(GSM signal strength)
`12/12/2018`	(Date)
`12:00:03`	(Time)
`1  54254`	(Value of selected input)

9. Test to ensure that pulses are recorded accurately. See Appendix 11 for troubleshooting experience.
10. If the signal strength is below 10%, FTP transmission will not be attempted. Try relocating the aerial, adding an extension cable if necessary (note: there will be a trade-off between signal loss in the cable and the improved aerial position). The Network LED flashes quickly when searching for a network, and slowly when connected to a network. The Ring LED is normally lit.

11. While the datalogger is running, the pushbutton steps through the available pulse input that is displayed on screen (if only one input is specified, the button has no effect). If a pulse occurs on the input currently displayed on screen the LED flashes green.
12. To retrieve data manually, power down the datalogger , remove the SD card and copy the files to a PC.

REFERENCES

1. ***Design of a Datalogger for MODBUS.*** SWEEG Software Document 39. 2018, T.A.Mitchell.
2. ***Options for Wireless Temperature Logging at the University of Exeter.*** SWEEG Briefing Paper 108. 2016, A.T.Rowson & T.A.Mitchell.
3. ***Gigalog S Datalogger.*** Controlord. http://www.controlord.fr/en/home.htm . Accessed 18/12/2018.
4. ***Text only Arduino Library for SSD1306 OLED displays.*** https://github.com/greiman/SSD1306Ascii. Accessed 3/5/18.
5. ***Real Time Clock RTC library.*** https://github.com/dtu-mekatronik/HCRTC, last accessed 26/2/2018.
6. ***Serial Monitor Deluxe.*** https://www.idogendel.com/en/products/serial-monitor-deluxe, last accessed 19/12/2018.

APPENDIX 1 COMPONENT LISTING

Table A1.1 Component listing.

Item	Detail	Source	Approximate Cost
Arduino Nano	Arduino Nano v3.0 (ATMEGA 328P) CH340G	eBay (scooterboy101)	£2.75
GSM Module	SIM800L EVB V.2.0	eBay (cayin35)	£6.49
Real Time Clock	DS1307 I2C RTC Real Time Clock Module	eBay (abaxus_uk)	£1.62
SD Card Interface	Arduino compatible SD Card Module	eBay (hobbycomponents)	£2.99
OLED Module	I2C OLED display 128 x 64 white	eBay (majikthi5e)	£4.95
Power Stabiliser	LM2596 DC-DC Buck Converter	eBay (umtmedia)	2 @ £4.35
Bicolour LEDs	3 mm, Red/Green common anode	Onecall SC07621[8]	£0.30
Series Resistors[9]	470 Ω ¼ W	Onecall RE03756	2 @ £0.06
Power Supply	9V DC 2.5A	Onecall PW04187	£16.78
Power Socket	2.1 mm DC Power Socket	Onecall AV15144	£0.45
Input Terminals	5 way PCB mounting	Onecall CN15851	2 @ £0.76
Pushbutton	Momentary pushbutton	Onecall SW04408	£0.13
Aerial	GSM Antennae	Onecall RF00692	£9.30
Enclosure	90x58x88mm ventilated DIN mounting	Onecall EN82496	£7.39
Battery	GP CR2032	Onecall BT00125	£1.63
Total			**£65.12**

[8] This device is common cathode, so would require slight code and wiring modification
[9] Further resistors of this type are required to connect the meter to the input terminals.

APPENDIX 2 PIN USAGE

Table A2.1 Pin usage on the Arduino.

Pin	Allocation	Note
D0 (RX)	Spare	Used for built-in USB interface for echoing data to computer during testing; potential future use for RS485 MODBUS
D1 (TX)	Spare	Used for built-in USB interface for echoing data to computer during testing; potential future use for RS485 MODBUS
D2	RX from GSM	Receives data from GSM
D3	TX to GSM	Transmits data to GSM
D4	GSM Reset	Hold Low to reset GSM module
D5	Button	Pushbutton, connect between D5 and Ground
D6	Pulse input 1	
D7	Pulse Input 2	
D8	Pulse Input 3	
D9	Pulse Input 4	
D10 (SPI)	SD Card (CS)	Can use other digital pins, but SD.h always reserves pin D10 anyhow
D11 (SPI)	SD Card (MOSI)	
D12 (SPI)	SD Card (MISO)	
D13 (SPI)	SD Card (SCK)	
A0 (D14)	LED Common Anode	Set permanently high (could wire to +5V instead, or to ground for a common cathode LED)
A1 (D15)	Green LED Cathode	470Ω resistor. For common cathode LED, reverse output state in code.
A2 (D16)	Red LED cathode	470Ω resistor. For common cathode LED, reverse output state in code.
A3 (D17)	Spare	
A4 (I2C)	RTC & OLED (SDA)	RTC address is 0x68 and OLED is 0x3C
A5 (I2C)	RTC & OLED (SCL)	Addresses may vary between similar modules
A6	Spare	Analogue input only on this pin
A7	Spare	Analogue input only on this pin
Vin	7-12 V DC Supply +ve	Power supply to device (to power socket centre pin), and available on a terminal for use to power meter pulse circuitry.
5V	5 V supply from Arduino	Power to RTC, SD, OLED display, and available on a terminal for use to power meter pulse circuitry.
3.3V	3.3 V supply from Arduino	Not Used
RST	Not used	
AREF	Not used	
GND	To RTC, SD, OLED, power supply -ve contact	Availabkle on a terminal for use to power meter pulse circuitry.

