ME 507 Mechanical Control System Design

Fall 2016

Dr. John Ridgely

Term Project: Color Companion

Samuel Adler, Louis Roseguo

1.0 - Introduction

Though the rise of products like the smartwatch may generate the perception of wearable devices as a new innovation, people have always been making what technologies they have wearable. The history of wearable technology extends back more than 100 years, with the first confirmed wristwatch showing up in the late 1800s. Going even further back, early forms of eyeglasses were worn by Italian monks as far back as the 13th century. In the present day, people remain dependant on wearable technology to provide instantaneous, aesthetically pleasing information or feedback. With that in mind, we set out to create a device that can provide instantaneous feedback to the user in response to their movements. The concept involved using a GPS module to read the wearer's speed and course (cardinal direction) over ground, then outputting corresponding values to a strip of color LEDs which would change color based on the direction of motion and scale the intensity based on the user's speed. A digital readout, or any other type of numerical display, is great for logging data or monitoring an athlete's progress over time, but isn't as intuitive as colored lights, and likely isn't as engaging either, especially to more casual users. This device is intended for use by runners or bikers to promote engagement and provide information simultaneously. In addition, the lights double as a safety feature, making sure those around the user are aware of their presence when they might otherwise have difficulty noticing them, such as at night.

2.0 - Specifications

Table 1: Project Specifications

Aspect	Target	Condition
Battery Life	1 Hour	Minimum
Resolution	45 Degrees/Color	Maximum
Refresh Rate	10 Hz	Minimum
Wearability	Components Consolidated	Absolute
Persistence of Operation	Functions Without Satellite Fix	Absolute

3.0 - Design Development

3.1 - Initial Development

Initial planning of the ‘Color Companion’ involved determining the scope of our project. What features of motion we wanted our device to track, and how we wanted them to be displayed in an easily understandable way quickly, were our first priorities. After some deliberation, the group decided that the most useful and easily discernable information that an LED strip could show were the direction and speed of a person. After this, it was just a matter of determining hardware for it.

3.2 - Hardware Design

The hardware has a few main components, a printed circuit board, GPS, IMU, Pyboard, and a neopixel LED strip (with everything else being smaller components like capacitors, resistors and the like). First, a circuit board was designed using the program Eagle, making files to make the printed circuit board from.

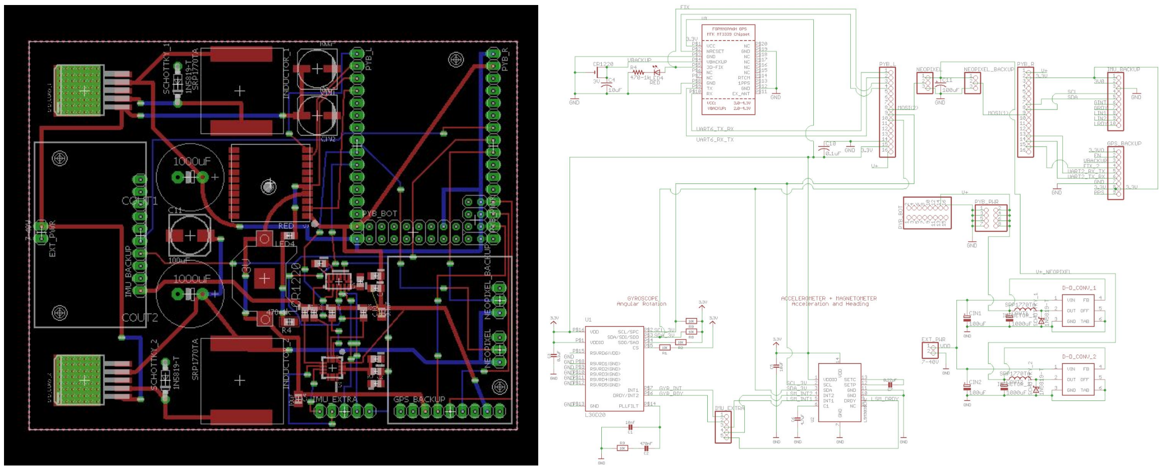

Figure 1: Eagle Board Schematics

The way the printed circuit board (Figure 2) was designed allowed for the breakout boards to be implemented in case there were issues with the GPS or IMU components, though it was necessary for the Pyboard. Components that produce large amounts of heat had extra vias placed near them in order to better dissipate the heat. Holes were placed in different regions of the board so it could be attached to an article of clothing or other item that would be on someone’s body at all times. The placement of the LED headers were left on the edge of the device to prevent any unnecessary overlap of the parts.

