MULTI-SENSOR ARRAY (VERSION 4)

Design Document

Team: Michael Petersen
Jen Stoddard
Wesley Mahurin

Project: MSAv4

Date Prepared: November 20, 2013

This page is intentionally blank

Figure 1: 28 July 2013 :Balloon Bonanza Test Flight

This page is intentionally blank

Table of Contents

List of Figures

List of Tables

Revision History

Date	Author	Revision Made
20 November, 2013	Petersen, Michael	Original Document

List of Abbreviations and Definitions

AIAA	American Institute of Aeronautics and Astronautics
ECIB	External Controls and Indicators Board
ESB	External Sensors Board
FAA	Federal Aviation Administration
GPSIM	GPS System Interface Module
HARBOR	High Altitude Research Balloon for Outreach and Research
MSAv3	Multi-Sensor Array (Version 3)
MSAv4	Multi-Sensor Array (Version 4)
OUR	Office of Undergraduate Research
Rpi	Raspberry Pi
SID	System Interface Daughterboard
SSH	Secure Shell

1.0 Introduction

The purpose of this project is to produce a new MSAv4 (Multi-Sensor Array version 4) which is capable of gathering in-situ data from the polluted atmospheric inversions zones common in urbanized mountain basins and from the lower stratosphere at the edge of space. The MSAv4 is to be used by the High Altitude Reconnaissance Balloon for Outreach and Research (HARBOR) team; which is a local research group that studies atmospheric pollution, ozone levels, and anthropogenic climate change.

The MSAv4 consists of accelerometers, magnetometers, temperature sensors, humidity sensors, and pressure sensors. The MSAv4 also supports an expandable array of particulate and gas sensors; with a simple user interface. This system has flown to the mid-stratosphere (30-34km above sea level) on five HARBOR missions.

The MSAv4 will be replacing the MSAv3 (Rob Eckel version) that has been used by HARBOR for prior missions. The MSAv3 is a single board data logger facilitated by a PIC-18 microcontroller. The MSAv3 is capable of monitoring flight dynamic data and some environmental data. It is modular in design and is partially expandable. The primary limitations of the MSAv3 are small code memory, slower processor speed, and the required programming skills necessary to add "guest" packages.

This projected was requisitioned by Dr. John Sohl from the Weber State University Physics Department. Dr. Sohl is the current Director of HARBOR and the primary contact for questions regarding the design and use of the MSAv4.

The development of the MSAv4 was funded in part by the Office of Undergraduate Research with funds donated by the Ralph Nye Charitable Foundation. A grant proposal was submitted to the Office of Undergraduate Research by our team in April of 2013 and was accepted. Additional funds were allocated by the AIAA (American Institute of Aeronautics and Astronautics) and the Val A. Browning Foundation.

2.0 Scope

This documents contains the hardware and software design of the MSAv4; including the main interface board, external sensor board, external control board, and GPS interface board. It describes the overall use of the Raspberry Pi single board computer to meet the requirements shown on page 124 in the Concept Document. Though set-up is discussed, this document does not explain the inner workings of the Raspberry Pi nor go into depth with the libraries used to interface with it. This document does not describe all of the procedures for the construction of individual parts(i.e. it is assumed that the reader is capable of PCB construction and assembly). This document also does not describe in detail the science for which the MSAv4 is designed to conduct, though testing and calibration are covered.

3.0 Design Overview

The MSAv4 project requires the design and construction of four primary physical components:

1. Data logging computer to store sensor data
2. Interface boards for both onboard and external devices
3. Controls and Indicators board that can be accessed from the outside
4. Enclosures to house and protect the MSAv4 components

The project also requires the design of a software system to perform six main functions:

1. Initiate the main program when powered up
2. Start the data logging program when the mission pin is pulled
3. Monitor battery voltage and initiate a safe shutdown in case of low voltage
4. Monitor altitude and initiate a warning buzzer on descent, a few thousand feet above the ground. And then turn off on landing (Or produce a less frequent chirp)
5. Read, process, and store sensor data at a preset sample rate, and store data in .CSV formatted files
6. Initiate safe shutdown at the end of the mission

3.1 Requirements

Dr. Sohl provided a list of primary and secondary design concepts for the MSAv4 design on 30 November 2012. See page 124 for the original concept document.

3.1.1 Primary Project Goals

Five primary project goals were provided in the concept document. All of the goals were met; except for GPS capability. A list of secondary goals was also supplied, but they were not addressed in this project. The primary goals are as follows:

1. Easy operation on the flight line with external controls and indicators
2. Total weight (not including the gas and dust sensor assembly) shall be less than the current weight which is 0.9 lbs (408 g)
3. Total size with foam insulation box shall be no larger than 12 " square
4. The unit shall have on-board data memory storage that is accessible via either USB or SD card
5. The unit shall have the ability to have "guest" packages attached with programming that is easy to modify

3.1.2 Functional Requirements

The functional requirements are based in part on the existing MSAv3 system. The new system must replace the old system as well as provide new functionality. The following list of requirements encompasses all of the existing functions as well as new functions.

- The MSAv4 shall have external controls and indicators
- The MSAv4 shall have an on-board data memory storage either USB or SD
- The MSAv4 shall have a system where "guest" packages can be easily added
- Data shall be time stamped
- The MSAv4 shall read 3-axis acceleration in both the ± 2g and ± 8g range
- The MSAv4 shall read 3-axis gyroscopic motion
- The MSAv4 shall take 3-axis magnetic field measurements
- The MSAv4 shall log GPS data; including time, longitude, latitude, altitude, and number of tracking satellites
- The MSAv4 shall monitor internal temperature
- The MSAv4 shall monitor external ambient temperature between -50ºC and 50ºC
- The MSAv4 shall monitor atmospheric absolute pressure between 1 mbar and 1000 mbar (0.75 torr to 750 torr)
- The MSAv4 shall monitor both external and internal relative humidity
- The MSAv4 shall monitor battery voltage and initiate a safe shutdown when the per-cell voltage drops below 3.2V per cell
- The MSAv4 shall monitor particulates with a particle sensor and flow meter
- The MSAv4 shall be capable of monitoring gas compositions; including Carbon Dioxide, Ozone, and various Nitrogen Oxides
- The MSAv4 shall be equipped with a motion and altitude triggered external siren

3.1.3 Performance Requirements

No specific performance requirements are called out for in the concept document; however, it is known from experience that the MSAv4 must be capable of operating for a minimum of 4 hours to ensure that data are collected throughout the entire mission.

3.1.4 Interface Requirements

The design document specifies that at least one communication interface must be made available to guest sensors. A list of interfaces is given at the end of the concept document in order of preference. Dr. Sohl granted our team freedom to chose which of the interfaces to use. The three main data interfaces chosen for this project are I2C, analog (0V - 1.65V) and RS-232 (UART). Guest sensors must also be provided 3.3V and 5.0V power and ground. To reduce heat through power dissipation at the regulators, a 7.4V battery shall be used.

The MSAv4 must have onboard data storage. SD was chosen for memory storage since it comes ready with the Raspberry Pi computer.

3.1.5 Environmental Requirements

The MSAv4 will operate in extreme environments and conditions. The list of environmental requirements are as follows:

- The MSAv4 components shall be rated for 50ºC to -50ºC
- The MSAv4 shall be capable of dissipating heat at 100, 000 feet above sea level
- The MSAv4 shall be capable of measuring pressure to 1 mbar
- The MSAv4 shall be capable of 16g forces during flight and landing
- The MSAv4 shall be resistant to Radio Frequency Interference

3.2 Constraints

3.2.1 Cost

Dr. Sohl did not constrain the cost of the MSAv4 system or the cost of development. However, it was recommended that a research grant be acquired to cover the cost of development. A grant proposal was submitted and the amount of $898 was given to cover development, testing, and building costs. Dr. Sohl did mention that he would prefer a system that was close to or less than $100 dollars per unit for the HARBOR LITE program which has not yet started.

3.2.2 Physical

The FAA limits the weight of science packages flown on research balloons to 12 lbs. The MSAv4 is one of several instruments flown during a HARBOR missions. The mass budget requirement for the MSAv4 is 0.9 lbs (408 grams), not including the gas sensor array or the particle sensor system.

HARBOR requires that packages be smaller than 12" cube, preferably 8" cube. Also, packages must be separated from other packages by a minimum distance of 6 feet.

3.2.3 Scheduling

An MSAv4 prototype shall be available for the 2013 HARBOR flight season; this requirement was met with a limited function (temperature, humidity, and particle readings) MSAv4. The syllabus gives a hard project deadline of December 2013.

3.2.4 Safety

The MSAv4 must be designed to be light and compact. The MSAv4 flies in a stack of instruments that must be separated by 6 feet to minimize impact in the event of a midair collision with other aircraft. For this reason, little or no metal should be used in the construction of the inner enclosure and insulated box. Weight and materials are the most important factor with regard to safety. Additionally, there is a requirement for the

MSAv4 to initiate a siren after descending below a few thousand feet above the ground. This is for the benefit of individuals on the ground who may not be aware of the package descending toward them. There is no FAA requirement with regard to warning sirens for research balloon packages.

3.3 Applicable Standards

The MSAv4 requires the use of several standards; including I2C, RS-232 (UART), and ANSI-C.

3.3.1 I2C

The MSAv4 uses the I2C bus as the primary means of communicating with sensors and external devices. All of the devices attached to the MSAv4 use the standard I2C protocol for communication; except for the GPS. The GPS transmit RS-232 signals, but they are converted to I2C by the XR20M1280 chip. I2C devices connected to the bus must run at high speed (400kHz).

3.3.2 RS-232

The MSAv4 is capable of communicating directly with one device over the single available UART channel. The RPi outputs TTL levels (0v to 3.3V) which are inverted; a mark is low and a space is high. However, there are no default sensors which use this channel. The XR20M1280 UART to I2C converter chip also uses Inverted TTL levels to communicate with RS-232 devices. The GARMIN GPS unit uses standard RS-232 protocol; which means that the signal from the GPS unit must be converted to TTL levels and inverted.

3.3.3 ANSI-C

The MSAv4 source code is compiled with the open source GCC compiler; which is compatible with ANSI-C, ISO-C, and some features of C89 and C90. The GCC compiler standards are maintained at:

http://gcc.gnu.org/onlinedocs/gcc/Standards.html

3.4 Dependencies

3.4.1 Power Supply

The MSAv4 is battery powered. The battery of choice is a 7.4V LiPo (Lithium Polymer) battery with a minimum capacity of 4 hours. The MSAv4 draws 310mA in standby and normal mode; it only draws 180mA in data logging mode. The recommended minimum battery capacity is 1100mAh.

3.4.2 I2C Bus (BSC0)

The MSAv4 communicates to external devices via an I2C bus. The RPi (version 1 - model B) was used in the development of the MSAv4. This version has two I2C busses;

labeled BSC0 and BSC1. The RPi (version1 - model B) defaults to the BSC0 bus for I2C communications. For this project, the I2C bus is operated at 400 KHz (I2C high speed). All I2C devices and sensors shall be chosen which are capable of operating at high speed.

3.5 Theory of Operation

The MSAv4 is a data logging computer for monitoring flight dynamics data, environmental data, and GPS coordinates. It is composed of four primary components:

1. Main computer, data logger, and processor
2. Sensor and device interface
3. External controls and indicators
4. External and internal sensors

The MSAv4 system uses a RPi (Raspberry Pi) single board computer as the primary platform for gathering, processing, and storing data. Linux scripts are used to initiate the main program, safe shutdown, monitor CPU temperature, and disable unused chips to save power.

The RPi's I2C bus is used as the primary link to sensors, clocks, and other external devices. The BCM2835 C library(McCauley, 2013) is used to control the RPi's GPIO, SPI, UART, and I2C pins.

The SID (System Interface Daughterboard) houses all of the standard flight dynamic and environmental sensors, ADCs, amplifiers, power regulators, and I/O connectors. The SID is designed to connect directly to the RPi via a 2x13 GPIO header; an additional unused 1x8 header provides physical mounting support. The SID supplies regulated 3.3V and 5.0V to the entire system; including the RPi.

The SID is the primary link from the RPi to all external devices and components; except for remote SSH (Secure Shell); which is accessed via the Ethernet port.

The SID is controlled and monitored from the ECIB (External Controls and Indicators Board) Board. The ECIB contains status LEDS, an external power switch, a mission pull pin, and a Descent Warning buzzer.

The ESB (External Sensor Board) contains a temperature and humidity sensors that are linked to the SID. It is mounted inside the walls of the main enclosure with access to the outside environment. See Figure 2 below for a detailed overview of how the main components are connected.

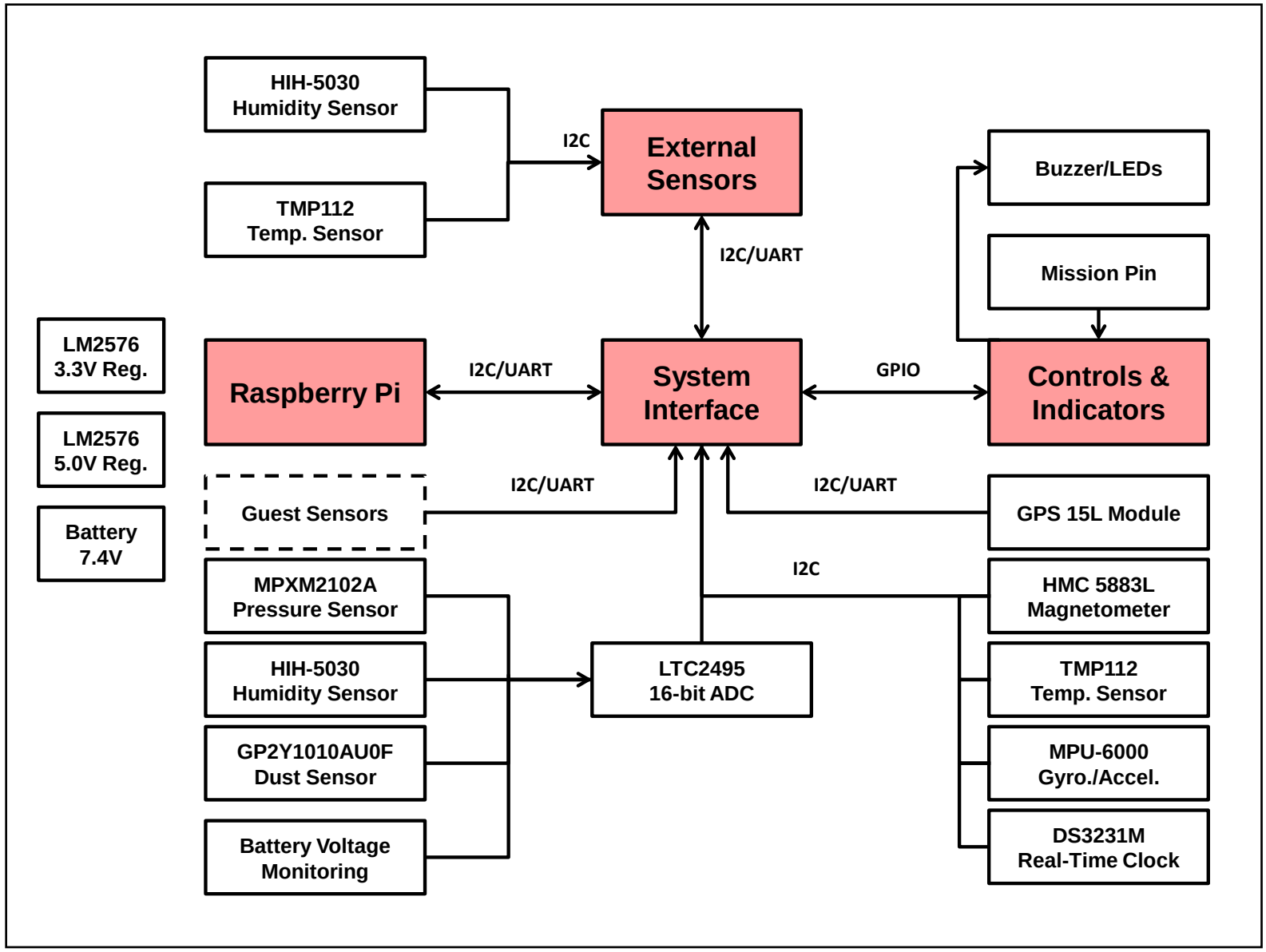

Figure 2: MSAv4 Block Diagram

3.5.1 Raspberry Pi

The Raspberry Pi is a single board computer with a 1 GHz processor and SD hard drive with up to 32 GB of memory. It has easily accessible GPIO, UART, I2C, and SPI pins; located on a 2x13 male head on top of the board. It can be plugged directly into an HDMI or RCA style monitor and accessed with a USB mouse and keyboard. It can also be accessed remotely via SSH. See Figure 3 below for an overview of the RPi.

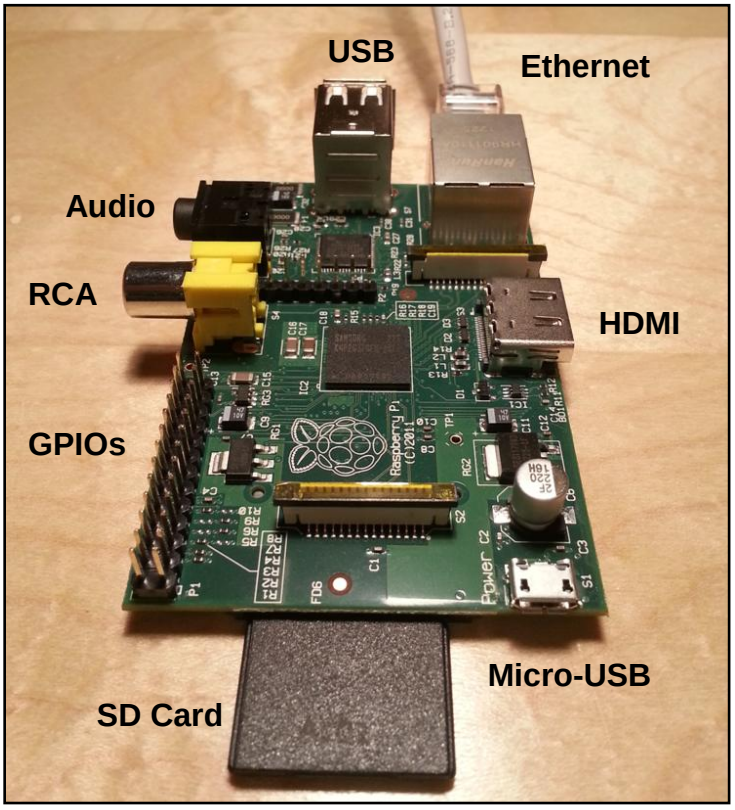

Figure 3: Raspberry Pi (Model B - Version 1)

The RPi executes the main data logging program and stores collected data on the SD hard drive. All code written for the MSAv4 is compiled directly on the RPi using a open source Linux compiler known as GCC. The RPi runs on 5VDC and draws approximately 400mA peak under normal operation. The RPi is programmed through SSH via the Ethernet port. This makes it possible to reprogram the RPi on the fly during flight missions. Several free tools can be downloaded for the RPi that can be used to test the I2C bus, UART channels, and other GPIOs; these tools will be covered in more detail in section 4.

3.5.2 System Interface Daughterboard

The SID is the primary interface between sensors, controls, and external devices and the RPi. It is designed to mount directly on to the RPi via the 2x13 GPIO header row.

The main components of the SID are:

- 16-bit 16-channel ADC
- Magnetometer
- Accelerometer / Gyroscope
- Real Time Clock and 3.3V Clock Memory Battery
- 3.3V Regulator
- 5.0 V Regulator
- Pressure sensor with Instrumentation Amplifier
- Humidity Sensor
- Temperature Sensor
- Status LED
- Pushbutton
- I2C, 3.3V, 5.0V, GPIO, and ADC Header Connections
- ECIB Harness and Connector
- ESB Harness and Connector

Figure 4 provides a detailed breakout diagram of all components mounted to the SID.