APPENDIX 3 MAIN CODE LISTING

This is the code used to provide datalogger and GSM functionality.

```
//T.A.Mitchell May 2018
//Modified for new hardware
//Modbus commands have been commented out for now

//LIBRARIES
#include <EEPROM.h> //EEPROM I/O
#include <SPI.h> //SPI Bus for SD Card
#include <SD.h> //Simple SD card library
#include <Wire.h> //I2C Comms for RTC on A4 and A5
#include <SSD1306Ascii.h> //OLED Display minimalist text library
#include <SSD1306AsciiWire.h> //OLED Display minimalist text library - extensions to support i2c
```

```
#include <HCRTC.h> //RTC
#include <SoftwareSerial.h> //Software serial for comms on any digital pins
//#include <SimpleModbusMaster.h> //Modbus RTU

//DEFINITIONS to insert into code at compile time (so end up in flash memory not RAM)
#define I2CDS1307Add 0x68 //I2C address for the RTC
#define I2COLedAdd 0x3C //I2C address for OLED display

//Indicator LED Pins
#define PinLEDG 16 //Green LED cathode on pin 15 (= A1) - Flash on pulse on input channel 1
#define PinLEDR 15 //Red LED cathode on pin 17 (= A2) - Initalisation Error LED (ON), Set Clock (Flash)
#define PinLEDAnode 14 //Anode pin, could just use permanent +5V

//GSM Pins
#define GsmRST 4 //GSM reset pin, pull low to reset the GSM
#define GsmTx 2 //GSM Tx pin, receive pin on Arduino
#define GsmRx 3 //GSM Rx pin, transmit pin on Arduino

//SD Card Select (SPI bus)
#define SdCS 10 //Set ChipSelect Pin - use 10 as SD.h reserves this pin anyhow

//Pushbutton
#define PinButton 5 //Digital pin connected to button

//Modbus
//#define Timeout 1000 //Maximum time for slave to respond (ms)
//#define Polling 200 //Maximum scan rate of master to allow slave to return to idle (ms)
//#define RetryCount 10 //Maximum retries if slave returns response Timeout or error
//#define TxEnablePin 4 //Pin to set RS485 interface to transmit or receive  (set pin high to Transmit,
sets DE and RE on RS485 PCB high. DE must be high to transmit and RE low to receive)

//Misc
#define LongIntMax 4294967295 //Maximum value of an unsigned long integer, used in timer check in case
        value has reset to zero in the interval
#define ClockInt 100 //Interval to check RTC (ms) to trigger next log to file.  Should happen several
        times a second
#define LcdUdInt 2000 //LCD Update Interval (ms).  Should happen just over once a second so clock
        seconds increment nicely
#define ButInt 500 //Maximum button press interval (ms), stops multiple triggering of button command
        actions due to fast running of code. 500ms is suitable, short enough not to be noticeable by
        user, but long enough to debounce
#define MsgDelay 5000 //Delay (ms) after certain LCD messages before proceeding to next action that
        might write to the LCD, e.g. 5,000ms
#define LEDPulseTime 200 //Time to light LED for to indicate a pulse (note: could light continuously if
        this is longer than debounce
#define EEPROMCFGDataStart 16 //Address of start of sample config file data, first two bytes are length
        of remaining data (bytes 0-15 are 4 x 4 byte meter reads)

//AT Commands
//Read commands from eeprom and write straight to port, the only variables are the output filename and
        number of bytes to write.
//Maximum number of setup commands, used to size arrays.  The number of commands is less than or equal
        to this number and is read from the config file
#define MaxGSMTestCmds 5 //Number of GSM Modem test commands
#define MaxGSMInitCmds 10 //Number of GSM Modem initialisation commands
#define MaxFTPInitCmds 10 //Number of FTP initialisation commands

//GLOBAL VARIABLES
char LoggerStatus = 'H'; //Logger status, initially H meaning wait to synch to full hour before writing
        to file (including after reboot), G = Go (logging)
unsigned int LogInterval = 1; //Logging interval in seconds (overwritten with value from config file)
//byte QtyRegs = 0; //Number of registers defined in config file to be read
byte DispReg = 0; //Input to display on LCD (zero based) - changeable using button
byte UploadRate = 1; //How often to upload data: 1 = Hourly, 2 = Daily
byte NewSave = 1; //Whether an attempt has previoisly been made to FTP the data (used to only save
        readings to EEPROM on first attempt and set appropriate file for save when next new file
        available, and to trigger ftp retry of previous file)
char MultiCol; //Multicolumn format for text files (Y/N)
byte SetupError = 0; //1 if error occurred during setup, logging will not start
File DataFile; //Reference to file
char TempStr10[11]; //Temporary string read from file
char OldFileName[13]; //Filename for previous log write to file
char FileName[13]; //Name of current logging file
char FTPFile[13]; //Filename for previous logging period: file to FTP
unsigned int GsmSig = 0; //GSM Signal Strength %
unsigned int GSMTimeout = 100; //Timeout for GSM response in ms (individual characters)
unsigned int GSMLongTimeout = 5000; //Timeout for GSM response in ms (initial response, may be delayed
        by modem action)
unsigned int GSMVeryLongTimeout = 65535; //Timeout for GSM response in ms (after write operation) (max
        value is 65,535ms)
byte FirstModemCall = 1; //Ensures initialisation of modem and access point profile on first call to
        ftptest or if tests return an error

//EEPROM Addresses
unsigned int CFGFileLen;  //Length of sample config file data in bytes
byte SigStrCmdLen; //Length of signal Strength Command
byte SigStrRespLen; //Length of signal strength response (exc. signal strength parameter)
byte NGSMTestCmds; //Number of GSM test commands
byte GSMTestCmdLen[MaxGSMTestCmds]; //Length of each GSM Test Command
byte GSMTestRespLen[MaxGSMTestCmds]; //Length of each GSM Test Response
byte NGSMInitCmds; //Number of GSM Initialisation commands
```

```
byte GSMInitCmdLen[MaxGSMInitCmds]; //Length of each GSM Initialisation Command
byte GSMInitRespLen[MaxGSMInitCmds]; //Length of each GSM Initialisation Response
byte NFTPInitCmds; //Number of FTP Initialisation commands
byte FTPInitCmdLen[MaxFTPInitCmds]; //Length of each FTP Initialisation Command
byte FTPInitRespLen[MaxFTPInitCmds]; //Length of each FTP Initialisation Response

unsigned int SigStrStartAddr; //Signal Strength command data start address
unsigned int TestCmdStartAddr; //GSM test command data start address
unsigned int GSMInitStartAddr; //GSM Initialisation command data start address
unsigned int FTPInitStartAddr; //FTP Initialisation command data start address
unsigned int FTPSendStartAddr; //FTP Send Commands start address

byte GSMFileCmdLen; //Length of GSM Set Filename command
byte GSMFileRespLen; //Length of GSM Set Filename command response
byte GSMFTPStartCmdLen; //Length of GSM Start FTP command
byte GSMFTPStartRespLen; //Length of GSM Start FTP command response
byte GSMFTPReqCmdLen; //Length of GSM Request FTP Data send command
byte GSMFTPReqRespLen; //Length of GSM Request FTP Data send command response
byte GSMFTPEndCmdLen; // Length of GSM End FTP Data send command
byte GSMFTPEndRespLen; // Length of GSM End FTP Data send command response

byte GSMFTPEndSessLen; // Length of GSM End FTP Session command
byte GSMFTPEndSessRespLen; // Length of GSM End FTP Session command response

//Pulse Inputs
byte PinIn[4] = {6, 7, 8, 9}; //Input pins for the pulse inputs
byte OldInputState[4] = {0, 0, 0, 0}; //Old Input states when last read, so only triggers count on
        leading edge, 0 = high (open)
byte NPulseInputs = 1; //Number of pulse inputs in use - overwritten from config file
byte PulseDebounce = 50; //Pulse Input debounce - overwritten from config file.  Applied to both contact
        closed time and contact open time: conctact must be closed for at least this time.  At least
        this time must also elapse from contact opening to re-closing.
unsigned int PulseWeight[4]; //Pulse weighting, e.g. 10 = 10 units per pulse, possibly not much point
        for electricity and gas, usually fraction of  unit per pulse and the readings are integers
unsigned long PulseReading[4] = {0, 0, 0, 0}; //Meter readings, defaults are used if readings file
        doesn't exist
char PulseName[4][9]; //Pulse input names, first subscript is the number of array elements, the second
        is the size of each element (including null termination)
unsigned long PrevPulseTime [4] = {0, 0, 0, 0}; //Timer of previous input state change used for debounce
byte NewPulse [4] = {0, 0, 0, 0}; //Set 1 if new pulse detected and not yet reached debounce time

//Timer Variables
unsigned long PreviousMillis; //Old timer value used to trigger RTC check
unsigned long PrevLcdUd = 0; //Timer at last LCD update
unsigned int Secs = 65534; //Elapsed seconds (read from RTC) since values last logged, large value
        ensures logging starts immediately
byte OldSecs; //Previous seconds value read from RTC - to detect change in RTC seconds to trigger
        logging on an interval defined in seconds
byte OldMinute; //Previous minutess value read from RTC - to detect change in RTC minutes to keep
        seconds counter accurate during lengthy FTP
byte OldHour; //Used for synch to midnight start - only start logging when hour increments and to
        trigger FTP upload when hour changes
byte OldDay; //Used to trigger FTP upload when day changes

//Previous button Press
unsigned long OldButTime = 0; //Time of last button press

//Modbus Error Flags
//byte MErrorMin = 1; //MODBUS error condition, set to zero (no error) whenever modbus reading
        successful for all registers and reset to 1 (error) after each write to file

//MODBUS data structures
//#define TotalNoOfRegisters 10 //Max number of registers to log, used to allocate memory, same as total
        packets as all packets are sized for one register

// Create an array of Packets to be configured
//Packet Packets[TotalNoOfRegisters];

//Create an array to hold returned data in its raw form of unsigned integers
//unsigned int HoldingRegs[TotalNoOfRegisters];

//byte RegOffset = 0; //Bugfix for values being returned in wrong array subscript in HoldingRegs.
        Determined by testing for each baud and parity setting.  0 if no adjustment, 1 adds one to
        subscript and highest number becomes zero

//INITIALISE DEVICES through Function declarations
HCRTC HCRTC; //Initialise RTC library first as it includes a Wire.begin(); statement
SSD1306AsciiWire oled; //Initialise OLED display
SoftwareSerial gsm(GsmTx, GsmRx); //Comms to GSM

//__________

//Setup Function reads or creates a config file
//Sets up the serial port as modbus slave
//Updates the RTC if the config file requests this (then writes to config file to stop future RTC update
        on reset)

void setup() {
  ////Local variables for config
  //unsigned long SerBaud; //Baud Rate
  //byte SerParity; //Parity N/O/E = None/Odd/Even
```

```
//byte SerStop; //1 or 2
//byte SerConfig; //Value to set Data bits, parity and stop bits
byte DataRead; //Raw data from file
//unsigned int SlaveId, SlaveReg, LineStartPos, x = 0; //Parameters to read slave ID and registers
      from file
//byte Func, InvalidEntry = 0, CommaCount = 0; //Various variables used to read register parameters
       from file
byte RtcData[6]; //Store elements of date and time to write to RTC to set it
byte i; //Counter to loop over characters

//Initialise RTC
//Note the RTC libraries for RTC and OLED issue wire.beginTransmission(Address) , wire.send ,
      wire.endTransmission() as required
HCRTC.RTCRead(I2CDS1307Add);

//Initialise GSM serial comms
gsm.begin(4800); //##Try faster baud rate, gsm should autodetect
//Serial.begin(9600); //for testing, e.g. Serial.println(F("Test")); writes out to serial monitor

//OLED Display
oled.begin(&Adafruit128x64, I2COLedAdd); // set up the OLED's number of columns and rows
oled.setFont(System5x7); //Standard font, at x2, 11 characters per line
oled.clear();
oled.set2X(); //Double font size
oled.setCursor(0,2); //column pixel, row in 8 pixel blocks. For larger font, there are 4 lines of
      text.  2 centres error messages vertically

//Initialise LED Pins
pinMode(PinLEDAnode, OUTPUT);
pinMode(PinLEDG, OUTPUT);
pinMode(PinLEDR, OUTPUT);

digitalWrite(PinLEDAnode, HIGH); //+5V supply
digitalWrite(PinLEDG, HIGH); //Cathode high = off
digitalWrite(PinLEDR, HIGH); //Cathode high = off

//Initialise Button Pin
pinMode(PinButton, INPUT_PULLUP);

//Initialise Pulse Input Pins
//Wire 10k resistor between each input and ground
//Wire the meter between +5V and input
pinMode(PinIn[0], INPUT);
pinMode(PinIn[1], INPUT);
pinMode(PinIn[2], INPUT);
pinMode(PinIn[3], INPUT);

//Initialise RS485 Pin
//pinMode(TxEnablePin, OUTPUT);
//digitalWrite(TxEnablePin,LOW);//Receive 485

//Initialise SD card pin
pinMode(SdCS, OUTPUT);
digitalWrite(SdCS, HIGH);

//Initialise GSM reset pin
pinMode(GsmRST, OUTPUT);
digitalWrite(GsmRST, HIGH);

//Retrieve length of sample config file from EEPROM
CFGFileLen = EEPROMReadUint16(EEPROMCFGDataStart);

//Open the SD Card
if (!SD.begin(SdCS)) //If fail to initialise SD comms trigger error condition and message
{
  oled.println(F("  SD card"));
  oled.println(F("   error"));
  digitalWrite(PinLEDR, LOW); //RED LED ON
  SetupError = 1;
  //Exit setup
  return;
}
else //card is present
{
  //Does the config file exist?
  if (!SD.exists(F("gsmlog.cfg"))) //If config file missing create an example and trigger error
       condition and message
  {
    //If no config file, create an example
    //from the data on the EEPROM
    SdFile::dateTimeCallback(DateTime); //Set datestamp for file
    DataFile = SD.open(F("gsmlog.cfg"), FILE_WRITE); //create example file
    if (DataFile)
    {
      //Create example
      //First part of file (above GSM commands)
      EEPROMToFile(EEPROMCFGDataStart + 2, CFGFileLen); //First two bytes are length of remaining data
       so not transferred

      //Second Part of file (GSM Commands)
      //Signal Strength Check
```

```
unsigned int NextAddress = EEPROMCFGDataStart + 2 + CFGFileLen; //Next EEPROM Address = CFG File
