CIWS-WM-Node

Firmware Documentation

Firmware ver. 1.0.0

Project Organization

The CIWS-WM-Node 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.

Consider the following C code:

```
int main()
{
	int a = 2;
	int b = a * 4;
	printf("The result is %d\n", b);
	int c = b * 4;
	printf("The result is %d\n", c);
	return 0;
}
```

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:

```
int times4(int input)
{
	return input * 4;
}


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:

```
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;
}


int times4(int input)
{
      return input * 4;
}
```

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-WM-Node 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:

```
#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-WM-Node. The aim of this section was to illustrate why the project is organized the way it is.

Main Loop: Computational_Firmware.ino

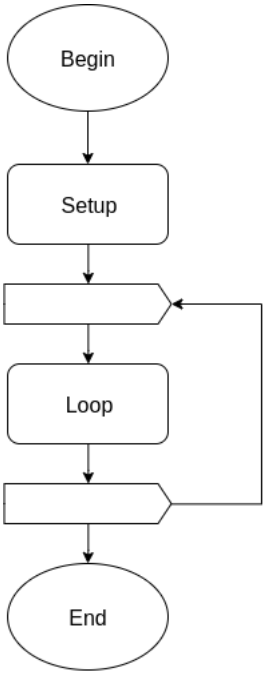
The file `Computational_Firmware.ino` is the program's starting point. All Arduino projects have a `.ino` file. Ours contains four functions:

- `void setup()`
- `void loop()`
- `void INT0_ISR()`
- `void INT1_ISR()`

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:



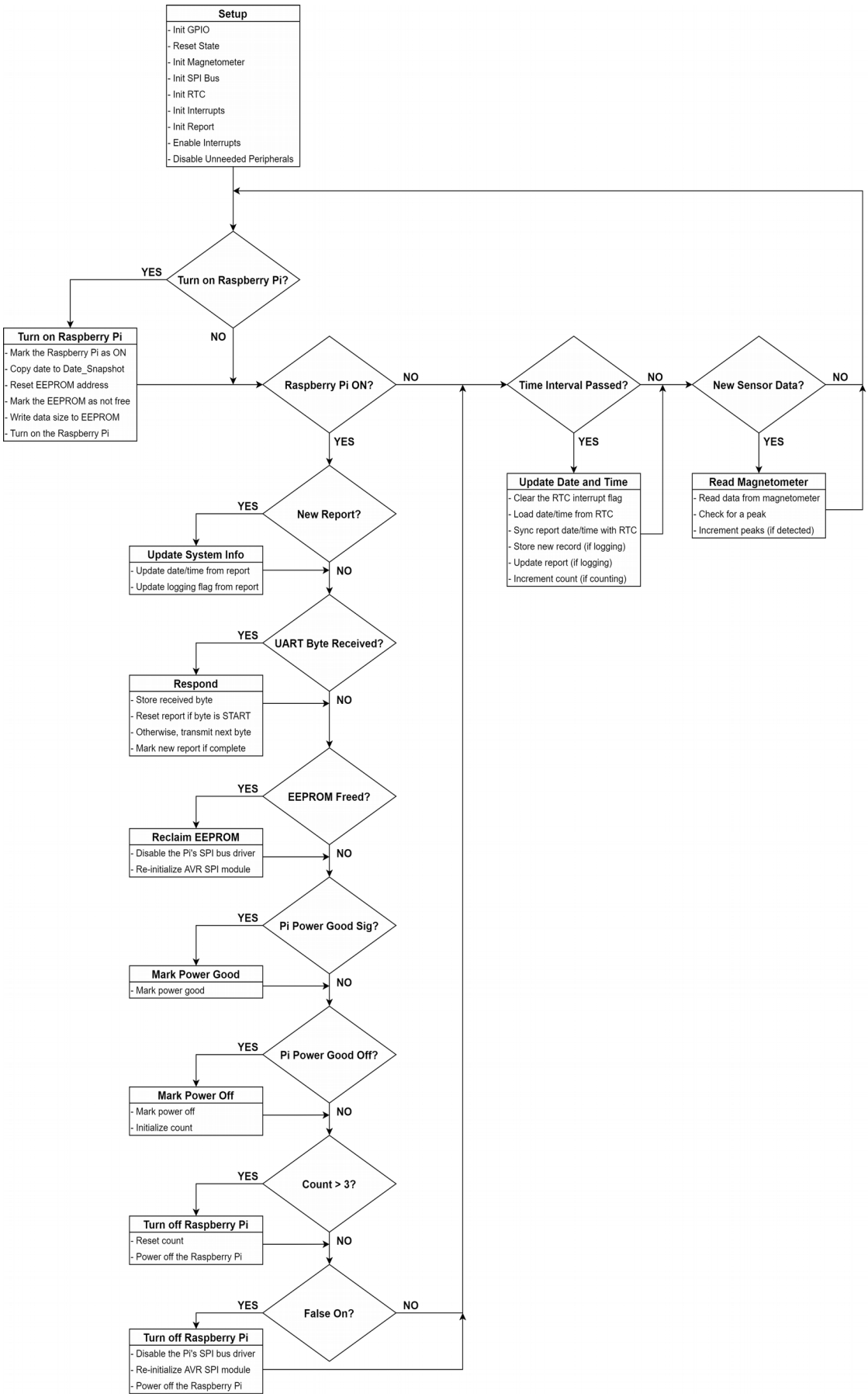
The `setup()` function does the following:

- Initializes GPIO pins
- Initializes the system state data structure
- Initializes the magnetometer
- Initializes the AVR SPI module
- Initializes the real-time clock
- Sets up the magnetometer and real-time clock interrupt handlers
- Initializes Raspberry Pi report data
- Stops the clock for all unused peripherals

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

1. Check if Raspberry Pi activation button is pressed
2. Swap report data with the Raspberry Pi
3. Negotiate the SPI bus with the Raspberry Pi
4. Check if four seconds are up
5. Update timestamp
6. Check if magnetometer data is ready
7. Process incoming data to count peaks

A flowchart describing the firmware is shown on the next page.


Setup
- Init GPIO
- Reset State
- Init Magnetometer
- Init SPI Bus
- Init RTC
- Init Interrupts
- Init Report
- Enable Interrupts
- Disable Unneeded Peripherals
Turn on Raspberry Pi?
YES
NO
Turn on Raspberry Pi
- Mark the Raspberry Pi as ON
- Copy date to Date_Snapshot
- Reset EEPROM address
- Mark the EEPROM as not free
- Write data size to EEPROM
- Turn on the Raspberry Pi
Raspberry Pi ON?
NO
YES
Time Interval Passed?
NO
YES
Update Date and Time
- Clear the RTC interrupt flag
- Load date/time from RTC
- Sync report date/time with RTC
- Store new record (if logging)
- Update report (if logging)
- Increment count (if counting)
New Sensor Data?
NO
YES
Read Magnetometer
- Read data from magnetometer
- Check for a peak
- Increment peaks (if detected)
New Report?
YES
NO
Update System Info
- Update date/time from report
- Update logging flag from report
UART Byte Received?
YES
NO
Respond
- Store received byte
- Reset report if byte is START
- Otherwise, transmit next byte
- Mark new report if complete
EEPROM Freed?
YES
NO
Reclaim EEPROM
- Disable the Pi's SPI bus driver
- Re-initialize AVR SPI module
Pi Power Good Sig?
YES
NO
Mark Power Good
- Mark power good
Pi Power Good Off?
YES
NO
Mark Power Off
- Mark power off
- Initialize count
Count > 3?
YES
NO
Turn off Raspberry Pi
- Reset count
- Power off the Raspberry Pi
False On?
YES
NO
Turn off Raspberry Pi
- Disable the Pi's SPI bus driver
- Re-initialize AVR SPI module
- Power off the Raspberry Pi

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 should be kept as short as possible.

System State: state.h and state.cpp

These files define two C/C++ structs, `State` and `SignalState`.

`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.
- `unsigned long romAddr` – Pointer to the current address in EEPROM.
- `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 readMag` – Boolean flag: True if magnetometer data is ready, false if it is not.
- `bool newReport` – Boolean flag: True if a transaction with the RPi is complete, false if it is not.
- `bool RPiON` – Boolean flag: True if power is supplied to RPi, false if it is not.
- `bool powerGood` – Boolean flag: True if the RPi signals after power-on, false if not.
- `bool romFree` – Boolean flag: True if the RPi signals it is finished with EEPROM, false if not.
- `bool RPiFalseON` – Boolean flag: True if the RPi is unresponsive on power-on, false if it is not.
- `byte interval` – Stores the time interval between data records.