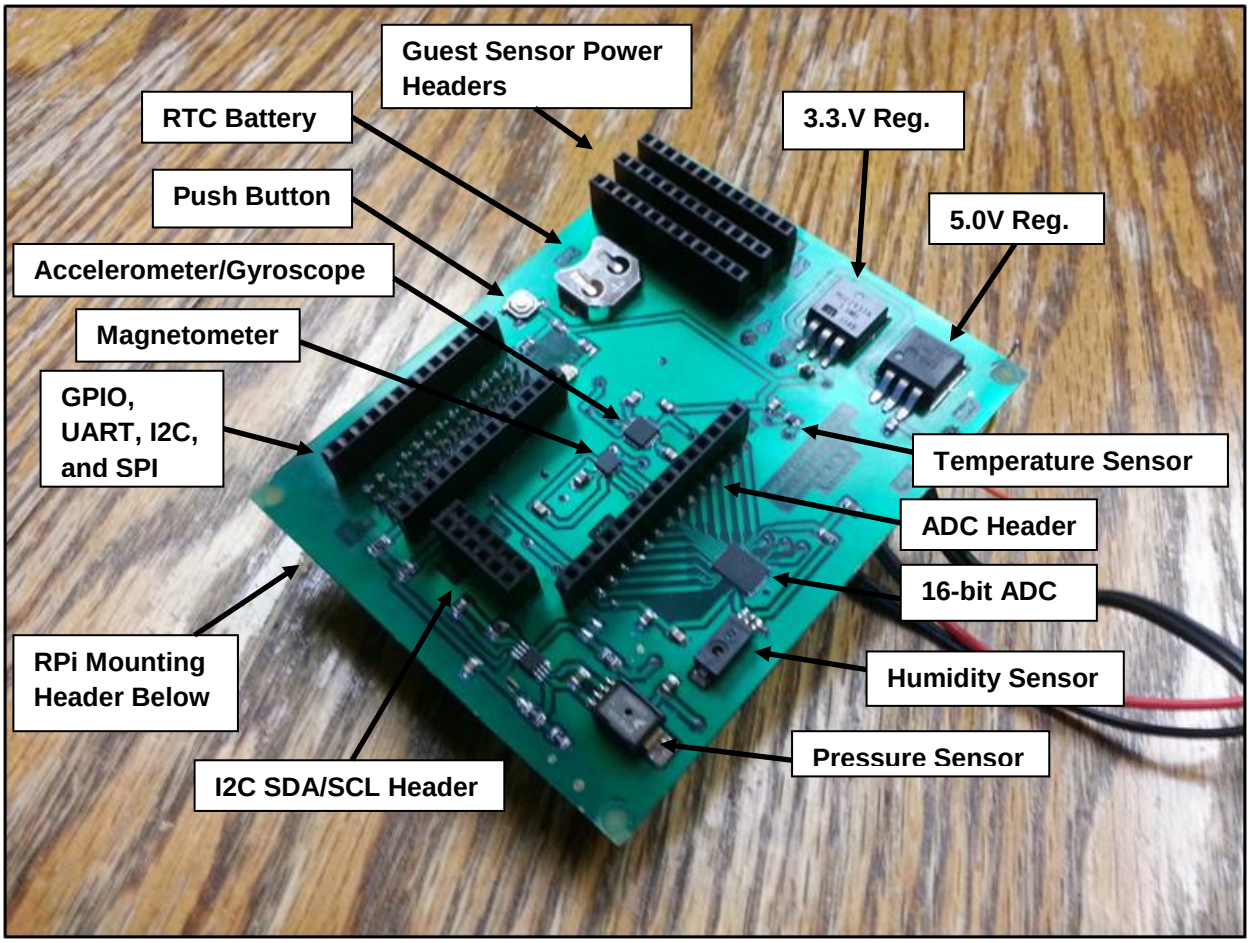

Figure 4: System Interface Daughterboard

The SID is designed to supply power to guest sensors and provides inputs to the ADC, the I2C bus, and all of the RPi GPIO pins. This makes adding new guest sensors and devices simple.

3.5.3 External Controls and Indicators Board

The ECIB is designed to allow the user to control the RPi once it is sealed into the flight box; which is taped and strapped before flight. From here, the user can turn the main power on and off, and initiate the flight program. Pulling the flight pin initiates the flight program.

The main components of the ECIB are:

- ON/OFF Power Switch
- RPi Status, RPi Power, and Battery Power LEDs
- Descent Warning Buzzer (Look-Up Alarm)
- Mission Pull Pin

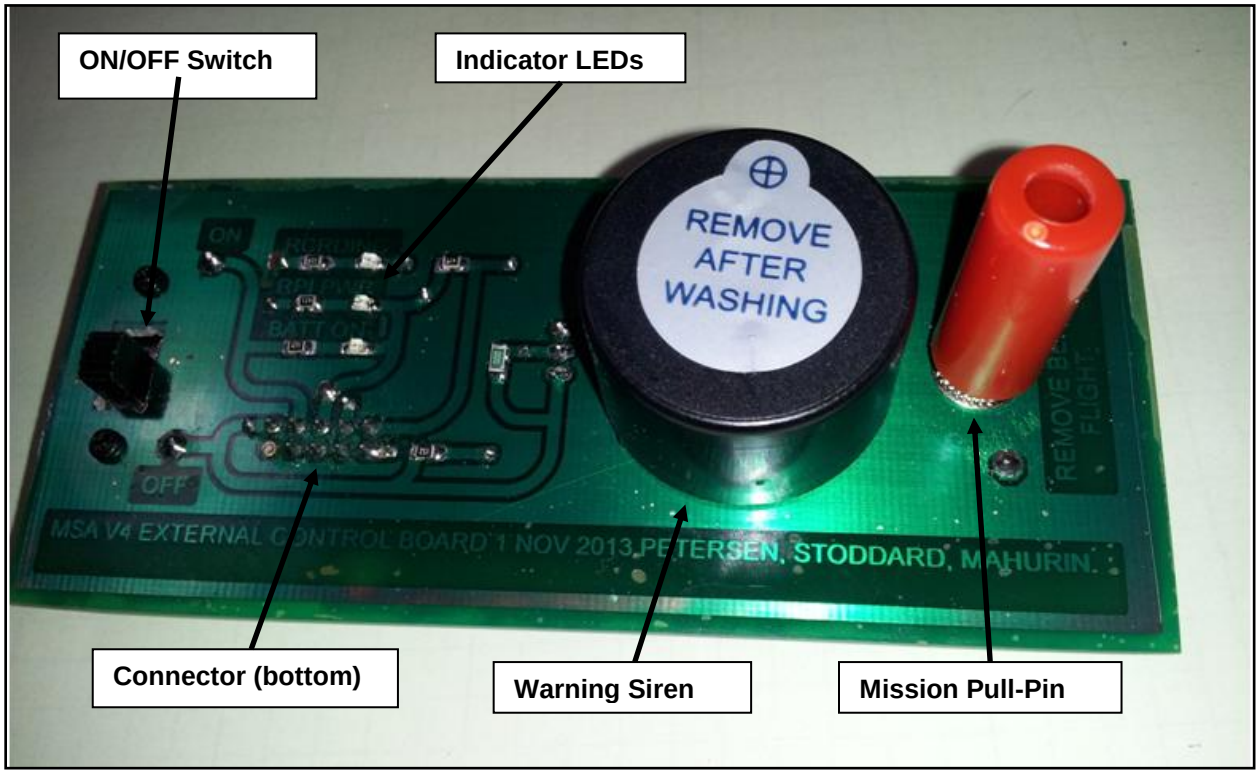

Figure 5: External Controls and Indicators Board

3.5.4 External Sensor Board

The ESB is self explanatory. This board is exposed to the outside environment to collect temperature and humidity data. The main components of the ESB are:

- HIH5030 Humidity Sensor
- TMP112 Temperature Sensor

Figure 6 provides a detailed breakout diagram of the ESB.

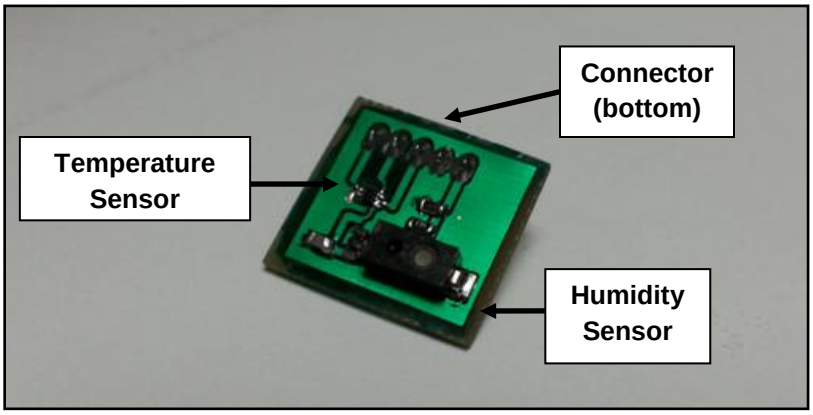

Figure 6: External Sensors Board

3.5.5 GPS Interface Module

The GPSIM is the interface board that links the GARMIN 15xL GPS receiver module to the SID. The 15xL transmits GPS data and coordinates via RS-232 at normal levels. The RPi UART channel requires TTL levels (0 to 3.3V) and the signal must be inverted. The GPSIM has a connector for the GPS module and one for the SID. It is designed to be mounted to the top of the internal enclosure along with the GPS receiver.

The main components of the GPSIM are:

- RS-232 to TTL Signal Converter/Inverter
- UART to I2C Converter Chip
- GPS Module

The GPSIM board was not completed for this project. A board has been designed and etched, but is still needs to be completely assembled and tested. This will be finished at a later date.

4.0 Design Details

This section provides a detailed description of the design and development of the MSAv4 system. All of the necessary information for setting up the RPi to work with the MSAv4 is provided in the first section. Sections 4.2 - 4.5 describe the design, layout, and construction of the four primary PCBs manufactured for the MSAv4:

1. External Sensor Board
2. External Controls and Indicators Board
3. GPS Interface Board
4. System Interface Daughterboard

Sections 4.6 - 4.7 provide a brief summary of the design and construction of the two enclosures built for the MSAv4:

1. Internal MSAv4 enclosure
2. External Flight Box

Section 4.8 provides a very detailed description of the design and implementation of the MSAv4 operational flight program. Section 4.9 provides a list of requirements and design concepts that were not sufficiently met.

4.1 Raspberry Pi

This project uses the RPi (version 1 - Model B), manufactured by Element14. The operating system used for this particular device is 2013-07-26-wheezy-rasbpian.img. The operating system was downloaded and installed from the raspberrypi.org website:

< http://www.raspberrypi.org/>

Instructions for installing the operating system are also found at the raspberrypi.org website. The RPi was chosen because it is light weight (40 grams), it is relatively inexpensive ($35), and because it comes with a free Linux operating system.

The Raspberry Pi is a single board computer with a 1 GHz processor and SD hard drive that can have up to 32 GB of memory. It has easily accessible GPIO, UART, I2C, and SPI pins; located on a 2x13 male header located on the top of the board. It can be plugged directly into an HDMI or RCA style monitor and accessed with a USB mouse and keyboard. It can also be accessed remotely via SSH. See page 20 for an overview of the RPi.

4.1.1 SSH and Setting Up the RPi Static IP Address for Communication

The first step in using the RPi for the MSAv4 platform is to enable the SSH tool so that the RPi can be directly connected to another computer via Ethernet. The best way is to

access the RPi from another computer running Linux; such as Ubuntu. A new RPi can only be accessed via HDMI on a computer monitor or TV and a USB keyboard. The following lines of code are entered into the command line (ctrl + alt + T) to enable the SSH tools on the RPi.

```
sudo bash
ssh-keygen
<enter>
<enter>

ls /root/.ssh
```

This command verifies that the private keys are generated.

```
service ssh start
service ssh status
```

This command verifies that SSH is running.

```
sudo update-rc.d ssh defaults
sudo reboot
```

After reboot, it is necessary to set up the Static IP so that the RPi can be accessed from the Ethernet port. To do this, the gateway address between the computer used with the RPi and the RPi itself must be matched. The following line of code accesses the IP file on the RPi. This is the code used for the MSAv4 RPi:

```
sudo nano /etc/network/interfaces
```

The file is then modified to look like this:

```
auto lo
iface lo inet loopback

# setup for Raspberry Pi to connect to Laptop via Ethernet
auto eth0
iface eth0 inet static
address 172.16.1.11
 mask 255.255.255.0
 gateway 172.16.1.2

# Make sure Laptop is configured as follows
# auto eth0
# iface eth0 inet static
# address 172.16.1.21
# mask 255.255.255.0
# gateway 172.16.1.1

allow-hotplug wlan0
iface wlan0 inet manual
wpa-roam /etc/wpa_supplicant/wpa_supplicant.conf
```

```
iface default inet dhcp
```

The laptop used to program the RPi was setup as mentioned above. The RPi was rebooted again to make the changes take effect. An Ethernet cable was connected between the RPi and the project laptop. From the laptop, the following code was entered into the command line gains access to the RPi:

```
sudo ssh pi@172.16.1.11

<enter password: raspberry>
```

4.1.2 Unlocking the I2C Bus and Using i2c-tools to Verify

The RPi image for this project comes with the I2C busses disabled. The following lines of code are used to enable the bus:

```
sudo nano /etc/modprobe.d/raspi-blacklist.conf
```

In this file, the following line is commented out to look like this:

```
#blacklist i2c_bcm2708
```

The file is saved and closed. Next, the modules file is accessed with:

```
sudo nano /etc/modules
```

In this file, the following line is added:

```
i2c-dev
```

The file is saved and closed. To access the I2C bus it is necessary to install the Linux tool *i2c-tools*. To do this the RPi is connected to the internet and the following Linux command is used:

```
sudo apt-get install i2c-tools
```

After the installation process is complete, a user name is added to the bus:

```
sudo adduser pi i2c
sudo reboot
```

The *i2c-tools* program is used to verify that I2C devices are connected to the I2C bus. Entering the following code generates a list with the slave address of any I2C devices connected to the bus.

```
sudo i2cdetect -y 0
```

This tool was used to verify each device used for the MSAv4 project.

4.1.3 Unlocking the UART Pins and Using Mini-com to Verify

As with the I2C bus, the UART pins are initially disabled for the user. The pins are, by default, used for logging kernel data from the RPi and for trouble shooting at the factory. To recover functionality of these pins, it is necessary to modify the *cmdline.txt* file in the *boot* folder. The following line of code accesses the file:

```
sudo nano /boot/cmdline.txt
```

The line:

```
dwc_otg.lpm_enable=0 console=ttyAMA0 115200 kgdboc=ttyAMA0 115200 console=tty1
root=/dev/mmcblk0p2 rootfstype=ext4 elevator=deadline rootwait
```

is changed to:

```
dwc_otg.lpm_enable=0 console=tty1 root=/dev/mmcblk0p2 rootfstype=ext4
elevator=deadline rootwait
```

The file is saved and closed. The *inittab* file is then changed to prevent the RPi from automatically spawning a UART channel.

```
sudo nano /etc/inittab
```

The following line is commented out to look like this:

```
#T0:23:respawn:/sbin/getty -L ttyAMA0 115200 V + 100
```

The file is saved and closed. After the RPi is rebooted, the UART pins are then available for use. The Linux tool *minicom* was used to verify the function of the UART pins. To install *minicom* the following line of code is used in the RPi command line; while connected to the internet:

```
sudo apt-get install minicom
```

After minicom is installed, the following line of code is used to open the terminal and set the baud rate to 4800:

```
sudo minicom -b 4800 -o -D /dev/ttyAMA0
```

The *minicom* tool was used to verify both transmitted and received data over the UART pins.

4.1.4 The Raspberry Pi GPIO Pins

The MSAv4 makes special use of the RPi GPIO pins; particularly the I2C, which is used as the primary communication link between sensors and other devices. Several GPIO pins are used to monitor the pull-pin and push-button status, as well as control an LED

and buzzer. The pin break-out and pin numbering for the RPi can be found in Figure 7 below. A photograph of the GPIO pins on the RPi can be found on page 20.

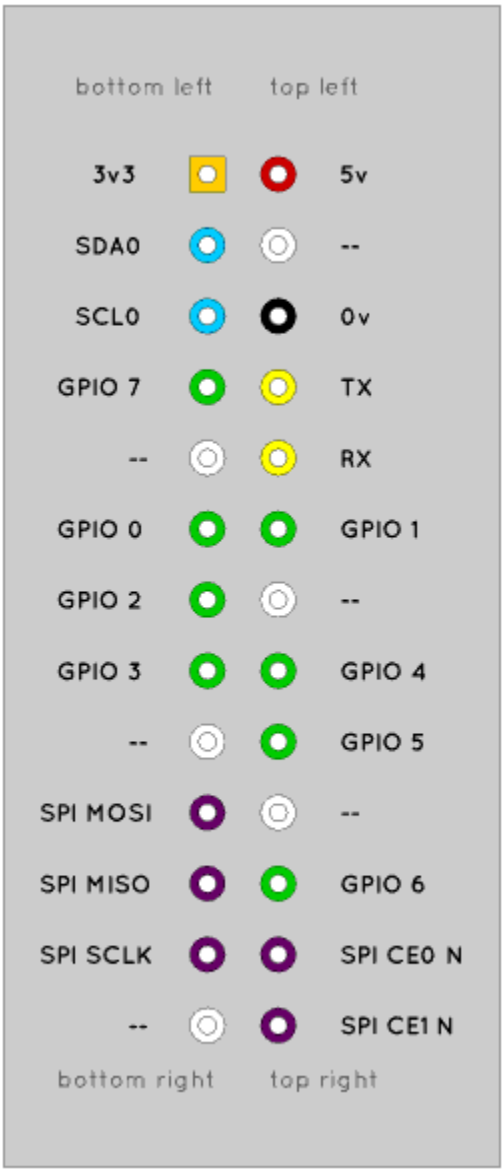

Figure 7: RPi GPIO Pin Out and P1 Header with Numbering
Source: elinux.org(2012)

4.1.5 Conserving Power in Data Logging Mode

The RPi draws between 310mA and 390mA during normal operation; this use includes the current draw of the HDMI output chip and the USB/Ethernet chips. One half of the current consumption by the RPi is from the HDMI and USB chips. To prolong battery life during flight, these chips must be powered down. This step is handled in the operational flight program, but it is mentioned here because it requires the use of two Linux scripts. The first script shuts down the HDMI video chip:

```
sudo /opt/vc/bin/tvservice -off
```

The C code for this script is:

```
system("/opt/vc/bin/tvservice -off"); // shut off PAL/HDMI outputs to save power
```

The next line of code suspends the USB/Ethernet controller; which generates approximately 8000 interrupts per seconds under normal operation.

```
sudo sh -c "echo 1 > /sys/devices/platform/bcm2708/usb/bussuspend"
```

The C code for this script is:

```
system("sh -c \"echo 1 > /sys/devices/platform/bcm2708/usb/bussuspend\" ");
// shut off USB chip to save power
```

Both of these chips are enabled when the MSAv4 is rebooted or powered up for the first time.

4.2 External Sensor Board

The external sensor board measures temperature and humidity outside of the MSAv4 flight box. The external board is a simple PCB designed to hold a TMP112 digital temperature sensor and an HIH5030 analog humidity sensor. It measures 3/4" x 3/4" and weighs 14.26 grams (with harness). The same type of each sensor on the External Sensor Board is also mounted on the SID for measuring internal humidity and temperature. Similar functions are called in the operational flight program to read these sensors; though the TMP112 has a different slave address (0x49), and the HIH5030 uses a different ADC channel (channel 3).