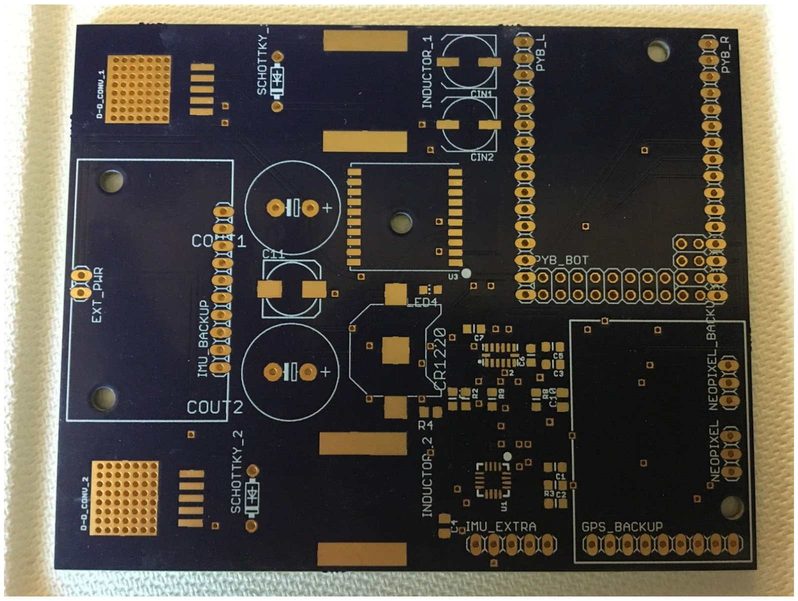
Figure 2: Printed Circuit Board (no components)

The GPS (upper middle square in Figure 3) was picked for being able to tell a change in speed to a resolution of 0.1 m/s, while also providing an accurate reading for the direction the user is facing. The only shortcoming of the model chosen was that the signal was weak to nonexistent indoors, though this could be offset by getting an antenna attachment implemented into the design. The IMU (blue breakout board on the left in Figure 3) was chosen as a backup for when the GPS isn't working indoors and for its acceleration reading capabilities. In addition, incorporating the IMU allows for potential further sensory info to be collected and displayed.

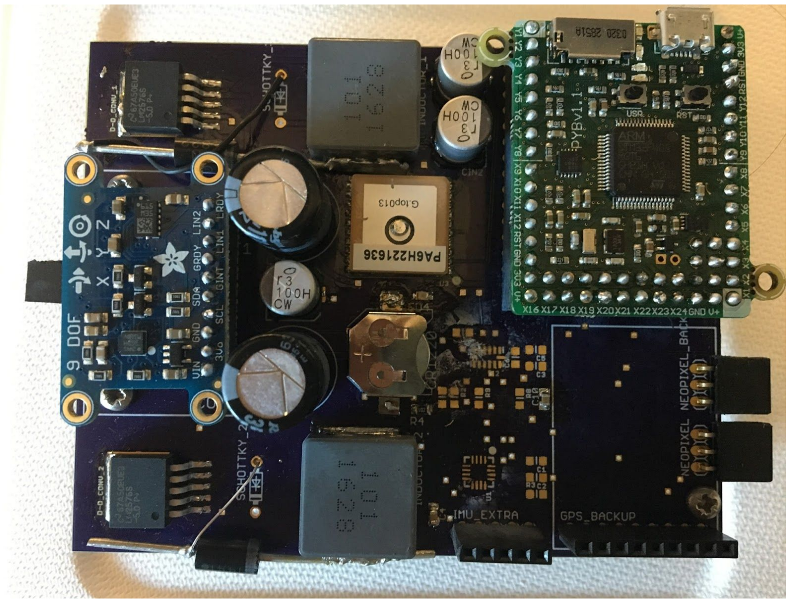
Figure 3: Printed Circuit Board (components attached)

The neopixel LED strip (Figure 4) was chosen for having different programmable addresses for each individual light. That means that any light can have a different RGB value than the one next to it, which allows for more interesting light patterns to be developed. The range of light values and combinations also made this an appealing choice for our device because it allows for more ways to show distinct colors for different directions.

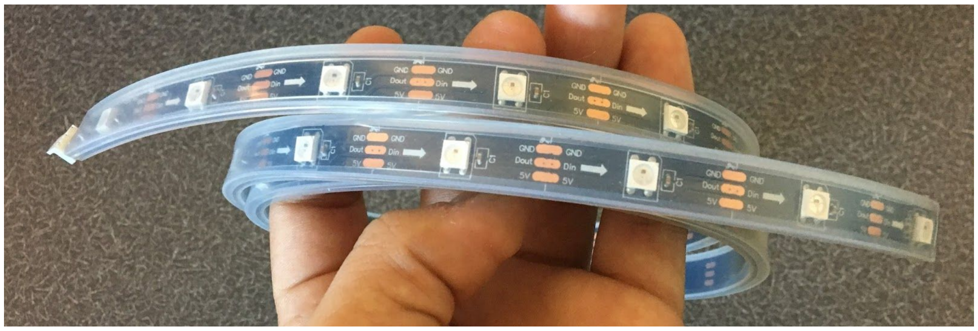

Figure 4: Neopixel LED Strip

The power source (Figure 5) chosen for the project was selected because it’s compact size and power supply were enough to power the circuit board and lights attached. They also ran for a decent amount of time without needing multiple recharges.