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.

- `float x[2] = {0, 0};` – Input signal from the magnetometer.
- `float a = 0.95;` - DC Removal filter pole.
- `float y[2] = {0, 0};` - Output signal from DC Removal filter.
- `bool triggered = false;` - Boolean flag for software-based schmitt trigger.

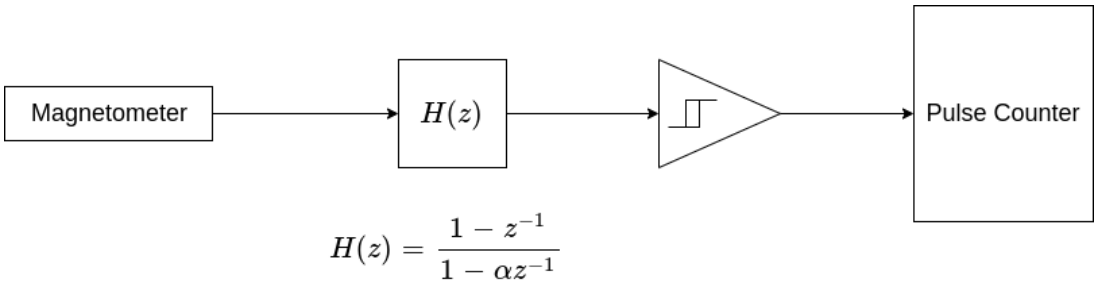
The use of these values is detailed in the next section, **Peak Detection**.

Peak Detection: detectPeaks.h and detectPeaks.cpp

These files define a single function:

```
bool peakDetected(volatile SignalState_t* signalState)
```

The function `peakDetected()` 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.



$$H(z) = \frac{1 - z^{-1}}{1 - \alpha z^{-1}}$$

Source:
https://www.embedded.com/design/configurable-systems/4007653/DSP-Tricks-DC-Removal