4.3 External Controls and Indicators Board

The ECIB (External Controls and Indicators Board) gives users the capability to know if the battery if hooked up, the RPi is powered, or if the RPi is recording at a glance with three LEDs. The ECIB incorporates a “pull before flight” pin, a piezo buzzer, and an on/off switch.

When the battery is connected to the ECIB, the bottom LED marked, “BATT ON” lights up. To power the RPi, select “ON”, with the slide switch on the left side of the ECIB. Note that the buzzer will sound to indicate the RPi is powered, additionally, the middle LED marked, “RPI PWR”, will light. After the RPi is powered ON, pulling the “REMOVE BEFORE FLIGHT” pin, will cause the “RCRDING” LED to flash, and the buzzer will cease. At this point the RPi is in data logging mode.

Selecting “OFF” on the slide switch on the left side of the ECIB at any time will kill power to the RPi. The power switch is designed to break the ground circuit to the SID. Experimentation has shown that a ground still exists through the resistors on the ECIB, this causes the battery light to illuminate and the piezo buzzer to sound. This is undesirable since it puts the RPi and other components at risk due to unanticipated voltages. This issue will be addressed in a future update.

The "RCRDING" LED is linked to the status LED, mounted on the SID. It is also controlled by the same GPIO (Pin 11) as the status LED. The LED illuminates by grounding through Pin 11.

The piezo buzzer is a 3.3V, 100dB, 3.5kHz alarm. It is controlled by GPIO (Pin 18). The RPi sends a 3.3V signal to the gate of a 2N7000 transistor to cause the buzzer to sound. The 2N7000 provides the buzzer a path to ground and limits the current from the RPi GPIO.

The pull pin is designed to drop the node voltage measured by GPIO (Pin 15) to below 1V when it is inserted. When the pin is pulled, the node voltage is pulled to 3.3V by a 4.7KΩ resistor. The RPi acknowledges that the pin has been released by flashing the "RCRDING" LED in quick 50ms bursts and sounding the buzzer in unison with the LED. When the "RCRDING" LED stabilizes at 1s pulses, it is in data logging mode.

The ECIB circuit board is a two-sided circuit board created using Eagle V6.5.0 by CADSOFT (See Figure 8).

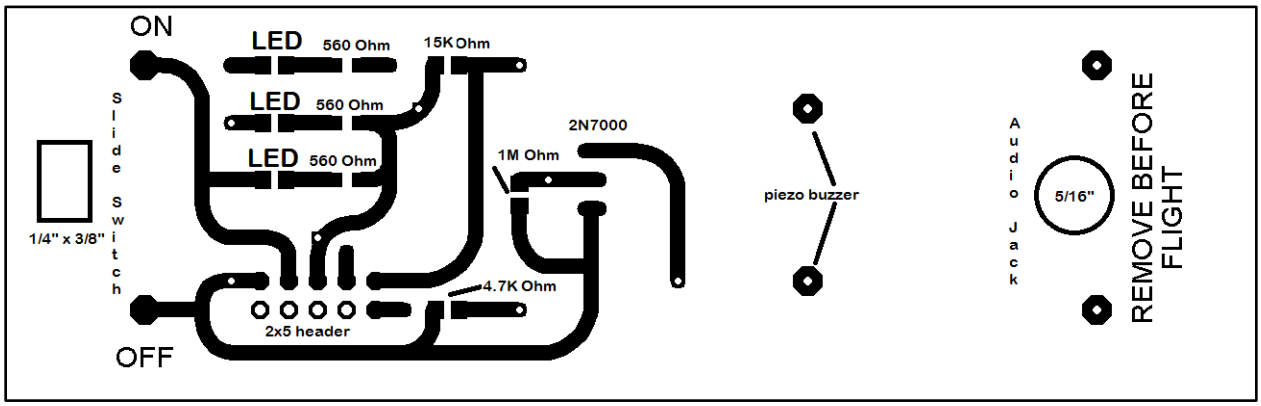

Figure 8: ECIB Board Plan (Front View)

4.4 GPS Interface Module

The GPSIM converts UART to I2C allowing the MSAv4 to link with the GARMIN 15xL receiver module. This makes use of the I2C bus rather than using up the only UART channel on the RPi. The GPSIM is equipped with a XR20M1280 UART to I2C converter with a 128 byte FIFO. This section describes the interface between the XR20M1280, the GPS receiver module, and the operational flight program.

4.4.1 XR20M1280

This device is the main component of the GPS Interface Module. The XR20M1280 converts between UART and I2C or SPI. We connect pin 24 high to select the I2C interface. The address of this device is set to 0x35 on this board by tying both address pins to ground.

The 4MHz crystal is used along with device registers, DLM, DLL and DLD to set the baud rate for the UART to 4800 to match the GPS device.

4.4.1.1 Line Control Register (LCR)

The LCR is at address 0x18 and for regular operation it is set to 0x03, which sets the device to one stop bit with an eight bit word. However, LCR must be changed to access some registers. To read or write to the enhanced registered LCR must be set equal to 0xBF. When LCR is set to this value, reading LCR will not give 0xBF, but enhanced registers can still be accessed.

4.4.1.2 Divisor

The divisor is used to determine the baud rate to be used and it can be set using the Divisor LSB Register (DLL), Divisor MSB Register (DLM), and Divisor Fractional Register (DLD) at addresses 0x00, 0x08, and 0x10 respectively. To access these registers, LCR[7] must be set to 1. Each component of the divisor is calculated using the equations found in section 1.8 of the datasheet. To set the baud rate to 4800 baud with a 4 MHz crystal and a 16X sampling clock, the divisor must equal 52.

4.4.1.3 FIFO Control Register (FCR)

The FIFO Control Register is used to enable and clear the FIFO. This register can be written to at address 0x10 when LCR is not equal to 0xBF and EFR[4] equals 0 (default), but cannot be read from. To clear and enable the FIFOs write 0x07 to this register.

4.4.1.4 Line Status Register (LSR)

This register is used to detect errors in data transfer. It is a read-only register at address 0x28 when LCR is not equal to 0xBF. The bits for this register are shown in Table 1.

Table 1: XR20M1280 LSR Register Map

Register Name	Read/ Write	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
LSR	RD	RXFIFIO Global Error	THR & TSR Empty	THR Empty	RX Break	RX Framing Error	RX Parity Error	RX Over-Run Error	RX Data Ready

4.4.1.5 Receive Holding Register (RHR)

RHR is a read-only value with address 0x00 when LCR[7] equals zero. This register holds one byte from the FIFO and after being read immediately shifts in the next byte to be read.

4.4.2 Connecting the GPS Interface Module

The board is connected to the GPS device through a 4 pin header (not all of the GPS pins are used). Because the output of the GPS is -5V to +5V, we clamp the signal between 0 and 3.3V using two Schottky diodes. The signal is also inverted with a Schmitt triggered NAND gate inverter before being sent to the XR20M1280.

Another 4 pin header connects the board to the I2C bus and power supply via the SID.

4.5 System Interface Daughterboard

The SID is the primary interface between the RPi and the rest of the sensors, devices, and components of the MSAv4 (See Figure 4 on page 22). The SID performs two main functions:

1. Supply regulated power to the RPi, onboard sensors, external devices and guest sensors.
2. Provide a data link from onboard sensors and guest packages to the RPi for data collection, processing, and storage.

4.5.1 Connecting the SID to the RPi

The SID is designed to connect directly to the RPi via a 2x16 row of GPIO header pins located on the top of the RPi board. The profile of the board is designed to cover the top of RPi with about 1/4" overhang around the edges. Figure 9 below shows how the SID mounts to the RPi.

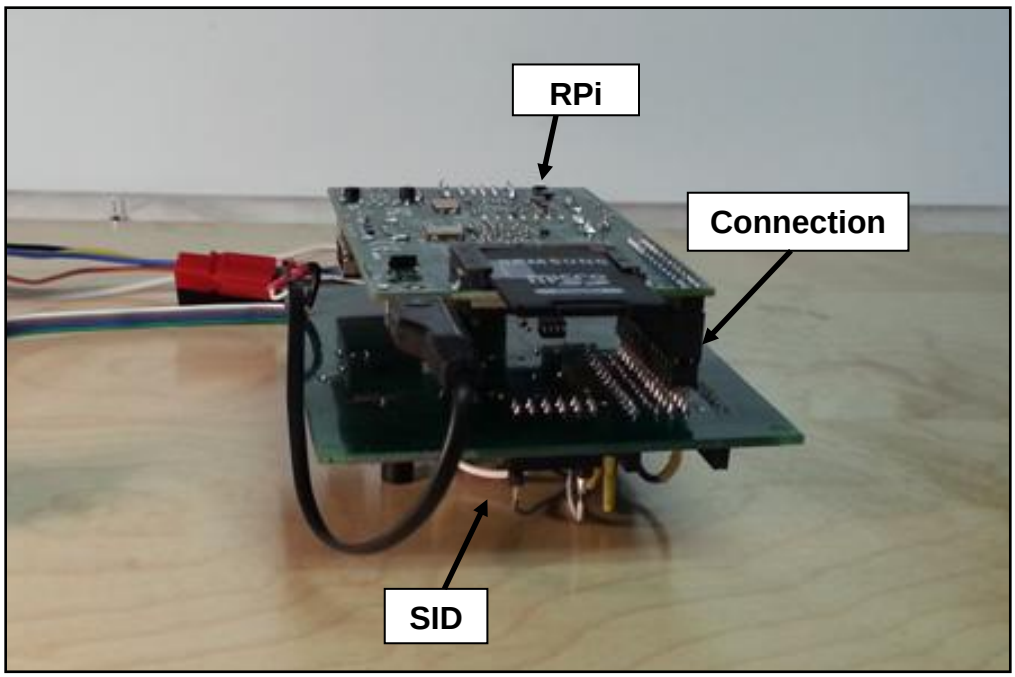

Figure 9: SID Mounted to RPi (Shown up-side down for visibility)

An additional header located in the center of the RPi is used for added support, though none of the pins in the connections are used. This mechanical connection provides for better contact to the pins and reduces noise and interference.

4.5.2 Power Regulation

The MSAv4 is powered by a 7.4V 1100mAh LiPO battery. The SID is equipped with two separate voltage regulators for supplying power to the RPi, sensors, and guest packages. The SID features two 750mA linear voltage regulators (MIC2937A / 5V / 3.3V) with reversed battery protection. The RPi is the only standard 5V device in the MSAv4 system. It runs on 5V @ 180mA (while running the MSAv4 flight program); 560mA remain for other 5V devices. The 3.3V sensors and devices on the MSAv4 consume a total max current of 14.4mA (See Table 2), which leaves 736mA for other 3.3V devices.

Table 2: Supply Current for Standard Equipped 3.3V Devices

Device	Part Number	Units	Supply Current	Total Current
Instrumentation Amplifier	MAX 4208	1	2.3mA	2.3mA
Magnetometer	HMC 5883L	1	0.1mA	0.1mA
Real-Time Clock	DS 3231M	1	0.65mA	0.65mA
Accel./Gyro.	MPU 6000	1	3.9mA	3.9mA
Pressure Sensor	MPX2102A	1	6mA	6mA
Humidity Sensor	HIH 5030	2	0.5mA	1mA
Temperature Sensor	TMP 112	2	0.01mA	0.01mA
16-bit ADC	LTC 2495	1	0.3mA	0.3mA
3.3 Voltage Regulator	MIC 2937A / 3.3V	1	0.16 mA	0.16 mA
Total				14.42 mA

Power is made available to external guest packages via three supply headers mounted to the SID; one supply header for each rail (3.3V, 5.0V, and Ground). See Figure 10 for the location of these headers.

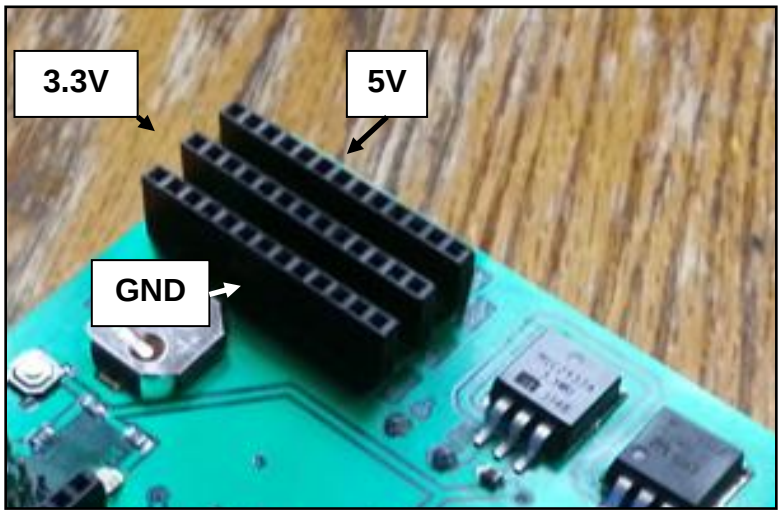

Figure 10: External Power Supply Rails

4.5.3 I2C Bus

The SID makes use of the GPIO pins 3 and 5, which are SDA and SCL respectively, of the RPi I2C bus. The I2C bus is held at a constant 3.3V potential by two 1.5KΩ pull-up resistors. As per the I2C standard, they are pulled down when the master or slaves are using the bus. A 2x6 female header allows six more external devices to be connected to the I2C bus on the SID. All of the onboard I2C devices are connected directly to the bus. Devices have been selected that are capable of (I2C High Speed) 400 KHz operation. It is recommended that any new devices incorporated into the MSAv4 system also adhere to this operation speed. See Figure 11 for the location of the I2C headers.

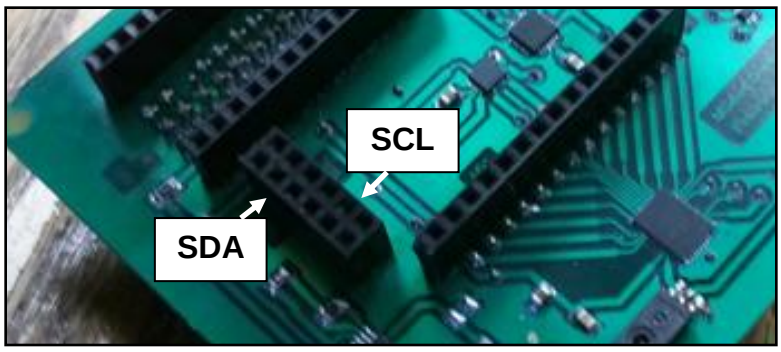

Figure 11: External I2C Bus Connections

4.5.4 Analog to Digital Converter and External Sensors

The SID is equipped with a 16-bit 16 channel ADC that is capable of handling 8 differential inputs or 16 single ended inputs. It has a programmable gain from 1 to 256. It contains an onboard temperature sensor that is used to compensate the onboard humidity sensor that is mounted adjacent to the ADC. The pressure sensor and humidity sensors are linked directly to the ADC, leaving 14 available channels for guest sensors. A 14 pin female header provides a direct link to the ADC. See Figure 12 below for the location and orientation of the ADC input header.

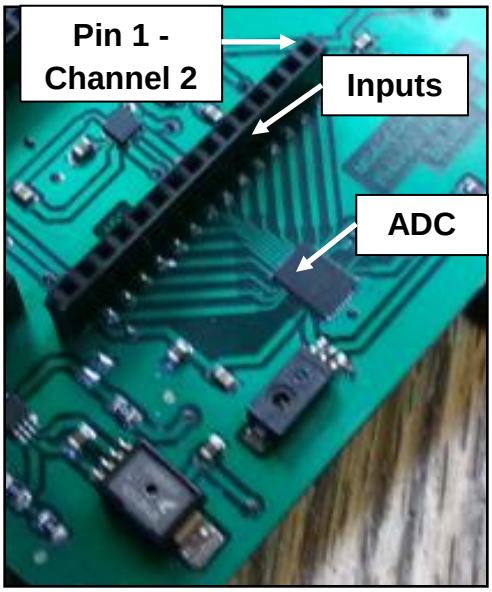

Figure 12: ADC Circuit and External Connections

The first pin in the header is channel 2 of the ADC. The channel numbers increase consecutively along with pin number from channel 2 at pin 1 to channel 16 at pin 14. The reference voltage of the ADC is set to 3.3V, this means that the maximum allowable input is limited 1.65V (One half V_{REF}). The voltage step-size of the 16-bit count value from an ADC read is 0.000025177(Volts / ADC Count). This provides adequate resolution for the sensor values chosen for the MSAv4. The design of new

guest packages must factor in the abovementioned maximum signal input and output step-size.

4.5.5 System Interface Daughterboard Sensors and Devices

This section provides a brief description of the standard suite of sensors built onto the SID.

4.5.5.1 TMP 112

The TMP 112 is a small, low power temperature sensor with built in I2C functionality. It is designed to operate at temperatures between -40ºC and +125 ºC. The TMP 112 has an internal 12-bit ADC and performs conversions continuously. There are two TMP 112 sensors used on the MSAv4; one is mounted directly to the SID. The other is mounted to the external sensor board and is positioned to take temperature readings outside of the MSAv4 flight box. The onboard sensor is positioned near the two voltage regulator and is intended to monitor board temperature. This is for informational purposes, only. This system is not designed to initiate shutdown if the board temperature exceeds a particular value. However, it is possible to add this functionality if necessary. Because there are two of the same chip used with the MSAv4, it is necessary to give them two different addresses. The slave address prefix for the TMP 112 is a hex value 0x4X. Leaving the AD0 pin floating or connected to low produces the slave address 0x48; connecting AD0 to high gives the slave address 0x49. The onboard temperature sensor is addressed 0x48 and the external sensor 0x49. See the schematics on pages 81 and 83 for more detail.

4.5.5.2 HMC 5883L

The HMC 5883L is a 3-axis magnetometer. It has an internal 12-bit ADC with a field resolution of ±8 Gauss. The HMC 5883L can operation in temperatures between -30ºC and +85 ºC; because it is enclosed in the insulated flight box it will rarely see temperatures below 0 ºC. It is capable of communicating on the I2C bus; with a slave address of 0x1E. The datasheet specifies that traces and metal components should not be places directly below the magnetometer on a double sided-board. A sample layout is provided in the datasheet.

4.5.5.3 MPU 6000

The MPU 6000 is a combination accelerometer and gyroscope in one package. It comes equipped with three 16-bit ADC for digitizing the gyroscopes and three more 16-bit ADC for digitizing the accelerometers. It stores data samples in a 1024 byte FIFO. The accelerometer portion has a programmable full-range scale in modes of 2g, 4g, 8g, and 16g. The gyroscope is also programmable in modes of ±250º/s, ±500º/s, ±1000º/s, and ±2000º/s. Chip select is wired high to enable this chip. AD0 pin is wired low to set the slave address to 0x69. This chip communicates via the I2C bus. This chips comes equipped with internal crystal oscillators; the datasheet suggests using one of the

gyroscope reference clocks for more stable performance. The MPU 6000 can operation in temperatures between -40ºC and +85 ºC.

4.5.5.3 LTC2495

The LTC2495 is a 16-bit, 16 channel ADC. The LTC2495 is capable of communication on the I2C bus; with a slave address of 0x14. ADC channels can be configured as either single-ended or differential; with a programmable gain from 1 to 256. This chip is the primary link between analog input signals and the MSAv4. The reference voltage (V_{REF}) is set to 3.3V; therefore the ADC input voltage is limited to 1.65 V ($V_{input} = V_{REF}/2$). All devices using the ADC directly must be limited between 0V and 1.65V. The step size of this ADC is 25.177µV ($V_{step} = 1.65V / 2^{16}$). The LTC2495 can operation in temperatures between -40ºC and +85 ºC.