 template start address + 2 bytes for length of that data plus the data itself
SigStrStartAddr = NextAddress;
DataFile.print(F("Sig Str L: "));
SigStrCmdLen = EEPROM.read(NextAddress);
DataFile.println(SigStrCmdLen);
NextAddress ++;

DataFile.print(F("Sig Str C: "));
EEPROMToFile(NextAddress, SigStrCmdLen);
DataFile.println(); //New line at end of data
NextAddress += SigStrCmdLen;

DataFile.print(F("Sig Str R D: "));
EEPROMToFile(NextAddress, 1);
DataFile.println(); //New line at end of data
NextAddress ++;

DataFile.print(F("Sig Str R L: "));
SigStrRespLen = EEPROM.read(NextAddress);
DataFile.println(SigStrRespLen);
NextAddress ++;

//Comms Tests
TestCmdStartAddr = NextAddress;
NGSMTestCmds = EEPROM.read(NextAddress);
DataFile.print(F("N Comms Test Cmds (<7): "));
DataFile.println(NGSMTestCmds);
NextAddress ++;

for(byte GSMTestCmdNo = 1; GSMTestCmdNo <= NGSMTestCmds; GSMTestCmdNo++)
{
  DataFile.print(F("L Comms Test C "));
  DataFile.print(GSMTestCmdNo);
  DataFile.print(F(": "));
  GSMTestCmdLen[GSMTestCmdNo - 1] = EEPROM.read(NextAddress);
  DataFile.println(GSMTestCmdLen[GSMTestCmdNo - 1]);
  NextAddress ++;

  DataFile.print(F("Comms Test C "));
  DataFile.print(GSMTestCmdNo);
  DataFile.print(F(": "));
  EEPROMToFile(NextAddress, GSMTestCmdLen[GSMTestCmdNo - 1]);
  DataFile.println(); //New line at end of data
  NextAddress += GSMTestCmdLen[GSMTestCmdNo - 1];

  DataFile.print(F("L Comms Test C "));
  DataFile.print(GSMTestCmdNo);
  DataFile.print(F(" Valid Resp: "));
  GSMTestRespLen[GSMTestCmdNo - 1] = EEPROM.read(NextAddress);
  DataFile.println(GSMTestRespLen[GSMTestCmdNo - 1]);
  NextAddress ++;

  DataFile.print(F("Comms Test C "));
  DataFile.print(GSMTestCmdNo);
  DataFile.print(F(" Valid Resp: "));
  EEPROMToFile(NextAddress, GSMTestRespLen[GSMTestCmdNo - 1]);
  DataFile.println(); //New line at end of data
  NextAddress += GSMTestRespLen[GSMTestCmdNo - 1];
}

//GSM Initialisation
GSMInitStartAddr = NextAddress;
NGSMInitCmds = EEPROM.read(NextAddress);
DataFile.print(F("N GSM Init Cmds (<11): "));
DataFile.println(NGSMInitCmds);
NextAddress ++;

for(byte GSMInitCmdNo = 1; GSMInitCmdNo <= NGSMInitCmds; GSMInitCmdNo++)
{
  DataFile.print(F("L GSM Init C "));
  DataFile.print(GSMInitCmdNo);
  DataFile.print(F(": "));
  GSMInitCmdLen[GSMInitCmdNo - 1] = EEPROM.read(NextAddress);
  DataFile.println(GSMInitCmdLen[GSMInitCmdNo - 1]);
  NextAddress ++;

  DataFile.print(F("GSM Init C "));
  DataFile.print(GSMInitCmdNo);
  DataFile.print(F(": "));
  EEPROMToFile(NextAddress, GSMInitCmdLen[GSMInitCmdNo - 1]);
  DataFile.println(); //New line at end of data
  NextAddress += GSMInitCmdLen[GSMInitCmdNo - 1];

  DataFile.print(F("L GSM Init C "));
  DataFile.print(GSMInitCmdNo);
  DataFile.print(F(" R: "));
  GSMInitRespLen[GSMInitCmdNo - 1] = EEPROM.read(NextAddress);
  DataFile.println(GSMInitRespLen[GSMInitCmdNo - 1]);
  NextAddress ++;
```

```
  DataFile.print(F("GSM Init C "));
  DataFile.print(GSMInitCmdNo);
  DataFile.print(F(" R: "));
  EEPROMToFile(NextAddress, GSMInitRespLen[GSMInitCmdNo - 1]);
  DataFile.println(); //New line at end of data
  NextAddress += GSMInitRespLen[GSMInitCmdNo - 1];
}

//FTP Initialisation
FTPInitStartAddr = NextAddress;
NFTPInitCmds = EEPROM.read(NextAddress);
DataFile.print(F("N FTP Init Cmds (<11): "));
DataFile.println(NFTPInitCmds);
NextAddress ++;

for(byte FTPInitCmdNo = 1; FTPInitCmdNo <= NFTPInitCmds; FTPInitCmdNo++)
{
  DataFile.print(F("L FTP Init C "));
  DataFile.print(FTPInitCmdNo);
  DataFile.print(F(": "));
  FTPInitCmdLen[FTPInitCmdNo - 1] = EEPROM.read(NextAddress);
  DataFile.println(FTPInitCmdLen[FTPInitCmdNo - 1]);
  NextAddress ++;

  DataFile.print(F("FTP Init C "));
  DataFile.print(FTPInitCmdNo);
  DataFile.print(F(": "));
  EEPROMToFile(NextAddress, FTPInitCmdLen[FTPInitCmdNo - 1]);
  DataFile.println(); //New line at end of data
  NextAddress += FTPInitCmdLen[FTPInitCmdNo - 1];

  DataFile.print(F("L FTP Init C "));
  DataFile.print(FTPInitCmdNo);
  DataFile.print(F(" R: "));
  FTPInitRespLen[FTPInitCmdNo - 1] = EEPROM.read(NextAddress);
  DataFile.println(FTPInitRespLen[FTPInitCmdNo - 1]);
  NextAddress ++;

  DataFile.print(F("FTP Init C "));
  DataFile.print(FTPInitCmdNo);
  DataFile.print(F(" R: "));
  EEPROMToFile(NextAddress, FTPInitRespLen[FTPInitCmdNo - 1]);
  DataFile.println(); //New line at end of data
  NextAddress += FTPInitRespLen[FTPInitCmdNo - 1];
}

//FTP Send Commands
FTPSendStartAddr = NextAddress;

DataFile.print(F("L GSM Set Filename C (to start of filename): "));
GSMFileCmdLen = EEPROM.read(NextAddress);
DataFile.println(GSMFileCmdLen);
NextAddress++;

DataFile.print(F("GSM Set Filename C (to start of filename): "));
EEPROMToFile(NextAddress, GSMFileCmdLen);
DataFile.println(); //New line at end of data
NextAddress += GSMFileCmdLen;

DataFile.print(F("L GSM Set Filename C R: "));
GSMFileRespLen = EEPROM.read(NextAddress);
DataFile.println(GSMFileRespLen);
NextAddress++;

DataFile.print(F("GSM Set Filename CR: "));
EEPROMToFile(NextAddress, GSMFileRespLen);
DataFile.println(); //New line at end of data
NextAddress += GSMFileRespLen;

DataFile.print(F("L GSM Start FTP C: "));
GSMFTPStartCmdLen = EEPROM.read(NextAddress);
DataFile.println(GSMFTPStartCmdLen);
NextAddress++;

DataFile.print(F("GSM Start FTP C: "));
EEPROMToFile(NextAddress, GSMFTPStartCmdLen);
DataFile.println(); //New line at end of data
NextAddress += GSMFTPStartCmdLen;

DataFile.print(F("L GSM Start FTP CR: "));
GSMFTPStartRespLen = EEPROM.read(NextAddress);
DataFile.println(GSMFTPStartRespLen);
NextAddress++;

DataFile.print(F("GSM Start FTP CR: "));
EEPROMToFile(NextAddress, GSMFTPStartRespLen);
DataFile.println(); //New line at end of data
NextAddress += GSMFTPStartRespLen;

DataFile.print(F("L GSM Req FTP Data send C: "));
GSMFTPReqCmdLen = EEPROM.read(NextAddress);
```

```
    DataFile.println(GSMFTPReqCmdLen);
    NextAddress++;

    DataFile.print(F("GSM Req FTP Data send C (exc. Data L): "));
    EEPROMToFile(NextAddress, GSMFTPReqCmdLen);
    DataFile.println(); //New line at end of data
    NextAddress += GSMFTPReqCmdLen;

    DataFile.print(F("L GSM Req FTP Data Send CR: "));
    GSMFTPReqRespLen = EEPROM.read(NextAddress);
    DataFile.println(GSMFTPReqRespLen);
    NextAddress++;

    DataFile.print(F("GSM Req FTP Data Send CR: "));
    EEPROMToFile(NextAddress, GSMFTPReqRespLen);
    DataFile.println(); //New line at end of data
    NextAddress += GSMFTPReqRespLen;

    DataFile.print(F("L GSM End FTP Data send C: "));
    GSMFTPEndCmdLen = EEPROM.read(NextAddress);
    DataFile.println(GSMFTPEndCmdLen);
    NextAddress++;

    DataFile.print(F("GSM End FTP Data send C: "));
    EEPROMToFile(NextAddress, GSMFTPEndCmdLen);
    DataFile.println(); //New line at end of data
    NextAddress += GSMFTPEndCmdLen;

    DataFile.print(F("L GSM End FTP Data Send CR: "));
    GSMFTPEndRespLen = EEPROM.read(NextAddress);
    DataFile.println(GSMFTPEndRespLen);
    NextAddress++;

    DataFile.print(F("GSM End FTP Data Send CR: "));
    EEPROMToFile(NextAddress, GSMFTPEndRespLen);
    DataFile.println(); //New line at end of data
    NextAddress += GSMFTPEndRespLen;

    DataFile.print(F("L GSM End FTP Sess C: "));
    GSMFTPEndSessLen = EEPROM.read(NextAddress);
    DataFile.println(GSMFTPEndSessLen);
    NextAddress++;

    DataFile.print(F("GSM End FTP Sess C: "));
    EEPROMToFile(NextAddress, GSMFTPEndSessLen);
    DataFile.println(); //New line at end of data
    NextAddress += GSMFTPEndSessLen;

    DataFile.print(F("L GSM End FTP Sess CR: "));
    GSMFTPEndSessRespLen = EEPROM.read(NextAddress);
    DataFile.println(GSMFTPEndSessRespLen);
    NextAddress++;

    DataFile.print(F("GSM End FTP Sess CR: "));
    EEPROMToFile(NextAddress, GSMFTPEndSessRespLen);
    DataFile.println(); //New line at end of data
    NextAddress += GSMFTPEndSessRespLen;

    //Close file
    DataFile.close();
    oled.println(F("Config file"));
    oled.println(F("  created"));
    digitalWrite(PinLEDR, LOW); //RED LED ON
    SetupError = 1;
    //Exit setup
    return;
  }
  else //Failed to create a config file
  {
    oled.println(F("Config file"));
    oled.println(F("write fail"));
    digitalWrite(PinLEDR, LOW); //RED LED ON
    SetupError = 1;
    //Exit setup
    return;
  }
}
else //Config file is present, read it
{
  DataFile = SD.open(F("gsmlog.cfg"), FILE_WRITE); //open file, leave datestamp as is
  if (DataFile)
  {
    DataFile.seek(0);//go to start of file
    DataFile.seek(FindNext(DataFile,':') + 1);//Find the first parameter

    if (DataFile.peek() == 'Y') //Update RTC specified (Peek reads without changing current position
     in file)
    {
      DataFile.write('N'); //Change to N so clock not set again on next reset
      DataFile.seek(FindNext(DataFile,':') + 1);
```

```
  //Loop over Year (5), Month (4),Date (3), Hour (2), Minute (1), Second (0)

  for(i = 5; i < 255; i--)
  {
    memset(TempStr10, 0, sizeof(TempStr10)); //Clear the temporary string
    TempStr10[0] = DataFile.read();
    TempStr10[1] = DataFile.read();
    RtcData[i] = atoi(TempStr10);
  }
  //Set RTC
  //Display message to press button to set clock
  oled.setCursor(0,1);
  oled.println(F(" Press OK"));
  oled.println(F("  to set"));
  oled.println(F("  clock"));

  //Show a yellow LED
  digitalWrite(PinLEDR, LOW);
  digitalWrite(PinLEDG, LOW);

  //Wait for keypress
  do
  {
  } while (digitalRead(PinButton) == HIGH); //Keep looping until button pressed

  //Set clock
  HCRTC.RTCWrite(I2CDS1307Add, RtcData[5], RtcData[4], RtcData[3], RtcData[2], RtcData[1],
 RtcData[0], 1);  //Set Clock, last parameter is weekday and not used
  delay(100); //Delay to let the clock be set

  //Turn Yellow LED off
  digitalWrite(PinLEDR, HIGH);
  digitalWrite(PinLEDG, HIGH);

  //Message displaying new time
  HCRTC.RTCRead(I2CDS1307Add); //Read RTC
  oled.clear();
  oled.setCursor(0,1); //column pixel, row in 8 pixel blocks.
  oled.println(F("Clock set:"));
  oled.println(HCRTC.GetDateString()); //Display date
  oled.println(HCRTC.GetTimeString()); //Display time
  delay(MsgDelay); //Pause so can read LCD diagnostic message
}
else //don't need to update time
{
  DataFile.seek(FindNext(DataFile,':') + 1); //Move to clock set value, but will not be used
}

//Wait for keypress to start (to allow user to synch reading with meter)?
DataFile.seek(FindNext(DataFile,':') + 1); //Find parameter
if (DataFile.peek() == 'Y') //Wait for keypress specified
{
  DataFile.write('N'); //Change to N so no wait for keypress on next reboot

  //Display Message
  oled.clear();
  oled.setCursor(0,2);
  oled.println(F(" Press OK"));
  oled.println(F(" to start"));

  //Show a yellow LED
  digitalWrite(PinLEDR, LOW);
  digitalWrite(PinLEDG, LOW);

  //Wait for keypress
  do
  {
  } while (digitalRead(PinButton) == HIGH); //Keep looping until button pressed

  //Turn Yellow LED off
  digitalWrite(PinLEDR, HIGH);
  digitalWrite(PinLEDG, HIGH);

  //Clear message
  oled.clear();
}

//Number of pulse inputs
ReadCharFromFile();
NPulseInputs = atoi(TempStr10);

//Pulse Weights (only read for pulse inputs that exist)
for (byte InputNo = 0; InputNo < NPulseInputs; InputNo++)
{
  ReadNumberFromFile(5);
  PulseWeight[InputNo] = atoi(TempStr10);
}

//Pulse Input Names (only read for pulse inputs that exist)
for (byte InputNo=0; InputNo<NPulseInputs; InputNo++)
{
```

```
  DataFile.seek(FindNext(DataFile,':') + 1);
  i = 0;
  do
  {
    DataRead = DataFile.read();
    if(DataRead != 13) //Dont add CR
     {
      PulseName[InputNo][i] = DataRead;
      i++;
     }
  } while ((DataRead != 13) && (i < 9)); //Stop reading after CR or 9 characters
}

//Pulse Input Debounce
ReadNumberFromFile(3);
PulseDebounce = atoi(TempStr10);

//Log Rate (read as multiple chars)
ReadNumberFromFile(5);
LogInterval = atoi(TempStr10);

//Upload Rate
DataFile.seek(FindNext(DataFile,':') + 1);
switch (DataFile.read())
{
 case 'D': //Daily
    UploadRate = 2;
  break;
  default: //Hourly
    UploadRate = 1;
   break;
}

//Column Format
DataFile.seek(FindNext(DataFile,':') + 1);
MultiCol = DataFile.read();

//Starting Meter Reads
DataFile.seek(FindNext(DataFile,':') + 1);
if (DataFile.peek() == 'Y') //Meter readings will be read from file
{
  DataFile.write('N'); //Change to N so EEPROM values used on subsequent reboot
  //Read values from file for each input
  //Uses long integers so values up to 4,294,967,295 possible
  //But will rollover at 999,999,999
  for (byte InputNo = 0; InputNo < NPulseInputs; InputNo++)
    {
      ReadNumberFromFile(10);
      //If ten digits long, discard first digit as rolls over at 999,999,999
      if ((TempStr10[9] >= 48) && (TempStr10[9] <= 57)) //10th character is a numeric digit
      {
        for (byte j = 0; j < 10; j++)
        {
          TempStr10[j] = TempStr10[j + 1];
        }
      }
      PulseReading[InputNo] = atol(TempStr10);
    }
    //The reads are written to EEPROM at end of setup
}
else
{
  //Skip lines in file
  for (byte InputNo = 0; InputNo < NPulseInputs; InputNo++)
  {
    DataFile.seek(FindNext(DataFile,':') + 1);
  }
  //Read values from EEPROM
  for (byte InputNo = 0; InputNo < NPulseInputs; InputNo++)
  {
    PulseReading[InputNo] = EEPROMReadUint32(4 * InputNo);
  }
}
//GSM parameters - these are saved to EEPROM not RAM
//There are checks in the Write functions that total datalength does not exceed eeprom size
unsigned int NextAddress = EEPROMCFGDataStart + 2 + CFGFileLen;

//Signal Strength Request
SigStrStartAddr = NextAddress;

//Length of signal strength command
ReadCharFromFile();
SigStrCmdLen = atoi(TempStr10);
EEPROMWriteUint8(NextAddress, SigStrCmdLen);
NextAddress++;

//Signal Strength Command
DataFile.seek(FindNext(DataFile,':') + 1);
FileToEEPROM(NextAddress,SigStrCmdLen);
NextAddress += SigStrCmdLen;
```

```
//Signal Strength Response Delimeter
DataFile.seek(FindNext(DataFile,':') + 1);
FileToEEPROM(NextAddress,1);
NextAddress ++;

//Signal Strength Response length