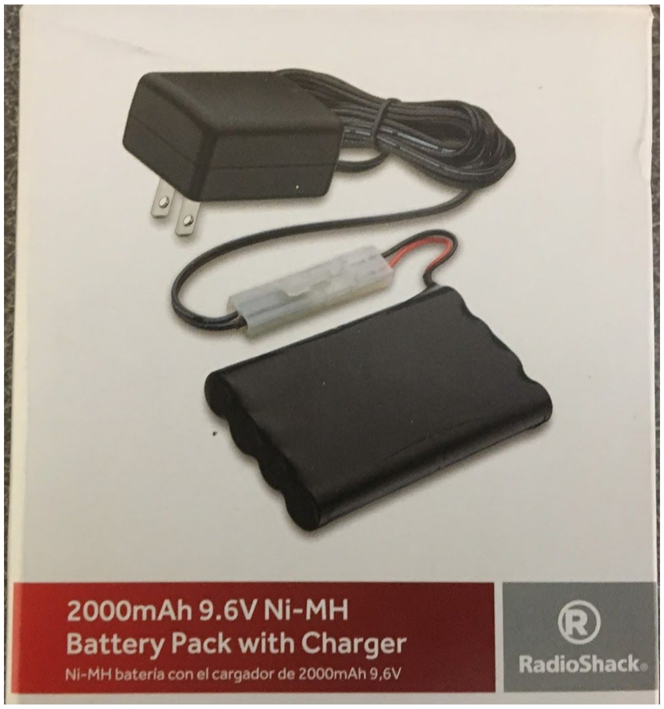
2000mAh 9.6V Ni-MH
Battery Pack with Charger
Ni-MH bateria con el cargador de 2000mAh 9,6V
RadioShack

Figure 5: Battery Power Source

3.3 - Software Design

The Task Diagram (Figure 6) was developed based off of the timing we were expecting to use and the priority we believed the function held. For the changing lights, we felt that the timing had to not be too slow that a person's perception would notice it. With that in mind, a timing of 100 Hz was selected. The GPS timing was selected because the device can only update at a speed of 10 Hz, so the function runs a bit faster at a timing of 20 Hz. The IMU was set for a speed of 500 Hz because it could collect data that fast. The calculating task needs to go fast enough to consistently supply new data to the change lights task, so it was set at a timing twice as fast at 200 Hz.

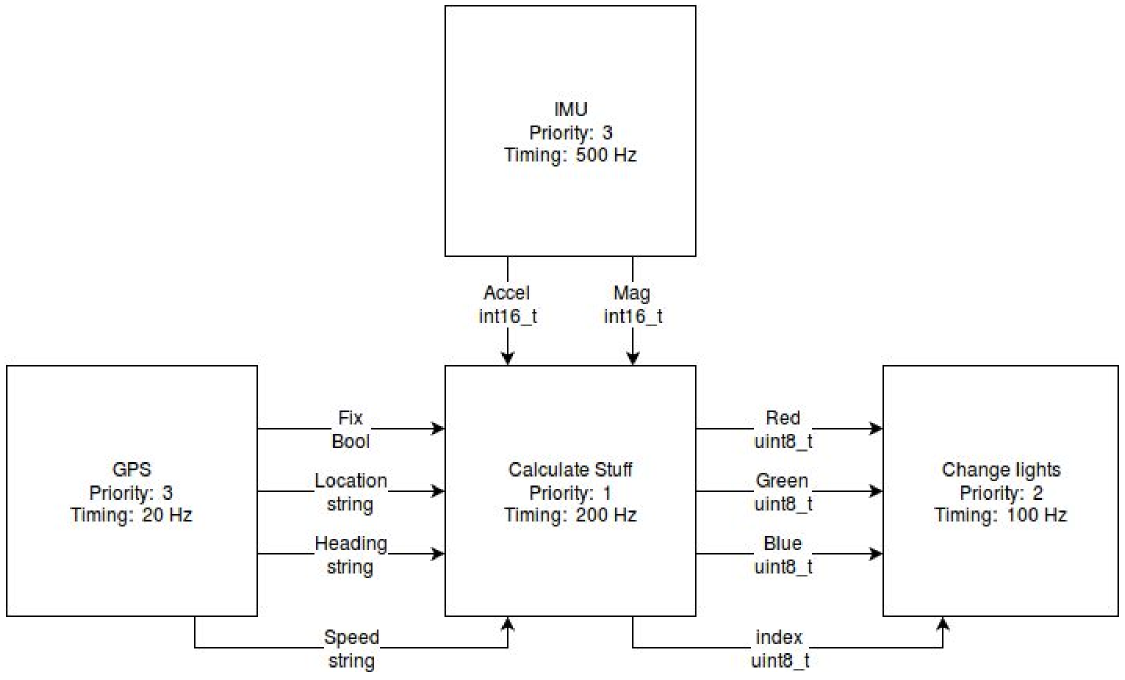

Figure 6: Task Diagram

The most important component of the project is the GPS module (Figure 7), which gathers the speed and course data. The microcontroller communicates with the GPS over serial, and receives a regular stream of lines in the form of NMEA "sentences." These sentences come in a standard format, which can be read by a sentence parser [3] after cleaning it up a bit. The Pyboard only runs the GPS code if it has a satellite fix. If it is reading for the first time after running the IMU code, it expects the GPS buffer to be full of outdated data, and reads it all in order to clear the buffer and allow the GPS to begin reading again before reading lines until it acquires an RMC (Recommended Minimum Navigation Information) sentence and passes it on. If it has just read an RMC sentence and still has a satellite fix, then it skips the buffer dump to save time.