4.5.5.4 HIH5030

The HIH5030 is a 3.3V analog humidity sensor with a single-ended output. It is designed to read a range of humidity from 0%RH to 100%RH. It is capable of operating in temperatures ranging from -40ºC and +85 ºC. It has a voltage divided output to ensure it doesn't exceed the 1.65V max allowable input to the ADC. The Internal sensor is connected to channel 1. The external sensor is connected to channel 3. Both channels are configured with a gain of 1.

4.5.5.5 MPX2102A

The MPX2102A is a differential type analog pressure sensor that is optimized for 16V, but had to be designed to be used with 3.3V; This sensor has been used by HARBOR for many years and the previous circuit was designed by Rob Eckel. This circuit is the same circuit used on the previous version MSAv3. It has since been discovered that this sensor is not linear in the range of 0 torr to 150 torr (altitudes from 0 feet to 40,000 feet). It was necessary to write code capable distinguishing between the linear and non-linear regions of the sensor. It is possible to reconstruct pressure data from the sensor at pressures below 150 torr. The MPX2102A is designed to operate at temperatures between -40ºC and +125 ºC. The effects of temperature on the full scale span of the sensor or very small, according to the datasheet.

4.5.5.6 DS3231M

The DS3231M is a 3.3V I2C RTC (Real Time Clock) with a back-up lithium battery (with a life of approximately 5 years). This device is programmed with the current date and provides a time stamp for sensor readings. It also provides the current date which is printed in the header of each data file. The DS3231M RTC is mounted on the bottom of the SID. The version used on the MSAv4 comes with its own internal crystal.

4.5.5.7 MAX 4208

The MAX 4208 is a 3.3 V instrumentation amplifier optimized for the pressure sensor circuit. The gain is set to 100 v/v. The output of this amplifier is fed to the ADC for further processing.

4.5.6 Special Function Circuits

The SID has three special function circuits:

1. Shutdown Button
2. Battery Voltage Monitoring Circuit.
3. Status LED

The shutdown button and Status LED are located next to each other on the board (See Figure 13). The Status LED provides some indication of which mode the MSAv4 is operating in:

1. Stand-By Mode
2. Data Logging Mode
3. Power Down Mode

Upon initial start-up, the MSAv4 is put into stand-by mode and the status light flashes in quick 50 ms bursts twice per second. This indicates that the MSAv4 is waiting for the mission pin to be pulled. After the pin is pulled , the USB, Ethernet port, and HDMI ports are disabled to save power, and the MSAv4 enters data logging mode. This is important to note because it will be impossible to SSH into the MSAv4 while it is running in data logging mode. If the pull-pin is not inserted, the MSAv4 will move directly into data logging mode. Also during this time the status light flashes on and off for 1 second intervals.

The shutdown button is continuously monitored by the MSAv4 during the data logging process. When pressed, the MSAv4 is put into power down mode and a safe shutdown is initiated. This mode closes all open files to save the data from corruption. It also powers down the RPi.

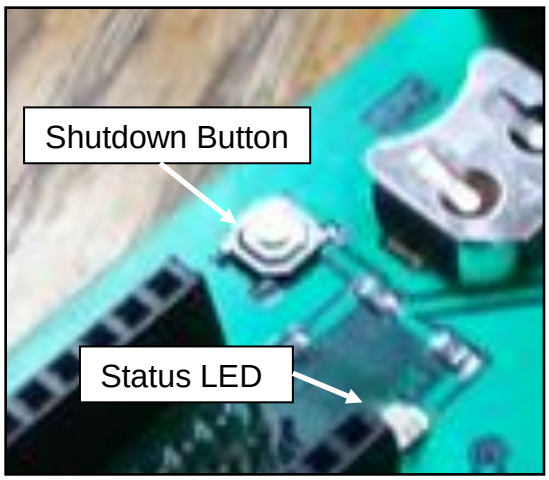

Figure 13: SID Pushbutton and LED

The battery voltage monitoring circuit is a simple voltage divider used to scale the battery voltage down to a level acceptable by the ADC. The MSAv4 monitors this value, accounting for the scaling factor, and initiates a safe shutdown when the battery per-cell voltage drops below a preset threshold. This system is described in more detail on page 58.

4.6 MSAv4 Internal Enclosure

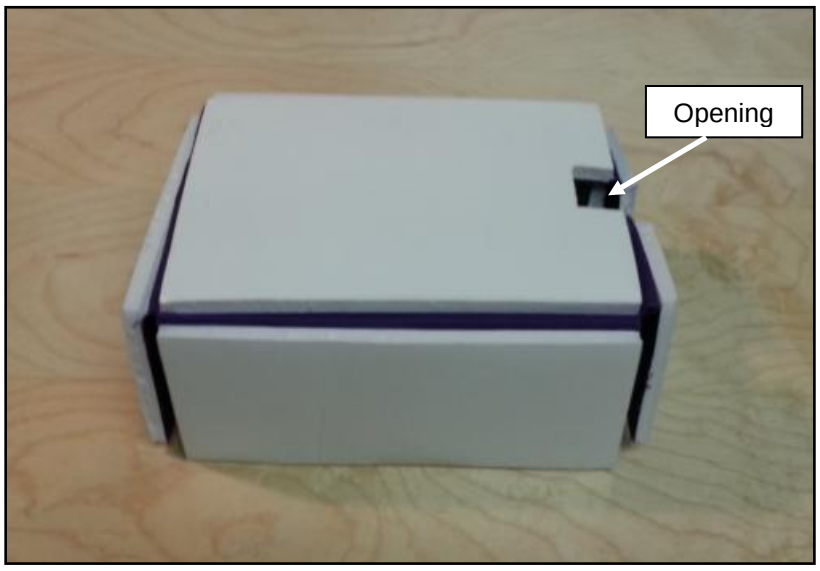

Figure 14: MSAv4 Internal Enclosure (Outside View)

The internal enclosure protects the RPi and SID from excessive vibration, shock, and allows for easy insertion into the MSAv4 Flight Box. The internal enclosure is constructed with white Elmer's Glue and made entirely of 3/16" thick white foam board covered fabric to add strength.

Table 3: MSAv4 Internal Enclosure Parts and Dimensions

Piece	Dimensions	Quantity
Long Box Side	10" x 1 11/16"	2
Short Box Side	3 7/8" x 1 11/16"	2
Box Bottom	3.5" x 10"	1
Long Lid Side	10 5/8" x 1 11/16"	2
Short Lid Side	3 7/8" x 1 11/16"	2
Lid Top	3 5/8" x 10 5/8"	1
Exterior Fabric	14 1/2" x 8 1/2"	2
Velcro (hook and loop) pieces	1/4" x 1/4"	6
5 minute epoxy	N/A	1 tube
White Elmer's Glue	N/A	1 bottle

The bottom box of the enclosure is constructed of five pieces of 3/16" foam board; two Long Sides, two Short Sides, one Box Bottom, and one piece of Exterior Fabric. The four Box Sides are glued lengthwise along the Box Bottom. The Lid is held into place by six 1/4" by 1/4" pieces of Velcro (hook and loop). The internal enclosure features a 1" x 1" opening along the top of the box to allow for the MSAv4 wire harness (See Figure 14Figure 15).

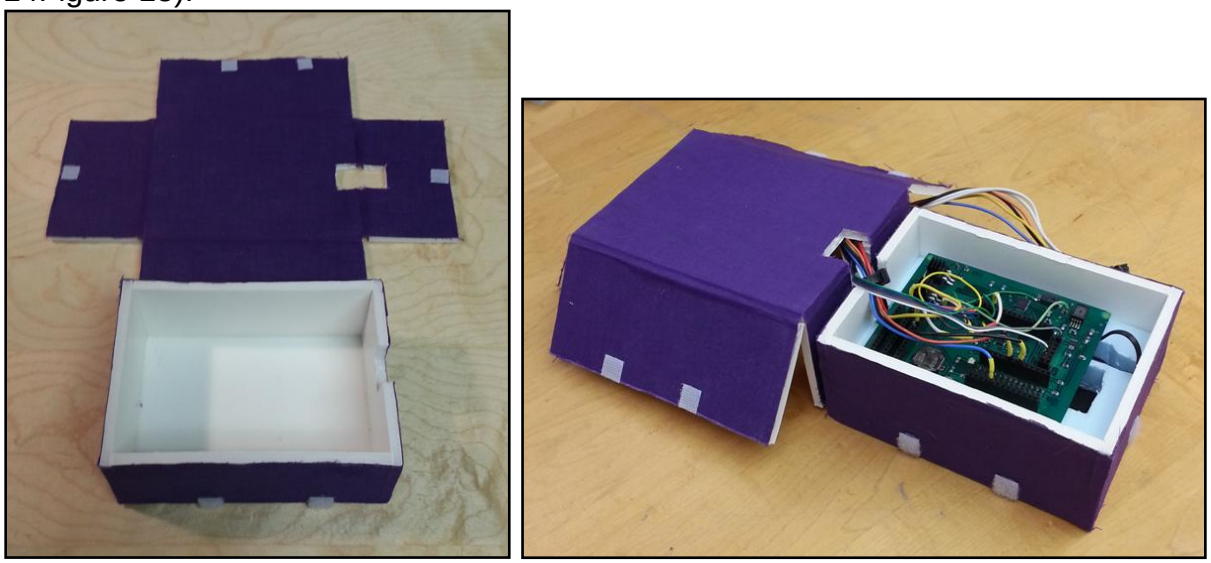

Figure 15: MSAv4 Internal Enclosure (Inside View)

A high density foam with a stenciled cutout to accept the RPi is inserted into the internal enclosure to prevent side to side motion. A piece of foam presses against the top edges of the SID to prevent vertical motion (See Figure 15).

There are plans to line the interior with aluminum tape to create an RFI shield around the RPi and SID; however, no RFI was detected during flight tests.

4.7 MSAv4 Flight Box

The MSAv4 Flight Box is a 7.5" x 8.5" x 8" Styrofoam enclosure that insulates the batteries and electronics from the external environment, and provides protection against the impact of landing and colliding with other instruments after the balloon bursts. Styrofoam is inexpensive and easy to work with. Low density foam is used to fills gaps to prevent internal components from shifting around during the flight; this provides additional insulation as well. Sensors and controls are embedded into the wall of the Styrofoam enclosure to gain access to the external environment. See Figure 16 below.

Figure 16: MSAv4 Flight Box (Front View)

The internal Enclosure is designed to fit inside of the Flight box and leave enough room for batteries, a pump, and other parts for the future particle sensor circuit; including a mass air flow sensor. Compartments are also cut out of the lid to house small PCBs for controlling external sensors; such as the particle sensor and CO2 gas sensor (Not Shown). See Figure 17 for a detailed interior view of the MSAv4 Flight Box.

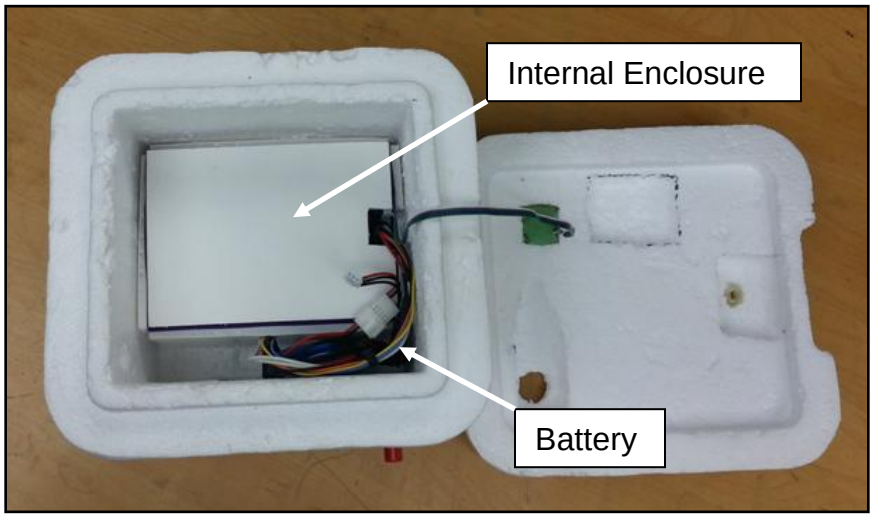

Figure 17: MSAv4 Flight Box (Top View)

During flight tests, the MSAv4 was flown with a pump and particle sensor. The flight box was easily modified to incorporate the new components. Small holes were created for the inlet and exhaust tubes to allow the pump to circulate air through the particle sensor (See Figure 18 below).

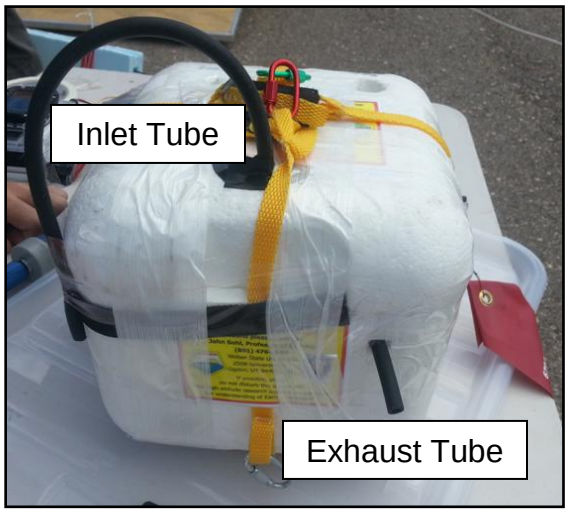

Figure 18: MSAv4 Flight Box with Particle Sensor

The pump, extra battery with voltage regulator, PCB with controllers, and the particle sensor were positioned in the open spaces of the Flight Box. The addition took only a few hours and held up well during test flights. Figure 19 below illustrates where the internal components were positioned for the particle sensor and related components. Duck tape is used to hold sensors and PCBs in place; this is because the tape is light weight and water proof. It provides a water resistant seal in the event of a water landing (Which happened during the 2 June, 2013 test flight; no components were damaged).

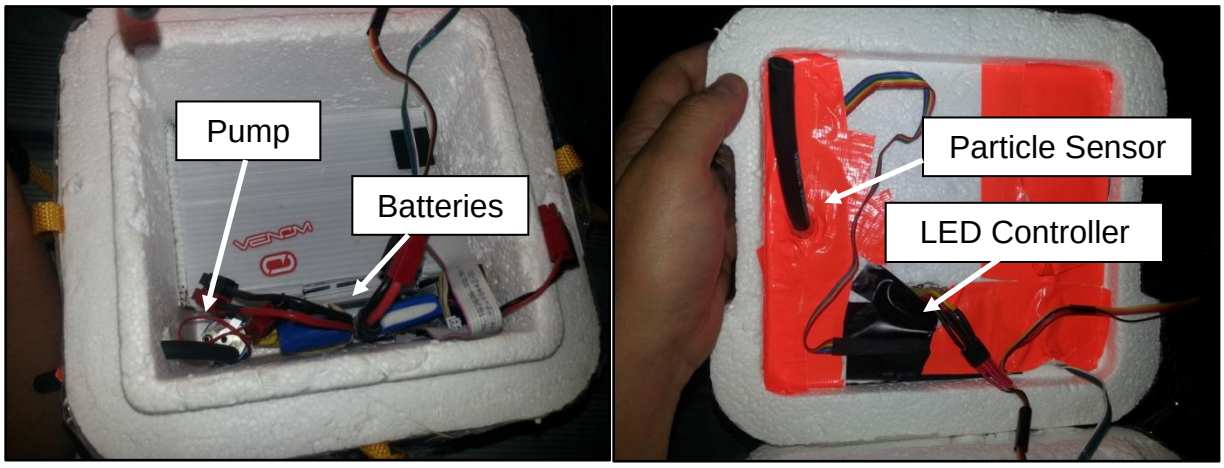

Figure 19: MSAv4 Flight Box with Particle Sensor System

4.8 Interface Software

The Multi-Sensor Array software is written completely in C, except for a minor Linux script for initiating the main program. The BCM2835 C library (McCauley, 2013) is used to interface with the I2C bus, UART pins, and the rest of the GPIOs on the Raspberry Pi. The RPi has two internal I2C busses; labeled BSC0 and BSC1. This project uses the RPi (version 1 - Model B) which defaults to the BSC0 bus. The C library had to be modified to use BSC0; since it was originally written for the RPi (version 2) which uses the BSC1 bus(Arjan, 2012). However, after the release of the BCM2835 C library (version 1.32) it is now possible to open the source code for the library and uncomment a single line so that it will function for a RPi (version1) (McCauley, 2013). The following snippet of code shows how the library was installed on the RPi:

```
tar zxvf bcm2835-1.32.tar.gz
cd bcm2835-1.32
./configure

********************************************************************************
If there is an issue with the last step it is because the system clock on the RPi is
not accurate and it appears that the new file was created before it was installed.
Try:

sudo find -exec touch \{\} \;

********************************************************************************

make
sudo make check
sudo make install
```

The note about the system clock was provided by Soybomb(2002).

POSIX threads are used to allow accurate timing of sensor reads. Flight dynamics data are read at a rate of 100 milliseconds per read. The ADC data are read at a rate which allows each active channel to be set twice, a 200 millisecond conversion rates with one discarded reading, and a final reading. The total time to read one channel is approximately 600 milliseconds. As the number of channels that made active is increased, the longer it takes to read the ADC data. Sections 4.8.2.2 Timing Thread and 4.8.2.3 Scheduling Thread go into more detail.

4.8.1 Linux Scripts

To get the MSAv4 program to automatically initiate the flight program when powered up, it is necessary to setup a short script file. Based on code found online(O'Hanlon, 2012), the following lines of code describe how to create the script that starts the MSAv4 code upon power up. First, the file must be created:

```
sudo nano /etc/init.d/run_msa
```

This creates the necessary file in the initialization folder. Then the following is written into the file to call the MSAv4 executable upon startup:

```
#! /bin/sh
       #/home/pi/msa_v4
       ## BEGIN INIT INFO
       Provides: msa_v4
       Required-Start:
       Required-Stop:
       Default-Start:
       Default-Stop: 0 1 6
       Short-Description: Simple script to start a data logging program at boot
       Description: A simple script which will start / stop a data logging program at
boot / shutdown.
       ## END INIT INFO

       # If you want a command to always run, put it here

       # Carry out specific instructions when asked to by the system
       case "$1" in
       start)
       echo"Starting msa_v4"
       #run applications you want to start
       sudo /home/pi/msa_v4
       ;;
       stop)
       echo"Stopping msa_v4"
       #killall msa_v4
       ;;
       *)
       echo"Usage:/home/pi/msa_v4 {start/stop}"
       exit 1
       ;;
       esac

       exit
```

After saving and closing the file, it is necessary to make it executable every time the RPi is powered up. The following lines of code show how this is done:

```
sudo chmod 755 /etc/init.d/run_msa
sudo update-rc.d run_msa defaults
```

Rebooting the RPi verifies that this method works. There may be instances when it is necessary to start up the run_msa script manually or to prevent it from starting up automatically. The following lines show how this is accomplished:

Manually starting the program:

```
sudo /etc/init.d/run_msa start
```

Manually stopping the program:

```
sudo /etc/init.d/run_msa stop
```

Removing the script from automatic startup:

```
sudo update-rc.d -f run_msa remove
```

4.8.2 MSAv4 Operational Flight Program

The MSAv4 operational flight program is broken into three main threads:

1. Main
2. Sample rate timing
3. Scheduling data reads and saving

The operational flight program also includes the functions necessary for reading each sensor and module.