ReadNumberFromFile(3);
SigStrRespLen = atoi(TempStr10);
EEPROMWriteUint8(NextAddress, SigStrRespLen);
NextAddress++;

//Communication Test Commands
TestCmdStartAddr = NextAddress;

//Number of Test Commands
ReadNumberFromFile(3);
NGSMTestCmds = atoi(TempStr10);
EEPROMWriteUint8(NextAddress, NGSMTestCmds);
NextAddress++;

//Parameters for each Test Command
for(byte GSMTestCmdNo = 1; GSMTestCmdNo <= NGSMTestCmds; GSMTestCmdNo++)
{
  //Command Length
  ReadNumberFromFile(3);
  GSMTestCmdLen[GSMTestCmdNo - 1] = atoi(TempStr10);
  EEPROMWriteUint8(NextAddress, GSMTestCmdLen[GSMTestCmdNo - 1]);
  NextAddress++;

  //Command
  DataFile.seek(FindNext(DataFile,':') + 1);
  FileToEEPROM(NextAddress,GSMTestCmdLen[GSMTestCmdNo - 1]);
  NextAddress += GSMTestCmdLen[GSMTestCmdNo - 1];

  //Response Length
  ReadNumberFromFile(3);
  GSMTestRespLen[GSMTestCmdNo - 1] = atoi(TempStr10);
  EEPROMWriteUint8(NextAddress, GSMTestRespLen[GSMTestCmdNo - 1]);
  NextAddress++;

  //Response
  DataFile.seek(FindNext(DataFile,':') + 1);
  FileToEEPROM(NextAddress,GSMTestRespLen[GSMTestCmdNo - 1]);
  NextAddress += GSMTestRespLen[GSMTestCmdNo - 1];
}

//GSM Initialisation Commands
GSMInitStartAddr = NextAddress;

//Number of Initialisation Commands
ReadNumberFromFile(3);
NGSMInitCmds = atoi(TempStr10);
EEPROMWriteUint8(NextAddress, NGSMInitCmds);
NextAddress++;

//Parameters for each Test Command
for(byte GSMInitCmdNo = 1; GSMInitCmdNo <= NGSMInitCmds; GSMInitCmdNo++)
{
  //Command Length
  ReadNumberFromFile(3);
  GSMInitCmdLen[GSMInitCmdNo - 1] = atoi(TempStr10);
  EEPROMWriteUint8(NextAddress, GSMInitCmdLen[GSMInitCmdNo - 1]);
  NextAddress++;

  //Command
  DataFile.seek(FindNext(DataFile,':') + 1);
  FileToEEPROM(NextAddress,GSMInitCmdLen[GSMInitCmdNo - 1]);
  NextAddress += GSMInitCmdLen[GSMInitCmdNo - 1];

  //Response Length
  ReadNumberFromFile(3);
  GSMInitRespLen[GSMInitCmdNo - 1] = atoi(TempStr10);
  EEPROMWriteUint8(NextAddress, GSMInitRespLen[GSMInitCmdNo - 1]);
  NextAddress++;

  //Response
  DataFile.seek(FindNext(DataFile,':') + 1);
  FileToEEPROM(NextAddress,GSMInitRespLen[GSMInitCmdNo - 1]);
  NextAddress += GSMInitRespLen[GSMInitCmdNo - 1];
}

//FTP Initialisation Commands
FTPInitStartAddr = NextAddress;

//Number of Initialisation Commands
ReadNumberFromFile(3);
NFTPInitCmds = atoi(TempStr10);
EEPROMWriteUint8(NextAddress, NFTPInitCmds);
NextAddress++;

//Parameters for each Test Command
```

```
for(byte FTPInitCmdNo = 1; FTPInitCmdNo <= NFTPInitCmds; FTPInitCmdNo++)
{
  //Command Length
  ReadNumberFromFile(3);
  FTPInitCmdLen[FTPInitCmdNo - 1] = atoi(TempStr10);
  EEPROMWriteUint8(NextAddress, FTPInitCmdLen[FTPInitCmdNo - 1]);
  NextAddress++;

  //Command
  DataFile.seek(FindNext(DataFile,':') + 1);
  FileToEEPROM(NextAddress,FTPInitCmdLen[FTPInitCmdNo - 1]);
  NextAddress += FTPInitCmdLen[FTPInitCmdNo - 1];

  //Response Length
  ReadNumberFromFile(3);
  FTPInitRespLen[FTPInitCmdNo - 1] = atoi(TempStr10);
  EEPROMWriteUint8(NextAddress, FTPInitRespLen[FTPInitCmdNo - 1]);
  NextAddress++;

  //Response
  DataFile.seek(FindNext(DataFile,':') + 1);
  FileToEEPROM(NextAddress,FTPInitRespLen[FTPInitCmdNo - 1]);
  NextAddress += FTPInitRespLen[FTPInitCmdNo - 1];
}

//FTP Commands
FTPSendStartAddr = NextAddress;

//Command to set target filename (AT+FTPPUTNAME=)
//Length
ReadNumberFromFile(3);
GSMFileCmdLen = atoi(TempStr10);
EEPROMWriteUint8(NextAddress, GSMFileCmdLen);
NextAddress++;

//Command
DataFile.seek(FindNext(DataFile,':') + 1);
FileToEEPROM(NextAddress,GSMFileCmdLen);
NextAddress += GSMFileCmdLen;

//Response Length
ReadNumberFromFile(3);
GSMFileRespLen = atoi(TempStr10);
EEPROMWriteUint8(NextAddress, GSMFileRespLen);
NextAddress++;

//Response
DataFile.seek(FindNext(DataFile,':') + 1);
FileToEEPROM(NextAddress,GSMFileRespLen);
NextAddress += GSMFileRespLen;

//Command to request write (AT+FTPPUT=1)
//Length
ReadNumberFromFile(3);
GSMFTPStartCmdLen = atoi(TempStr10);
EEPROMWriteUint8(NextAddress, GSMFTPStartCmdLen);
NextAddress++;

//Command
DataFile.seek(FindNext(DataFile,':') + 1);
FileToEEPROM(NextAddress,GSMFTPStartCmdLen);
NextAddress += GSMFTPStartCmdLen;

//Response Length
ReadNumberFromFile(3);
GSMFTPStartRespLen = atoi(TempStr10);
EEPROMWriteUint8(NextAddress, GSMFTPStartRespLen);
NextAddress++;

//Response
DataFile.seek(FindNext(DataFile,':') + 1); //response string includes cr-lf after OK,
FileToEEPROM(NextAddress,GSMFTPStartRespLen);
NextAddress += GSMFTPStartRespLen;

//command to write data (AT+FTPPUT=2,) (last value is bytes to write, needs to match response
 above)
//Length
ReadNumberFromFile(3);
GSMFTPReqCmdLen = atoi(TempStr10);
EEPROMWriteUint8(NextAddress, GSMFTPReqCmdLen);
NextAddress++;

//Command
DataFile.seek(FindNext(DataFile,':') + 1);
FileToEEPROM(NextAddress,GSMFTPReqCmdLen);
NextAddress += GSMFTPReqCmdLen;

//Response Length
ReadNumberFromFile(3);
GSMFTPReqRespLen = atoi(TempStr10);
EEPROMWriteUint8(NextAddress, GSMFTPReqRespLen);
```

```
NextAddress++;

//Response
DataFile.seek(FindNext(DataFile,':') + 1);
FileToEEPROM(NextAddress,GSMFTPReqRespLen);
NextAddress += GSMFTPReqRespLen;

//command to close transfer (AT+FTPPUT=2,0)
//Length
ReadNumberFromFile(3);
GSMFTPEndCmdLen = atoi(TempStr10);
EEPROMWriteUint8(NextAddress, GSMFTPEndCmdLen);
NextAddress++;

//Command
DataFile.seek(FindNext(DataFile,':') + 1);
FileToEEPROM(NextAddress,GSMFTPEndCmdLen);
NextAddress += GSMFTPEndCmdLen;

//Response Length
ReadNumberFromFile(3);
GSMFTPEndRespLen = atoi(TempStr10);
EEPROMWriteUint8(NextAddress, GSMFTPEndRespLen);
NextAddress++;

//Response
DataFile.seek(FindNext(DataFile,':') + 1);
FileToEEPROM(NextAddress,GSMFTPEndRespLen);
NextAddress += GSMFTPEndRespLen;

//command to close FTP Session (AT+SAPBR=0,1)
//Length
ReadNumberFromFile(3);
GSMFTPEndSessLen = atoi(TempStr10);
EEPROMWriteUint8(NextAddress, GSMFTPEndSessLen);
NextAddress++;

//Command
DataFile.seek(FindNext(DataFile,':') + 1);
FileToEEPROM(NextAddress,GSMFTPEndSessLen);
NextAddress += GSMFTPEndSessLen;

//Response Length
ReadNumberFromFile(3);
GSMFTPEndSessRespLen = atoi(TempStr10);
EEPROMWriteUint8(NextAddress, GSMFTPEndSessRespLen);
NextAddress++;

//Response
DataFile.seek(FindNext(DataFile,':') + 1);
FileToEEPROM(NextAddress,GSMFTPEndSessRespLen);
NextAddress += GSMFTPEndSessRespLen;

/*//MODBUS parameters
//Baud rate (read as multiple chars)
memset(TempStr10, 0, sizeof(TempStr10)); //Clear the temporary string
DataFile.seek(FindNext(DataFile,':') + 1);
i=0;
do
{
  DataRead = DataFile.read();
  TempStr10[i] = DataRead;
  i++;
} while ((DataRead != 13) && (i < 7)); //read up to 6 chars or carriage return
SerBaud = atol(TempStr10); //can be over 65535 so  use atol (ascii to long) instead of atoi
 (ascii to int)

//Parity
DataFile.seek(FindNext(DataFile,':') + 1);
SerParity = DataFile.read();

//Stop Bits
DataFile.seek(FindNext(DataFile,':') + 1);
SerStop = DataFile.read();

//Set SerConfig from constants
switch (SerParity)
{
  case 'N':
    if(SerStop == '1')
      {
        SerConfig = SERIAL_8N1; //This breaks the MODBUS RTU standard but is used by some
 devices
      }
    else //2 stop bits by default
      {
        SerConfig = SERIAL_8N2;
      }
  break;
  case 'E':
    if(SerStop == '2')
```

```
      {
        SerConfig = SERIAL_8E2; //This breaks the MODBUS RTU standard
      }
    else //1 stop bit by default
      {
        SerConfig = SERIAL_8E1;
      }
  break;
  default: //Odd Parity
    if(SerStop == '2')
      {
        SerConfig = SERIAL_8O2; //This breaks the MODBUS RTU standard
      }
    else //1 stop bit by default
      {
        SerConfig = SERIAL_8O1;
      }
  break;
}

//Set the RegOffset bugfix variable depending on selected comms parameters
//Note: 8E2 not tested, no adjustment made
if ((SerBaud == 19200) || (SerBaud == 115200) || (SerConfig == SERIAL_8N2) || (SerConfig ==
 SERIAL_8E1))
{
  RegOffset = 1;
}

//Registers to log
DataFile.seek(FindNext(DataFile,':') + 1);

do //scan the remaining data in the file to see if there are 2 commas on each line
{
  LineStartPos = DataFile.position(); //save this position, so can return here
  CommaCount = 0; //reset the count
  do
  {
    DataRead = DataFile.read();
    if (DataRead == ',')
    {
      CommaCount++;//count up the commas
    }

  } while ((DataRead != 255) && (DataRead != 13)); //keep going until end of file or carriage
 return

  if (CommaCount == 2) //if it's got 2 commas assume a valid slaveid, type and register
  {
    DataFile.seek(LineStartPos);
    i=0;
    memset(TempStr10, 0, sizeof(TempStr10)); //Clear the temporary string
    do
    {
      DataRead = DataFile.read();
      TempStr10[i] = DataRead;
      i++;
    } while ((DataRead != ',') && (i<7));
    SlaveId = atoi(TempStr10); //slave id
    DataRead = DataFile.read(); //this should be the register type
    switch (DataRead)
    {
      case 'C':
        Func = READ_COIL_STATUS;
      break;
      case 'S':
        Func = READ_INPUT_STATUS;
      break;
      case 'I':
        Func = READ_INPUT_REGISTERS;
      break;
      default: //H - Holding registers
        Func = READ_HOLDING_REGISTERS;
      break;
    }
    DataFile.read(); //move to the next char
    i=0;
    memset(TempStr10, 0, sizeof(TempStr10)); //Clear the temporary string
    do
    {
      DataRead = DataFile.read();
      TempStr10[i] = DataRead;
      i++;
    } while ((DataRead != 13) && (i<7));
    DataFile.read(); //move to the next char
    SlaveReg = atoi(TempStr10); //get the register number

    // Initialize packets (pointer to packet, slave device ID, Function code constant, address
to read (zero based),number of registers to read,subscript of first returned value in
HoldingRegs array
    //x is the zero based subscript value
    modbus_construct(&Packets[x], SlaveId, Func, SlaveReg, 1, x);
```

```
            x++;
          }
          else //not two commas on line, assume end of file
          {
            InvalidEntry = 1;
          }

        } while((!InvalidEntry) && (x<TotalNoOfRegisters)); //keep reading in master packets until there
         are no more

        //Store actual number of registers to read - x was incremented after final value so is the total
         number of registers
        QtyRegs = x;

        // Initialize the Modbus comms using the default hardware serial port (Pins D0 and D1) and
         parameters read in from file, packet definitions and array to hold data locally
        modbus_configure(&Serial, SerBaud, SerConfig, Timeout, Polling, RetryCount, TxEnablePin,
         Packets, x ,HoldingRegs); //Changed from max allowable registers (TotalNoOfRegisters) to actual
         (x)
        */

        //close the config file
        DataFile.close();

        //Write Meter reads to eeprom memory for future use
        WriteReadsToEEPROM();

        //Initialise Old Times to trigger first log
        HCRTC.RTCRead(I2CDS1307Add); //Update RTC data
        OldSecs = HCRTC.GetSecond();
        OldHour = HCRTC.GetHour();
        OldDay = HCRTC.GetDay();
      }
      else //couldn't open the config file
      {
        oled.println(F("Config file"));
        oled.println(F("won't open"));
        digitalWrite(PinLEDR, LOW); //RED LED ON
        SetupError = 1;
        //Exit Setup
        return;
      }
    }
  }
  if(!SetupError)
  {
    //Clear Display ready for main loop
    oled.clear();
    oled.set1X();
    oled.setCursor(0,0); //column pixel, row in 8 pixel blocks.
    oled.print(F("GSM Signal:"));
  }
}

//__________

void loop() {
  //Only do the loop actions if no setup error
  if (!SetupError)
  {
    byte RtcData[6];
    unsigned long CurrentMillis = millis();

    //byte MError = 0; //MODBUS error counter for each register

    //Poll the MODBUS devices
    //modbus_update();

    //Store any errors and reset so registers read subsequently
    /*MError = 0; //Reset number of errors this time
    for (byte i=1; i<QtyRegs; i++)
    {
      if (Packets[i - 1].connection == false)
      {
        MError++;
        Packets[i - 1].connection = true; //Reset the error
      }
    }
    if (MError == 0)
    {
      MErrorMin = 0;
    }*/

    if (IntervalCheck(PreviousMillis, ClockInt)) //Read the RTC to see if it's time to log
    {
      PreviousMillis = CurrentMillis; //Reset Timer

      HCRTC.RTCRead(I2CDS1307Add); //Update RTC data
      RtcData[0] = HCRTC.GetSecond(); //Get the seconds from the RTC
      RtcData[1] = HCRTC.GetMinute(); //Minutes
      if (RtcData[0] != OldSecs  || (RtcData[1] != OldMinute)) //Seconds or minutes have changed on the
```

```
    RTC
  {
    if ((RtcData[0] > OldSecs))
    //Seconds greater than old value
    {
      Secs = Secs + RtcData[0] - OldSecs; //Increment seconds counter
      //Check if we've missed whole minutes
      if (RtcData[1] != OldMinute)
      {
        if(RtcData[1] > OldMinute)
        {
          Secs = Secs + (60 * ( RtcData[1] - OldMinute));
        }
        else
        {
          //Minute rollover over hour
          Secs = Secs + (60 * ( RtcData[1] + (60 - OldMinute)));
        }
      }
    }
    if ((RtcData[0] <= OldSecs))
    //Minutes must have changed; if seconds havent, then minutes have as evaluated by higher level
     IF
    {
      //Seconds have returned to zero or are same : add difference in seconds through zero (if
     seconds the same this will add a minute)
      Secs = Secs + RtcData[0] + (60 - OldSecs); //Increment seconds counter

      //Add change in minutes
      //Minute rollover already accounted for in seconds calculation so subtract 1 from the minute
     increment in the below calculations
      if(RtcData[1] > OldMinute)
      {
        Secs = Secs + (60 * ( RtcData[1] - OldMinute - 1));
      }
      else
      {
        //Minute rollover over hour
        Secs = Secs + (60 * ( RtcData[1] + (60 - OldMinute - 1)));
      }
    }

    OldSecs = RtcData[0]; //Update old value to new value
    OldMinute = RtcData[1];

    //Synch to hour start check