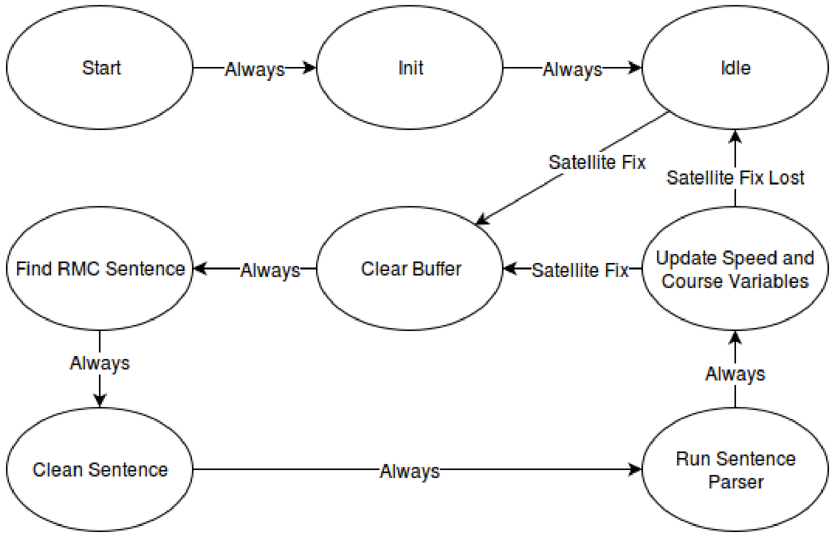

Figure 7: GPS State Diagram

The IMU code was made to go off whenever the GPS isn't able to get a proper signal fix, and the original concept relied on its accelerometer and magnetometer (Figure 8). Code for the magnetometer was written, but it ultimately wasn't implemented. The magnetometer code would configure a register to single-conversion mode, which would cause the magnetometer to update its readings. Then the pyboard would read and return the data. The accelerometer data updated continuously, so the microcontroller only needed to read and return the data after concatenating the high and low bytes.

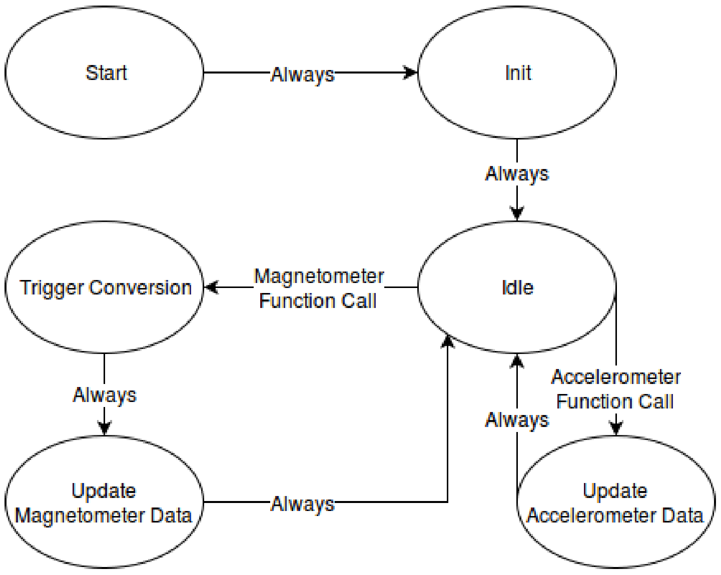

Figure 8: IMU State Diagram

The code was set up to calculate the direction based off of a 360° circle, with different directions taking up different sections of 45°, making for the full circle. This data would be pulled from the GPS function and applied to the calculating task (Figure 9). The velocity data would also be taken from the GPS, and calculated to determine how much to scale the lights by dividing the speed given (in km/h) by .36. This would allow for maximum intensity to be reached when someone reaches 45 km/h, the top speed reached by a person on foot, Usain Bolt.