4.8.2.1 Main Thread

The main thread performs the following functions:

- Creates POSIX Threads for the timer and scheduler
- Initiates the I2C bus
- Sets up GPIOs
- Creates .CSV files for flight dynamics, environmental, and GPS data
- Waits for mission pin to be pulled
- Disables USB, Ethernet, and HDMI ports to save power
- Main Loop
    - Monitors the push button
    - Advances and track the mission timer
    - Flashes the status light at 1 second intervals to indicate recording
    - Monitors the battery voltage
    - Initiates shutdown
- Closes files
- Closes I2C bus
- Closes GPIOs

The main loop monitors the push button, mission timer, and battery voltage; it also flashes the status led to let the user know that the MSAv4 is in data logging mode. When any of the monitored systems throw a flag, the main loop initiates a safe shutdown to protect flight and the LiPO battery. Figure 20 shows a flowchart for the main loop.

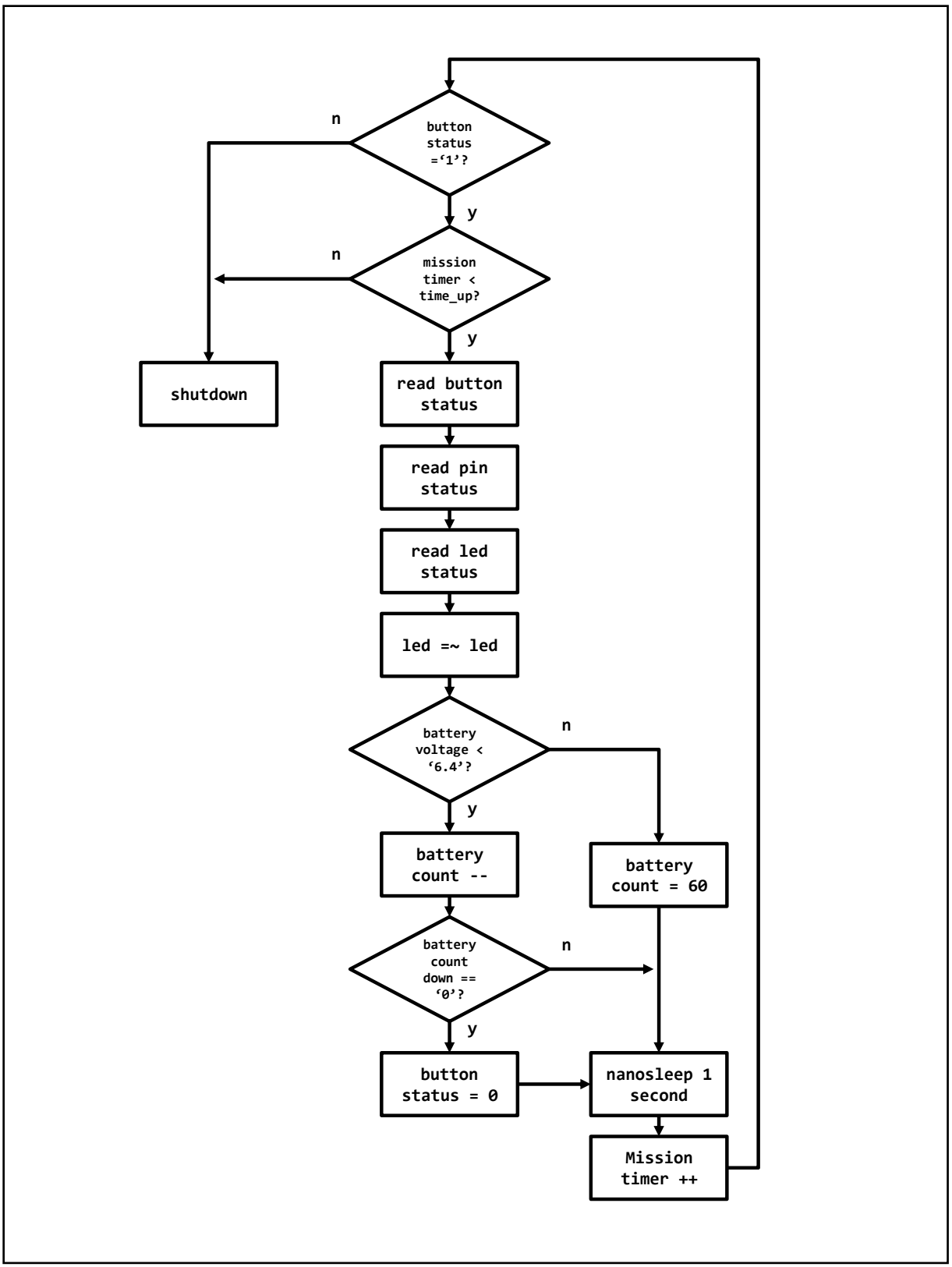

Figure 20: Main Loop Flowchart

4.8.2.2 Timing Thread

The timing thread is designed to post a semaphore at 100 millisecond intervals to set the base data sample rate to 10Hz. The following code defines the timing thread:

```
void *schedule_timer(void *arg)
{
	int milisec = 100; // length of time to sleep, in milliseconds
	struct timespec req = {0};
	req.tv_sec = 0;
	req.tv_nsec = milisec * 1000000L; // set delay to 100 milliseconds

	while(1)
	{
		//nanosleep(&req, (struct timespec *)NULL); // sleep 100 milliseconds
		nanosleep(&req, NULL);
		sem_post(&sem); // make semaphore available
	} // end while loop

} // end schedule_timer
```

4.8.2.3 Scheduling Thread

The scheduling thread is where all of the data reading functions are called. This function takes the semaphore whenever it is available and begins a read cycle. This thread contains two main reading cycles:

1. flight dynamics data (accelerometer, gyroscope, and magnetometer)
2. environmental data (temperature, humidity, pressure, etc,...)

The flight dynamics data are sampled at a different rate than the environmental data due to the larger conversion time of the ADC. To implement this, it was necessary to break the read cycle of the ADC up into three different phases:

1. Set slave address
2. Clear buffer after one read
3. Read and record data

Nested switch statements are used to step through each phase at a 200ms interval for each phase. The actual conversion time for the LTC2495 ADC is typically 160ms; 200ms was chosen to simplify the process. The environmental data does not have to be sampled quickly since the balloon has a relatively slow ascent rate. **Error! Reference ource not found.** shows a top level flowchart for the scheduling thread. The switch statement blocks each denote a switch statement used to perform the same function on a different channel. These blocks can be expanded as new ADC channels are used to accommodate more sensors. See the operation flight program in the appendix.

The scheduling thread also saves the data files every 5 minutes to prevent data loss. This is done decrementing a 1 second counter for 300 seconds (5 minutes) until it

reaches zero, and then closing and reopening the files in append mode. The counter is reset to 5 minutes and the process starts over again.

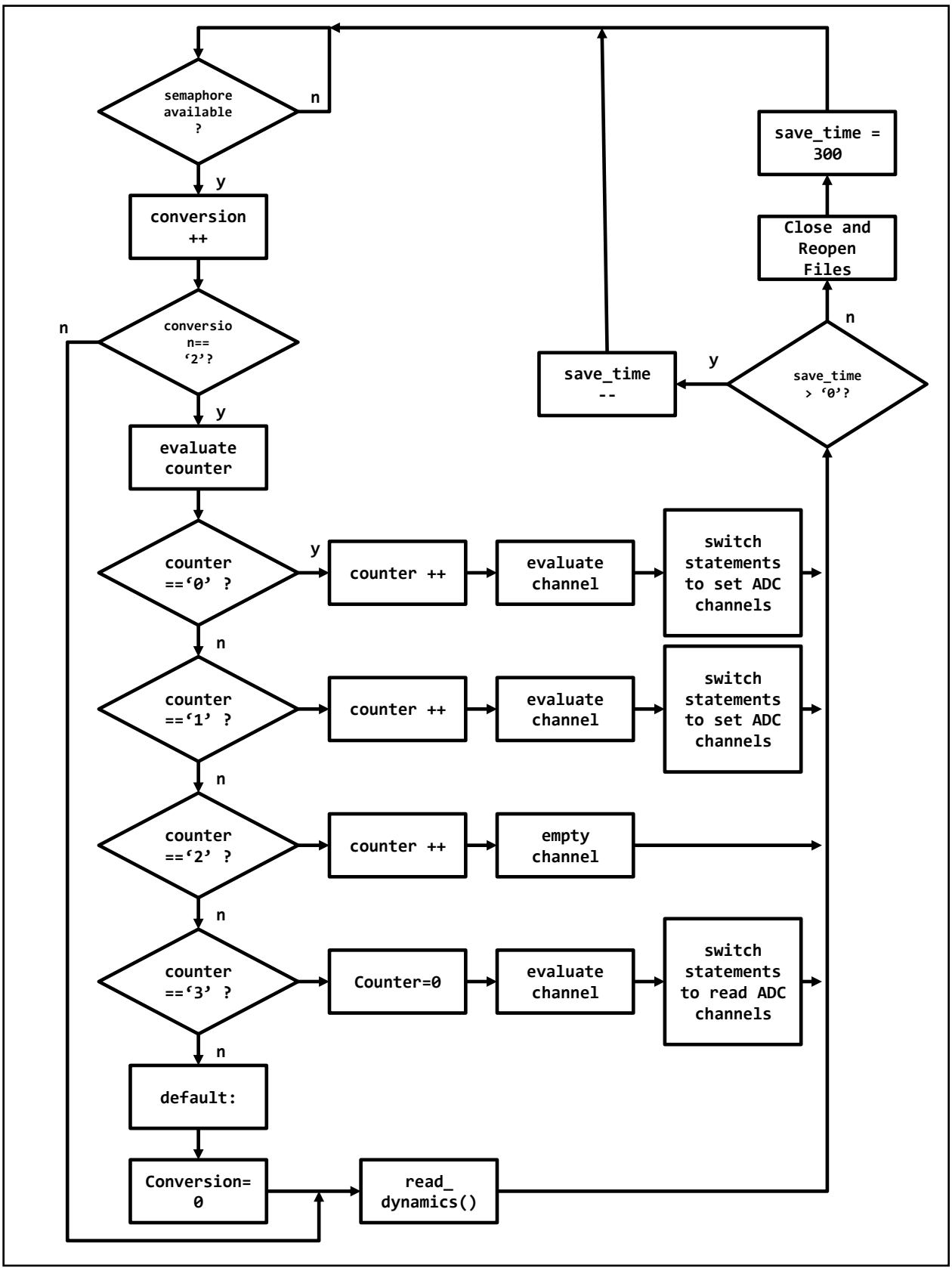

Figure 21: Scheduling Thread Flowchart (Top Level)

4.8.2.4 Data Reading Functions

There are 20 different functions defined after main. Several functions work together to perform one read; this refers to the three different phases for reading channels on the ADC. The functions are:

1. Reading the ADC temperature sensor
    a. set_channel_itemp()
    b. read_channel_itemp()
2. Reading the pressure sensor
    a. set_channel_press()
    b. read_channel_press()
3. Reading the humidity sensors (There are two sensors)
    a. set_channel_humid()
    b. set_channel_ext_humid()
    c. read_channel_humid()
    d. read_channel_ext_humid()
4. Reading the particle sensor
    a. set_channel_prtcl()
    b. read_channel_prtcl()
5. Reading the battery voltage
    a. set_channel_battery()
    b. read_channel_battery()
6. Reading the I2C temperature sensors (There are two sensors)
    a. read_tmp112()
    b. read_tmp112_ext()
7. Reading the flight dynamic sensors
    a. read_mpu6000()
    b. read_hmc5883()
    c. read_ds3231()
    d. read_dynamics(); This function calls functions a, b, and c
8. Reading the current date to place at the top of each data file
    a. get_date()
9. Discarding the first read after the channel is reset
    a. empty_current_channel()

4.8.2.4.1 Functions for Reading the LTC2495 ADC

The ADC reading functions all have the same general design. They each configure the ADC channel for their respective sensor to be either single ended or differential, to have a specific gain, and to have a specific slave address. The function reads three bytes from the data registers and processes them to find the signal voltage. After determining the signal voltage, each function then processes the value as determined by the

datasheet and calibration to produce flight data in engineering units. The raw voltage signal is not recorded, but can be extrapolated from the code if necessary.

The functions for reading the pressure sensor of the ADC will be described below as an example of how these functions are designed.

SET_CHANNEL_PRESS()
Every ADC function follows the same general pattern for setting up the ADC channel. All of the current sensors using the ADC are set as single ended inputs with a gain of 1. This was done for simplicity, and does not indicate that future sensor must be set the same. The following snippet of code describes how all of the ADC functions set the channel:

```
void set_channel_press(void)
{
	bcm2835_i2c_setSlaveAddress(LTC2495);
	bcm2835_i2c_write(chan_0, sizeof(chan_0));
// set ADC to read the pressure sensor

}// end set_channel_press
```

The variable `chan_0` is a two element array of unsigned characters holding 2-bytes of data which characterize the ADC channel. The first byte determines the channel number and input type (single ended or differential). The second byte sets the channel gain. The following line shows how the internal temperature channel was set:

```
unsigned char chan_0[]  = {0xb0, 0x80};
// 0xb080 selects ADC channel 0, single ended input, +IN, gain X 1
```

Note: It may have been more simple to create a generic function for setting the ADC since it only requires 2 bytes for every channel. The 2-byte channel array could be passed to the function to set the channel input type and gain.

EMPTY_CURRENT_CHANNEL()
This function simply reads the first data that is held in the ADC data register and ignores it. This is because the initial data has leftover information from a previous sample which may have been from another sensor. This step ensures that the ADC data are from the correct sample, channel, and sensor. The following code shows the function definition:

```
void empty_current_channel(void)
{
	unsigned char junk2[3];
	bcm2835_i2c_setSlaveAddress(LTC2495);
	bcm2835_i2c_read(junk2, sizeof(junk2)); // junk data

}// end empty_current_channel
```

All of the ADC read cycles use this same function.

READ_CHANNEL_PRESS()
This function reads three bytes from the default data register of the ADC. The first bit of the 2 byte value is unused. The remaining bits form a 15 bit binary value representing the ADC voltage. This value is converted to decimal and multiplied by the voltage step-size of the ADC, which is 0.000025177(Volts / Count LSB). The remaining value is the input signal voltage which can then be processed based on the type of sensor being read. Here is a sample of the function to read the pressure sensor:

```
void read_channel_press(FILE *gen_fp)
{
	bcm2835_i2c_setSlaveAddress(LTC2495);

	unsigned char p_buf[3];   // buffers for capturing the pressure

	// pressure sensor variables
	/*omitted variables for brevity*/
	..........
	..........

/*********************************************************************/
// read pressure sensor
/*********************************************************************/

	bcm2835_i2c_read(p_buf, sizeof(p_buf));
	// read the data register from the LTC2495
	p_buf[0] = p_buf[0] & 0x7f; // mask the first bit (unused)
	pressure = (p_buf[0] * 0x10000) + (p_buf[1] * 0x100) + (p_buf[2]);
	pressure = pressure/0x40;
	v_press = pressure * 0.000025177; // multiply by step size to get voltage

	/*omitted data processing for brevity*/
	..........
	..........

}// end read_channel_press
```

Note: All of the ADC function follow the same general format. It may have been better to simply write a generic function to read the ADC voltage and then pass the value to another function for processing.

4.8.2.4.1 Other I2C Device Functions
The other I2C devices are read using a different process than the ADC. Each device has a relatively unique procedure for setting up the sensor, reading the sensor, and

processing the data. Each function is designed according to the datasheet requirements of the related I2C device.

READ_MPU6000
This function reads the MPU6000 accelerometer/gyroscope. The MPU6000 defaults to a sleep state upon power up. It is necessary to write 0x00 to the Power Management Register 1 (0x6B) to wake it up. The Power Management Register 1 also configures the clock source, which is used to set the sample rate. The data sheet recommends using one of the gyroscope clocks as the clock reference for improved stability.

Table 4: Power Management 1 Register

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
6B	107	Device _Reset	Sleep	Cycle	-	Temp_Dis	CLKSEL[2:0]		

The possible clock source configurations are summarized in Table 5 below.

Table 5: MPU6000 Clock Source Configurations

CLKSEL	Clock Source
0	Internal 8MHz oscillator
1	PLL with X axis gyroscope reference
2	PLL with Y axis gyroscope reference
3	PLL with Z axis gyroscope reference
4	PLL with external 32.768kHz reference
5	PLL with external 19.2MHz reference
6	Reserved
7	Stops the clock and keeps the timing generator in reset

The slave address of MPU6000 used on the MSAv4 is set to 0x69.

AFS_SEL (register 0x1C) sets the full scale range for accelerometer readings. The MSAv4 sets AFS_SEL to 0 (2g) and 2 (8g) to achieve an LSB sensitivity of 16384 LSB/g and 4096 LSB/g respectively. See Table 8.

FS_SEL (register 0x1B) sets the full scale range for gyroscope readings. The MSAv4 sets FS_SEL to 0 (250 º/s) and 2 (1000 º/s) to achieve an LSB sensitivity of 131 LSB/º/s and 32.8 LSB/º/s respectively. See Table 7.

Full scale sensitivity is configured by writing values to bit-3 and bit-4 in the 0x1C (for the accelerometer) and 0x1B (for the gyroscope) registers (See pages 14-15 of the MPU6000 Register Map Document: RM-MPU-6000A-00) . The requirements document on page 124 specifies the sensitivity for each type of reading.

Table 6: MPU6000 Configuration Registers

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
1B	27	XG_ST	YG_ST	ZG_ST	FS_SEL[1:0]		-	-	
1C	28	XA_ST	YS_ST	ZA_ST	AFS_SEL[1:0]		-	-	-

Table 7: Gyroscope Full Scale Settings

FS_SEL	Full Scale Range
0	± 250 º/s
1	± 500 º/s
2	± 1000 º/s
3	± 2000 º/s

Table 8: Accelerometer Full Scale Settings

AFS_SEL	Full Scale Range
0	± 2g
1	± 4g
2	± 8g
3	± 16g

Data from gyroscope and accelerometer reads are stored in two blocks of six registers each. There are two registers to hold a 16-bit value for each axis. The accelerometer data are stored in registers from address 0x3B to 0x40. The output is a 16-bit 2's complement value. Data are read from the most recent sample with burst reads on the I2C bus.

Table 9: Accelerometer Data Registers

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
3B	59	ACCEL_XOUT[15:8]							
3C	60	ACCEL_XOUT[7:0]							
3D	61	ACCEL_YOUT[15:8]							
3E	62	ACCEL_YOUT[7:0]							
3F	63	ACCEL_ZOUT[15:8]							
40	64	ACCEL_ZOUT[7:0]							

These registers store the most recent accelerometer measurements. Accelerometer measurements are written to these registers at a sample rate defined in register 0X19, the Sample Rate Divider. For the MSAv4, the Sample Rate Divider is left at the default value.

The gyroscope data are stored in registers from address 0x43 to 0x48. As with the accelerometer data, the output is a 16-bit 2's complement value. Data are read from the most recent sample with burst reads (repeated starts with no stop) on the I2C bus.

Table 10: HMC5883 Data Output Registers

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
43	67	GYRO_XOUT[15:8]							
44	68	GYRO _XOUT[7:0]							
45	69	GYRO _YOUT[15:8]							
46	70	GYRO _YOUT[7:0]							
47	71	GYRO _ZOUT[15:8]							
48	72	GYRO _ZOUT[7:0]							

These registers store the most recent gyroscope measurements. Gyroscope measurements are written to these registers at a sample rate defined in register 0x19, the Sample Rate Divider. For the MSAv4, the Sample Rate Divider is left at the default value.

READ_HMC5883

This function reads the 3-axis magnetometer I2C chip. The first three writeable registers are for configuring the sample rate, gain, and sampling mode. The MSAv4 is designed to collect 8 averaged samples at 15 Hz. The gain is set to 5 and the chip is put into single measurement mode.