    if(LoggerStatus == 'H') //Awaiting a new hour at the start of logging
    {
      RtcData[2] = HCRTC.GetHour();
      if(RtcData[2] != OldHour)  //Change status to Go if hour has changed on the RTC
      {
        LoggerStatus = 'G';
        OldHour = RtcData[2]; //Delay FTPing until first whole hour of logging has occurred...
        OldDay = HCRTC.GetDay(); //...or whole day as appropriate
        Secs = 0;
      }
    }
  }
}
if ((Secs >= LogInterval)  && (LoggerStatus != 'H')) //time to log and not waiting for new hour
{
  Secs = Secs - LogInterval; //Reset seconds counter
  //Assemble filename from date and time
  //Retrieve each part of date and time as a numeric value (byte data type)
  RtcData[0] = HCRTC.GetSecond(); //Seconds
  RtcData[1] = HCRTC.GetMinute(); //Minutes
  RtcData[2] = HCRTC.GetHour(); //Hours
  RtcData[3] = HCRTC.GetDay(); //Date
  RtcData[4] = HCRTC.GetMonth(); //Month
  RtcData[5] = HCRTC.GetYear(); //Year
  //Store Previous filename as this will be the file to FTP
  memcpy(OldFileName, FileName, sizeof(FileName));

  //Create the filename from the time in the form YYMMDDHH.csv
  byte j=0; //j is character number in the filename (zero based, 0..7 for the date part)

  //loop over parts of the date and time as stored in RtcData
  //Start with year, then month...seconds

  for (byte i = 5; ((i >= 3) || (((i >= 2) && (UploadRate == 1)))); i--) //For each part of date
  {
    //Add the two digits to the filename: all parts of date and time are 1 or 2 digits
    FileName[j] = RtcData[i] / 10 + 0x30; //get first digit by integer division by 10 and convert to
     ascii code of digit by adding 30
    j++;
    FileName[j] = RtcData[i] % 10 + 0x30; //get second digit by remainder when divided by 10 and
     convert to ascii code of digit by adding 30
    j++;
  }

  //add file extension
```

```
FileName[j+0] = '.';
FileName[j+1] = 'c';
FileName[j+2] = 's';
FileName[j+3] = 'v';
FileName[j+4] = 0; //end the string


//Open the log file
SdFile::dateTimeCallback(DateTime); //Set datestamp for file
DataFile = SD.open(FileName, FILE_WRITE); //create it if it doesn't exist
if (DataFile)
{
  if (MultiCol == 'Y') //Multi column format for Excel
  {
    if(!DataFile.size()) //If file is zero length then add header
    {
      DataFile.print(F("Date Time"));
      for (byte InputNo = 0; InputNo < NPulseInputs; InputNo++)
      {
        DataFile.print(F(","));
        DataFile.print(PulseName[InputNo]);
      }
      DataFile.println(); //New line
    }

    //Insert Date and Time into file
    DataFile.print(HCRTC.GetDateString());
    DataFile.print(F(" ")); // space between DATE and TIME
    DataFile.print(HCRTC.GetTimeString());
    for (byte InputNo = 0; InputNo < NPulseInputs; InputNo++)
    {
      DataFile.print(F(","));
      DataFile.print(PulseReading[InputNo]);
    }
    DataFile.println(); //New line
  }
  else //Single Column Satchwell format
  {
    for (byte InputNo = 0; InputNo < NPulseInputs; InputNo++) //Write separate line for each input
    {
      //Input Name
      DataFile.print(PulseName[InputNo]);
      DataFile.print(F(","));
      //Date  '-' delimited
      DataFile.print(RtcData[3]);
      DataFile.print(F("-"));
      DataFile.print(RtcData[4]);
      DataFile.print(F("-"));
      DataFile.print(RtcData[5]);
      DataFile.print(F(","));
      //Time
      DataFile.print(HCRTC.GetTimeString());
      DataFile.print(F(","));
      //Reading and new line
      DataFile.println(PulseReading[InputNo]);
    }
  }
  DataFile.close(); //close the file
}

//If on the hour or day as applicable, send data via FTP
if (((RtcData[2] != OldHour) && (UploadRate == 1)) || ((RtcData[3] != OldDay) && (UploadRate ==
   2)) || NewSave == 0) //Check for hour or day rollover, or FTP retry
{
  //If new data file available, flag new save
  if (((RtcData[2] != OldHour) && (UploadRate == 1)) || ((RtcData[3] != OldDay) && (UploadRate ==
   2)))
  {
    NewSave = 1;
  }

  //Save latest readings to EEPROM
  //Only do this if first attempt to FTP this batch of data (NewSave == 1)
  if (NewSave == 1)
  {
    WriteReadsToEEPROM();
  }

  //FTP the file
  byte FTPSuccess = 0; //FTP successful?
  if (NewSave == 1)
  {
    //Update the filename on first attempt, same filename used for retries
    memcpy(FTPFile, OldFileName, sizeof(OldFileName));
  }

  //Prepare the GSM and Test Comms
  byte FTPOK = 0;
  FTPOK = FTPTest();

  if (FTPOK == 1) //Only do the below if comms OK
```

```
        {
          //Check file exists
          if (SD.exists(FTPFile))
          {
            DataFile = SD.open(FTPFile, FILE_READ); //Open File
            //Check File opened
            if (DataFile)
            {
              //Check file not empty
              if(DataFile.size())
              {
                //FTP operation
                FTPSuccess = FTPSend();
              }
              DataFile.close();
            }
          }
        }

        //Reset oldHour and oldDay
        OldHour = RtcData[2];
        OldDay = RtcData[3];

        //Set NewSave to 1 if FTP successful, will next FTP when day/hour increments, or to 0 if FTP
         unsuccessful, to retry FTP of same file on logging interval throughout the period until next new
         file available
        if(FTPSuccess == 1) //Only if FTP successful, ; if not successful FTP will retry whenever files
         updated
        {
           NewSave = 1;
        }
        else
        {
           NewSave = 0;
        }
      }
    } //End of writing results for logging interval

    //Update Signal Strength
    UpdateGSMSigStr();

    //Check input Statuses and increment readings
    ReadPulseInputs();
  }
}

//__________

//Callback Function to set timestamp for files
void DateTime(uint16_t* DDate, uint16_t* TTime)
{
  HCRTC.RTCRead(I2CDS1307Add); //Update RTC data

  // return date using FAT_DATE macro to format fields
  *DDate = FAT_DATE((2000 + HCRTC.GetYear()), HCRTC.GetMonth(), HCRTC.GetDay());

  // return time using FAT_TIME macro to format fields
  *TTime = FAT_TIME(HCRTC.GetHour(), HCRTC.GetMinute(), HCRTC.GetSecond());
}

//__________

//Copy data from EEPROM to file
void EEPROMToFile(unsigned int StartAddress,unsigned int NBytes)
{

  for (unsigned int j = 0; j < NBytes; j++)
  {
    DataFile.write(EEPROM.read(StartAddress + j));  //Write converts to ASCII characters
  }
}

//__________

//Copy data from EEPROM to GSM
void EEPROMToGSM(unsigned int StartAddress,unsigned int NBytes, byte CRLF)
{
  gsm.flush();
  for (unsigned int j = 0; j < NBytes; j++)
  {
    gsm.print((char)EEPROM.read(StartAddress + j));
  }
  if (CRLF == 1)
  {
    gsm.println();//Send CR + LF
  }
}

//__________

//Read an unsigned 16 bit integer from 2 EEPROM addresses
```

```
unsigned int EEPROMReadUint16(unsigned int StartAddress)
{
  byte TempBytes[2]; //Temporary store for the upper and lower byte
  for (byte j = 0; j < 2; j++)
  {
    TempBytes[j] = EEPROM.read(StartAddress + j); //First argument is address, second is value
  }
  return ((int)(TempBytes[0]) << 8) + ((int)(TempBytes[1]));
}

//__________

//Read an unsigned 32 bit integer from 4 EEPROM addresses
unsigned long EEPROMReadUint32(unsigned int StartAddress)
{
  byte TempBytes[4]; //Temporary store for the bytes, unsigned long is four bytes
  for (byte j = 0; j < 4; j++)
  {
    TempBytes[j] = EEPROM.read(StartAddress + j); //Argument is address
  }
  //Convert to unsigned long
  return ((long)(TempBytes[0]) << 24)  + ((long)(TempBytes[1]) << 16) + ((long)(TempBytes[2]) << 8) +
         ((long)(TempBytes[3]));
}

//__________

//Write an unsigned 8 bit integer to EEPROM address with EEPROM overrun check
void EEPROMWriteUint8(unsigned int StartAddress, byte ValToWrite)
{
  //Check the data will fit in EEPROM, E2END returns maximum address which varies by Arduino model
  if(StartAddress > E2END)
  {
    oled.println(F("Config file"));
    oled.println(F("too large! "));
    digitalWrite(PinLEDR, LOW); //RED LED ON
    SetupError = 1;
    return;
  }

  EEPROM.write(StartAddress, ValToWrite); //First argument is address, second is value
}

//__________

//Write an unsigned 16 bit integer to 2 EEPROM addresses
void EEPROMWriteUint16(unsigned int StartAddress, unsigned int ValToWrite)
{
  //Check the data will fit in EEPROM, E2END returns maximum address which varies by Arduino model
  if(StartAddress + 1 > E2END)
  {
    oled.println(F("Config file"));
    oled.println(F("too large! "));
    digitalWrite(PinLEDR, LOW); //RED LED ON
    SetupError = 1;
    return;
  }

  byte TempBytes[2];
  TempBytes[0] = (byte) ((ValToWrite & 0x0000FF00) >> 8  );
  TempBytes[1] = (byte) ((ValToWrite & 0X000000FF)       );

  for (byte j = 0; j < 2; j++)
  {
    EEPROM.write(StartAddress + j, TempBytes[j]); //First argument is address, second is value
  }
}

//__________

//Write an unsigned 32 bit integer to 4 EEPROM addresses
void EEPROMWriteUint32(unsigned int StartAddress, unsigned long ValToWrite)
{
  //Check the data will fit in EEPROM, E2END returns maximum address which varies by Arduino model
  if(StartAddress + 3 > E2END)
  {
    oled.println(F("  EEPROM"));
    oled.println(F(" overflow!"));
    digitalWrite(PinLEDR, LOW); //RED LED ON
    SetupError = 1;
    return;
  }

  byte TempBytes[4]; //Temporary store for the bytes, unsigned long is four bytes
  TempBytes[0] = (byte) ((ValToWrite & 0xFF000000) >> 24 );
  TempBytes[1] = (byte) ((ValToWrite & 0x00FF0000) >> 16 );
  TempBytes[2] = (byte) ((ValToWrite & 0x0000FF00) >> 8  );
  TempBytes[3] = (byte) ((ValToWrite & 0X000000FF)       );

  for (byte j = 0; j < 4; j++)
  {
```

```
    EEPROM.write(StartAddress + j, TempBytes[j]); //First argument is address
  }
}

//__________

//Copy data from file to EEPROM
void FileToEEPROM(unsigned int StartAddress,unsigned int NBytes)
{
  //Check the data will fit in EEPROM, E2END returns maximum address which varies by Arduino model
  if((StartAddress + NBytes - 1) > E2END)
  {
    oled.println(F("Config file"));
    oled.println(F("too large! "));
    digitalWrite(PinLEDR, LOW); //RED LED ON
    SetupError = 1;
    return;
  }

  //Write Data
  for (unsigned int j = 0; j < NBytes; j++)
  {
    EEPROM.write(StartAddress + j, DataFile.read());
  }
}

//__________

//Function to find the next instance of a character in the sd card file
unsigned int FindNext(File DF,byte CharToFind)
{
  byte DataRead;
  do
  {
    DataRead = DF.read();
  } while ((DataRead != 255) && (DataRead != CharToFind)); //end of file or character position
  return DF.position();
}

//__________

byte FTPSend() //Return 1 if OK and 0 if error
{
  //Send data to ftp server
  char RespChar = 255; //Read individual digits of response
  byte RespValid = 0; //Valid response from modem?
  unsigned int TempAddress = FTPSendStartAddr + 1; //Skip length of command
  gsm.flush(); //Finish write operation
  GSMFlushInput();  //Flush input data
  //Set Destination Filename, AT+FTPPUTNAME="xxx"
  EEPROMToGSM(TempAddress, GSMFileCmdLen,0); //Final parameter 0 - don't send CRLF
  TempAddress += GSMFileCmdLen;
  TempAddress += 1; //Skip response length
  //Send double quotes
  gsm.write(34);
  //Send Filename
  for (byte CharNo = 0; CharNo < 12; CharNo++)
  {
    gsm.print(FTPFile[CharNo]);
    if(FTPFile[CharNo + 1] == 0)
    {
      break; //End at null character
    }
  }
  //Send double quotes and CRLF
  gsm.write(34);
  gsm.println();
  //Read response
  RespValid = ReadGSMResponse(GSMFileRespLen, TempAddress,GSMLongTimeout);
  if(!RespValid)
  {
    return 0;
  }
  TempAddress += GSMFileRespLen;
  TempAddress += 1; //Skip length of next command

  //Start file writing session, AT+FTPPUT=1
  gsm.flush(); //Finish write operation
  GSMFlushInput();  //Flush input data
  EEPROMToGSM(TempAddress, GSMFTPStartCmdLen,1);
  TempAddress += GSMFTPStartCmdLen;
  TempAddress += 1;

  //Read fixed part of response
  RespValid = ReadGSMResponse(GSMFTPStartRespLen, TempAddress,GSMLongTimeout);
  if(!RespValid)
  {
    return 0;
  }
  TempAddress += GSMFTPStartRespLen;
  TempAddress += 1; //Skip length of next command
```

```
//Read maximum number of bytes than can be transmitted
unsigned long NCharToTx = 0;
//Read the response character by character until either buffer empty or 4 digits read, convert to
      unsigned long
for (byte CharNo = 0; CharNo < 4; CharNo++)
{
  RespChar = GSMRead(GSMLongTimeout);

  if(RespChar != 0) //0 returned if serial buffer empty, in which case skip adding digits
  {
    NCharToTx *= 10; //Multiply existing digits by 10
    NCharToTx += (RespChar - '0'); //convert ASCII value to int by subtracting ASCII value of 0 (48)
       from ASCII value of digit, add to value
  }
}
//Initialise file length and position
unsigned long FileLen;
FileLen = DataFile.size();
unsigned long FilePos = 0; //Position of next character to transmit, zero based

//if max transmit length >0, Send data in packets until all file sent
if (NCharToTx > 0)
{
  gsm.flush(); //Finish write operation
  do //Repeat until all data sent (NCharToTx > 0)
  {
    //Write data in packets reading length of next packet each time
    //Send Command AT+FTPPUT=2,NCharToTx ...don't forget to increment position in eeprom to read next

    GSMFlushInput();  //Flush input data

    //Adjust number of characters to transmit if characters remaining < transmit limit
    if(NCharToTx > (FileLen - FilePos))
    {
      NCharToTx = (FileLen - FilePos);
    }

    //Send Command
    EEPROMToGSM(TempAddress, GSMFTPReqCmdLen, 0);

    gsm.println(NCharToTx);

    TempAddress += GSMFTPReqCmdLen;
    TempAddress += 1; //Skip length of response

    //Read only fixed part of response
     RespValid = ReadGSMResponse(GSMFTPReqRespLen, TempAddress,GSMLongTimeout);
    if(!RespValid)
    {
      return 0;
    }
    //Discard rest of response (or it will be returned when trying to read response after bytes
       written)
    GSMFlushInput();

    //Send the characters
    for (unsigned long FileChar = 0; FileChar < NCharToTx; FileChar++)
    {
