**CIWS-MWM Datalogger**

Firmware Documentation

Firmware ver. 1.1.0

# Project Organization

1. The CIWS-MWM Datalogger project is organized using the standard approach using header files and source files, applied to an Arduino IDE environment. For those unfamiliar, C/C++ code is separated into header files and source files. The following example aims to make it clear why this is the case.
2. Consider the following C code:
3. int main()
4. {
5. int a = 2;
   1. int b = a \* 4;
   2. printf(“The result is %d\n”, b);
   3. int c = b \* 4;
   4. printf(“The result is %d\n”, c);
   5. return 0;
6. }
7. Obviously, this is a trivial example to illustrate how much more complex code is handled in this project. This code can be much more modular (and readable) by introducing a function:
8. int times4(int input)
9. {
10. return input \* 4;
11. }

int main()

{

int a = 2;

int b = times4(a);

printf(“The result is %d\n”, b);

int c = times4(b);

printf(“The result is %d\n”, c);

return 0;

}

The code is now more readable and modular, but now the main() function is at the bottom of the file (and it has to be, otherwise the function times4() would not be recognized in main(). Fortunately, programmers are good at finding ways around problems. The following code listing works just as well as the previous example:

1. int times4(int input);

int main()

{

int a = 2;

int b = times4(a);

printf(“The result is %d\n”, b);

int c = times4(b);

printf(“The result is %d\n”, c);

return 0;

}

1. int times4(int input)
2. {
3. return input \* 4;
4. }
5. Now, the main() function is much closer to the top of the file, making it easy to find the program’s starting point. The main() function recognizes the times4() function because it’s declared above main() using what’s called a function prototype, while the rest of times4() is defined below main(). This works really well for a handful of functions, but defining every function in one file becomes a mess in a project like the CIWS-MWM firmware. To counter this, two new files are created: times4.h and times4.cpp. The function prototype goes in times4.h, and the function definition goes in times4.cpp, leaving the file containing main() nice and clean:
6. #include “times4.h”

int main()

{

int a = 2;

int b = times4(a);

printf(“The result is %d\n”, b);

int c = times4(b);

printf(“The result is %d\n”, c);

return 0;

}

Again, though this example seems trivial, it is a very good way to organize a large firmware project like CIWS-MWM. The aim of this section was to illustrate why the project is organized the way it is.

# Main Loop: Firmware.ino

The file Firmware.ino is the program’s starting point. All Arduino projects have a .ino file. Ours contains six functions:

* void setup()
* void loop()
* void INT0\_ISR()
* void INT1\_ISR()
* storeNewRecord()
* byte bcdtobin(byte bcdValue, byte sourceReg)

The functions setup() and void() in the .ino file are actually called in the following manner:

int main()

{  
 setup();

while(1)

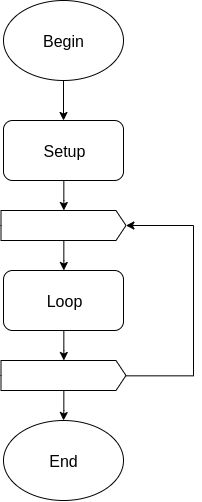
{

loop();

}

}

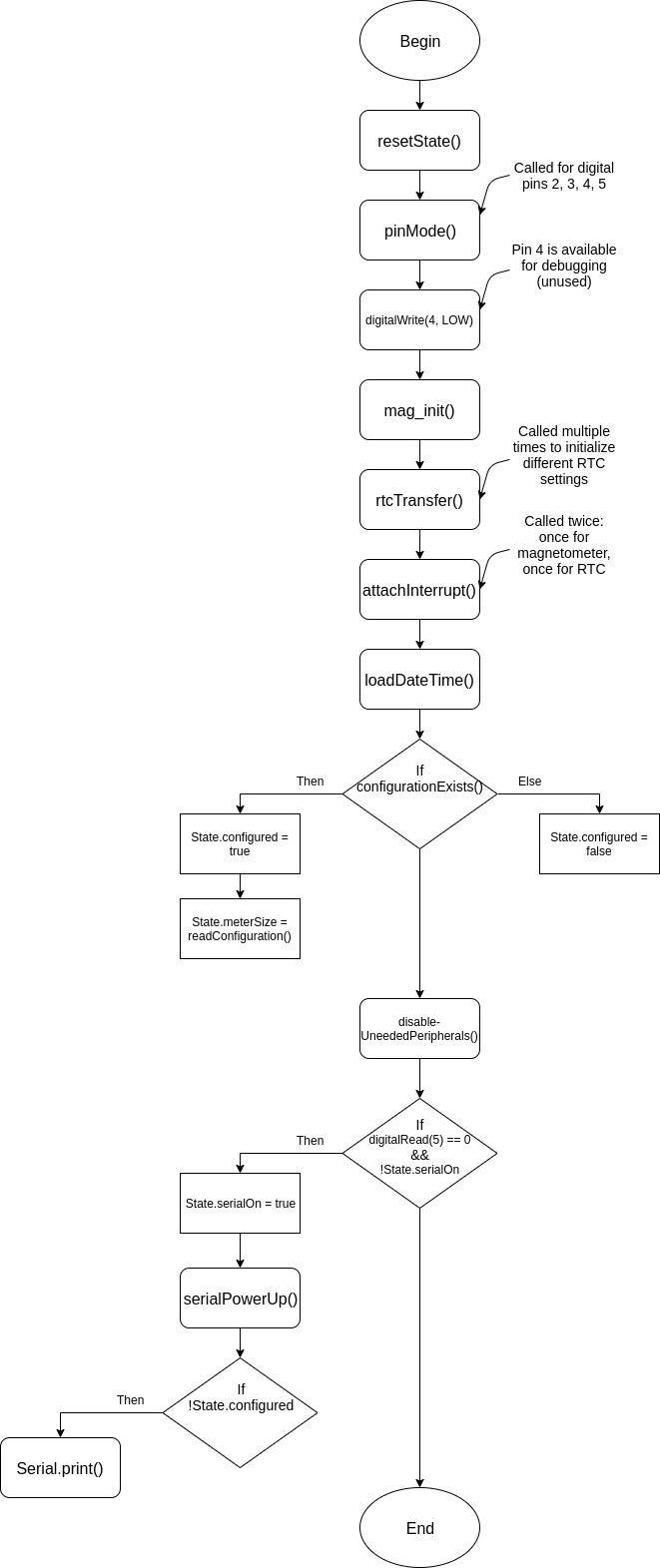
This way, setup() is called once, and loop() is called repeatedly until the microcontroller is reset. A flowchart of this process is shown here:



The setup() function does the following:

* Initializes the system state data structure
* Initializes GPIO Pins
* Initializes the magnetometer
* Initializes the real-time clock
* Sets up the magnetometer and real-time clock interrupt handlers
* Checks that the datalogger has valid configuration data
* Stops the clock for all unused peripherals
* Opens the serial interface if the serial activation button is pressed.

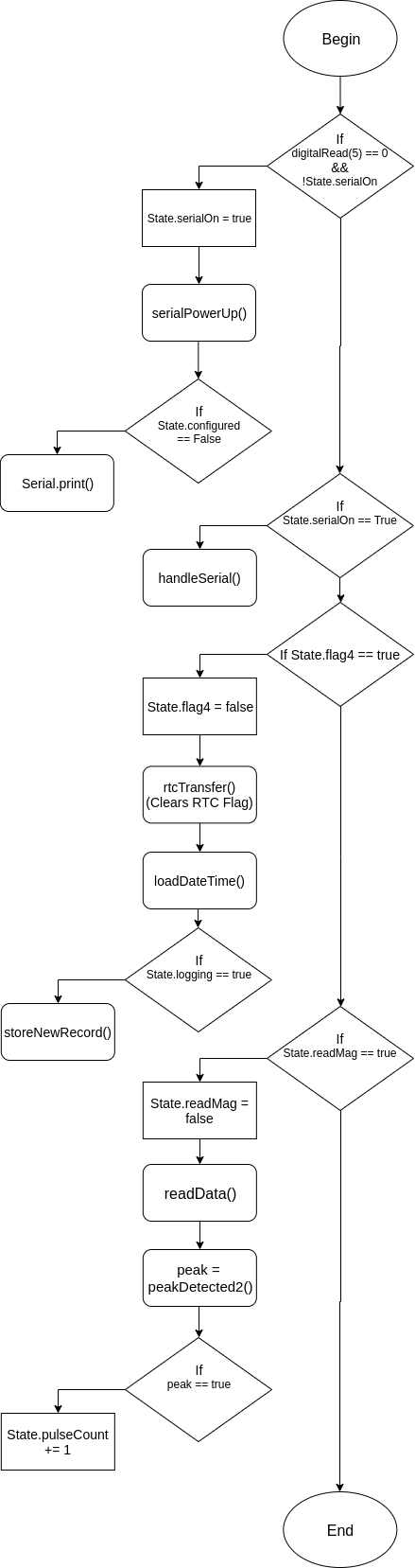
A flow chart of the setup() function is shown on the next page.



The loop() function is the datalogger firmware’s main loop, and performs the following actions:

1. Check if the serial activation button is pressed
2. Run the serial menu
3. Check if four seconds are up
4. Check if magnetometer data is ready
5. Process incoming data to count peaks

A flowchart of the loop() function is shown on the next page.



The functions INT0\_ISR() and INT1\_ISR() are both interrupt service routines. For those unfamiliar with interrupts, an interrupt service routine is a function executed when an event in hardware occurs. INT0\_ISR() code executes when the voltage on digital pin 2 transitions from low to high, and INT1\_ISR() code executes when the voltage on digital pin 3 transitions from high to low.

The voltage signal on digital pin 2 is controlled by the magnetometer. When the magnetometer has new data ready to report, it sets the voltage on pin 2 high, causing INT0\_ISR() to execute.

The voltage signal on digital pin 3 is controlled by the real time clock. When four seconds have elapsed, the real time clock sets the voltage on pin 3 low, causing INT1\_ISR() to execute.

Both interrupt service routines simply set a flag to true. The main loop checks these flags, and if they are set, responds accordingly. This is good practice; interrupt service routines need to be kept as short as possible.

The function storeNewRecord() is primarily responsible for storing data records to the datalogger’s SD Card by performing the following actions:

1. Gather date and time information
2. Activate the SPI module’s clock
3. Open a file on the SD Card
4. Print the following fields separated by commas:
   1. Timestamp
   2. Record Number
   3. Pulse Count
5. Close the file
6. Increment the record number
7. Deactivate the SPI module’s clock

The function bcdtobin() is responsible for converting the Binary Coded Decimal (BCD) data from the real-time clock into standard binary data, and takes a BCD value and a bitmask corresponding to the real-time clock from which the BCD value came from. This conversion is accomplished by multiplying the top 4 bits by ten and adding that number to the bottom four bits.

