# Powerduino Software Overview

Xinao Wang
Adrian Beltran
Ronald Meyer
hammondmay@gmail.com

# Contents

# 1. Flow charts

## 1.1 PC software flow chart

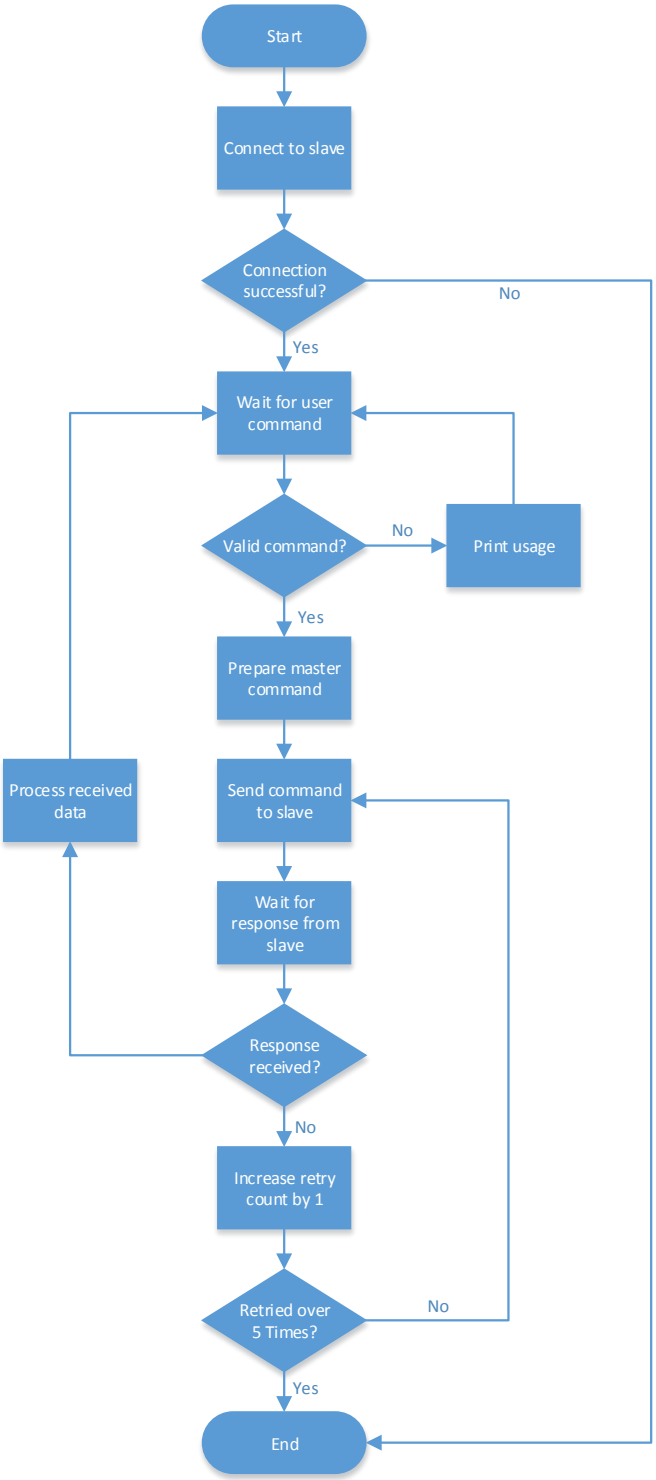



Figure 1. Flow chart for PC program

## 1.2 Microcontroller software flow chart

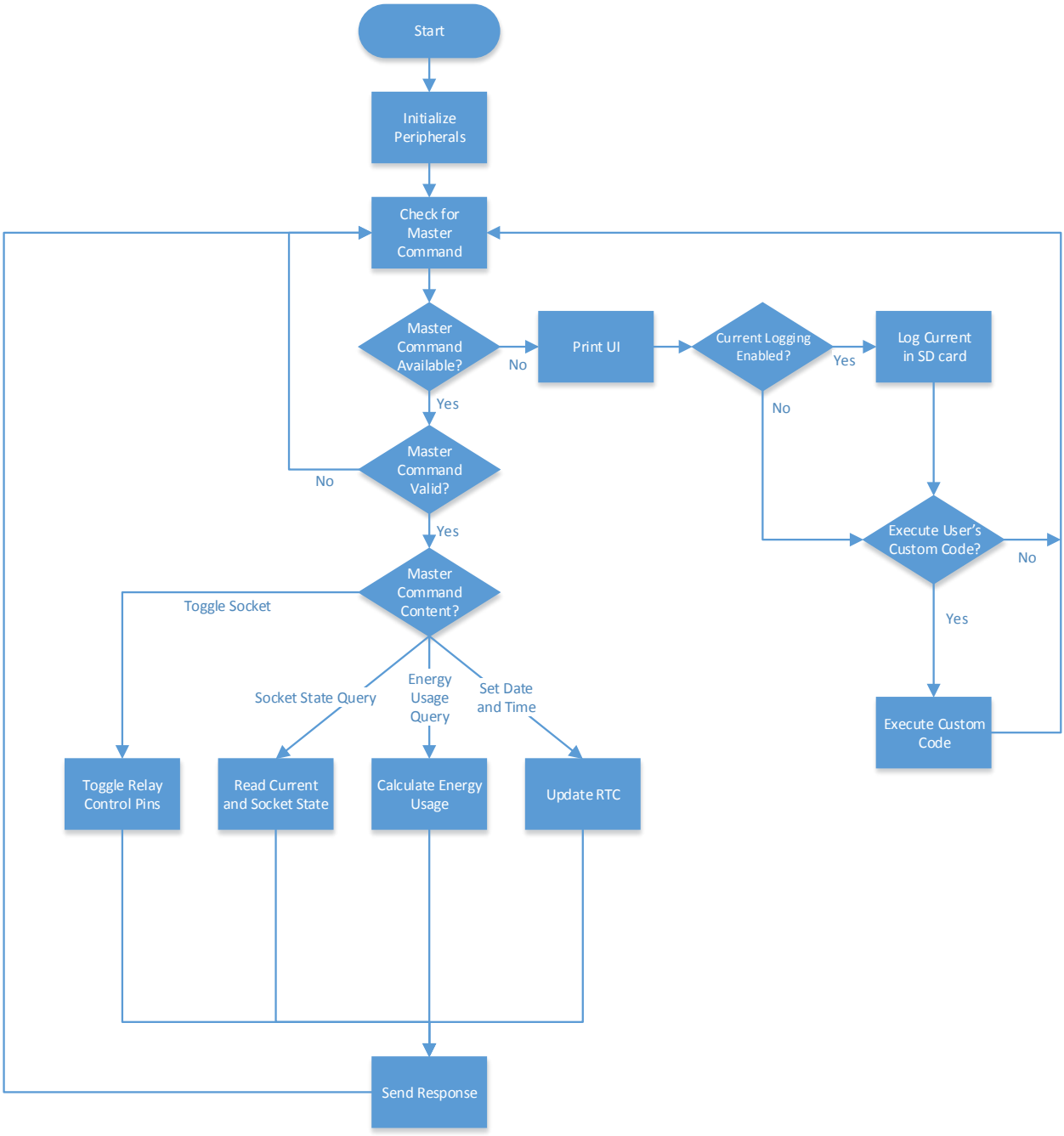



Figure 2. Flow chart for microcontroller program

# 2. Functionality overview

## 2.1 Communication protocol

The communication protocol of this project is relatively straightforward. PC program send commands over WiFi network, and microcontroller receives and responds to PC command via the WiFi module over serial.

To ensure the power strip receives PC's command, a Stop-and-wait protocol was used. Each time PC sends a command, it waits until acknowledgement from power strip is received, otherwise the command will be sent again after timeout.

### 2.1.1 Master commands

Master commands (commands generated from PC) always starts with a 2-byte header. The first byte of the header is always MASTER_COMMAND_TRANSMISSION_START, followed by the number of bytes of data to be sent to slave(power strip).