      DataFile.seek(FilePos);
      gsm.print((char)DataFile.read());
      FilePos++;
    }
    gsm.flush(); //Finish write operation

    //Read response when data teansmission complete: OK<CR><LF><CR><LF>+FTPPUT: 1,1,.....(ignore rest)
       (use max delay GSMVeryLongTimeout as can take a while)
    TempAddress -= GSMFTPStartRespLen;
    TempAddress -= GSMFTPReqCmdLen;
    TempAddress -= 2; //Skip lengths of commands
    //Read only fixed part of response
    RespValid = ReadGSMResponse(GSMFTPReqRespLen, TempAddress,GSMLongTimeout);
    if(!RespValid)
    {
      return 0;
    }

    //Increment next command address back to the request send command for next loop iteration
    TempAddress += GSMFTPStartRespLen;
    TempAddress += 1; //Skip length of next command
  } while (FilePos < FileLen);  //So will end when no more characters to send
}
else
{
  return 0; //Return error if NCharToTx returned by modem is 0
}
//Close the FTP send and session

//Increment EEPROM location to read over both the FTP send and response commands to the desired FTP
      close
TempAddress += GSMFTPReqCmdLen; //Request command
```

```
  TempAddress += 1; //Request command length of response
  TempAddress += GSMFTPReqRespLen; //Request command valid response
  TempAddress += 1; //Skip length of End command
  //Send AT+FTPPUT=2,0
  gsm.flush();
  GSMFlushInput();  //Flush input data
  //Send Command
  EEPROMToGSM(TempAddress, GSMFTPEndCmdLen, 1);
  TempAddress += GSMFTPEndCmdLen;
  TempAddress += 1; //Skip length of response

  //Await response  OK<CR><LF><CR><LF>+FTPPUT: 1,.....(ignore rest) (use max delay GSMVeryLongTimeout)
  RespValid = ReadGSMResponse(GSMFTPEndRespLen, TempAddress, GSMVeryLongTimeout);
  if(!RespValid)
  {
    return 0;
  }
  gsm.flush();
  GSMFlushInput();  //Flush input data
  //Increment next command address
  TempAddress += GSMFTPEndRespLen;
  TempAddress += 1; //Skip length of next command

  //AT+SAPBR=0,1 to end FTP session
  //Send Command
  EEPROMToGSM(TempAddress, GSMFTPEndSessLen, 1);
  TempAddress += GSMFTPEndSessLen;
  TempAddress += 1; //Skip length of response

  //Await response OK
  RespValid = ReadGSMResponse(GSMFTPEndSessRespLen, TempAddress,GSMLongTimeout);
  if(!RespValid)
  {
    return 0;
  }
}

//__________

//Test GSM is ready for FTP operation, reset and reconfigure as necessary
byte FTPTest() //Return 1 if OK and 0 if error
{
  //Update Signal Strength
  UpdateGSMSigStr();
  //If no signal then try rebooting modem
  if (GsmSig == 0)
  {
    ResetGSM();
    UpdateGSMSigStr();
  }
  if (GsmSig >= 10)
  {
    //Comms Tests - if first call to this sub don't test, all parameters will be reset anyway
    unsigned int TempAddress = TestCmdStartAddr + 2; //+1 for number of commands, +1 for length of first
        command
    byte RespValid = 0; //Valid response from modem?
    if(FirstModemCall == 0)
    {
      for(byte TestCmdNo = 0; TestCmdNo < NGSMTestCmds; TestCmdNo++)
      {
        gsm.flush(); //Finish write operation
        GSMFlushInput();  //Flush input data
        //Send Command
        EEPROMToGSM(TempAddress, GSMTestCmdLen[TestCmdNo], 1);
        TempAddress += GSMTestCmdLen[TestCmdNo];
        TempAddress += 1; //Skip length of response

        //Receive Response
        RespValid = ReadGSMResponse(GSMTestRespLen[TestCmdNo], TempAddress,GSMLongTimeout);
        if(!RespValid)
        {
          FirstModemCall = 1;
          break; //Skip further tests
        }
        TempAddress += GSMTestRespLen[TestCmdNo];
        TempAddress += 1; //Skip length of next command
      }
    }
    //If any of the tests failed, issue a hardware reset and Send all the config commands
    if(FirstModemCall == 1)
    {
      ResetGSM();
      gsm.println(("ATE0")); //Turn off command echo, will return <CR> <LF> OK but response not checked
      //GSM Initialisation Commands
      TempAddress = GSMInitStartAddr + 2; //+1 for number of commands, +1 for length of first command
      for(byte InitCmdNo = 0; InitCmdNo < NGSMInitCmds; InitCmdNo++)
      {
        gsm.flush(); //Finish write operation
        GSMFlushInput();  //Flush input data
        //Send Command
        EEPROMToGSM(TempAddress, GSMInitCmdLen[InitCmdNo], 1);
```

```
        TempAddress += GSMInitCmdLen[InitCmdNo];
        TempAddress += 1; //Skip length of response

        //Receive Response
        RespValid = ReadGSMResponse(GSMInitRespLen[InitCmdNo], TempAddress,GSMLongTimeout);
        if(!RespValid)
        {
          return 0;
        }

        TempAddress += GSMInitRespLen[InitCmdNo];
        TempAddress += 1; //Skip length of next command
      }
    }
    FirstModemCall == 0; //Initialisation not required in future unless tests indicate a reset is
         required

    //FTP Initialisation Commands
    TempAddress = FTPInitStartAddr + 2; //+1 for number of commands, +1 for length of first command
    for(byte InitCmdNo = 0; InitCmdNo < NFTPInitCmds; InitCmdNo++)
    {
      gsm.flush(); //Finish write operation
      GSMFlushInput();  //Flush input data
      //Send Command
      EEPROMToGSM(TempAddress, FTPInitCmdLen[InitCmdNo], 1);
      TempAddress += FTPInitCmdLen[InitCmdNo];
      TempAddress += 1; //Skip length of response

      //Receive Response
      RespValid = ReadGSMResponse(FTPInitRespLen[InitCmdNo], TempAddress,GSMLongTimeout);
      if(!RespValid)
      {
        return 0;
      }
      TempAddress += FTPInitRespLen[InitCmdNo];
      TempAddress += 1; //Skip length of next command
    }
    return 1;
  }
  else
  {
    return 0; //Signal strength < 10%
  }
}

//__________

void GSMFlushInput() //Flush input buffer of GSM by reading all available data
{
  char RespChar2 = 255;
  do
  {
    RespChar2 = GSMRead(GSMTimeout);
  } while (RespChar2 != 0);
}

//__________


//Read GSM response with timeout
char GSMRead(unsigned int TimeOutLen)
{
  //Set timeout datum
  unsigned long GSMTimeoutDatum = millis();
  do
  {
    ReadPulseInputs();
  } while ((!gsm.available()) && (!IntervalCheck(GSMTimeoutDatum, TimeOutLen)));
  if(gsm.available())
  {
    return gsm.read();
  }
  else
  {
    return 0; //Return null character
  }
}

//__________

byte IntervalCheck(unsigned long Datum, unsigned long Interval) //return 1 if interval has elapsed
{
 unsigned long CurrentTime = millis();
 if ((CurrentTime < Datum) && (((LongIntMax - Datum) + (CurrentTime)) > Interval))
 {
  return 1;
 }
 else if ((CurrentTime < Datum) && (((LongIntMax - Datum) + (CurrentTime)) <= Interval))
 {
   return 0;
 }
```

```
 else if ((CurrentTime - Datum) > Interval)
 {
   return 1;
 }
 else
 {
   return 0;
 }
}

//__________

//Function to read value of button input, return 1 if button pressed (allowing for debounce)
byte ReadButton()
{
  if(IntervalCheck(OldButTime, ButInt))
  { //Minimum interval between keypresses to avoid multiple triggers
    // read the button value
    if (!digitalRead(PinButton)) //Button is LOW when pressed
    {
      OldButTime = millis(); //Store time of this button press for debounce
      //Force LCD refresh ASAP - using oldBLTime as just set to millis, saves creating another variable
      if(OldButTime > LcdUdInt)
      { //Setting previous update time to zero will trigger
        PrevLcdUd = 0;
      }
      else
      {
        PrevLcdUd = LongIntMax - LcdUdInt;
      }
      return 1; //button pressed
    }
    else
    {
      return 0; //button not pressed
    }
  }
  else //Insufficient time since previous button action: ignore button state
  {
    return 0;
  }
}

//__________

//Function to read character from file to temporary string
void ReadCharFromFile()
{
  byte DataRead; //Raw data from file
  DataFile.seek(FindNext(DataFile,':') + 1);
  DataRead = DataFile.read();
  memset(TempStr10, 0, sizeof(TempStr10)); //Clear the temporary string
  TempStr10[0] = DataRead;
}

//__________

byte ReadGSMResponse(byte RespLen, unsigned int StartAddr, unsigned int TimeoutLen1) //Read response
         from GSM, if not expected return 0 else return 1.  Parameters are response length, response
         start address
{
  char RespChar3 = 255;
  unsigned long GSMTimeoutDatum = millis();
  //Wait for response

  for (byte CharNo = 0; CharNo < RespLen + 2; CharNo++) //Read input including CR + LF before data
  {
    RespChar3 = GSMRead(TimeoutLen1); //Read input
    if(CharNo > 1) //First characters returned are CR LF, skip these
     {
      if(RespChar3 != (char)EEPROM.read(StartAddr + CharNo - 2)) //Does response match expected
         response?
      {
        return 0; //Return error
      }
    }
  }
  return 1; //Response matches
}

//__________

void ReadNumberFromFile(byte MaxDigits)
{
  byte DataRead; //Raw data from file
  DataFile.seek(FindNext(DataFile,':') + 1);
  byte CharNo = 1;
  memset(TempStr10, 0, sizeof(TempStr10)); //Clear the temporary string
  do
  {
    DataRead = DataFile.read();
```

```
    TempStr10[CharNo - 1] = DataRead;
    CharNo++;
  } while ((DataRead != 13) && (CharNo < MaxDigits + 1));
}

//__________

//Function to read pulse inputs, check button and update display
void ReadPulseInputs()
{
  for (byte InputNo = 0; InputNo < NPulseInputs; InputNo++)
  {
    //Store current input state
    byte InValue = digitalRead(PinIn[InputNo]); //Set 1 if high (closed) and 0 if low (open)
    if (InValue != OldInputState[InputNo])
    {
      //State has changed, log the time and store to oldvalue
      PrevPulseTime[InputNo] = millis();
      OldInputState[InputNo] = InValue;
      NewPulse[InputNo] = true;
    }
    else
    {
      //State has not changed
      if((IntervalCheck(PrevPulseTime[InputNo],PulseDebounce)) && (NewPulse[InputNo] == true) &&
         (InValue == 1))
      {
        //Debounce period exceeded, pulse not yet counted and contact is closed
        //Increment the reading
        PulseReading[InputNo] += PulseWeight[InputNo];
        //Record that the puse has been logged for this contact closure
        NewPulse[InputNo] = false;
        //Flash the Green LED if the input = that on the display
        if(InputNo == DispReg)
        {
          digitalWrite(PinLEDG, LOW);
        }
      }
      //(do nothing if state not changed and debounce not reached, pulse already logged or contact has
        opened)
    }
  }
  //Turn LED off after interval
  if(!digitalRead(PinLEDG)); //LED is on (low)
  {
    if((IntervalCheck(PrevPulseTime[DispReg],LEDPulseTime)))
    {
      digitalWrite(PinLEDG, HIGH);
    }
  }

  // read the button and take action if pressed (ReadButton Function returns 1)

  if (ReadButton())
  //Action on keypress
  {
    //Increase register displayed on LCD
    if(DispReg == (NPulseInputs - 1))
    {
      DispReg = 0;
    }
    else
    {
      DispReg++;
    }
    //Clear pulse indicator LED (as the time datum used to turn it off will now be for the wrong input
    digitalWrite(PinLEDG, HIGH);
  }

  //Update the LCD periodically
  if(IntervalCheck(PrevLcdUd, LcdUdInt))
  {
    //Read the clock
    HCRTC.RTCRead(I2CDS1307Add);

    //Display Signal Strength
    oled.set1X();
    oled.setCursor(75,0); //column pixel, row in 8 pixel blocks.
    oled.print(F("    ")); //Erase signal strength
    oled.setCursor(75,0);
    oled.print(GsmSig);
    oled.print(F("%"));

    //Display day of month
    oled.setCursor(0,2);
    oled.print(F("  "));
    oled.setCursor(0,2);
    oled.print(HCRTC.GetDateString()); //Date

    //Display HH:MM:SS
    oled.setCursor(0,4);
```

```
    oled.print(HCRTC.GetTimeString()); //Time

    //Display register subscript (1-based)
    oled.set2X();
    oled.setCursor(0,6);
    oled.print(DispReg + 1);

    //Display register value (unsigned int - so 5 digits)
    oled.setCursor(25,6);
    oled.print(F("      "));
    oled.setCursor(25,6);
    oled.print(PulseReading[DispReg]);

    //Reset LCD refresh timer
    PrevLcdUd=millis();
  }
}

//__________

//Function to Hardware Reset the GSM
void ResetGSM()
{
  //Hardware reset
  digitalWrite(GsmRST, LOW); //Reset, the GSM lights go out
  unsigned long TimeDatum = millis();
  do
  {
    ReadPulseInputs();
  } while (!IntervalCheck(TimeDatum, 1000)); //1 second reset
  digitalWrite(GsmRST, HIGH); //Return to normal operation, Network LED flashes fast while searching for
        a network then slow when a network is found.
  TimeDatum = millis();
  do
  {
    ReadPulseInputs();
  } while (!IntervalCheck(TimeDatum, 30000)); //30s to reconnect to network
}

//__________

//Update Signal Strength
void UpdateGSMSigStr()
{
  char SigStrDelim = (char)EEPROM.read(SigStrStartAddr + 1 + SigStrCmdLen); //Retrieve response
        delimiter from EEPROM
  unsigned long GSMTimeoutDatum = millis();
  gsm.flush(); //No need to flush input as read and discarded up to : delimiter
  memset(TempStr10, 0, sizeof(TempStr10)); //Clear the temporary string
  EEPROMToGSM(SigStrStartAddr + 1, SigStrCmdLen, 1); //Send the signal strength request

  do
  {
    ReadPulseInputs();
  } while ((!gsm.available()) && (!IntervalCheck(GSMTimeoutDatum, GSMTimeout)));

  if(gsm.available())
  {
    do
    {
      TempStr10[0] = GSMRead(GSMTimeout);
    }
    while (TempStr10[0] != SigStrDelim); //Read response up to the delimiter

    //Read the bytes that are the signal strength

    for (byte CharNo = 0; CharNo < SigStrRespLen; CharNo++)
    {
      TempStr10[CharNo] = GSMRead(GSMTimeout);
    }

    GsmSig = atoi(TempStr10) * 3; //Convert to number, multiply by 3 to give a rough percentage, before
         multiplying 0-9 is marginal quality, 10-19 is good quality and 20-31 is excellent quality. 99 =
         not detectable/unknown

    if (GsmSig == 297) //99 unscaled = not known
    {
      GsmSig = 0;
    }
  }
  else
  {
    GsmSig = 0;
  }
}

//__________

//Function to write latest meter reads to EEPROM
//Write Meter reads to eeprom memory for future use
void WriteReadsToEEPROM()
```

```
{
  for (byte InputNo = 0; InputNo < NPulseInputs; InputNo++)
  {
    EEPROMWriteUint32((4 * InputNo),PulseReading[InputNo]);
  }
}
```