# System State: state.h and state.cpp

1. These files define two C/C++ structs, State and SignalState.
2. State keeps track of several important values:

* byte pulseCount – The number of pulses in the current sample period.
* byte lastCount – The number of pulses in the previous sample period.
* unsigned int totalCount – The number of pulses since logging started.
* unsigned long recordNum – The record number of the current sample period.
* bool logging – Boolean flag: True if the device is logging, false if it is not.
* bool flag4 – Boolean flag: True if four seconds has passed, false if not.
* bool serialOn – Boolean flag: True if serial interface is active, false if it is not.
* bool SDin – Boolean flag: True if an SD card has been initialized, false if not.
* bool readMag – Boolean flag: True if magnetometer data is ready, false if it is not.
* char meterSize – Stores the meter size, used to compute water flow.
* bool configured – Boolean flag: True if valid configuration data exists in memory.
* char filename[13] – Stores the current filename.

This State struct is initialized using the function void resetState(volatile State\_t\* State). The pulse counts are initialized to zero, the record number is initialized to one, and the boolean flags are initialized to false.

SignalState keeps track of values used for processing the magnetometer signal. A few values are left over from an older signal processing algorithm. These are noted, and will be removed in a future release.

* float x[2] = {0, 0}; – Input signal from the magnetometer.
* float s = 0; - From old algorithm.
* float sf[2] = {0, 0}; - From old algorithm.
* float a = 0.95; - DC Removal filter pole.
* float y[2] = {0, 0}; - Output signal from DC Removal filter.
* float offset = -0.005; - From old algorithm.
* bool slopeWasPositive = false; - From old algorithm.
* bool slopeIsPositive = false; - From old algorithm.
* bool triggered = false; - Boolean flag for software-based schmitt trigger.

The use of these values is detailed in the next section, **Peak Detection**.

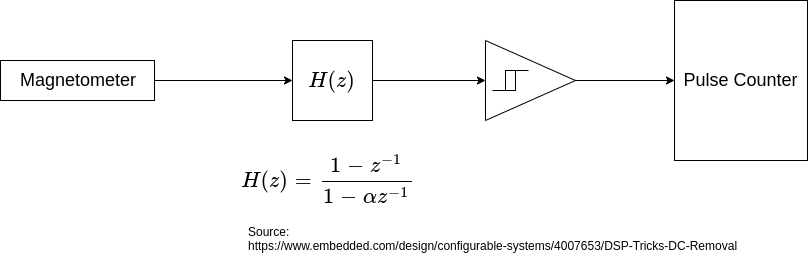
# Peak Detection: detectPeaks.h and detectPeaks.cpp

These files define two functions:

1. bool peakDetected(volatile SignalState\_t\* signalState)
2. bool peakDetected2(volatile SignalState\_t\* signalState)

The function peakDetected() is deprecated, and will be entirely removed by the next version.

The function peakDetected2() is responsible for detecting peaks corresponding to water flow in the signal read by the magnetometer. Before going over the code, it is crucial to understand the filtering algorithm implemented by the code.

The function peakDetected2() implements the two middle blocks of the above diagram. The block *H(z)* removes the signal’s DC offset. This centers the signal at zero, which permits a simple peak-detecting algorithm called a software schmitt trigger, the triangular block in the diagram.

The schmitt trigger watches for the signal to cross its high and low thresholds. When the signal crosses the high threshold, the schmitt trigger will count it as a peak, but will not count any other high-threshold crossing as a peak until the signal crosses the low threshold. This ensures that noise in the magnetometer’s signal cannot be counted as extra peaks. This peak detection is implemented in the following way:

Begin:

/\*\* H(z) \*\*/

y[n] = a \* y[n - 1] + x[n] - x[n - 1]

/\*\* Schmitt Trigger \*\*/

If triggered:

If y[n] < -1:

triggered is false

If not triggered:

If y[1] > 1:

triggered is true

return peak = true

End

1. A C/C++ struct called SignalState stores the current input sample (x[n]), the previous input sample (x[n - 1]), and the previous output sample (y[n – 1]). SignalState also holds a and triggered. Threshold values for the Schmitt Trigger are declared inside the function peakDetected2(). This information is all that is necessary to detect if a peak is occurring in the magnetometer’s signal. SignalState is defined in the file state.h.
2. The value a is the pole of *H(z)*. For this filter, it is desirable for the pole to be close to, but not equal to, one. For any digital filter, a pole greater than or equal to one causes instability.

This function is only called once every time a data sample is collected from the magnetometer.

# Magnetometer: magnetometer.h and magnetometer.cpp

The following functions are defined in magnetometer.cpp and magnetometer.h:

1. bool mag\_init()
2. void read\_mag(int8\_t\* data)
3. void mag\_transfer(int8\_t\* data, uint8\_t reg, uint8\_t numBytes, uint8\_t RW)
4. readData(volatile SignalState\_t\* SignalState)
5. initializeData(volatile SignalState\_t\* SignalState)

These functions are responsible for the initialization and reading of an LIS3MDL magnetometer. Above, they are listed in order of appearance, but in the following sections, they will be covered in order from low-level to high-level functionality.

The function mag\_transfer() takes an array of data, a register number, the length of the array of data, and a read/write flag as input, and is responsible for writing data and reading data to and from the magnetometer. Most of the time, data will be read from the magnetometer, but data does get written to the magnetometer when initialized on startup. The data is read and written via an I2C serial bus, and the Arduino IDE’s Wire library is used for I2C communication.