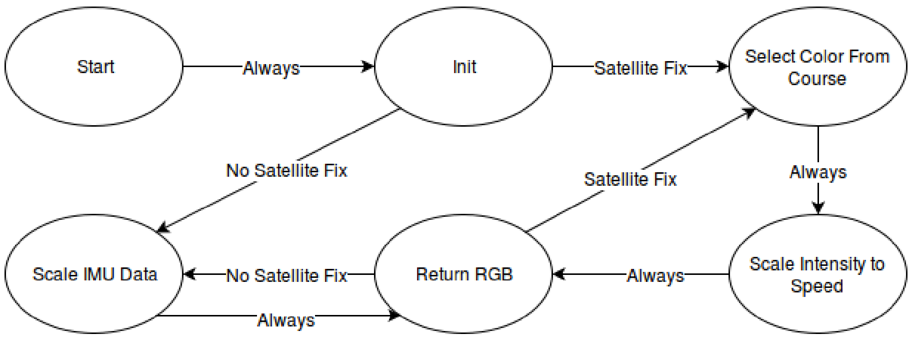

Figure 9: Calculate State Diagram

The lights.py file was made to handle the information gathered by the GPS and output data that can be used by the LED strip to show that output data. Depending on the direction and velocity of the user, the lights file will give an appropriate directional color. It was also intended to include a way to account for errors in the code, as seen in the state diagram (Figure 10).

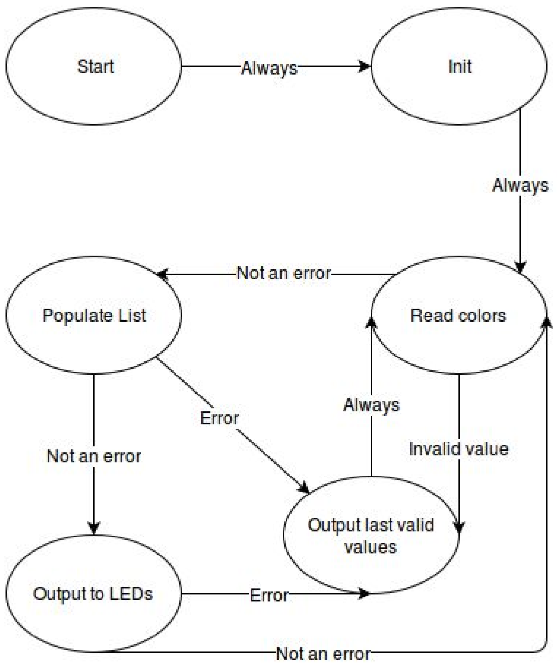

Figure 10: Lights State Diagram

4.0 - Results

The system worked as programmed/built for the most part. There were many difficulties in getting the GPS to connect to the satellite, largely due to poor weather conditions as well as most of the programming taking place indoors. Tests involving the code gathering data and changing the color and intensity of the LED strip appropriately were largely successful. It was entertaining to see the light change based off of movement. This movement could be validated by being compared to our phone's compass capabilities. Unfortunately, the lights would occasionally get locked up after switching out of the IMU function, and would get stuck on a color of light and not change intensity. The effect of the light intensity was harder to gauge accuracy on since there wasn't an easy way to determine our speed and the luminosity of the lights at any given time, but it was clear that the lights were getting brighter and dimmer appropriately for how much faster or slower someone was moving. An unexpected result of the project was the effectiveness of the lights outdoor. During midday especially, the sun would make it difficult to see the color or brightness of the lights, except at higher intensities. Further development of this project would ideally take care of this issue, though it would likely be a hardware fix instead of a software one. The lights still functioned properly at night and in less bright areas, where LED color and small changes in intensity were easier to notice.

The final code can be found at the code repository for this project on GitHub:
https://github.com/Kaiapuni/ME-507-Project

5.0 - Appendices

5.1 - References

[1] Adler, Samuel and Roseguo, Louis. *"ME-507-Project,"* https://github.com/Kaiapuni/ME-507-Project/tree/master.

[2] Bednarik, Jan. *"Micropython-ws2812,"* https://github.com/JanBednarik/micropython-ws2812.

[3] inmcm. *"micropyGPS."* https://github.com/inmcm/micropyGPS.

Namespace Documentation

boot Namespace Reference

Detailed Description

Author:

Louis

boot runs at startup and imports the modules used by the project.

docstring Namespace Reference

Created on Tue Dec 6 21:50:20 2016.

Detailed Description

This is the main code run by the pyboard.
This module takes GPS data and turns it into useful values to output to the Neopixels.

Author:

: Louis
This module pulls data from the GPS module and makes it convenient to read.

Author:

: Louis
This module converts data from the IMU into values ranging from 0-255 for the Neopixels.

Author:

: Louis
This module facilitates communicating with the Adafruit 9DOF LSM303 and L3DG20 IMU.

Author:

: Louis
This module contains functions that use all the other modules to write to the Neopixels.

gpsreader Namespace Reference

Functions

def **checksum** (packet)

This function implements the same XOR checksum as the gps.

def **readsum** (packet)

This function should read the checksum on incoming packets.

def **parse** (packet)

This function returns a list containing the separated components of a packet.

def **RMC** (gps)

This function searches for and returns a GPRMC sentence from the gps.

def **fastRMC** (gps)

Identical to RMC, but without the buffer dump.

def **nmeaupdate** (parser, sentence)

This function updates the NMEA sentence parser.

def **quickupdate** (gps, parser)

This function runs RMC and nmeaupdate together.

def **quickerupdate** (gps, parser)

This function runs RMC and nmeaupdate together.