APPENDIX 4 EEPROM_WRITE CODE

This code is used in place of the normal code in Appendix 3, to write the example gsmlog.cfg file to EEPROM by copying the file from the SD card. The first available EEPROM address for modem command data is shown on the OLED display.

```
//EEPROM_Write_V1
//Reads a file from the SD card and writes it to EEPROM starting at the start address
//Used to write the example config file (excluding FTP commands) to the EEPROM
//Similar code will be used in the main datalogging program to write the GSM settings from the config
file to EEPROM

#include <EEPROM.h> //EEPROM I/O
#include <SPI.h> //SPI Bus for SD Card
#include <SD.h> //Simple SD card library
#include <Wire.h> //I2C Comms for RTC on A4 and A5
#include <SSD1306Ascii.h> //OLED Display minimalist text library
#include <SSD1306AsciiWire.h> //OLED Display minimalist text library - extensions to support i2c

//DEFINITIONS to insert into code at compile time (so end up in flash memory not RAM)
#define I2COLedAdd 0x3C //I2C address for OLED display
#define SdCS 10 //Set ChipSelect Pin - use 10 as SD.h reserves this pin anyhow
#define EEPROMStartAdd 16 //First address to write to on EEPROM 0-15 are the 4 x 4 byte meter readings
//Indicator LED Pins
#define PinLEDG 16 //Green LED cathode on pin 15 (= A1) - Flash on pulse on input channel 1
#define PinLEDR 15 //Red LED cathode on pin 17 (= A2) - Initalisation Error LED (ON), Set Clock (Flash)
#define PinLEDAnode 14 //Anode pin, could just use permanent +5V


File DataFile; //Reference to file
SSD1306AsciiWire oled; //Initialise OLED display

void setup()
{
  //Initialise SD card pin
  pinMode(SdCS, OUTPUT);
  digitalWrite(SdCS, HIGH);

  //Initialise LED Pins
  pinMode(PinLEDAnode, OUTPUT);
  pinMode(PinLEDG, OUTPUT);
  pinMode(PinLEDR, OUTPUT);

  digitalWrite(PinLEDAnode, HIGH); //+5V supply
  digitalWrite(PinLEDG, HIGH); //Cathode high = off
  digitalWrite(PinLEDR, HIGH); //Cathode high = off

  //Initialise OLED
  Wire.begin(); //This would not be needed if the RTC was being used and initialised first
  oled.begin(&Adafruit128x64, I2COLedAdd); // set up the OLED's number of columns and rows
  oled.setFont(System5x7); //Standard font legible at x2, 11 characters per line
  oled.clear();
  oled.set2X(); //Illegible at 1x
  oled.setCursor(0,4); //column pixel, row in 8 pixel blocks.  4th row is first Functioning row on the
        faulty display

  //Initialise serial for debug output
  Serial.begin(9600);

  //Open the SD Card
  if (!SD.begin(SdCS)) //If fail to initialise SD comms trigger error condition and message
  {
    oled.println(F("SD Card"));
    oled.println(F("Error"));
    digitalWrite(PinLEDR, LOW); //RED LED ON
    //Exit setup
    return;
  }
  else //card is present
  {
    //Does the config file exist?
    if (SD.exists("gsmlog.cfg")) //If config file missing create an example and trigger error condition
         and message
    {
      DataFile = SD.open("gsmlog.cfg", FILE_READ); //open file, leave datestamp as is
      if (DataFile)
      {
        unsigned int FileLength; //Length of file to write in bytes
        byte TempBytes[2]; //Temporary store for the upper and lower byte of file size
```

```
        //Write file length to first two bytes
        FileLength = DataFile.size();
        TempBytes[0] = (byte) ((FileLength & 0x0000FF00) >> 8  );
        TempBytes[1] = (byte) ((FileLength & 0X000000FF)       );

        for (byte j = 0; j < 2; j++)
        {
          EEPROM.write(EEPROMStartAdd + j, TempBytes[j]); //First argument is address, second is value
        }

        //Write file to subsequent addresses
        for (unsigned int i = 0; i < FileLength ;i++)
        {
          EEPROM.write(EEPROMStartAdd + 2 + i,DataFile.read()); //Address, Data.  Address is start
         address plus 2 bytes for length value
        }
      }
      else
      {
        //couldn't open the config file
        oled.println(F("Config file"));
        oled.println(F("won't open"));
        digitalWrite(PinLEDR, LOW); //RED LED ON
        //Exit Setup
        return;
      }
      digitalWrite(PinLEDG, LOW); //GREEN LED ON
      digitalWrite(PinLEDR, LOW); //Red LED ON

      //Read data from EEPROM and write to Serial for debugging
      //Data length
      unsigned int DataLength; //Lentgth of data in bytes
      byte TempBytesRead[2]; //Temporary store for the upper and lower byte of file size

      for (byte j = 0; j < 2; j++)
      {
        TempBytesRead[j] = EEPROM.read(EEPROMStartAdd + j); //First argument is address, second is value
      }
      DataLength = ((int)(TempBytesRead[0]) << 8) + ((int)(TempBytesRead[1]));
      Serial.print("Bytes of Data: ");
      Serial.print(DataLength);
      Serial.println("");

      //Data
      for (unsigned int i = 0; i < DataLength ;i++)
      {
        Serial.print((char)EEPROM.read(EEPROMStartAdd + 2 + i)); //Address
        //Serial.print(" ");
      }
      Serial.println("");
      oled.println(F("Next Addr.: "));
      oled.println(EEPROMStartAdd + 2 + DataLength);
      digitalWrite(PinLEDR, HIGH); //Red LED OFF
    }
    else
    {
      oled.println(F("No Config"));
      oled.println(F("File on Card"));
      digitalWrite(PinLEDR, LOW); //RED LED ON
      //Exit setup
      return;
    }
  }
}

void loop()
{
  //Main loop not used
}
```

APPENDIX 5 GSM CODE

This code is used in place of the normal code in Appendix 3 for modem testing and to send text messages to keep the GSM account active.

```
//Uses serial monitor as terminal to talk to GSM modem

#include <SoftwareSerial.h>
SoftwareSerial gsm(2,3); //rx pin, tx pin

void setup()
{
 Serial.begin(9600);
 gsm.begin(4800);
}

void loop()
{
 if(gsm.available())
```

```
 {
  Serial.println(gsm.readString());
 }
 if(Serial.available())
 {
  gsm.println(Serial.readString());
 }
}
```

APPENDIX 6 CODE VERSION HISTORY

Table A5.1 Version history: main code.

Version	Notes
009	First tested version
010	Display layout improved
011	Pulse inputs changed from using internal pullup resistors to floating, and pulse detection changed to detect change to high state.

Table A5.2 Version history: EEPROM_Write code.

Version	Notes
001	First tested version

APPENDIX 7

APPENDIX 8 EEPROM MAP

Usage of the EEPROM memory is as listed below. Note: parameters are repeated according to the stated number of commands of each type.

Table A6.1 EEPROM memory map.

Start Address	Number of Bytes	Bytes Per Value	Data Type	Description	Example Value
0	16	4	Unsigned Long	Saved meter readings	0 to 999999
16	2	2	Unsigned Int	Length of example gsmlog.cfg file in bytes	409
	varies	1	Char	Data for example gsmlog.cfg file	(text)
	1	1	byte	Lenth of GSM signal strength command	6
	varies	1	char	GSM signal strength command	AT+CSQ
	1	1	char	Signal Strength Response Character before data	:
	1	1	byte	Signal Strength Response value: number of characters	3
	1	1	byte	Number of Comms Test Commands	3
	1	1	byte	Length of first test command	8
	varies	1	char	First test command	AT+CREG?
	1	1	byte	Length of first test command valid response	9
	varies	1	char	First test valid response	+CREG:0,1
	1	1	byte	Length of second test command	12
	varies	1	char	Second test command	AT+SAPBR=1,1
	1	1	byte	Length of second test command valid response	2
	varies	1	char	Second test valid response	OK
	1	1	byte	Length of third test command	11
	varies	1	char	Third test command	AT+FTPSTATE
	1	1	byte	Length of third test command valid response	11
	varies	1	char	Third test valid response	+FTPSTATE:0
	1	1	byte	Number of GSM Initialisation commands	6
	1	1	byte	Length of first GSM initialisation command	9
	varies	1	char	First GSM initialisation command	AT+CFUN=1
	1	1	byte	Length of first GSM initialisation command valid response	2
	varies	1	char	First GSM initialisation command valid response	OK
	1	1	byte	Length of second GSM initialisation command	10
	varies	1	char	Second GSM initialisation command	AT+CGATT=1
	1	1	byte	Length of second GSM initialisation command valid response	2
	varies	1	char	Second GSM initialisation command valid response	OK
	1	1	byte	Length of third GSM initialisation command	29
	varies	1	char	Third GSM initialisation command	AT+SAPBR=3,1,"CONTYPE","GPRS"
	1	1	byte	Length of third GSM initialisation command valid response	2
	varies	1	char	Third GSM initialisation command valid response	OK
	1	1	byte	Length of fourth GSM initialisation command	33
	varies	1	char	Fourth GSM initialisation command	AT+SAPBR=3,1,"APN","giffgaff.com"
	1	1	byte	Length of fourth GSM initialisation command valid response	2
	varies	1	char	Fourth GSM initialisation command valid response	OK
	1	1	byte	Length of fifth GSM initialisation command	30
	varies	1	char	Fifth GSM initialisation command	AT+SAPBR=3,1,"USER","giffgaff"
	1	1	byte	Length of fifth GSM initialisation command valid response	2
	varies	1	char	Fifth GSM initialisation command valid response	OK
	1	1	byte	Length of sixth GSM initialisation command	12
	varies	1	char	Sixth GSM initialisation command	AT+SAPBR=1,1
	1	1	byte	Length of sixth GSM initialisation command valid response	2
	varies	1	char	Sixth GSM initialisation command valid response	OK