The function mag\_init() does not take any input, and is responsible for initializing the magnetometer when the datalogger is started up. It utilizes the function mag\_transfer() to write initialization data to the magnetometer. The following table lists the data written to each of the magnetometer’s control registers, along with the behavior due to the data written.

|  |  |  |
| --- | --- | --- |
| **Register** | **Data** | **Behavior** |
| Control Register 1 | 0x32 | - Temperature sensor disabled  - X-Axis in Medium Performance Mode  - Y-Axis in Medium Performance Mode  - Output data rate ~560-570 Hz  - Self-test disabled |
| Control Register 2 | 0x00 (Default) | - Output data scaled to +/- 4 gauss |
| Control Register 3 | 0x00 | - Low-power mode disabled  - Sample: Continuous-conversion mode |
| Control Register 4 | 0x04 | - Z-Axis in Medium Performance Mode |
| Control Register 5 | 0x80 | - Set FAST READ |

More data for these registers can be found in section 7 of the LIS3MDL datasheet. The key takeaway from this table is that the magnetometer is set to a high output data rate, medium performance mode for all axes, reads on a +/- 4 gauss scale, and does what the datasheet refers to as FAST READ. FAST READ means that only the top half of the two-byte data sample are reported by the magnetometer. This is beneficial in part because most signal noise is contained in the lower half byte; however, the resulting signal is less smooth, so a signal consisting of two-byte data samples is being considered for a future version. The combination of a fast output data rate and all axes set to medium performance mode results in an output data rate of about 570 Hz (measured experimentally).

The function read\_mag() takes an array of data, which it will populate with data from the magnetometer by calling mag\_transfer(data, OUT\_X\_H, 3, MAG\_READ); This reads the data output of the X, Y, and Z axes of the magnetometer.

The function readData() takes a SignalState structure, described previously. The function calls read\_mag() and stores the output byte from the x axis in the SignalState structure as a floating point number, allowing it to be processed by the functions in detectPeaks.cpp.

The function initializeData() actually does the same thing as the function readData(). They are likely to be merged in a future version.

# Real Time Clock: RTC\_PCF8523.h and RTC\_PCF8523.cpp

The following functions are defined in RTC\_PCF8532.h and RTC\_PCF8523.cpp:

* byte rtcTransfer(byte reg, byte flag, byte value)
* void registerDump()
* void loadDateTime(Date\_t\* Date)

These files also define a list of hexadecimal addresses for the RTC’s registers and a Date structure, which holds the current year, month, day, hour, minute, and second (updated every four seconds).

The function rtcTransfer() is responsible for transferring data to the RTC, and takes an eight-bit register number, a read/write flag, and an eight-bit value to write. This function utilizes the Arduino IDE’s Wire library for I2C communication with the RTC.

The function registerDump() prints the data in each register to a serial terminal, and is used to verify the contents of each RTC register.

The function loadDateTime() reads all of the RTC’s date and time registers, and stores the resulting data in a Date\_t structure. This function is called every four seconds.

The RTC is initialized in Firmware.ino using the following calls to rtcTransfer():

rtcTransfer(reg\_Tmr\_CLKOUT\_ctrl, WRITE, 0x3A);

rtcTransfer(reg\_Tmr\_A\_freq\_ctrl, WRITE, 0x02);

rtcTransfer(reg\_Tmr\_A\_reg, WRITE, 0x04);

rtcTransfer(reg\_Control\_2, WRITE, 0x02);

rtcTransfer(reg\_Control\_3, WRITE, 0x80);

These calls are likely to be combined into a single function in a future release. Below is a table detailing what each register configuration does:

|  |  |  |
| --- | --- | --- |
| **Register** | **Data** | **Behavior** |
| Timer CLOCKOUT Control | 0x3A | - Disable 1-second clock output  - Configure Timer A as countdown timer.  - Disable Timer B |
| Timer A Frequency Control | 0x02 | - Use a 1 Hz source clock for Timer A countdown. |
| Timer A Register | 0x04 | - Set Timer A to countdown from 4 (4-second timer). |
| Control Register 2 | 0x02 | - Clears any interrupt flags  - Enables Timer A Countdown Interrupt  - Disables all other interrupts in Control Register 2 |
| Control Register 3 | 0x80 | - Enables battery switch-over function in standard mode  - Clears any interrupt flags  - Disables interrupts in Control Register 3 |

# Configuration Data: configuration.h and configuration.cpp

The ATmega328p microcontroller used for this datalogger has a small chunk of Electrically Erasable Programmable Read-Only Memory, or EEPROM, in which datalogger configuration data is stored. 15 bytes of EEPROM are used, laid out in the following organization:

|  |  |
| --- | --- |
| **Address** | **Data** |
| 0x0 | Site Number (100’s Place) |
| 0x1 | Site Number (10’s Place) |
| 0x2 | Site Number (1’s Place) |
| 0x3 | Meter Size |
| 0x4 | Datalogger ID (100’s Place) |
| 0x5 | Datalogger ID (10’s Place) |
| 0x6 | Datalogger ID (1’s Place) |
| 0x7 | File Number (1000’s Place) |
| 0x8 | File Number (100’s Place) |
| 0x9 | File Number (10’s Place) |
| 0xA | File Number (1’s Place) |
| 0xB | Checksum 0 |
| 0xC | Checksum 1 |
| 0xD | Checksum 2 |
| 0xE | Checksum 3 |

Three functions are used in relation to this configuration data:

* bool configurationExists();
* uint8\_t readConfiguration(uint8\_t segment);
* void writeConfiguration(uint8\_t segment, char data);

The function configurationExists() checks the EEPROM for valid configuration data by reading the checksum bytes in addresses 0xB – 0xE using the readConfiguration() function. It then calculates two different values: Checksum 0 + Checksum 1 and Checksum 2 + Checksum 3. If they both equal 0xFF, then the configuration data is considered present and valid.

The function readConfiguration() first waits for completion of any EEPROM writes in progress. It then loads a read address into the EEPROM Address Register (EEAR) and initiates an EEPROM Read operation. It then returns the data in EEDR, the register in which data read from the EEPROM is stored.