Function Documentation

def gpsreader.checksum (***packet*****)**

This function implements the same XOR checksum as the gps.

def gpsreader.fastRMC (***gps*****)**

Identical to RMC, but without the buffer dump.

def gpsreader.nmeaupdate (***parser*****,** ***sentence*****)**

This function updates the NMEA sentence parser.

def gpsreader.parse (***packet*****)**

This function returns a list containing the separated components of a packet.

def gpsreader.quickerupdate (***gps*****,** ***parser*****)**

This function runs RMC and nmeaupdate together.

def gpsreader.quickupdate (***gps*****,** ***parser*****)**

This function runs RMC and nmeaupdate together.

def gpsreader.readsum (***packet*****)**

This function should read the checksum on incoming packets.

def gpsreader.RMC (***gps*****)**

This function searches for and returns a GPRMC sentence from the gps.

imumath Namespace Reference

Functions

def **amagnitude** (components)
This function calculates the magnitude of the acceleration.
def **aleds** (components)
This function calculates backup LED values based on accelerometer data.

Function Documentation

def imumath.aleds (***components*****)**

This function calculates backup LED values based on accelerometer data.

def imumath.amagnitude (***components*****)**

This function calculates the magnitude of the acceleration.

imureader Namespace Reference

Functions

def **accelenable** (imu, rate)

This function sets the accelerometer power mode and data rate.

def **printcontents** (imu, device, location)

This function prints a byte of data located within an I2C device.

def **accelaxes** (imu, x, y, z)

This function enables or disables the axes of the accelerometer.

def **accelp** (imu, mode)

This function determines whether the accelerometer operates in low-power mode.

def **accelsetup** (imu, rate, mode, x, y, z)

This function sets up the IMU CTRL_REG1 in a single step.

def **accelquickstart** (imu)

This function sets up the IMU with the preset configuration I expect to want to use.

def **ax** (imu)

This function reads the x axis of the accelerometer.

def **ay** (imu)

This function reads the y axis of the accelerometer.

def **az** (imu)

This function reads the z axis of the accelerometer.

def **acceleration** (imu)

This function returns a list containing each component of acceleration.

def **ax100** (imu)

This function exists for testing, and prints x acceleration 100 times.

def **ares** (imu, hires)

This function controls high resolution mode.

def **maginit** (imu)

This function enables readings from the magnetometer.

def **magdrdy** (imu)

This function checks the bit indicating whether a new set of measurements is available on the magnetometer.

def **magsingle** (imu)

This function configures the magnetometer for a single conversion.

def **magx** (imu)

This function reads the x-axis of the magnetometer.

def **magy** (imu)

This function reads the y-axis of the magnetometer.

def **magz** (imu)

This function reads the z-axis of the magnetometer.

def **magx100** (imu)

This function prints the x-axis of the magnetometer 100 times for testing purposes.
def **heading** (imu)
This function returns a list of the components of magnetometer reading.

Function Documentation

def imureader.accelaxes (*imu*, *x*, *y*, *z*)

This function enables or disables the axes of the accelerometer.

def imureader.accelenable (*imu*, *rate*)

This function sets the accelerometer power mode and data rate.

def imureader.acceleration (*imu*)

This function returns a list containing each component of acceleration.

def imureader.accelp (*imu*, *mode*)

This function determines whether the accelerometer operates in low-power mode.

def imureader.accelquickstart (*imu*)

This function sets up the IMU with the preset configuration I expect to want to use.

```
Currently: Enable, 400 Hz, Low-Power, All axes enabled.
```

def imureader.accelsetup (***imu*****,** ***rate*****,** ***mode*****,** ***x*****,** ***y*****,** ***z*****)**

This function sets up the IMU CTRL_REG1 in a single step.

def imureader.ares (***imu*****,** ***hires*****)**

This function controls high resolution mode.

def imureader.ax (***imu*****)**

This function reads the x axis of the accelerometer.

def imureader.ax100 (***imu*****)**

This function exists for testing, and prints x acceleration 100 times.

def imureader.ay (***imu*****)**

This function reads the y axis of the accelerometer.

def imureader.az (***imu*****)**

This function reads the z axis of the accelerometer.

def imureader.heading (*imu*)

This function returns a list of the components of magnetometer reading.

def imureader.magdrdy (*imu*)

This function checks the bit indicating whether a new set of measurements is available on the magnetometer.

def imureader.maginit (*imu*)

This function enables readings from the magnetometer.
Probably.

def imureader.magsingle (*imu*)

This function configures the magnetometer for a single conversion.

def imureader.magx (*imu*)

This function reads the x-axis of the magnetometer.

def imureader.magx100 (*imu*)

This function prints the x-axis of the magnetometer 100 times for testing purposes.

def imureader.magy (***imu*****)**

This function reads the y-axis of the magnetometer.

def imureader.magz (***imu*****)**

This function reads the z-axis of the magnetometer.

def imureader.printcontents (***imu*****,** ***device*****,** ***location*****)**

This function prints a byte of data located within an I2C device.

lights Namespace Reference

Functions

def **color** (direction)

Direction Color Choice.

def **intensity** (velocity)

Message for error.

def **brightness** (velocity)

This function returns a single factor for later use.

def **merge** (course, speed)

else: return -1

Variables

RED = tuple(2, 0, 0)

RGB address for red light.