1	1	byte	Number of FTP Initialisation commands	5
1	1	byte	Length of First FTP initialisation command	11
varies	1	char	First FTP initialisation command	AT+FTPCID=1
1	1	byte	Length of First FTP initialisation command valid response	2
varies	1	char	First FTP initialisation command valid response	OK
1	1	byte	Length of Second FTP initialisation command	29
varies	1	char	Second FTP initialisation command	AT+FTPSERV="web-ftp.ex.ac.uk"
1	1	byte	Length of Second FTP initialisation command valid response	2
varies	1	char	Second FTP initialisation command valid response	OK
1	1	byte	Length of Third FTP initialisation command	17
varies	1	char	Third FTP initialisation command	AT+FTPUN="web684"
1	1	byte	Length of Third FTP initialisation command valid response	2
varies	1	char	Third FTP initialisation command valid response	OK
1	1	byte	Length of Fourth FTP initialisation command	19
varies	1	char	Fourth FTP initialisation command	AT+FTPPW="Util-4EX"
1	1	byte	Length of Fourth FTP initialisation command valid response	2
varies	1	char	Fourth FTP initialisation command valid response	OK
1	1	byte	Length of Fifth FTP initialisation command	54
varies	1	char	Fifth FTP initialisation command	AT+FTPPUTPATH="/services.exeter.ac.uk/utilities/test/"
1	1	byte	Length of Fifth FTP initialisation command valid response	2
varies	1	char	Fifth FTP initialisation command valid response	OK
1	1	byte	Length of GSM Set Filename command	14
varies	1	char	GSM Set filename command (excluding filename and double quotation marks)	AT+FTPPUTNAME=
1	1	byte	Length of GSM Set Filename command valid response	2
varies	1	char	GSM Set Filename command valid response	OK
1	1	byte	Length of GSM Start FTP command	11
varies	1	char	GSM Start FTP command	AT+FTPPUT=1
1	1	byte	Length of GSM Start FTP valid response (exc. Final argument returned)	18
varies	1	char	GSM Start FTP valid response (exc. Final argument returned)	OK +FTPPUT:1,1,
1	1	byte	Length of GSM Request FTP Data send command	12
varies	1	char	GSM Request FTP Data send command (exc. Data length)	AT+FTPPUT=2,
1	1	byte	Length of GSM Request FTP Data send command valid response (exc data length confirmed)	11
varies	1	char	GSM Request FTP Data send command valid response (exc data length confirmed)	+FTPPUT: 2,
1	1	byte	Length of GSM End FTP Data send command	13
varies	1	char	GSM End FTP Data send command	AT+FTPPUT=2,0
1	1	byte	Length of GSM End FTP Data send command valid response	17
varies	1	char	GSM End FTP Data send command valid response	OK +FTPPUT:1,
1	1	byte	Length of GSM End FTP Session command	12
varies	1	char	GSM End FTP Session command	AT+SAPBR=0,1
1	1	byte	Length of GSM End FTP Session command valid response	2
varies	1	char	GSM End FTP Session command valid response	OK
			Remainder of memory unallocated	
1024			*End of memory (Arduino Nano)*	

APPENDIX 9 TYPICAL AT COMMANDS

AT commands are used to communicate with the GSM modem. These are ASCII text commands, each starting with AT and ending with the carriage return (ASCII 13) plus linefeed (ASCII 10) sequence. The modem typically responds with carriage return (ASCII 13) plus linefeed (ASCII 10) followed by data. The modem will typically echo the command before sending the response. This needs to be disabled by sending the command `ATE0`. Commands are manufacturer-specific and the commands below pertain tot the SIMCom SIM800L modem.

Table A7.1 Common AT commands for GSM modem .

AT Command	Expected Response[10]	Description
GSM Modem Initialisation Commands		
`ATE0`	`OK`	Turn command echo off (hard coded, not read from gsmlog.cfg)
`AT+CFUN=1`	`OK`	Set full functionality
`AT+CGATT=1`	*`OK`*	Start GPRS service
`AT+SAPBR=3,1,"CONTYPE","GPRS"`	`OK`	Set GPRS as service type
`AT+SAPBR=3,1,"APN","giffgaff.com"`	`OK`	Set APN for GiffGaff
`AT+SAPBR=3,1,"USER","giffgaff"`	`OK`	Set username for GiffGaff (note: password parameter "PWD" value is blank for GiffGaff so is not set)
`AT+SAPBR=1,1`	`OK`	Start session (first argument) using profile number 1 (second argument)
Communications Test Commands (if any fail, modem is rebooted and the initialisation commands above are sent)		
`AT+CREG?`	`+CREG:0,1`	Network registration status. Second returned parameter is 1 if registered to home network.
`AT+SAPBR=1,1`	`OK`	Start session (first argument) using profile number 1 (second argument)
`AT+FTPSTATE`	`+FTPSTATE:0`	Is an FTP transfer session ongoing?
FTP Initialisation Commands		
`AT+FTPCID=1`	`OK`	Set profile number for FTP session
`AT+FTPSERV="web-ftp.ex.ac.uk"`	`OK`	Set FTP server name
`AT+FTPUN="web684"`	`OK`	Set FTP server username
`AT+FTPPW="Util-4EX"`	`OK`	Set FTP server password
`AT+FTPPUTPATH="/services.exeter.ac.uk/utilities/test/"`	`OK`	Set destination path
FTP Data Transmission Commands		
`AT+FTPPUTNAME="test.txt"`	`OK`	Set destination filename
`AT+FTPPUT=1`	`OK` `+FTPPUT:1,1,y`	Create and open destination file for writing. *y* is the maximum number of bytes that can be written in a batch. If second parameter is not 1, this indicates an error.

`AT+FTPPUT=2,z`	`+FTPPUT: 2,z`
	
	`OK`
	`+FTPPUT:1,1,y`
		Request to start writing *z* bytes of data.
		Having received the response, the data should be sent to the modem.
		Once *z* bytes have been transmitted, the second response is received.

`AT+FTPPUT=2,0`	`OK`	Request to close the file
`AT+SAPBR=0,1`	`OK`	End session using profile number 1
Signal Strength check		
`AT+CSQ`	`+CSQ:a,b`	Signal strength check. *a* is signal quality (0-31, other values indicate error).

[10] Excluding carriage return (ASCII 13) plus linefeed (ASCII 10) sequences.

APPENDIX 10 GSM ACCOUNT DETAILS

The system has been tested with a GiffGaff SIM card

Username (for online account management): CEEArduino1

Password (for online account management): Ardu1n0NANO

Telephone number: 07713 100761

The account must be used once every six months to keep it active, either by transmitting data or a text message.

To transmit a text message, load the code listed in Appendix 5 to the Arduino. With the Arduino connected to the PC via USB open Serial Monitor Deluxe [6] (or another terminal program that allows transmission of non-printable ASCII characters) and enter the following commands:

```
AT+CMGF=1
```

```
AT+CMGS="01234 56789"
```
(replacing the number with a valid phone number)

Type text of message

Enter ASCII character 26 to end (enter \26\ in Serial Monitor Deluxe)

APPENDIX 11 EXAMPLE CONFIGURATION FILE

```
Update Time (N/Y): N
New Time (YYMMDDHHMMSS): 181211125800
Wait for keypress to start (N/Y): N
Max Pulse Input No (1/2/3/4): 2
Pulse Weight 1 (1-65535 units per pulse): 1
Pulse Weight 2 (1-65535 units per pulse): 1
Pulse Input 1 Name (1 to 8 characters): Input 1
Pulse Input 2 Name (1 to 8 characters): Input 2
Pulse Debounce (1-255ms): 50
Log Rate (1-65535 S): 300
Upload Rate (H/D): H
Multi Column Format (N/Y): Y
Specify Starting Readings (N/Y): N
Meter Reading Input 1 (0-999999): 5678
Meter Reading Input 2 (0-999999): 89745
Length of signal strength command: 6
Signal Strength Command: AT+CSQ
Signal Strength Response Data Delimiter: :
Signal Strength Response Length: 3
Number of Communication Test Commands: 3
Length of First Communication Test Command: 8
First Communication Test Command: AT+CREG?
Length of First Communication Test Command Valid Response: 9
First Communication Test Command Valid Response: +CREG:0,1
Length of Second Communication Test Command: 12
Second Communication Test Command: AT+SAPBR=1,1
Length of Second Communication Test Command Valid Response: 2
Second Communication Test Command Valid Response: OK
Length of Third Communication Test Command: 11
Third Communication Test Command: AT+FTPSTATE
Length of Third Communication Test Command Valid Response: 11
Third Communication Test Command Valid Response: +FTPSTATE:0
Number of GSM Initialisation Commands: 6
Length of First GSM Initialisation Command: 9
First GSM Initialisation Command: AT+CFUN=1
Length of First GSM Initialisation Command Valid Response: 2
First GSM Initialisation Command Valid Response: OK
Length of Second GSM Initialisation Command: 10
Second GSM Initialisation Command: AT+CGATT=1
Length of Second GSM Initialisation Command Valid Response: 2
Second GSM Initialisation Command Valid Response: OK
Length of Third GSM Initialisation Command: 29
Third GSM Initialisation Command: AT+SAPBR=3,1,"CONTYPE","GPRS"
Length of Third GSM Initialisation Command Valid Response: 2
Third GSM Initialisation Command Valid Response: OK
Length of Fourth GSM Initialisation Command: 33
Fourth GSM Initialisation Command: AT+SAPBR=3,1,"APN","giffgaff.com"
Length of Fourth GSM Initialisation Command Valid Response: 2
Fourth GSM Initialisation Command Valid Response: OK
Length of Fifth GSM Initialisation Command: 30
Fifth GSM Initialisation Command: AT+SAPBR=3,1,"USER","giffgaff"
Length of Fifth GSM Initialisation Command Valid Response: 2
Fifth GSM Initialisation Command Valid Response: OK
Length of Sixth GSM Initialisation Command: 12
Sixth GSM Initialisation Command: AT+SAPBR=1,1
Length of Sixth GSM Initialisation Command Valid Response: 2
Sixth GSM Initialisation Command Valid Response: OK
Number of FTP Initialisation Commands: 5
Length of First FTP Initialisation Command: 11
First FTP Initialisation Command: AT+FTPCID=1
Length of First FTP Initialisation Command Valid Response: 2
First FTP Initialisation Command Valid Response: OK
Length of Second FTP Initialisation Command: 29
Second FTP Initialisation Command: AT+FTPSERV="web-ftp.ex.ac.uk"
Length of Second FTP Initialisation Command Valid Response: 2
Second FTP Initialisation Command Valid Response: OK
Length of Third FTP Initialisation Command: 17
Third FTP Initialisation Command: AT+FTPUN="web684"
Length of Third FTP Initialisation Command Valid Response: 2
Third FTP Initialisation Command Valid Response: OK
Length of Fourth FTP Initialisation Command: 19
Fourth FTP Initialisation Command: AT+FTPPW="Util-4EX"
Length of Fourth FTP Initialisation Command Valid Response: 2
Fourth FTP Initialisation Command Valid Response: OK
Length of Fifth FTP Initialisation Command: 54
Fifth FTP Initialisation Command: AT+FTPPUTPATH="/services.exeter.ac.uk/utilities/test/"
Length of fifth FTP Initialisation Command Valid Response: 2
Fifth FTP Initialisation Command Valid Response: OK
Length of GSM Set Filename command (to start of filename): 14
GSM Set filename command (to start of filename): AT+FTPPUTNAME=
Length of GSM Set filename command Valid Response: 2
GSM Set filename command Valid Response: OK
Length of GSM Start FTP command: 11
GSM Start FTP command: AT+FTPPUT=1
Length of GSM Start FTP command Valid Response (excluding data buffer length returned): 19
GSM Start FTP command Valid Response (excluding data buffer length returned): OK

+FTPPUT: 1,1,
Length of GSM Request FTP Data send command: 12
```

```
GSM Request FTP Data send command (exc. Data length): AT+FTPPUT=2,
Length of GSM Request FTP Data send command Valid Response (exc. Data length): 11
GSM Request FTP Data send command Valid Response (exc. Data length): +FTPPUT: 2,
Length of GSM End FTP Data send command: 13
GSM End FTP Data send command: AT+FTPPUT=2,0
Length of GSM End FTP Data send command Valid Response: 17
GSM End FTP Data send command: OK

+FTPPUT: 1,
Length of GSM End FTP Session command: 12
GSM End FTP Session command: AT+SAPBR=0,1
Length of GSM End FTP Session command Valid Response: 2
GSM End FTP Session command: OK
```

APPENDIX 12 PULSE COUNT INPUTS

Pulse counting should be straightforward: digital inputs on the Arduino are "floating" at an indeterminate state when not connected, high when connected to +5V and low when connected to ground. The internal pull-up resistor can be enabled to ensure the input is high when disconnected and becomes low when connected to ground (e.g. via a switch contact). The pull-up option is enabled using the command `pinMode(x, INPUT_PULLUP);`, where *x* is the input number. This is how the inputs were originally configured, and worked fine with a mechanical switch contact.

Further testing used an ABB B21 111-100 electricity meter which has a pulse output. The specifications state that the output supports a DC voltage of 5 to 40 volts and a current of 2 to 100 mA. Although the schematics show a switch contact (e.g. relay), from on-line forums it would appear the output is a transistorised switch such as an open-collector output. The transistor circuit is driven by the power supply connected to the pulse output. When connected to the Arduino input as described above no pulses were detected. The inbuilt pullup resistor is in the order of tens of kilohms, indicating the resulting current in the circuit will be less than 1 mA and possibly insufficient to drive the pulse output circuit. The inputs were reconfigured as `pinMode(x, INPUT);`, which means the input would be floating. A 10kΩ resistor was then wired between the input and ground, and the meter's pulse output connected between +5V and the input. Pulses were detected when the Arduino was powered via USB, but not when it was powered by +9V on the V_{in} pin. It was thought the 5 volt supply might be insufficient to drive the circuit, since the minimum specificed voltage is 5 volts. The final attempt was to connect the pulse output of the meter to the 9 volt stabilised logger power supply, and the other terminal of the meter output to the Arduino input via a 470Ω resistor, with a further 470Ω resistor between the input and ground (Figure A11.1). This would reduce the voltage at the input to the desired figure of about 5 volts, and provide a current of about 10 mA to drive the meter's pulse output circuit. This was successful, but only if the pulse length on the meter was increased to about 250 ms.

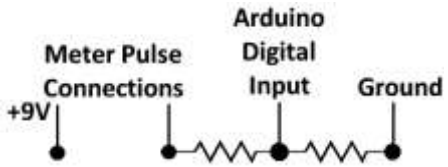

Figure A11.1. Pulse input connection for the ABB B21 111-100 electricity meter.

The above illustrates that whilst a simple switch contact can easily be detected, the pulse outputs of individual meters need to be tested and modifications might be necessary to ensure that pulses are detected reliably. Changes could be made to the code to check the pulse input state more frequently, and ensure that short-duration pulses are not missed.