The function writeConfiguration() takes both an address and a data byte. It first waits for completion of any EEPROM writes in progress. It loads a read address into the EEAR, loads data into the EEDR, enables EEPROM writes, and starts an EEPROM write.

# Power Reduction: powerSleep.h and powerSleep.cpp

The MWM Datalogger is a battery-powered logger, and as such, it is crucial that energy is saved in every possible part of the device to prolong the possible logging period. This is done in part with the functions listed here:

* void enterSleep();
* void disableUnneededPeripherals();
* void twiPowerUp();
* void twiPowerDown();
* void serialPowerUp();
* void serialPowerDown();
* void SDPowerUp();
* void SDPowerDown();

If you’ve been examining the source code, you’ve no doubt encountered these functions. That’s because most peripherals are powered down on start-up, and are only powered on when needed. These functions make use of functions from <avr/sleep.h> and <avr/power.h>. These libraries are available through the Arduino IDE, or by installing avr-libc. Power-up and power-down are accomplished by activating and deactivating the clock, respectively. When a peripheral receives no clock signal, it does nothing, and is effectively powered off.

The function enterSleep() puts the microcontroller into a standby mode in which very little power is consumed. The mode SLEEP\_MODE\_STANDBY is selected, sleep is enabled, and the microcontroller is put to sleep. The program is halted at this point. On wake-up, sleep is disabled and the function returns control. ***Use of this function causes data samples to be recorded incorrectly, and as such is unused. This function will be put in use again once the error is corrected.***

The function disableUnneededPeripherals() is called on startup in Firmware.ino. This function first disables the ADC by writing 0x00 to the ADCSRA register. Once this is done, this function calls the following functions from <avr/power.h>:

power\_adc\_disable();

power\_timer0\_disable();

power\_timer1\_disable();

power\_timer2\_disable();

power\_twi\_disable();

power\_usart0\_disable();

power\_spi\_disable();

This deactivates the clock to all of the above peripherals.

All of these peripherals are turned on again when needed using the corresponding PowerUp() function:

* I2C bus: twiPowerUp()
* Serial Port: serialPowerUp()
* SD Interface (SPI): SDPowerUp()

Each PowerUp() function is similar:

I2C Bus:

void twiPowerUp(){

power\_twi\_enable(); // From <avr/power.h>

\_delay\_us(1); // Pause execution for peripheral to start

Wire.begin(); // Initialize Arduino’s Wire library

\_delay\_us(1); // Pause execution for peripheral to start

return;

}

Serial Port:

void serialPowerUp(){

power\_usart0\_enable(); // From <avr/power.h>

\_delay\_ms(10); // Pause execution for peripheral to start

Serial.begin(9600); // Initialize Arduino’s Serial library

\_delay\_ms(10); // Pause execution for peripheral to start

Serial.print(F(">> Logger: Logger ready.\n>> User: "));

return;

}

SD Interface (SPI):

void SDPowerUp(){

power\_spi\_enable(); // From <avr/power.h>

return;

}

All of these peripherals are turned off again when needed using the corresponding PowerDown() function:

* I2C bus: twiPowerDown()
* Serial Port: serialPowerDown()
* SD Interface (SPI): SDPowerDown()

All of the PowerDown() functions simply call their respective power\_xxx\_disable() function from the <avr/power.h> library (as seen in disableUnneededPeripherals()).

# Serial User Interface: handleSerial.h and handleSerial.cpp

Before going over the functions defined by these two files, It’s useful to understand how the firmware processes user input generally. Recall from the main loop that the function handleSerial() is called once per loop, assuming the interface is active. The handleSerial() function is the central piece to the entire interface, and calls the other functions defined in these two files based on the user input.

There are a lot of functions, so hold on tight. All of these functions are used in relation to interaction between the user and the datalogger:

* void handleSerial(volatile State\_t\* State, Date\_t\* Date, volatile SignalState\_t\* SignalState);
* void setConfiguration(volatile State\_t\* State);
* void cleanSD(volatile State\_t\* State);
* void viewDateTime(Date\_t\* Date);
* void exitSerial(volatile State\_t\* State, Date\_t\* Date);
* void ejectSD(volatile State\_t\* State);
* void printHelp();
* void initSD(volatile State\_t\* State);
* void listFiles(void);
* void startLogging(volatile State\_t\* State, volatile SignalState\_t\* SignalState, volatile Date\_t\* Date);
* void stopLogging(volatile State\_t\* State);
* void updateDateTime(Date\_t\* Date);
* void RTC\_Doctor();
* char getInput();
* char getNestedInput();
* void printWater(State\_t\* State);
* void printConfig(State\_t\* State);
* void createHeader(State\_t\* State);
* void nameFile(State\_t\* State, Date\_t\* Date);
* void incrementFileNumber(void);

The handleSerial() function is passed the State, Date, and SignalState structures. These structs may or may not be used in the next function call, but each one has the potential of being used. This function checks for user input from the serial interface (remember, this function is called once per loop, whether the user inputs data or not). If it’s there, handleSerial() calls the function getInput(), which will be described later. For now, it is enough to know that it retrieves input that the user types.