ORANGE = tuple(2, 1, 0)

RGB address for orange light.

YELLOW = tuple(2, 2, 0)

RGB address for yellow light.

GREEN = tuple(0, 2, 0)

RGB address for green light.

LIGHTBLUE = tuple(0, 2, 2)

RGB address for light blue light.

BLUE = tuple(0, 0, 2)

RGB address for blue light.

PURPLE = tuple(2, 0, 2)

RGB address for purple light.

PINK = tuple(2, 0, 1)

RGB address for pink light.

out = **RED**

Red for North.

light_scale = int(velocity/.36)

Get magnitude for light intensity.

color = tuple([**light_scale***x for x in color])

Apply light intensity to color.

Function Documentation

def lights.brightness (*velocity*)

This function returns a single factor for later use.

def lights.color (***direction*****)**

Direction Color Choice.

def lights.intensity (***velocity*****)**

Message for error.
Color Intensity

```
Sets intensity of lights based off of velocity
```

def lights.merge (***course*****,** ***speed*****)**

else: return -1
This function returns RGB values ready for output to LEDs.

Variable Documentation

lights.BLUE = tuple(0, 0, 2)

RGB address for blue light.

lights.color = tuple([light_scale*x for x in color])

Apply light intensity to color.

lights.GREEN = tuple(0, 2, 0)

RGB address for green light.

int lights.light_scale = int(velocity/.36)

Get magnitude for light intensity.

lights.LIGHTBLUE = tuple(0, 2, 2)

RGB address for light blue light.

lights.ORANGE = tuple(2, 1, 0)

RGB address for orange light.

lights.out = RED

Light value sent to LED strip

lights.PINK = tuple(2, 0, 1)

RGB address for pink light.

lights.PURPLE = tuple(2, 0, 2)

RGB address for purple light.

lights.RED = tuple(2, 0, 0)

RGB address for red light.

lights.YELLOW = tuple(2, 2, 0)

RGB address for yellow light.

main Namespace Reference

Functions

def **compensate** ()
This function should allow IMU response when GPS data is unavailable.

Variables

imu = pyb.I2C(2, pyb.I2C.MASTER)
MASTER
baudrate
int **num_leds** = 29
LED_strip = WS2812(spi_bus = 2, led_count = **num_leds**)
list **rgb_data** = [(1, 0, 0)]***num_leds**
gps = pyb.UART(6, 9600, timeout=1000)
parser = MicropyGPS()
GPRMC = **gpsreader.parse**(**gpsreader.RMC(gps**))

Function Documentation

def main.compensate ()

This function should allow IMU response when GPS data is unavailable.

Variable Documentation

main.baudrate

main.GPRMC = gpsreader.parse(gpsreader.RMC(gps))

main.gps = pyb.UART(6, 9600, timeout=1000)

main.imu = pyb.I2C(2, pyb.I2C.MASTER)

main.LED_strip = WS2812(spi_bus = 2, led_count = num_leds)

main.MASTER

int main.num_leds = 29

main.parser = MicropyGPS()

list main.rgb_data = [(1, 0, 0)]*num_leds

ME_507_Project Namespace Reference

Functions

def **gpstopixels** (gps, parser, neopixels, num_leds)

This function should call all the requisite functions to update the neopixels.

def **fastgpstopixels** (gps, parser, neopixels, num_leds)

This function should call all the requisite functions to update the neopixels without the RMC buffer dump.

Function Documentation

def ME_507_Project.fastgpstopixels (*gps*, *parser*, *neopixels*, *num_leds*)

This function should call all the requisite functions to update the neopixels without the RMC buffer dump.

def ME_507_Project.gpstopixels (*gps*, *parser*, *neopixels*, *num_leds*)

This function should call all the requisite functions to update the neopixels.