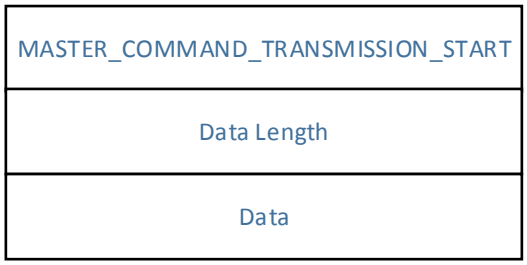



Figure 3. Master command structure

### 2.1.2 Slave commands

Slave commands (Microcontroller's response to master commands) is structurally similar to master commands, with a 2-byte header followed by data. Except the first byte in the header is SLAVE_COMMAND_ACK.

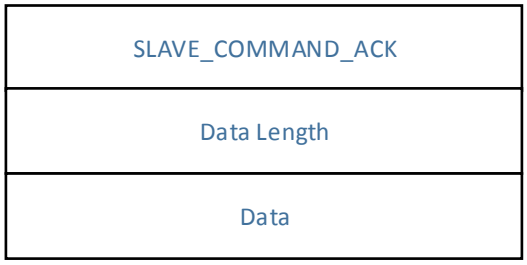
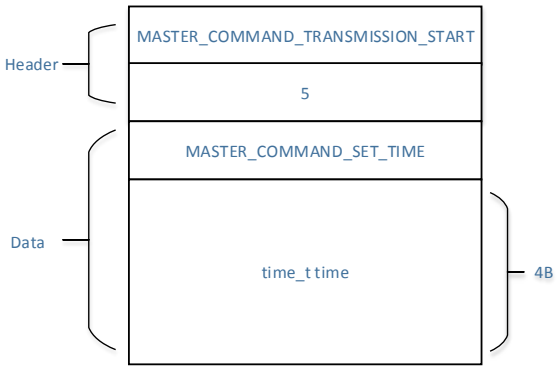



Figure 4. Slave command structure

### 2.1.3 Existing command pairs

Below are existing command pairs in the program. All slots are 1 byte if not otherwise noted. Master command on the left, slave response on the right.

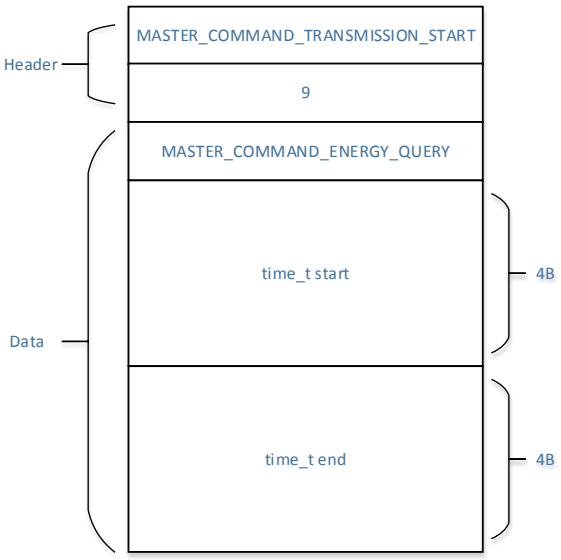
#### 2.1.3.1 Set time



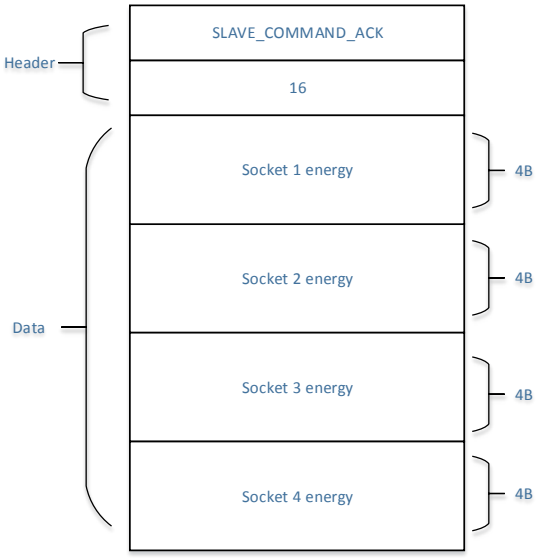

send_cmd_set_time()


send_default_ACK()