Next, handleSerial() calls another function based on what input it received. A table on the next page details what functions get called for what input is received:

|  |  |  |
| --- | --- | --- |
| **User Input** | **Function Call** | **Notes** |
| g | setConfiguration(State); | Edit datalogger configuration data. |
| c | cleanSD(State); | Delete files from an SD card. |
| d | viewDateTime(Date); | Display the current date/time. |
| e | exitSerial(State, Date); | Deactivate the serial interface. |
| E | ejectSD(State); | Tells the program that the SD card is not in use. |
| h | printHelp(); | Print a list of commands. |
| i | initSD(State); | Initialize the SD card. |
| l | listFiles(); | List files on the SD card. |
| p | printConfig(State); | Display the datalogger’s configuration data. |
| R | RTC\_Doctor(); | Debug the Real Time Clock. |
| s | startLogging(State, SignalState, Date); | Initiate logging session. |
| S | stopLogging(State); | Terminate a logging session. |
| u | updateDateTime(Date); | Edit the current date/time. |
| w | printWater(State); | Display current water use (if logging session is active). |

The datalogger expects single-character commands. If the user enters a string of characters, then each one is executed simultaneously. If any entered character is not listed in the list above, then the datalogger will report that the command is an unknown command.

The function getInput() calls the function Serial.read() to pull user input from the serial interface. It then prints the input to the terminal, so the user can see what he/she typed (unless the user typed a newline character).

Some of the functions that get called require their own input. This case is handled with the getNestedInput() function. This function is actually quite similar to handleSerial(). It loops until the user inputs a newline character. Each time it loops, the function checks for available serial data. If it’s there, it calls getInput(). It can be thought of as a nested handleSerial() for individual functions.

The function setConfiguration() takes a State struct as an input and prompts the user to enter configuration data for the datalogger’s site number, ID number, meter size, and file number. It also displays the current configuration setting for the user’s convenience. Pseudo code for this function is listed here:

Begin:

Let user know the datalogger is configuring

Prompt user for 3-digit site number and display current site number

Get user input and store as the new 3-digit site number

Prompt user for 3-digit ID number and display current ID number

Get user input and store as the new 3-digit ID number

Prompt user for new meter size

List meter options

Get user input

If input == 1

Display meter choice to user

Store new meter size in State

Else If input == 5

Display meter choice to user

Store new meter size in State

Else

Tell user the meter size was improperly configured.

Prompt user for 4-digit file number and display current file number

Get user input and store as the new 4-digit file number

Write configuration checksums

Call printConfig(State) (Prints configuration for user’s convenience)

End

As a side note, this version only has two meter options, while the project has expanded to include a few more. This is to be addressed in a future version of the code.

The function cleanSD() takes a State struct as an input, and allows the user to delete certain files on an SD card. This is accomplished using the SD library for Arduino. The pseudo code for this function is listed below:

Begin:

If the State.SDin flag is false

Tell the user that the SD card is not initialized

Exit this function

If the State.logging flag is false

Call listFiles() (Prints all files on the SD Card)

Prompt the user for a file to remove

Get user input

Warn the user that the file will permanently delete the file

Ask the user whether or not to continue

Get user input

If yes

Call SDPowerUp()

Call SD.begin()

Call SD.remove(filename)

Call SDPowerDown()

Call listFiles()

Else

Tell the user that files can’t be deleted while logging

End

The function viewDateTime() prints the date and time to the terminal, and takes a Date struct as an input. The pseudo code is listed below:

Begin:

Copy data from the Date struct

Print Month

Print Day

Print Year

Print Hour

If Minute < 10

Print 0

Print Minute

End

The function exitSerial() sets a flag in the State struct which tells the rest of the system that the serial interface is not active. It also calls serialPowerDown() from powerSleep.h, which helps extend the device’s battery life. The pseudo code is listed below:

Begin:

Call viewDateTime(Date)

Tell the user he/she is exiting the interface

Set serialOn flag in State false

Pause execution for one second (allows time for messages to finish)

Call serialPowerDown()

End

The function ejectSD() sets a flag in the State struct which tells the rest of the system that the SD card is “ejected”, or not initialized, possibly not even in the system. The pseudo code is listed below:

Begin:

If the State.logging flag is true

Tell the user the SD Card cannot be ejected while logging

Exit this function

If the State.SDin flag is false

Tell the user the SD Card is already ejected

Exit this function

Set the State.SDin flag false

Tell the user the SD Card may be removed

End

The function printHelp() prints a list of commands to the terminal for the user’s reference. The printing, as always, is done with the Arduino IDE’s Serial.print() function. The pseudo code is listed below:

Begin:

Print list of commands (using Serial.print())

End

The function initSD() takes a State struct as an input, and is responsible for initializing the Arduino SD library and letting the rest of the system know that the SD card is initialized. The pseudo code is listed below:

Begin:

Call SDPowerUp()

If SD.begin() returns successful

Set State.SDin flag True

Tell the user the SD Card is ready for use

Else

Set State.SDin flag False

Tell the user the SD Card could not be initialized.

Call SDPowerDown()

End

The function listFiles() uses Arduino’s SD library to display all of the files available on the SD Card for the user’s convenience. The pseudo code is listed below:

Begin:

Call SDPowerUp()

Open the root partition of the SD card

Rewind directory (starts listing files at beginning of card)

Print top of table (File name, file size)

While there’s still files

Open the next file

Print the file name

Print the file size

Close the file

Call SDPowerDown()

End

The function RTC\_Doctor() is designed for easy resetting of the real time clock. This feature was included as we developed the proper settings for the real time clock, and if we got something wrong, we could check RTC registers and even write data to them.