After configuring the output, a simple 6 byte burst read gathers all of the data from the same sample batch. The data registers begin just after the configuration registers at address 0x03 (See Table 11 for the complete register list). The output is a 16-bit 2's compliment value which is easily converted to a signed decimal.

Table 11: HMC5883 Register List

Address (Hex)	Name
00	Configuration Register A
01	Configuration Register B
02	Mode Register
03	Output X MSB Register
04	Output X LSB Register
05	Output Y MSB Register
06	Output Y LSB Register
07	Output Z MSB Register
08	Output Z LSB Register

READ_TMP112 and READ_TMP112_EXT
Reading the TMP112 I2C temperature sensors is an easy task. The slave address for each chip is set to 0x48 (internal temperature) and 0x49 (external temperature). No special register reading function are necessary to get data from the default data registers. The BCM2835_I2C_READ() function is used without specifying the register to read a 16 bit value. Only 12 of the 16 bits are used to hold the temperature value; thus the important 12 bits must be left justified. The resulting value is the 2's compliment of the temperature which must be converted to decimal and then multiplied by a scalar provided by the manufacturer.

READ_DS3231 and GET_DATE
These functions read the current time and date from the DS3231M Real Time Clock. Time stamps are added to the front of every data line and each file has the current date inserted at the top of the file header. There are many timekeeping registers in the DS3231, but only 6 are used for the MSAv4. The time register address are from 0x00 to 0x02. These time registers are set to output a 24 hour clock. A burst read is used to gather data from the output registers. The data values are in BCD format and must be converted to decimal for printing. This is achieved by isolating the high nibble and multiplying it by 10. This value is then added to the low nibble to get the decimal value. This conversion is done for seconds, minutes, and hours. The same method is applied to the date; which is printed in MM/DD/YY format. The date address registers are from 0x04 to 0x06. The slave address for the DS3231 is set to 0x68.

READ_XR20M1280
This section is reserved for code to read the XR20M1280 UART to I2C converter; specifically for reading the Garmin GPS 15L module.

4.8.2.5 Safe Shutdown
The MSAv4 is designed to shut down safely when one of three criteria are met:

1. Mission timer elapses
2. Battery voltage is critically low
3. MSAv4 shutdown button is pressed

Currently, the MSAv4 has a mission counter that is initialized at (14400 seconds) 4 hours. The mission time can be modified by changing the variable named `TIME_UP`. This prevents the MSAv4 from running unnecessarily after touchdown.

The MSAv4 also monitors the battery voltage to prevent damage to the LiPO batteries. The LiPO battery manufacture suggests that the per-cell voltage not drop below 3.2 Volts; dropping below 3.0 Volts can damage the battery. For the two cell MSAv4 battery, the shutdown voltage is set at 6.4 Volts (This value is divided down to meet the 1.65V maximum input requirement of the ADC). When voltages below 6.4V are detected, the

operational code sets the BATTERY_COUNT_DOWN flag and continues to watch the voltage for 60 seconds. If the voltage remains below the critical value for more than 1 minute, a safe shutdown is executed. Otherwise, the flag is reset.

The typical way to initiate a safe shutdown is to press and hold the shutdown button. The main loop constantly monitors the BUTTON_STATUS variable. If BUTTON_STATUS is low at the beginning of the main loop, a safe shutdown is initiated.

4.9 Design Concepts and Requirements not Completed for MSAv4

Development time limited the number of features that were designed into the MSAv4 during the 2013 project. Several features listed in the Requirements Document had to be postponed until a later time. These features include:

- Radio Telemetry
- External LCD Display (LEDs used instead)

The above features are considered secondary to the Concept Document provided by Dr. Sohl. However, they may still be incorporated at a later time. To ensure that the MSAv4 would be capable of at least replacing the MSAv3, the proposed new features were not completely designed; these features are:

- Temperature Adjusted Gas Sensors
- Particle sensor with flow meter
- Motion and Altitude Triggered Siren

The above features were partially developed until it was realized that time was too limited to pursue them.

A prototype version of the particle sensor was flown during flight tests. The prototype lacked a flow meter, but still had a pump and sensor. Data were collected, but cannot be used for scientific purposes because they lack calibration. Results from the particle sensor can be found in section 5.2 beginning on page 72.

A UART to I2C chip was selected and successfully tested with the GPS unit. However, the necessary PCB was not completed on time. The GPS module is a necessary component for triggering the warning siren. The MSAv4 is equipped with a siren, but lacks an altitude trigger and more time is necessary to program a motion trigger.

The MSAv4 is capable of logging CO2 gas sensor data from a CO2S-PPM (Wide Range), which has been flown with the MSAv3. But, additional time is necessary to write the code for reading the sensor and logging the data.

GPS capability is the only primary requirement not completed. Many of the components of the system were successfully demonstrated and a PCB was partially fabricated. This system must be completed at a later date.

Dr. Sohl has been informed of the current status of the MSAv4 project and has expressed satisfaction with the overall progress of the MSAv4 project.

5.0 Testing

Testing was used to validate that requirements were met. Items tested include:

- Dimensions (Weight and Size)
- Performance (Power consumption, Communication)
- Functionality (Switches, Code, Buzzer, Lights, etc,...)
- Hardness (Environmental, Shock, Vib, etc,...)

Testing was conducted in two primary settings; in the lab and on test flights.

5.1 Lab Testing

5.1.1 Dimension Requirement Testing

The design document calls for two main dimensions; the weight of the MSAv4 system without gas or particle sensing components, and the size of the external enclosure. The given value of the current MSAv3 also neglects the weight of the GPS module.

5.1.1.1 Weight

The core MSAv4 components are lighter than those of the MSAv3. The MSAv4 mass requirement of less than 0.9 lbs (408 grams) has been met at 406.9 grams. The MSAv3 and MSAv4 were measured and compared side by side; the results of this measurement are summarized in Table 12: Mass Comparison of MSAv3 vs. MSAv4Table 12 below.

Table 12: Mass Comparison of MSAv3 vs. MSAv4

Component	MSAv3 - grams	MSAv4 - grams	% Difference
Main Board	44.6	41.3	-7.99
Sensor Board(s)	51.7	44.1	-17.2
Primary Battery	68.0	71.4	4.76
Enclosure	118.4	84.8	-39.6
Flight Box	125.3	125.3	0
Controls	0.00	40.0	~
Total	408.0	406.9	-0.27

5.1.1.2 Size

The height, length, and width of the MSAv4 flight box are 7.5" x 8.5" x 8" respectively. The size dimension requirement of less than 12" x 12" x 12" has been met.

5.1.2 Performance Requirement Testing

The performance requirement that were directly tested in the lab were power consumption, battery life, and communication.

5.1.2.1 Power Consumption and Battery Life

The MSAv4 has been tested and is capable of lasting for the desired duration of over 4 hours. The MSAv4 must continue to operate throughout the mission. Missions can last anywhere from 2 to 6 hours. Typically, the ascent and descent take only 2 hours to complete; the rest of the time is set-up and recovery. The target operation life on one battery charge is at least 4 hours. Direct measurement of the MSAv4 current during operation is between 180mA and 190mA. With the supplied 1100mAh battery, the expected mission life is approximately 6 hours. The MSAv4 was tested by assembling the MSAv4, connecting power, initiating the flight program, and noting the length of time before shut-down due to a critically low battery voltage event; as monitored by the ADC. The 1100mAh battery lasted 4.5 hours before the MSAv4 initiated a safe shutdown due to low battery voltage. The battery was measured at 6.4V which is the prescribed shutdown voltage. The fact that the battery only lasted 4.5 hours and that the MSAv4's current draw is known from direct measurement, suggests that the battery is not performing as expected.

The power consumption of the MSAv4 was compared to the MSAv3. The MSAv3 is very efficient compared to the MSAv4, but has fewer capabilities. The following summarizes and compares the current consumption of both the MSAv3 and the MSAv4.

Table 13: Power Consumption Comparison of MSAv4 vs. MSAv3

Mode	MSAv3 - mA	MSAv4 - mA	Avg. % Difference
Normal	75 - 100	310-390	75.0
Data Logging	75 - 100	180-190	52.7

5.1.2.2 Communication

Each communication protocol was directly tested in the lab. The I2C bus performed as expected. All devices that have been connected to the bus have been recognized and have successfully transmitted and received data. The UART channel was found to operate at TTL levels (0 - 3VDC), but the signal is inverted (a mark is low). Several baud rates ranging from 115,000 baud to 4800 baud were tested and all of them were correct. Each GPIO pin was tested in both output and input mode and was found to operate correctly.

The sample rate for the flight dynamics sensors was determined from experimental data to be 10 Hz, which was expected. The sample rate for the environmental sensors has no preset value because it changes depending on the number of channels are being read from the ADC.

The data logging program successfully reads and processes all of the installed sensors and devices.

5.1.3 Functionality Requirement Testing

The functionality of each component and system was tested and it was verify that the MSAv4 performed all required tasks and operations at the appropriate times. The following sections provide explanations of how each function was tested and the results of the test.

5.1.3.1 On/Off Switch

The on/off switch was tested to verify that power to the RPi was severed in off mode. The design of the switch is supposed to cut the ground to all of the components on the SID board; including the regulators. It was discovered that the ECIB was proving a ground path through the LED resistors and allowing the regulators to produce a voltage, though too low to turn on the RPi. However, the siren turns on when the battery is connected and the switch is off. When the switch is turned on the siren shuts off, the RPi starts up, and everything functions as expected. The on/off switch circuit should be redesigned to cut the positive side of the battery and better isolate the components connected through the ECIB.

5.1.3.2 Mission Pull-Pin

The mission pull-pin was successfully tested to verify that voltage at the monitor node, RPi pin number 15, is high(3.3V) when the pin is pulled and low(Less than 1V) when the pin is inserted. It was verified that the Pull-pin meets the functional requirement. When the pin is inserted and the MSAv4 is powered up, the system stays in stand-by mode. When the pin is pulled, the system moves into data logging mode.

5.1.3.3 Shutdown Pushbutton

The shutdown pushbutton was successfully tested to verify that voltage at the monitor node, RPi pin number 12, is high(3.3V) when the button is not pressed and low(Less than 1V) when the button is pressed. It was verified that the pushbutton meets the functional requirement. When the button is pressed during the data logging mode, the MSAv4 initiates a safe shutdown.

5.1.3.4 Mission Timer Shutdown

The mission timer was successfully tested to verify that the MSAv4 will initiate a safe shutdown after operating for 4 hours. This was done by starting the MSAv4 and noting the time. Exactly 4 hours later, the MSAv4 initiated a safe shutdown.

5.1.3.5 Low Voltage Shutdown

The low voltage monitor was successfully tested to verify that the MSAv4 will initiate a safe shutdown when the battery voltage drops to 6.4V (3.2V per cell). This was done by starting the MSAv4 and noting the battery voltage. The MSAv4 was allowed to run beyond the mission timer restriction to allow battery voltage to continue decreasing. The MSAv4 initiated a safe shutdown and the battery voltage was measured at exactly 6.4V.

5.1.3.6 Automatic Startup

The automatic startup function was successfully tested to verify that the MSAv4 will automatically initiate the operational flight program when powered up. This was done by connecting the battery and switching the ECIB to the ON position. The MSAv4 successfully entered stand-by mode; indicating that the operation flight program had been initiated.

5.1.3.7 Saves Data Every 5 Minutes

The MSAv4 was successfully tested to verify that the flight data was being saved every five minutes by the task scheduler. This was done by first starting the MSAv4 and allowing it to log data for 4 minutes and then powering it down. The data files were opened and found to be empty. Two run were conducted at 6 minutes and 11 minutes to verify that data was being save every 5 minutes; and it was.

5.1.3.8 Siren Operation

The siren circuit was tested by modifying the code to activate the siren when the MSAv4 transitioned from stand-by mode to data logging mode; as part of the acknowledgment process. The siren functioned as expected.

5.1.3.9 Status LED Operation

The status LEDs, both on the SID and on the ECIB, were successfully tested to verify that they turned on and off when expected. This was done by simply starting the operational flight program and noting the duration and timing of the LED flashes. The LEDs functioned as expected.

5.1.3.10 Pressure Sensor Function and Calibration

The MPXM2102A pressure sensor was successfully calibrated to produce reliable pressure values between 0 torr and 760 torr. The pressure sensor was tested inside the environmental chamber at the Weber State University Physics Department; this chamber is capable of producing vacuums as low as 10^{-7} torr. The MSAv4 was placed inside the chamber and the ambient room pressure was noted at 760 torr; this value was confirmed by checking with two local weather reporting agencies. The MSAv4 was started and the time noted. The environmental chamber was switched on. The pressure began to rapidly decrease.

Figure 22: HARBOR Environmental Chamber

Direct pressure readings were taken at 1 minute intervals for 48 minutes; producing 47 measurements. The voltage output from the pressure sensor was plotted against the pressure readings at the appropriate times. It was discovered that the linear operating range of the MPX2102A pressure sensor is 150 torr to 760 torr (See Figure 23). This is in agreement with the datasheet. A formula was derived for the measured curve and compared to the datasheet; they are exactly the same.

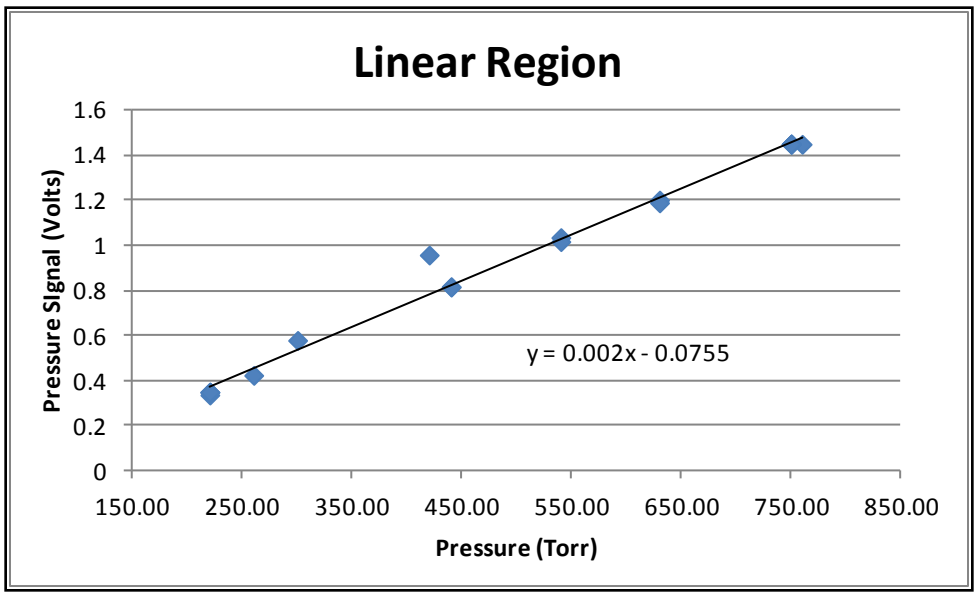

Figure 23: MPXM2102A Linear Pressure Range

For pressures below 150 torr, the MPXM2102A pressure sensor behaves logarithmically (See Figure 24). Ideally, more data points are necessary to accurately describe this curve but the vacuum chamber changes values rapidly in this region. It is recommended that an additional test be conducted to gather more data points in the transition region between 0.056 torr and 150 torr. A formula was derived to describe the curve in the non-linear region. The MSAv4 operational code was modified to use both curves to reconstruct the pressure. The linear model is used for sensor voltages above 0.319V. The non-linear model is used for sensor voltages at and below 0.319V.

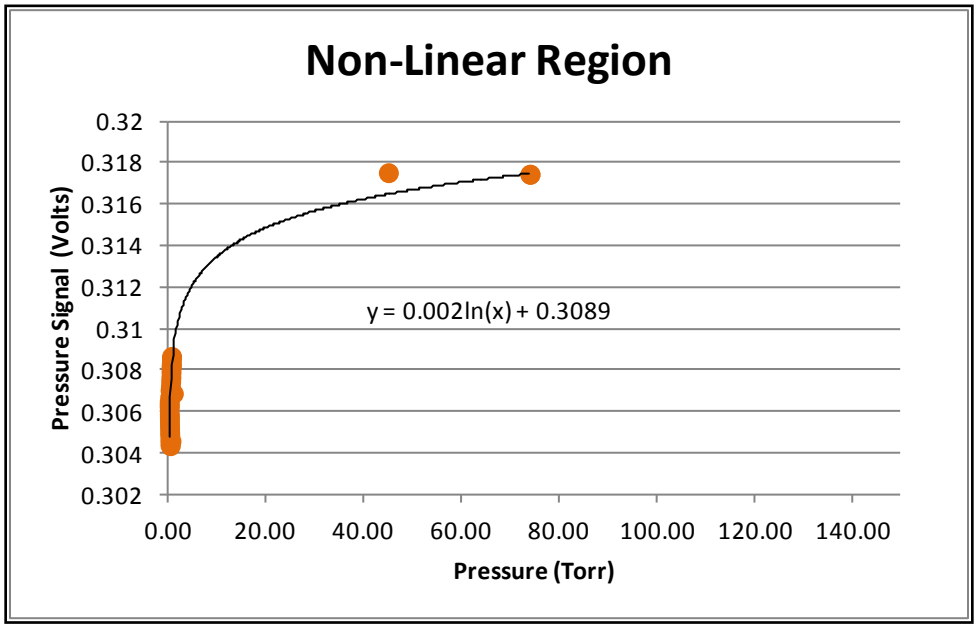

Figure 24: MPXM2102A Non-Linear Pressure Range

The reconstructed pressure data matched the actual data very well. Figure 25 shows the reconstructed data, shown as a purple line, transposed with the actual data, shown as yellow dots.

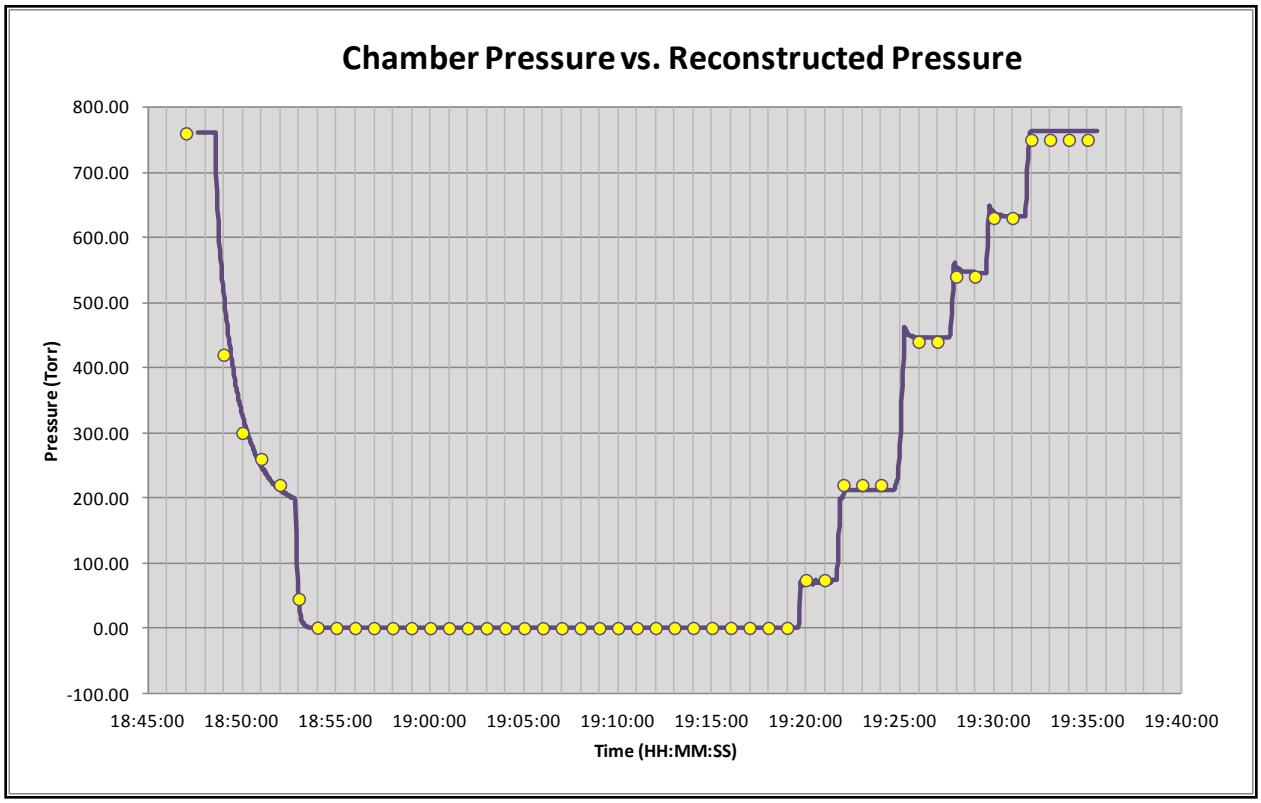

Figure 25: MPXM2102A Reconstructed Pressure

5.1.3.11 Humidity Sensor Function and Calibration

A partially successful calibration was conducted for the HIH5030 humidity sensor. The sensor was tested in three different environments; dry, average, and wet. Readings from the sensor were compared to a portable weather station used by HARBOR to calibrate the Ozonesonde. Humidity readings from the MSAv4 are in general agreement with the weather station sensor, but it was discovered that the MSAv4 sensor is more sensitive to changes in environmental humidity and probably more accurate than the portable weather station. The portable weather station reported that same humidity for several minutes when moved from a wet environment to a more dry environment. The MSAv4 sensor reports changes more rapidly. Calibration was then performed by taking the maximum voltage reading in a hot humid room with a show running, and then applying the sensor curve from the datasheet. Figure 26 shows a plot of the calibrated HIH5030 humidity and the value reported by the weather station. The region from 70-90% RH along the weather station axis denotes a time lag as the weather station slowly approached 100% RH.