The function `peakDetected()` 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
```

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 `peakDetected()`. 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`.

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 I^2C serial bus, and the Arduino IDE's Wire library is used for I^2C 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 loadDateTime(Date_t* Date)`
- `void copyDateTime(Date_t* Date1, Date_t* Date2)`
- `void setClockPeriod(uint8_t period)`

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 I^2C communication with the RTC.

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 function `copyDateTime()` reads data in a `Date_t` structure and stores the data in a second `Date_t` structure. This function is called when the Raspberry Pi is activated. The copied timestamp is the start time for the next batch of data in the EEPROM.

The function `setClockPeriod` adjusts the RTC interrupt clock period, thus adjusting the time interval between data records.

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

Power Reduction: powerSleep.h and powerSleep.cpp

The WM-Node 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 spiPowerUp();`
- `void spiPowerDown();`

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_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:

- I^2C bus: `twiPowerUp()`
- Serial Port: `serialPowerUp()`
- SPI bus: spi`PowerUp()`

These functions call their respective `power_xxx_enable()` function from the `<avr/power.h>` library.

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

- I^2C bus: `twiPowerDown()`
- Serial Port: `serialPowerDown()`
- SPI bus: spi`PowerDown()`

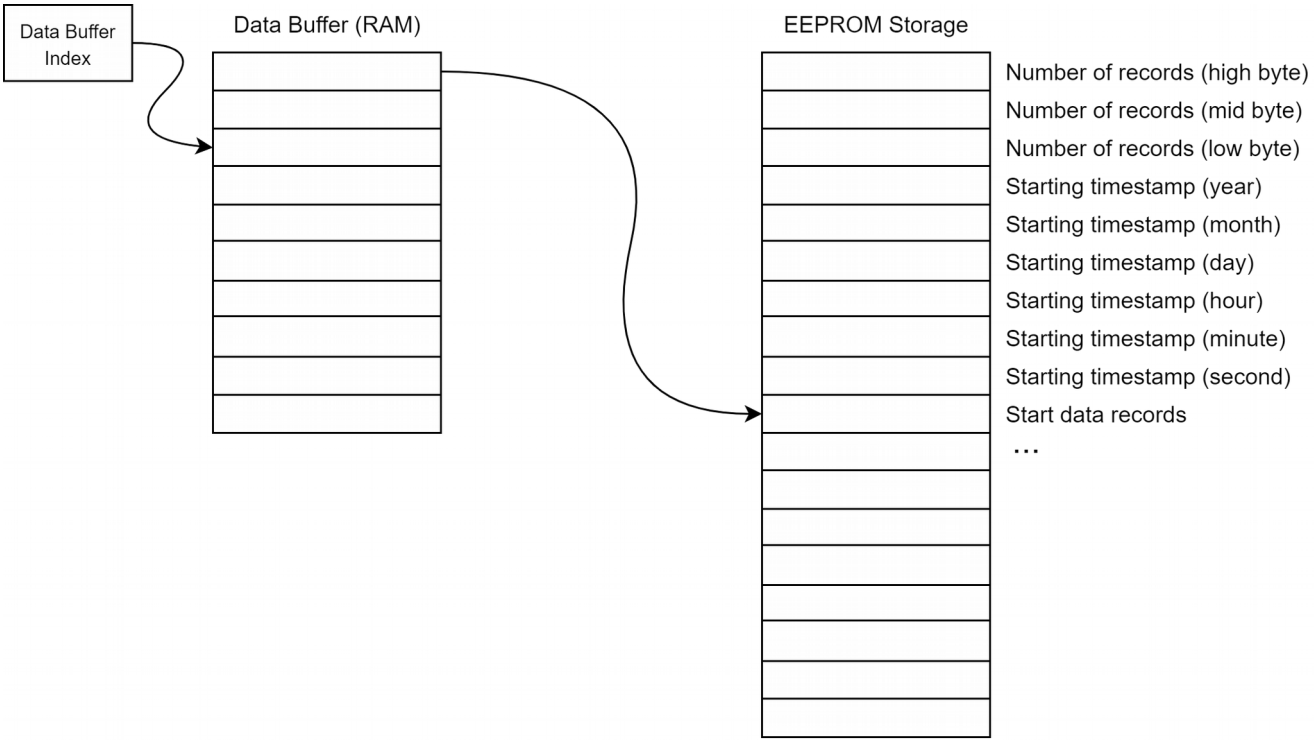
All of the `PowerDown()` functions simply call their respective `power_xxx_disable()` function from the `<avr/power.h>` library (as seen in `disableUnneededPeripherals()`).

Storing Data: storeNewRecord.h and storeNewRecord.cpp

These two files define three functions:

- `writeDataSize(State_t* State)`
- `writeDateAndTime(Date_t* Date)`
- `storeNewRecord(State_t* State, Date_t* Date, Date_t* Date_Snapshot)`

These functions are responsible for storing data in the 25LC1024 EEPROM chip. The following diagram illustrates the record storage architecture:



A data record may be stored in one of two storage areas:

1. Data buffer (128 byte-sized records)
2. EEPROM chip

The default storage for the data record is the EEPROM chip. As long as the EEPROM chip is not being used by the Raspberry Pi, all data records bypass the data buffer and are stored directly in the EEPROM.

If the EEPROM chip is tied up by the Raspberry Pi, then the data record is stored in the data buffer. This data buffer is a circular buffer implemented as an array in the microcontroller RAM. As soon as the Raspberry Pi releases the EEPROM back to the microcontroller, the data in the data buffer is copied to the beginning of the data records section of the EEPROM chip. Once this process is complete, the data buffer index resets to zero.

The data format is also shown in the above diagram. The number of records is stored in the first three bytes of the EEPROM (addresses 0, 1, and 2). The byte at address zero is the most significant byte, or high byte. The byte at address one is the middle byte, and the byte at address two is the least significant byte. This number corresponds to the number of data records after the timestamp data in the EEPROM.

The next six bytes of the EEPROM hold the starting timestamp. This timestamp is the timestamp associated with the first data record in the EEPROM, and is the only timestamp stored in the EEPROM. Because the data records are stored based on timing from the Real-Time Clock, the timestamps for each record can be determined from the original timestamp, and is done so by the Raspberry Pi. The timestamp is comprised of six bytes, or one byte per field. The fields are year, month, day, hour, minute, and second.