The function startLogging() takes State, SignalState, and Date structs as inputs, and is responsible for initiating a logging session. It Generates a new filename (that does not already exist on the SD card), sets the system state flags and values, writes a header for the data file, initializes data for the peak detection algorithm, and enables the magnetometer interrupt, so the datalogger will respond when the sensor signals that data is ready. The pseudo code is listed below:

Begin:

If State.SDin flag is True

If State.logging flag is True

Tell user the device is already logging

Else

Call nameFile(State, Date) (Generates a new file name)

Call SDPowerUp()

Loop While the current filename matches a file on the SD Card

Call incrementFileNumber()

Call nameFile(State, Date)

End Loop

Print file name for user's convenience

Set State.logging flag to True

Set State.recordNum = 1

Set State.pulseCount = 0

Set State.lastCount = 0

Set State.totalCount = 0

Call createHeader(State) (Writes a header to the data file)

Call initializeData(SignalState)

Enable magnetometer sensor interrupt

Tell user that the logging session has started

Else

Tell user the logging session couldn't start without an SD card

End

The function stopLogging() takes the State struct as an input. It sets a flag to stop the logging session, disables the magnetometer sensor interrupt, and increments the file number for the next logging session. Note that the file number checking implemented in the function startLogging() is a preventative measure, it should never have to execute the loop code. The pseudo code is listed below:

Begin:

If State.logging flag is True

Set State.logging flag False

Disable Magnetometer Sensor interrupt

Tell user that the logging session has ended

Call incrementFileNumber()

Else

Tell user that the device is not logging.

End

The function updateDateTime() takes the Date struct as an input. The function prompts the user for date and time information, and uploads that information to the RTC. The pseudo code is listed below:

Begin:

Tell user the device is updating the date and time

Prompt user for a new month in MM format

Receive 10's place (with call to getNestedInput())

Receive 1's place (with call to getNestedInput())

Prompt user for a new day in DD format

Receive 10's place (with call to getNestedInput())

Receive 1's place (with call to getNestedInput())

Prompt user for a new year in YY format

Receive 10's place (with call to getNestedInput())

Receive 1's place (with call to getNestedInput())

Prompt user for a new hour in HH format

Receive 10's place (with call to getNestedInput())

Receive 1's place (with call to getNestedInput())

Prompt user for a new minute in MM format

Receive 10's place (with call to getNestedInput())

Receive 1's place (with call to getNestedInput())

Convert month digits to Binary Coded Decimal

Convert day digits to Binary Coded Decimal

Convert year digits to Binary Coded Decimal

Convert hour digits to Binary Coded Decimal

Convert minute digits to Binary Coded Decimal

Upload new month to RTC (call to rtcTransfer)

Upload new day to RTC (call to rtcTransfer)

Upload new year to RTC (call to rtcTransfer)

Upload new minute to RTC (call to rtcTransfer)

Upload 0x00 (new second) to RTC (call to rtcTransfer)

call loadDateTime(Date) (Copies new date and time from RTC)

call viewDateTime(Date) (Prints Date and Time for user)

Tell user the date and time have been reset.

End

The function getInput(), as discussed previously, receives input from the user for the top level of the handleSerial() function. The user input is returned as the function's output. The pseudo code is listed below:

Begin:

Read input from user with call to Serial.read()

If input is not a newline

Print the input (so user can see what was typed)

End

The function getNestedInput(), as discussed previously, is very similar to the function getInput(). The difference is that getNestedInput() loops until it has received input from the user, while getInput() is only designed to check once per iteration of handleSerial(). As such, getNestedInput() is used by functions called from handleSerial() which also need user input. The pseudo code is listed below:

Begin:

Set finished to False

Loop while finished is False

input = getInput()

If input is not a newline

finished is True

End Loop

End

The function printWater() takes the State struct as an input, and prints water flow data for the user to see. The pseudo code is listed below:

Begin:

Tell the user that the data is from the previous sample

Copy State.lastCount to pulses

Copy State.totalCount to totalPulses

If State.meterSize is 1

Set convFactor = 0.033

Else If State.meterSize is 5

Set convFactor = 0.0087

Else

Set convFactor = 0

Total Flow = pulses \* convFactor

Total Flow Since Start = totalPulses \* convFactor

Average Flowrate = Total Flow \* 60 / 4

Print pulses

If convFactor is not 0

Print Total Flow

Print Average Flowrate

Print totalPulses

Print Total Flow Since Start

Else

Tell user the meter configuration is invalid.

End

The function printConfig() takes the State struct as an input. It reads the configuration data in the datalogger's EEPROM and prints it to the terminal. The pseudo code is listed below:

Begin:

Read and Print the 3-digit Site Number

Read and Print the 3-digit ID Number

Read and Print the 4-digit File Number

If State.meterSize is 1

Print 1"

Else If State.meterSize is 5

Print 5/8"

Else

Print INVALID

End

The function createHeader() takes the State struct as an input. It is responsible for writing a header to the current data file:

Site #: xxx

Datalogger ID #: xxx

Meter Size: x"

Time,Record,Pulses

The pseudo code is listed below:

Begin:

Call SDPowerUp()

Open the logging session's file

Write the 3-digit Site Number

Write the 3-digit ID Number

If State.meterSize is 1

Print 1"

Else If State.meterSize is 5

Print 5/8"

Else

Print INVALID

Print "Time,Record,Pulses\n"

Close the file

Call SDPowerDown()

End

The function nameFile() takes a both the State and Date structures as inputs, and names a file based on the site number and file number stored in the datalogger's EEPROM in the following format, where 's' denotes a site number digit and 'f' denotes a file number digit:

sss\_ffff.csv

The function incrementFileNumber() reads the file number digits from the datalogger's EEPROM, sums them together, adds one, and rewrites them back to the EEPROM.