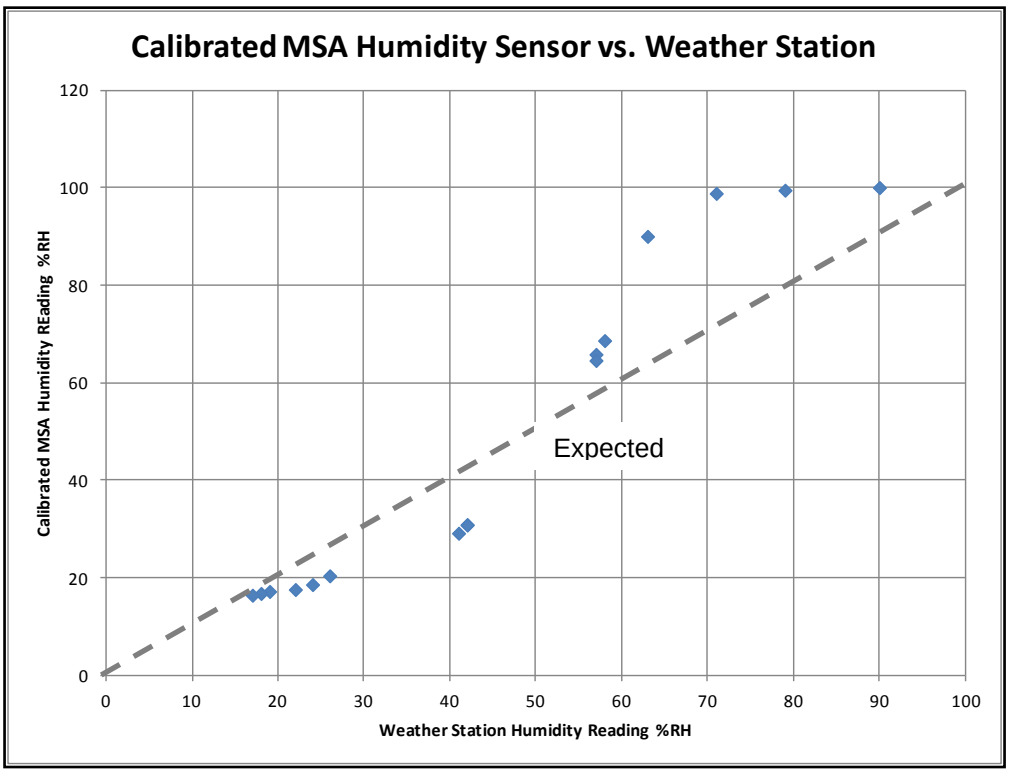

Figure 26: MSAv4 vs. Weather Station Humidity

Further work is required to reliably calibrate the HIH 5030 sensor. It is recommended that a more reliable device be used to calibrate the sensor; such as a sling psychrometer.

5.1.3.12 Temperature Sensor Function and Calibration

The TMP112 and LTC2495 temperature sensors are currently adjusted to the manufactures specifications. The onboard temperature sensors require a special calibration since they are more affected by the onboard components than by ambient room temperature. A calibration must be competed with a thermal scanner or no-touch laser thermometer. The external TMP112 is also set to manufacture specification and needs to be calibrated. Temperature readings have been compared to the weather station sensor, and they are with 0.2 ºC agreement; however, it is not known if the weather station sensor is accurate.

5.1.3.13 Magnetometer Function and Calibration

The magnetometer was successfully tested to verify that it was capable of measuring changes in a magnetic field. This was done by starting the MSAv4 and then choosing an orientation relative to magnetic north. The enclosure was then oriented on all six sides so that each side would face north for 1 minute. The magnetic axis orientation of the MSAv4 is shown in Figure 27 below.

Note: The magnetometer axis orientation does not exactly match the orientation of the accelerometer/gyroscope. However, it is possible to compensate for this in the code.

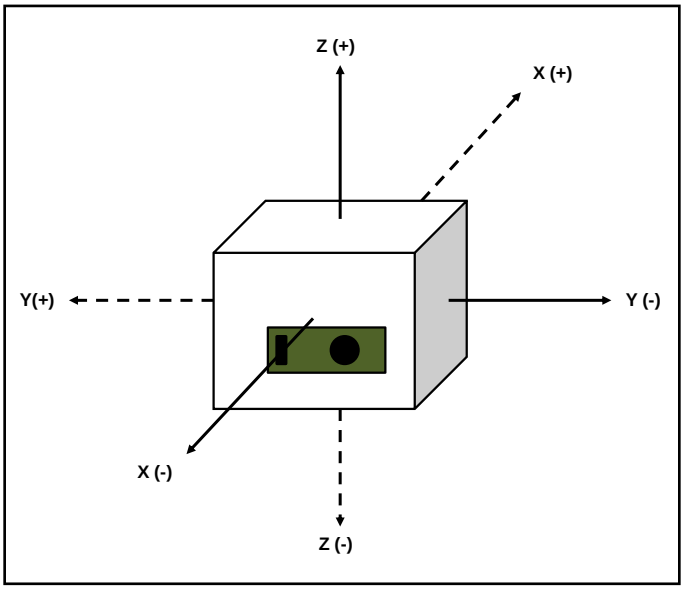

Figure 27: MSAv4 Magnetic Axis Orientation

The test indicated changes in magnetic flux with changes in orientation. The output values are stable, however it is not possible to determine which direction is north from the data. The results of the test are shown in Figure 28 below.

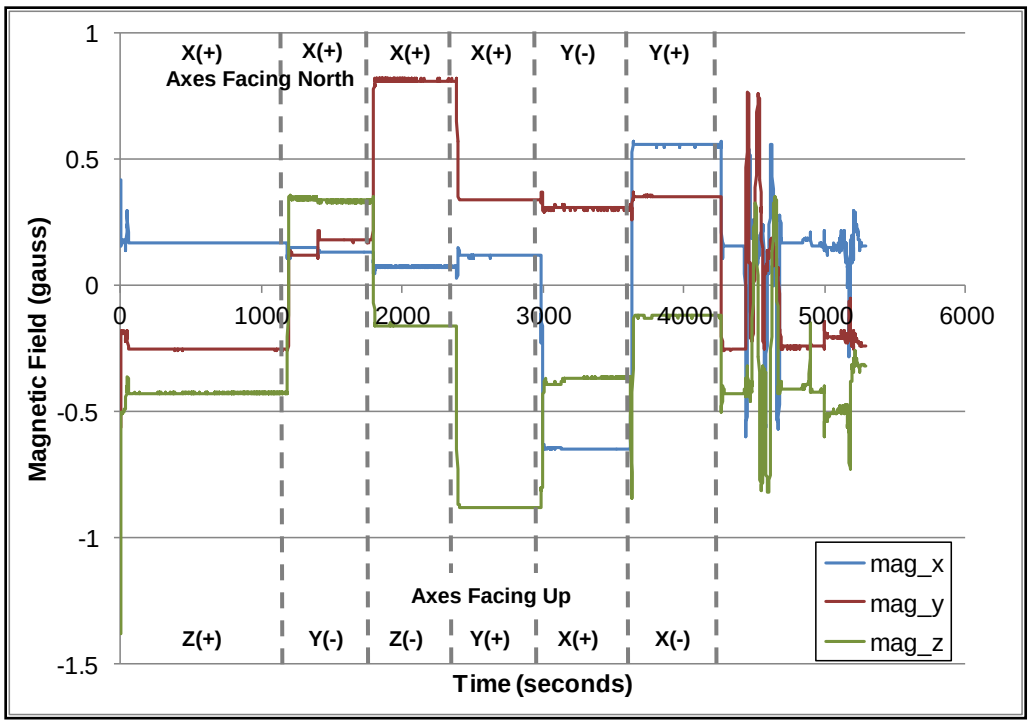

Figure 28: Magnetometer Test Results

There are six stable 1 minute intervals representing the six different orientations of the MSAv4 enclosure. There were electronic devices nearby that explain why the test results do not indicate which side is facing toward magnetic north.

A proper calibration must be conducted before magnetometer data can be used for scientific applications. A procedure for calibrating the magnetometer sensor has not been thoroughly discussed. More work remains.

5.1.3.14 Accelerometer/Gyroscope Function and Calibration

The accelerometer function was successfully tested to verify that correct readings are correlated with a particular axis. The accelerometer axis orientation is slightly different than the orientation of the magnetometer. The orientation is illustrated in Figure 29 below.

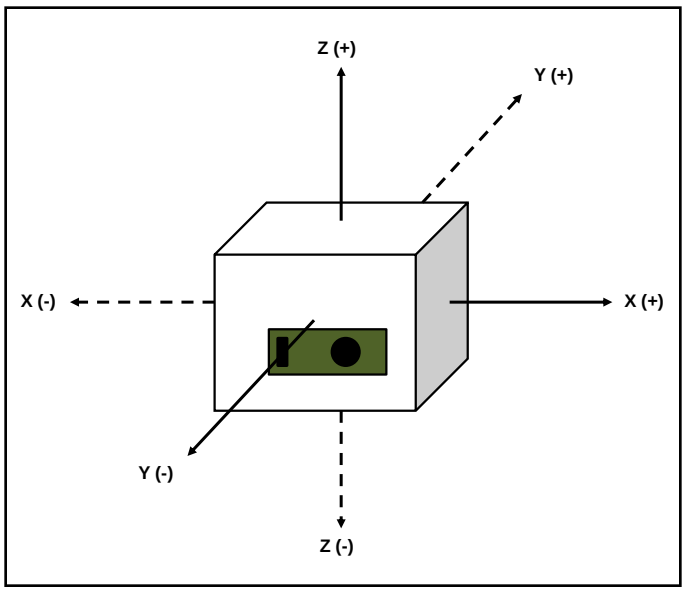

Figure 29: Accelerometer/Gyroscope Axis Orientation

Data was collected during the same trial run as the magnetometer data. The accelerometer data was easy to verify since stationary acceleration corresponds to the force of gravity at 1g. As expected, the accelerometer axis that was parallel with the force of gravity registered the appropriate positive or negative value depending on the sign of the axis. When the positive Z axis was pointed upward, a force of +1g was registered. When the negative Z axis was pointed upward, a force of -1g was registered; and likewise for all other axes. The results of the test are illustrated in Figure 30 below.

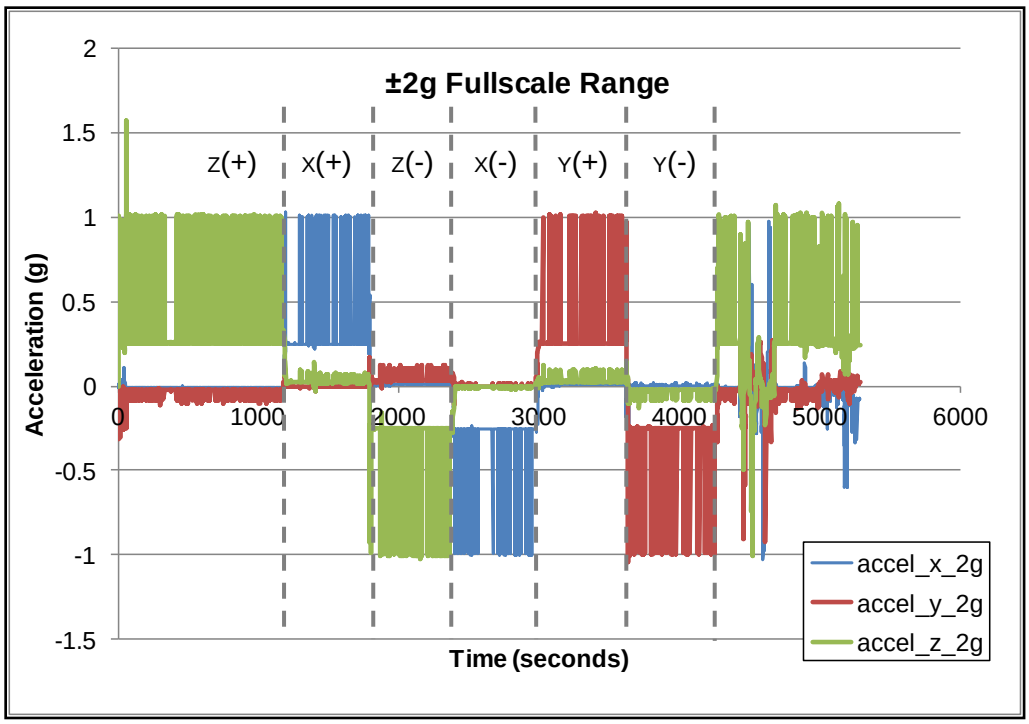

Figure 30: Accelerometer Test Results

Though gyroscopic forces are also measured by the MSAv4, the sensor was not calibrated. The procedure for calibrating the MSAv4 gyroscope entails placing the MSAv4 on a level rotating surface or disc with a known rate of rotation; keeping the rate constant. Then alternate MSAv4 orientations for all six sides. This should be done for several known rates of rotations. Then compare the rates with the readings gathered by the MSAv4. This experiment was not conducted for lack of time.

5.1.3.15 Real Time Clock Function

The RTC function was tested and verified by simply comparing the time stamp on data samples to the recorded times during calibration. The RTC is accurate to within a few seconds of real time. This can be reset at any time by the user. Additionally, the data was compared and found to be accurate as well.

5.1.5 Interface Hardware Testing

Harnesses and connections were tested during final assembly of the MSAv4. All systems are functioning as expected.

5.2 Flight Testing

Flight testing provide valuable data for evaluating the MSAv4 system. Though only a proof-of-concept prototype was flown during the flight season, several requirements were verified.

The MSAv4 functioned correctly for all flights. There were no unexpected interruptions, though the battery died on the first flight because the MSAv4 sat on the flight line for 2 hours prior to launch in data logging mode. The battery stopped approximately 2 hours into the flight; which was expected. This failure prompted a power switch to be included to the external control design so that battery life could be reserved.

There was initial concern that the MSAv4 might overheat in the low pressure environment due to a reduce ability for the MSAv4 to dissipate heat. The MSAv4 did generate a lot of heat on the flight line and the internal temperature stayed much warmer than the external temperature. However, the MSAv4 continued to function normally for five flights. The MSAv4 landed in a lake and in the desert on rough terrain. In instances where the MSAv4 hadn't already shut down due to low voltage, it still remained operational after touchdown and after burst; the two most traumatic events during the flight.

5.2.1 Particle Sensor Guest Package Testing

The particle sensor was used as the first guest package on the MSAv4. A function call was created to read ADC channel 2. A small PCB was build to regulate the signal to the Infrared LED used in the particle sensor. The PCB and sensor were embedded into the lid of the flight box and held in place with duck tape. A small vacuum pump with tubing was connected to the sensor. It was designed to suck air through the sensor from the outside and exhaust it through the side of the flight box (See Figure 18 and Figure 19). The particle sensor was flown on two missions during the balloon bonanza; 28 July 2013 and 31 July 2013. Data gathered from the first mission is illustrated in Figure 31.

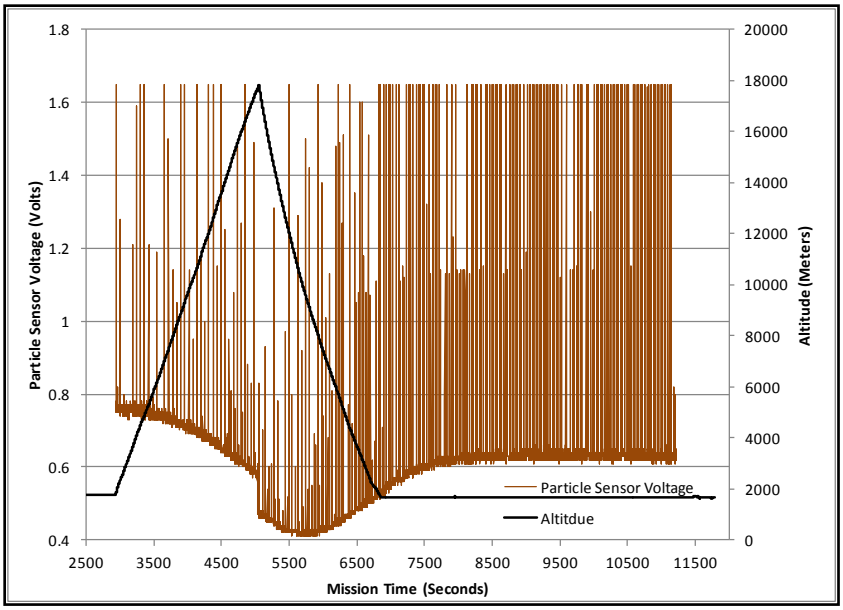

Figure 31: MSAv4 Particle Sensor Readings vs. Altitude
28 July, 2013

The data signal was cut off due to an improperly selected resistor in the voltage divider from the optical sensor. However, the density of particle counts is higher near the ground. Also, the density of counts increases after impact. This was expected because the MSAv4 landed in the desert where dirt and dust were being agitated by a breeze.

Figure 32: Landing Site in La Pointe, Utah
28 July, 2013

5.2.2 Internal and External Temperature Readings

The MSAv4 was flown with both external and internal temperature sensors. The MSAv3 had an internal temperature sensor but no external unit. The internal units were compared side by side and it was discovered that there are issues with the MSAv3 onboard temperature sensor. It is unstable compared to the MSAv4. Figure 33 shows a comparison of internal temperature data for both the MSAv4 and MSAv3. Notice that the MSAv3 provides unstable data for the first 1000 seconds and then fixes at 20.4 ºC for the duration of the flight; meanwhile the MSAv4 records a stable fluctuation in temperature changes as expected. The MSAv4 records a temperature dip between 4000 seconds and 12000 seconds; this occurs during flight where external temperatures dropped as low as -30 ºC; as shown in Figure 34.