#### 2.1.3.2 Request energy usage

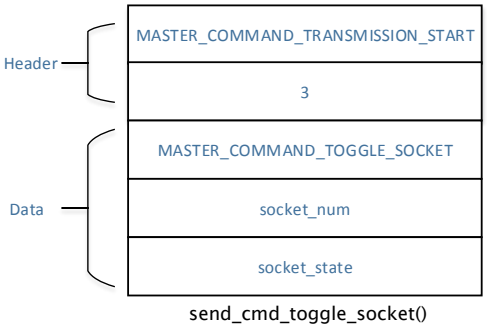



send_cmd_energy_query()

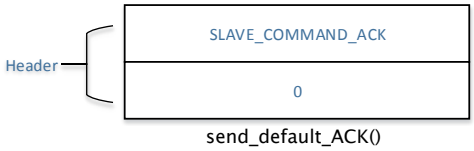



send_energy()

#### 2.1.3.3 Toggle socket

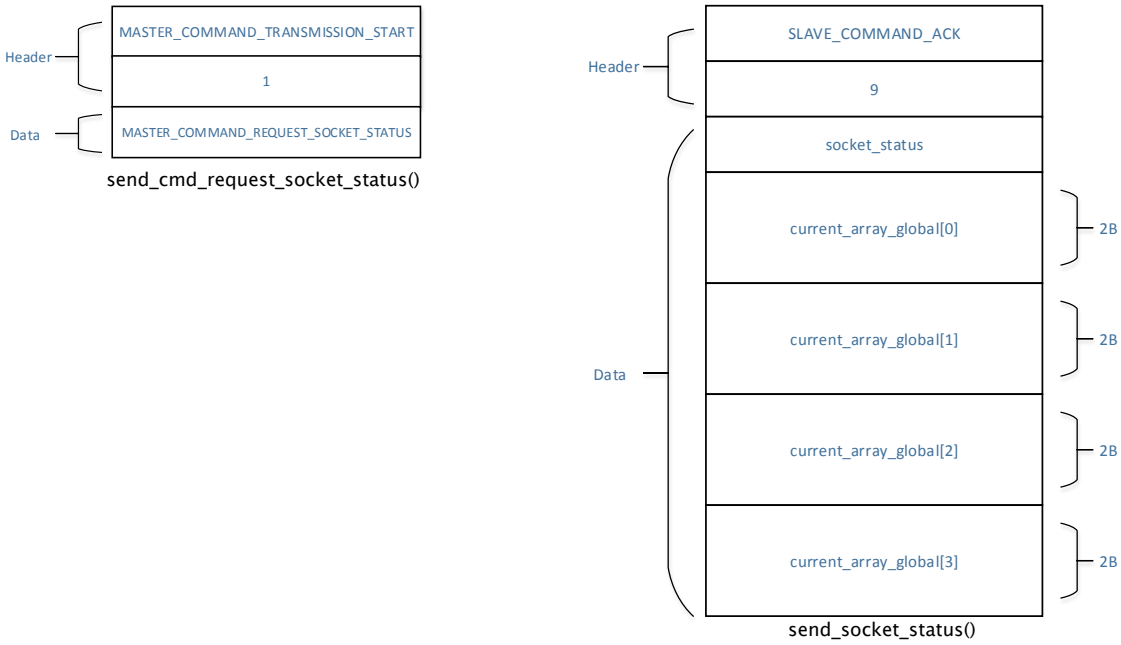



send_cmd_toggle_socket()

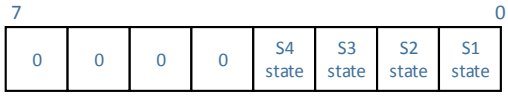



send_default_ACK()

#### 2.1.3.4 Request socket status




where socket_status byte in slave response is constructed as follows:



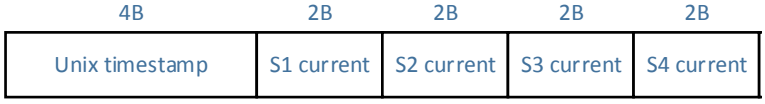

## 2.2 File structure

The microcontroller program uses SD card for data logging and storing settings and states. The file structures are described below.