The 25LC1024 EEPROM by Microchip is read and written to using an SPI bus, which is a common peripheral on many microcontrollers, including the ATmega328p, the microcontroller on the Arduino Pro board used. A brief description of the transaction with the EEPROM chip is given here. Further information can be found in the 25LC1024 datasheet (Microchip document DS22064D).

Every time the microcontroller writes to the EEPROM, it must enact two SPI transactions:

1. Send a write enable instruction
2. Send the data to be written

Sending the write enable instruction consists of sending the single byte 0x06. Writing this byte enables the EEPROM to accept a write instruction. This byte must be sent as a separate SPI transaction before any data writing can be attempted. There is no response from the EEPROM when the write enable instruction byte is sent.

After the write enable instruction byte has been sent to the EEPROM, the EEPROM is ready to accept data to be written. A data write transaction consists of the following bytes:

Byte	Field
0	Write Instruction
1	Address (MSB)
2	Address
3	Address (LSB)
4	Data byte 0
...	...
N	Data byte X

The maximum number of data bytes which can be written in a single transaction is 256, as long as all 256 bytes reside in the same page in the EEPROM chip. The firmware checks for page-crossings while writing and handles them accordingly; however, a page crossing will never occur in this project because of the following two facts:

1. Excepting the case given in 2., data is always written to the EEPROM in one-byte chunks
2. When the Raspberry Pi releases the EEPROM, a maximum of 128 bytes is written starting at address 0x009, making it impossible for the only multi-byte write to cross a page boundary.

Because the page boundary case is handled by the firmware, it is possible to increase the size of the data buffer without having to worry about page boundaries. It is also important to note that EEPROM writes take some time, so it is good practice to wait at least 5-6 milliseconds before beginning a write transaction.

The function `writeDataSize()` is used to write the current number of records to the EEPROM chip. This function is called when the user presses the activation button, so that the number of records on the EEPROM chip is written where the Raspberry Pi can find it. This functionality is there so that the Raspberry Pi, when reading the EEPROM, can know exactly how many bytes to read from the EEPROM chip.

The function `writeDateAndTime()` is used to write a timestamp to the EEPROM chip. This function is called when when the first data record is written to the EEPROM chip. As shown in the first diagram, the timestamp is written to the address range 0x003-0x008 of the EEPROM chip. This function is called by `storeNewRecord()`.

The function `storeNewRecord()` is used to store a data record to the EEPROM chip. Unlike the CIWS-MWM-Logger project, an individual record consists of only one byte, which is the number of pulses detected within the time interval of the record. The Raspberry Pi will fill in the record number and timestamp when it reads the data from the EEPROM. In addition to storing the current record to the EEPROM chip, this function also checks for records in the data buffer waiting to be written to the EEPROM. If there are records, they are written to the EEPROM first, followed by the current record. If the EEPROM is being used by the Raspberry Pi, then the current record is instead written to the data buffer.

The function `storeNewRecord()` performs one more important task. If the Raspberry Pi has not freed up the EEPROM when this function is called and the ROM data buffer is full, the program sets a flag in the system state structure, `RPiFalseON`, which indicates to the program that the Raspberry Pi has, for some reason, failed to release the EEPROM chip. The Raspberry Pi will then be shut down and data in the buffer will overwrite what is currently in the EEPROM. Unfortunately, this solution creates a potential for data loss if the Raspberry Pi freezes before releasing the EEPROM.

Communications: comm.h and comm.cpp

These two files define several functions useful for communicating with the EEPROM chip and the Raspberry Pi. These functions are:

- `updateReport(unsigned char* report, Date_t* Date, State_t* State)`
- `powerRPiON(void)`
- `powerRPiOFF(void)`
- `UART_Init(unsigned int ubrr)`
- `UART_Transmit(unsigned char data)`
- `UART_Receive(void)`
- `UART_End(void)`
- `spiInit(void)`
- `spiOff(void)`
- `spiSelectSlave(void)`
- `spiReleaseSlave(void)`
- `spiTranceive(unsigned char* data, unsigned char dataSize);`

The functions above provide an interface to SPI bus transactions, UART transactions, powering the Raspberry Pi on and off, and updating system information based on data in a “report”.

The function `updateReport()` is used to update system information and configuration based on data in a report from the Raspberry Pi. These reports are passed between the Raspberry Pi and the microcontroller one byte at a time. These reports are structured in the following way:

Byte	Field	R/W
0	Year	R/W
1	Month	R/W
2	Day	R/W
3	Hour	R/W
4	Minute	R/W
5	Second	R/W
6	Pulses in previous sample	R
7	Total pulses (MSB)	R
8	Total pulses	R
9	Total pulses (LSB)	R
10	Logging flag	R/W
11	Time interval	R/W

Each of these reports is preceded by a START byte, 0xEE. The START byte will restart any report transaction.

Whenever the Raspberry Pi sends a byte, the microcontroller responds with the byte it has stored corresponding to the same field in the report. If the byte sent by the Raspberry Pi is 0xFF, then the microcontroller does not overwrite the data in the corresponding field in the report; however, if the value is anything besides 0xFF (or 0xEE), the data will be overwritten if the field is writable (R/W in the table above). In other words, 0xFF signifies a read request, 0xEE signifies a start, and anything else is a write request.

The function `powerRPiON()` is used to power on the Raspberry Pi by setting pin PC2 (Arduino analog pin 2) high. This action triggers the power switching circuit which connects the battery to a 5 volt regulator, which powers the Raspberry Pi. This function also initializes the UART module by calling `UART_Init()`, deactivates the SPI module by calling `spiOff()`, and sets pin PB0 (Arduino digital pin 9) low, which enables the bus driver between the SPI bus and the Raspberry Pi, thus allowing the Raspberry Pi to read the EEPROM chip.

The function `powerRPiOFF()` is used to power off the Raspberry Pi by setting pin PC2 low. This action turns off the power switching circuit, essentially disconnecting the Raspberry Pi's regulator from the battery. The function `UART_End()` is called in order to disable the UART because it is not in use while the Raspberry Pi is off. The bus driver is disabled before this function is called, as is the SPI module re-initialized. Because of this, these functions are not present in `powerRPiOFF()`.

The function `UART_Init()` initializes the microcontroller's UART module at a baud rate of 9600 bps (bits per second). This baud rate was selected because it is a very common baud rate. The UART data frame format consists of a start bit, eight data bits, and a stop bit with no parity. Note that before anything else, this function has to call `serialPowerUp()`. The UART is only used to communicate with the Raspberry Pi.

The function `UART_Transmit()` takes an input byte and writes it to the UART data register, `UDR0`. The microcontroller automatically takes the data in `UDR0` and transmits it on the UART Tx pin.

The function `UART_Receive()` reads the UART data register. Reading the data register loads the byte received on the UART Rx pin.

The function `UART_End()` disables the UART module and calls `serialPowerDown()`.

The function `spiInit()` initializes the microcontroller's SPI module, which is used for writing data to the EEPROM chip. First, `spiInit()` calls `spiPowerUp()`, which activates the SPI module. The SPI pins are configured, and the SPI module is enabled in master mode with a 2 MHz clock. SPI interrupt flags are cleared, and the function `spiPowerDown()` is called, which deactivates the SPI module to save power. The module is reactivated whenever a transaction is about to take place.