File Documentation

boot.py File Reference

Namespaces

boot

gpsreader.py File Reference

Namespaces

gpsreader
docstring

Created on Tue Dec 6 21:50:20 2016. Functions

def **gpsreader.checksum** (packet)
This function implements the same XOR checksum as the gps.
def **gpsreader.readsum** (packet)
This function should read the checksum on incoming packets.
def **gpsreader.parse** (packet)
This function returns a list containing the separated components of a packet.
def **gpsreader.RMC** (gps)
This function searches for and returns a GPRMC sentence from the gps.
def **gpsreader.fastRMC** (gps)
Identical to RMC, but without the buffer dump.
def **gpsreader.nmeaupdate** (parser, sentence)
This function updates the NMEA sentence parser.
def **gpsreader.quickupdate** (gps, parser)
This function runs RMC and nmeaupdate together.
def **gpsreader.quickerupdate** (gps, parser)
This function runs RMC and nmeaupdate together.

imumath.py File Reference

Namespaces

imumath
docstring

Created on Tue Dec 6 21:50:20 2016. Functions

def **imumath.amagnitude** (components)
This function calculates the magnitude of the acceleration.
def **imumath.aleds** (components)
This function calculates backup LED values based on accelerometer data.

imureader.py File Reference

Namespaces

imureader
docstring

Created on Tue Dec 6 21:50:20 2016. Functions

def **imureader.accelenable** (imu, rate)
This function sets the accelerometer power mode and data rate.
def **imureader.printcontents** (imu, device, location)
This function prints a byte of data located within an I2C device.
def **imureader.accelaxes** (imu, x, y, z)
This function enables or disables the axes of the accelerometer.
def **imureader.accelp** (imu, mode)
This function determines whether the accelerometer operates in low-power mode.
def **imureader.accelsetup** (imu, rate, mode, x, y, z)
This function sets up the IMU CTRL_REG1 in a single step.
def **imureader.accelquickstart** (imu)
This function sets up the IMU with the preset configuration I expect to want to use.
def **imureader.ax** (imu)
This function reads the x axis of the accelerometer.
def **imureader.ay** (imu)
This function reads the y axis of the accelerometer.
def **imureader.az** (imu)
This function reads the z axis of the accelerometer.
def **imureader.acceleration** (imu)
This function returns a list containing each component of acceleration.
def **imureader.ax100** (imu)
This function exists for testing, and prints x acceleration 100 times.
def **imureader.ares** (imu, hires)
This function controls high resolution mode.
def **imureader.maginit** (imu)
This function enables readings from the magnetometer.
def **imureader.magdrdy** (imu)
This function checks the bit indicating whether a new set of measurements is available on the magnetometer.
def **imureader.magsingle** (imu)
This function configures the magnetometer for a single conversion.
def **imureader.magx** (imu)
This function reads the x-axis of the magnetometer.
def **imureader.magy** (imu)

This function reads the y-axis of the magnetometer.

def **imureader.magz** (imu)

This function reads the z-axis of the magnetometer.

def **imureader.magx100** (imu)

This function prints the x-axis of the magnetometer 100 times for testing purposes.

def **imureader.heading** (imu)

This function returns a list of the components of magnetometer reading.

lights.py File Reference

Namespaces

lights
docstring

Created on Tue Dec 6 21:50:20 2016. Functions

def **lights.color** (direction)
Direction Color Choice.
def **lights.intensity** (velocity)
Message for error.
def **lights.brightness** (velocity)
This function returns a single factor for later use.
def **lights.merge** (course, speed)
else: return -1

Variables

lights.RED = tuple(2, 0, 0)
RGB address for red light.
lights.ORANGE = tuple(2, 1, 0)
RGB address for orange light.
lights.YELLOW = tuple(2, 2, 0)
RGB address for yellow light.
lights.GREEN = tuple(0, 2, 0)
RGB address for green light.
lights.LIGHTBLUE = tuple(0, 2, 2)
RGB address for light blue light.
lights.BLUE = tuple(0, 0, 2)
RGB address for blue light.
lights.PURPLE = tuple(2, 0, 2)
RGB address for purple light.
lights.PINK = tuple(2, 0, 1)
RGB address for pink light.
lights.out = RED
Red for North.
lights.light_scale = int(velocity/.36)
Get magnitude for light intensity.
lights.color = tuple([light_scale*x for x in color])
Apply light intensity to color.

main.py File Reference

Namespaces

main
docstring

Created on Tue Dec 6 21:50:20 2016. Functions

def **main.compensate** ()
This function should allow IMU response when GPS data is unavailable.

Variables

main.imu = pyb.I2C(2, pyb.I2C.MASTER)
main.MASTER
main.baudrate
int **main.num_leds** = 29
main.LED_strip = WS2812(spi_bus = 2, led_count = num_leds)
list **main.rgb_data** = [(1, 0, 0)]*num_leds
main.gps = pyb.UART(6, 9600, timeout=1000)
main.parser = MicropyGPS()
main.GPRMC = **gpsreader.parse**(**gpsreader.RMC**(gps))

ME_507_Project.py File Reference

Namespaces

ME_507_Project
docstring

Created on Tue Dec 6 21:50:20 2016. Functions

def **ME_507_Project.gpstopixels** (gps, parser, neopixels, num_leds)
This function should call all the requisite functions to update the neopixels.
def **ME_507_Project.fastgpstopixels** (gps, parser, neopixels, num_leds)
This function should call all the requisite functions to update the neopixels without the RMC buffer dump.