# Table of Functions and Structs (Lexicographical Order)

|  |  |  |
| --- | --- | --- |
| **Function or Struct Name** | **Declaration** | **Definition** |
| bcdtobin() | storeNewRecord.h  Line 5 | Firmware.ino  Lines 406 - 417 |
| cleanSD() | handleSerial.h  Line 20 | handleSerial.cpp  Lines 270 - 324 |
| configurationExists() | configuration.h  Line 25 | configuration.cpp  Lines 3 - 25 |
| createHeader() | handleSerial.h  Line 35 | handleSerial.cpp  Lines 1028 - 1065 |
| Date\_t | RTC\_PCF8523.h  Lines 44 - 53 | RTC\_PCF8523.h  Lines 44 - 53 |
| disableUnneededPeripherals() | powerSleep.h  Line 8 | powerSleep.cpp  Lines 58 - 71 |
| ejectSD() | handleSerial.h  Line 23 | handleSerial.cpp  Lines 410 - 426 |
| enterSleep() | powerSleep.h  Line 7 | powerSleep.cpp  Lines 21 - 37 |
| exitSerial() | handleSerial.h  Line 22 | handleSerial.cpp  Lines 381 - 390 |
| getInput() | handleSerial.h  Line 31 | handleSerial.cpp  Lines 878 - 885 |
| getNestedInput() | handleSerial.h  Line 32 | handleSerial.cpp  Lines 904 - 919 |
| handleSerial() | handleSerial.h  Line 18 | handleSerial.cpp  Lines 53 - 143 |
| INT0\_ISR() | Firmware.ino  Lines 230 - 234 | Firmware.ino  Lines 230 - 234 |
| INT1\_ISR() | Firmware.ino  Lines 245 - 248 | Firmware.ino  Lines 245 - 248 |
| incrementFileNumber() | handleSerial.h  Line 37 | handleSerial.cpp  Lines 1088 - 1106 |
| initializeData() | magnetometer.h  Line 78 | magnetometer.cpp  Lines 320 - 331 |
| initSD() | handleSerial.h  Line 25 | handleSerial.cpp  Lines 483 - 499 |
| listFiles() | handleSerial.h  Line 26 | handleSerial.cpp  Lines 501 - 545 |
| loadDateTime() | RTC\_PCF8523.h  Line 57 | RTC\_PCF8523.cpp  Lines 137 - 170 |
| loop() | Firmware.ino  Lines 150 - 220 | Firmware.ino  Lines 150 - 220 |
| mag\_init() | magnetometer.h  Line 72 | magnetometer.cpp  Lines 85 - 167 |
| mag\_transfer() | magnetometer.h  Line 74 | magnetometer.cpp  Lines 231 - 258 |
| nameFile() | handleSerial.h  Line 36 | handleSerial.cpp  Lines 1067 - 1086 |
| peakDetected2() | detectPeaks.h  Line 15 | detectPeaks.cpp  Lines 98 - 126 |
| printConfig() | handleSerial.h  Line 34 | handleSerial.cpp  Lines 973 - 1009 |
| printHelp() | handleSerial.h  Line 24 | handleSerial.cpp  Lines 441 - 460 |
| printWater() | handleSerial.h  Line 33 | handleSerial.cpp  Lines 921 - 971 |
| RTC\_Doctor() | handleSerial.h  Line 30 | handleSerial.cpp  Lines 547 - 697 |
| readConfiguration() | configuration.h  Line 26 | configuration.cpp  Lines 27 - 34 |
| readData() | magnetometer.h  Line 77 | magnetometer.cpp  Lines 284 - 295 |
| read\_mag() | magnetometer.h  Line 73 | magnetometer.cpp  Lines 189 - 194 |
| registerDump() | RTC\_PCF8523.h  Line 56 | RTC\_PCF8523.cpp  Lines 76 - 101 |
| resetState() | state.h  Line 39 | state.cpp  Lines 3 - 17 |
| rtcTransfer() | RTC\_PCF8523.h  Line 55 | RTC\_PCF8523.cpp  Lines 36 - 55 |
| SDPowerDown() | powerSleep.h  Line 14 | powerSleep.cpp  Lines 192 - 197 |
| SDPowerUp() | powerSleep.h  Line 13 | powerSleep.cpp  Lines 173 - 178 |
| SignalState\_t | state.h  Lines 26 - 37 | state.h  Lines 26 - 37 |
| State\_t | state.h  Lines 8 - 22 | state.h  Lines 8 - 22 |
| serialPowerDown() | powerSleep.h  Line 12 | powerSleep.cpp  Lines 154 - 159 |
| serialPowerUp() | powerSleep.h  Line 11 | powerSleep.cpp  Lines 131 - 140 |
| setConfiguration() | handleSerial.h  Line 19 | handleSerial.cpp  Lines 176 - 248 |
| setup() | Firmware.ino  Lines 105 - 148 | Firmware.ino  Lines 105 - 148 |
| startLogging() | handleSerial.h  Line 27 | handleSerial.cpp  Lines 718 - 757 |
| stopLogging() | handleSerial.h  Line 28 | handleSerial.cpp  Lines 778 - 792 |
| storeNewRecord() | storeNewRecord.h  Line 4 | Firmware.ino  Lines 307 - 360 |
| twiPowerDown() | powerSleep.h  Line 10 | powerSleep.cpp  Lines 108 - 113 |
| twiPowerUp() | powerSleep.h  Line 9 | powerSleep.cpp  Lines 86 - 94 |
| updateDateTime() | handleSerial.h  Line 29 | handleSerial.cpp  Lines 808 - 864 |
| viewDateTime() | handleSerial.h  Line 21 | handleSerial.cpp  Lines 340 - 362 |
| writeConfiguration() | configuration.h  Line 27 | configuration.cpp  Lines 36 - 45 |