The function `spiSelectSlave()` sets the SPI chip select pin low, signaling to the EEPROM that a SPI transaction is about to take place. This function also calls `spiPowerUp()`, which activates the SPI module.

The function `spiReleaseSlave()` sets the SPI chip select pin high, signaling to the EEPROM that the transaction has been completed. This function also calls `spiPowerDown()`, which deactivates the SPI module.

The function `spiTranceive()` takes as input an array of bytes to transmit. This function iterates over the array and writes each byte to the SPI data register (`SPDR`). The function waits while each byte is transmitted, then reads the `SPDR`. The data in the `SPDR` is data received from the SPI slave device, if it happens to be transmitting data, and overwrites the original data to be transmitted. In this way, the `spiTranceive()` function may be used for both transmitting and receiving data from an SPI device; however, the receiving functionality is not used in this project.

The function `decimalToBCD()` converts a regular variable into the binary-coded-decimal format used by the PCF8523 RTC.

If any item regarding serial communication protocols is unclear, the user is encouraged to research the Serial Peripheral Interface (SPI) and Universal Asynchronous Receive-Transmit (UART) protocols in further depth.

Table of Functions and Structs

Function or Struct Name	Declaration	Definition
copyDateTime()	RTC_PCF8523.h Line 57	RTC_PCF8523.cpp Lines 126 - 134
Date_t	RTC_PCF8523.h Lines 44 - 53	RTC_PCF8523.h Lines 44 - 53
decimalToBCD()	comm.cpp Line 3	comm.cpp Lines 222 - 229
disableUnneededPeripherals()	powerSleep.h Line 8	powerSleep.cpp Lines 54 - 67
INT0_ISR()	Computational_Firmware.ino Lines 296 - 299	Computational_Firmware.ino Lines 310 - 313
INT1_ISR()	Computational_Firmware.ino Lines 295 - 298	Computational_Firmware.ino Lines 295 - 298
initializeData()	magnetometer.h Line 78	magnetometer.cpp Lines 294 - 305
loadDateTime()	RTC_PCF8523.h Line 56	RTC_PCF8523.cpp Lines 91 - 124
loop()	Computational_Firmware.ino Lines 148 - 286	Computational_Firmware.ino Lines 148 - 286
mag_init()	magnetometer.h Line 72	magnetometer.cpp Lines 69 - 141
mag_transfer()	magnetometer.h Line 74	magnetometer.cpp Lines 205 - 232
peakDetected()	detectPeaks.h Line 14	detectPeaks.cpp Lines 36 - 64
powerRPiOFF()	comm.h Line 20	comm.cpp Lines 126 - 132
powerRPiON()	comm.h Line 19	comm.cpp Lines 116 - 124
readData()	magnetometer.h Line 77	magnetometer.cpp Lines 258 - 269
read_mag()	magnetometer.h Line 73	magnetometer.cpp Lines 163 - 168
resetState()	state.h Line 40	state.cpp Lines 3 - 20
rtcTransfer()	RTC_PCF8523.h Line 55	RTC_PCF8523.cpp Lines 36 - 55
SignalState_t	state.h Lines 30 - 38	state.h Lines 30 - 38
State_t	state.h Lines 10 - 26	state.h Lines 10 - 26
serialPowerDown()	powerSleep.h Line 12	powerSleep.cpp Lines 147 - 152
serialPowerUp()	powerSleep.h Line 11	powerSleep.cpp Lines 127 - 133
setClockPeriod()	RTC_PCF8523.h Line 58	RTC_PCF8523.cpp Line 153 - 165
spiInit()	comm.h Line 27	comm.cpp Lines 171 - 183

spiOff()	comm.h Line 28	comm.cpp Lines 185 - 192
spiPowerDown()	powerSleep.h Line 14	powerSleep.cpp Lines 186 - 191
spiPowerUp()	powerSleep.h Line 13	powerSleep.cpp Lines 166 - 172
spiReleaseSlave()	comm.h Line 30	comm.cpp Lines 202 - 208
spiSelectSlave()	comm.h Line 29	comm.cpp Lines 194 - 200
spiTranceive()	comm.h Line 31	comm.cpp Lines 210 - 220
storeNewRecord()	storeNewRecord.h Line 12	storeNewRecord.cpp Lines 75 - 216
twiPowerDown()	powerSleep.h Line 10	powerSleep.cpp Lines 104 - 109
twiPowerUp()	powerSleep.h Line 9	powerSleep.cpp Lines 82 - 90
UART_End()	comm.h Line 25	comm.cpp Lines 162 - 169
UART_Init()	comm.h Line 22	comm.cpp Lines 134 - 149
UART_Receive()	comm.h Line 24	comm.cpp Lines 157 - 160
UART_Transmit()	comm.h Line 23	comm.cpp Lines 151 - 155
updateReport()	comm.h Line 17	comm.cpp Lines 5 - 114
usartReceiveComplete()	comm.h Line 15	comm.h Line 15
writeDataSize()	storeNewRecord.h Line 10	storeNewRecord.cpp Lines 9 - 44
writeDateAndTime()	storeNewRecord.h Line 11	storeNewRecord.cpp Lines 46 - 73