### 2.2.1 Energy logging file

The program stores the current reading of each socket to SD card every 10 seconds. The program will try to open a file with the name of that particular day, for example, if today is April 30th 2014, it will try to open a log file named 20140430, or create one if it doesn't exist.

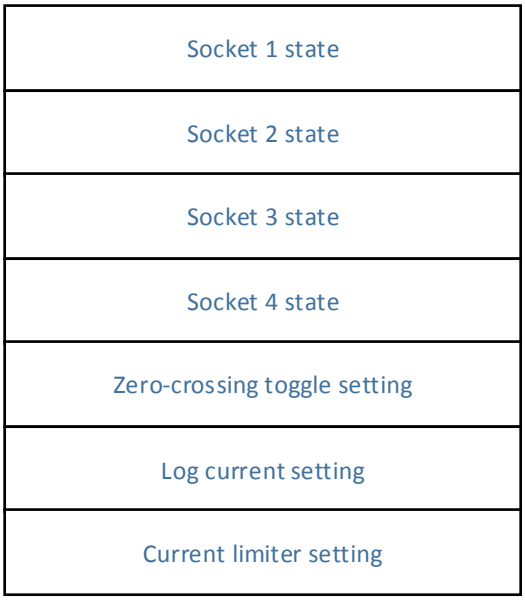
Each entry is then appended to the end of the file in little endian, with the following structure:




See append_current_log() for implementation.

### 2.2.1 Setting and states file

The microcontroller program also writes the state of each socket and settings to the SD card, so it remembers the state upon resetting. The file is called STATE, with the following structure:




Each slot is one byte, starting from the beginning of the file. See save_state() and recover_state() for implementation.

## 2.3 Current measurement

### 2.3.1 RMS current calculation

The output of current sensor is centered around 2.5V. When no current is present the output is 2.5V, output is greater than 2.5V for current in one direction, and smaller than 2.5V in the other direction. The current sensor output (0 - 5V) is divided down to 0 - 3.3V using a resistor divider. Therefore, the current sensor reading is centered around 1.6V after the division.

The resolution of Teensy's on-board ADC is 13 bits, meaning it reads 0V as 0 and 3.3V as 8192.

Each time read_current function is called, it takes 83 samples of current sensor reading in 200 microsecond interval, which is a total of 16600 microseconds, which is the period of a single 60Hz cycle (1/60 = 16.6 ms = 16666 us).

The samples are symmetrical around 1.6V, which is 8192 / 2 = 4096. To calculate RMS value, each sample is subtracted the average of entire sample array to make it center around 0, then the standard RMS method is used, taking the square root of the arithmetic mean of the squares of sample values.

The result RMS is still not current, but just the voltage reading from current sensor, to obtain current reading, a linear formula is applied, calculated from current sensor's datasheet.

I = ((4096 + RMS_value) / 8192) * 50 - 25

The resulting current is a floating number, to save space and make it easy to store on SD card and transport over serial, the float value is multiplied 1000 then stored as a 16-bit unsigned integer. For example if the result is 0.45A, 0.45 * 1000 = 450, so 450 is stored. The largest current can be stored in an int16_t is 65535 / 1000 = 65.535A, well over our current sensor's 20A maximum rating.

This method uses quite a lot of memory and has a 17ms delay every time it's called. Feel free to improve it, see current_reader class for implementation.

A timer interrupt fires every 0.3 seconds to call read_current function, and store the current reading in a global 4-element array, one for each socket. When a current reading is needed, the program will read the value from the array.

### 2.3.1 Moving average smoothing

However, the result of above method tend to fluctuate around the true reading, in order to achieve a smoother readout, each new current reading is taken average with 9 previous readings.