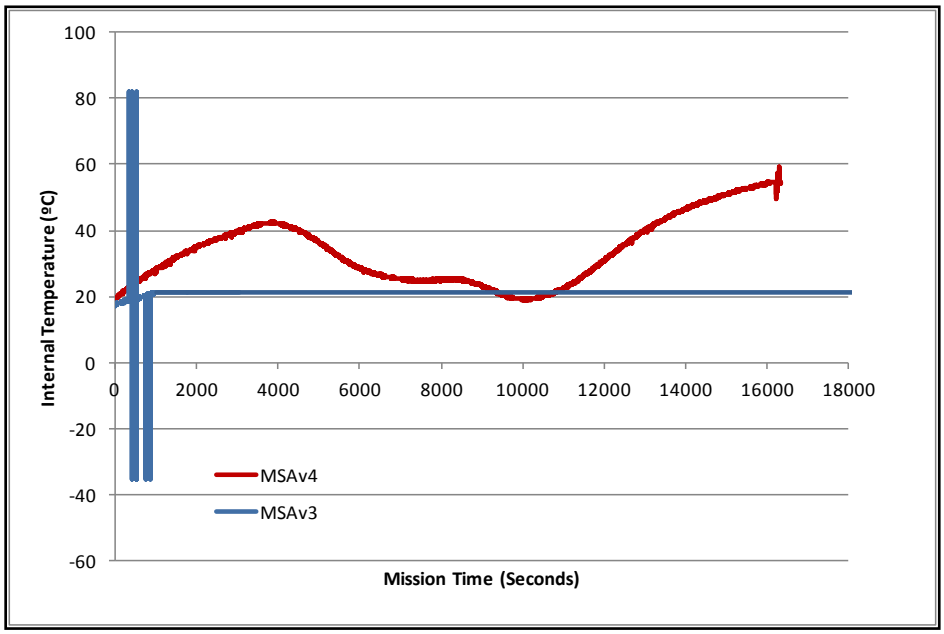

Figure 33: MSAv3 and MSAv4 Internal Temperature vs. Time 31 July, 2013

The external temperature sensor performed as designed, with temperature readings within expected limits. However, it was noticed that the external temperature was being affected by the increasing internal temperature. This has prompted a better mean of isolating the external temperature from internal heat sources. This will be corrected by moving the sensor closer to the outside environment and sealing the inside with more foam and tape. See Figure 34 for an illustration of external temperature compared to altitude during flight.

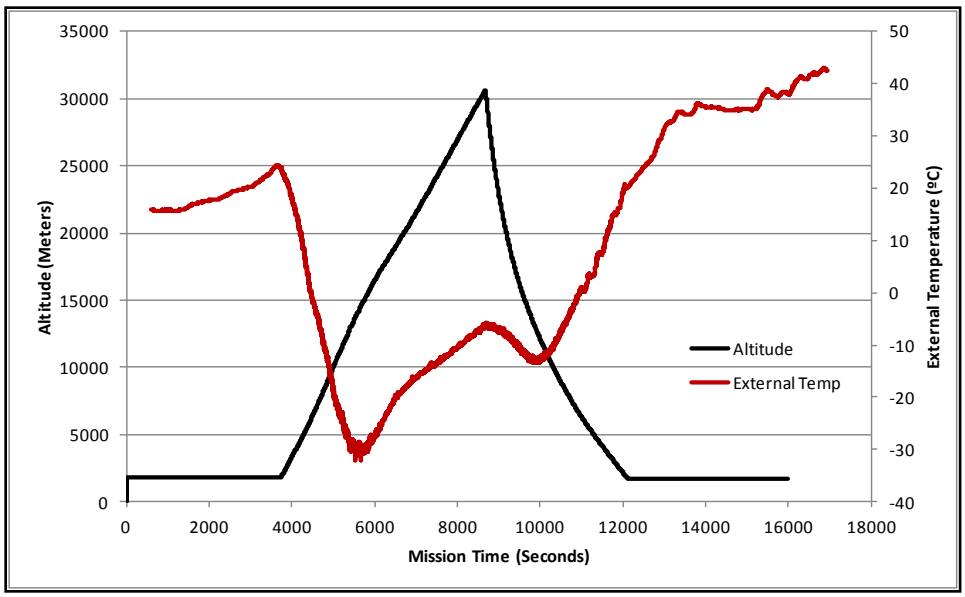

Figure 34: MSAv4 External Temperature and Altitude vs. Time 31 July, 2013

5.2.1 Hardness Requirement Testing

The 2013 summer flight tests demonstrated the MSAv4's ability to endure the temperatures, pressures, humidity, shock, and vibration forces typical of high altitude balloon flights. There are no indications that the MSAv4 was affected by RFI from nearby radio transmitters or any other kind of radiation. The MSAv4 landed in many types of different terrain; wooded, desert, and even aquatic. It survived every flight without damage. Besides power down due to low battery voltage, no data was lost or corrupted; all data streams appear smooth and uniform for the duration of the data logging period. Further testing will be conducted on future flights and calibrations.

Figure 35: Landing Point in La Pointe, Utah 31 July, 2013

6.0 Conclusion

The MSAv4 has been designed to meet a majority of the design requirements provided by Dr. Sohl. The MSAv4 is easy to operate on the flight line and has external controls and indicators. The MSAv4 is lightweight and compact. The MSAv4 has 8GB of onboard data memory that is easy to access via an Ethernet connection and SD card. The MSAv4 has the ability to easily accept guest packages, with programming that is easy to modify. Data collected by the MSAv4 is time stamped. The MSAv4 can communicate over UART, I2C, and Ethernet.

The MSAv4 is capable measuring the following parameters:

- Internal temperature
- External temperature
- Internal humidity
- External humidity
- 3-axis accelerations
- 3-axis gyroscopic motions
- 3-axis magnetic fields
- Atmospheric pressure
- Particle densities
- Battery voltage

There is still some work left to properly calibrate the gyroscope, magnetometer, and particle sensor. The MSAv4 processes raw data and reports calibrated engineering units. The MSAv4 has a functioning siren system; however, a function GPS is still needed to trigger the siren based on altitude.

The MSAv4 is capable of surviving the rigors of high altitude balloon flight. The internal and external enclosures provide adequate protection against cold temperatures, water, shock, and vibrations.

Delays resulted in a failure to complete the GPS module on time. The GPS module is planned to be completed at a later time.

Overall, the MSAv4 design has been a success. Plans are already in place to begin designing guest sensors that will operate with the MSAv4.

7.0 References

Works Cited

Arjan. (2012, 2). *googlegroups.com*. Retrieved from groups.google.com: https://groups.google.com/forum/#!topic/bcm2835/NGvE0VzC_Zg

cdaaawg. (2013, 03 16). *Turn off and on USB*. Retrieved from Raspberrypi.org: http://www.raspberrypi.org/phpBB3/viewtopic.php?p=284594

McCauley, M. (2013, October 9). *C Library for Broadcom BCM 2835 as used in Raspbery Pi*. Retrieved from airspayce.com: http://www.airspayce.com/mikem/bcm2835/

O'Hanlon, M. (2012, 6 10). *Raspberry Pi - run program at start-up*. Retrieved from stuff about code: http://www.stuffaboutcode.com/2012/06/raspberry-pi-run-program-at-start-up.html

S, A. (2012, 03 13). *RPi_low-level_peripherals*. Retrieved from elinux.org: http://elinux.org/File:GPIOs.png

Soybom. (2002, 10 23). *anandtech.com*. Retrieved from forums: http://forums.anandtech.com/showthread.php?t=910864

Appendix A

Schematics

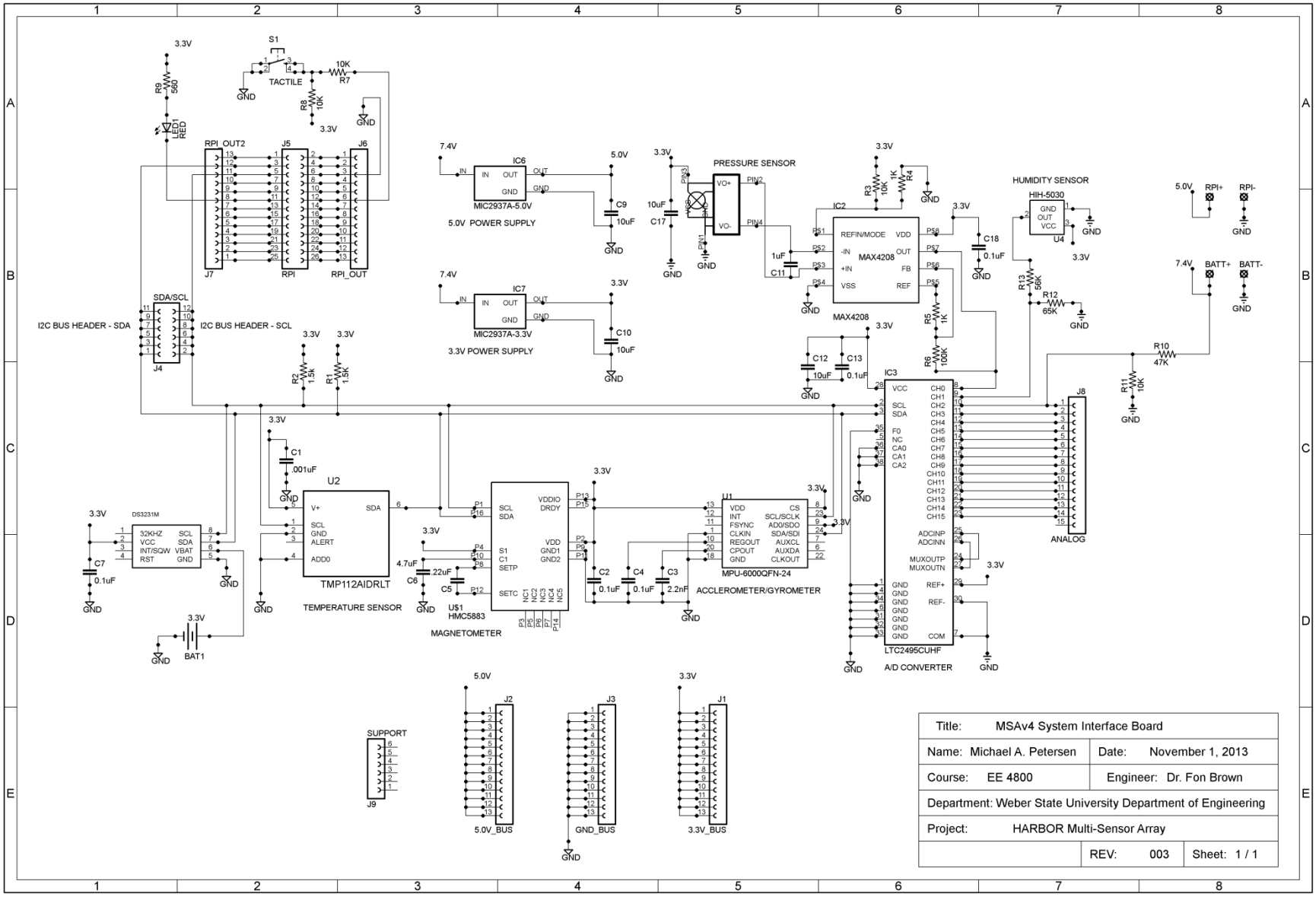

Figure 36: System Interface Daughterboard Schematic

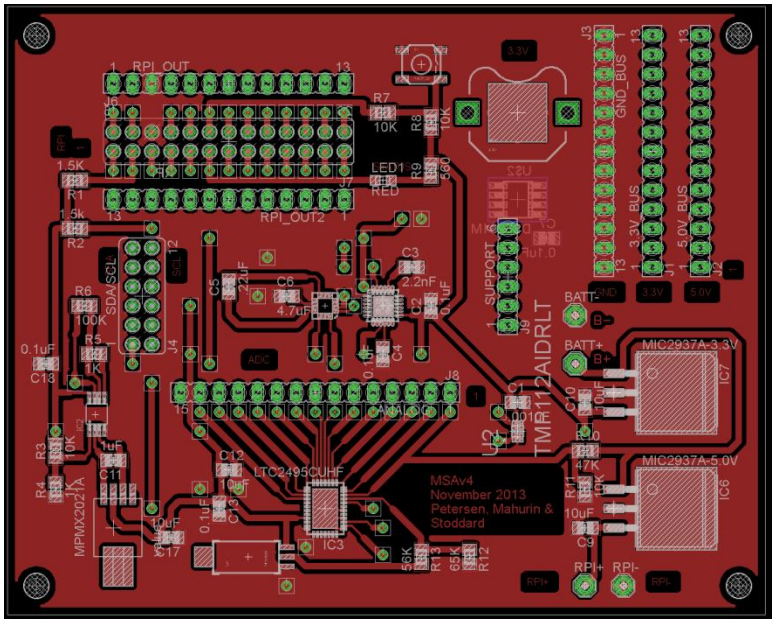

Figure 37: System Interface Board Layout (Top)

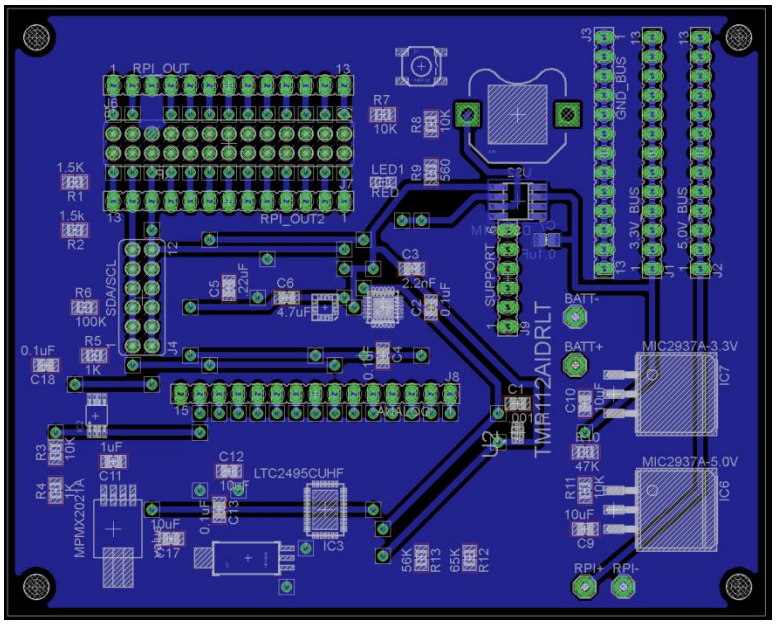

Figure 38: System Interface Board Layout (Bottom)

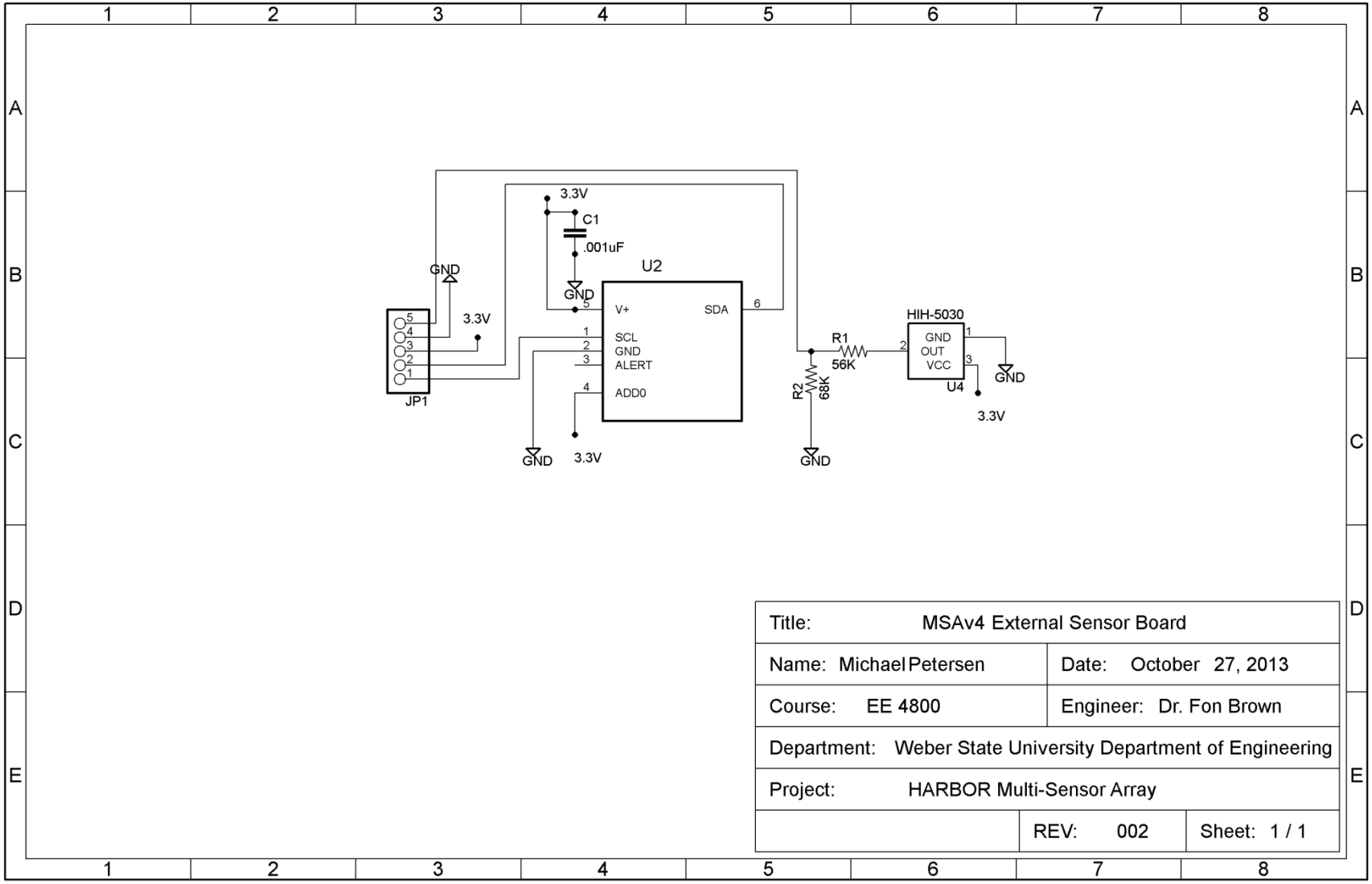

Figure 39: External Sensor Board Schematic

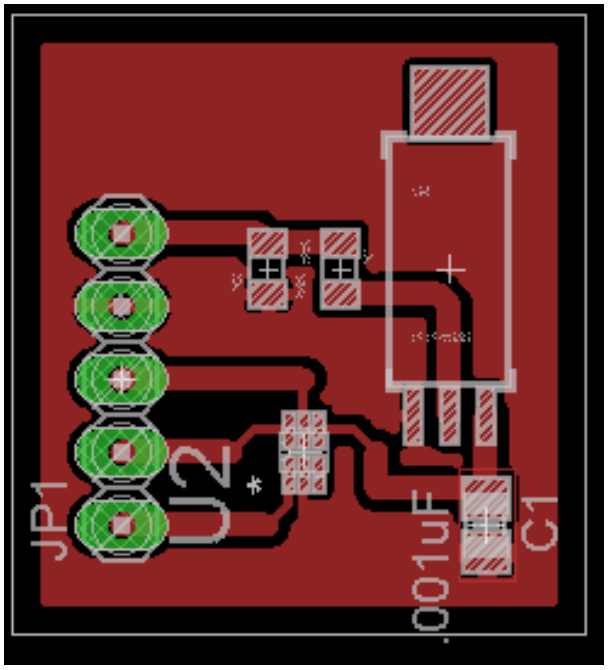

Figure 40: External Sensor Board Layout

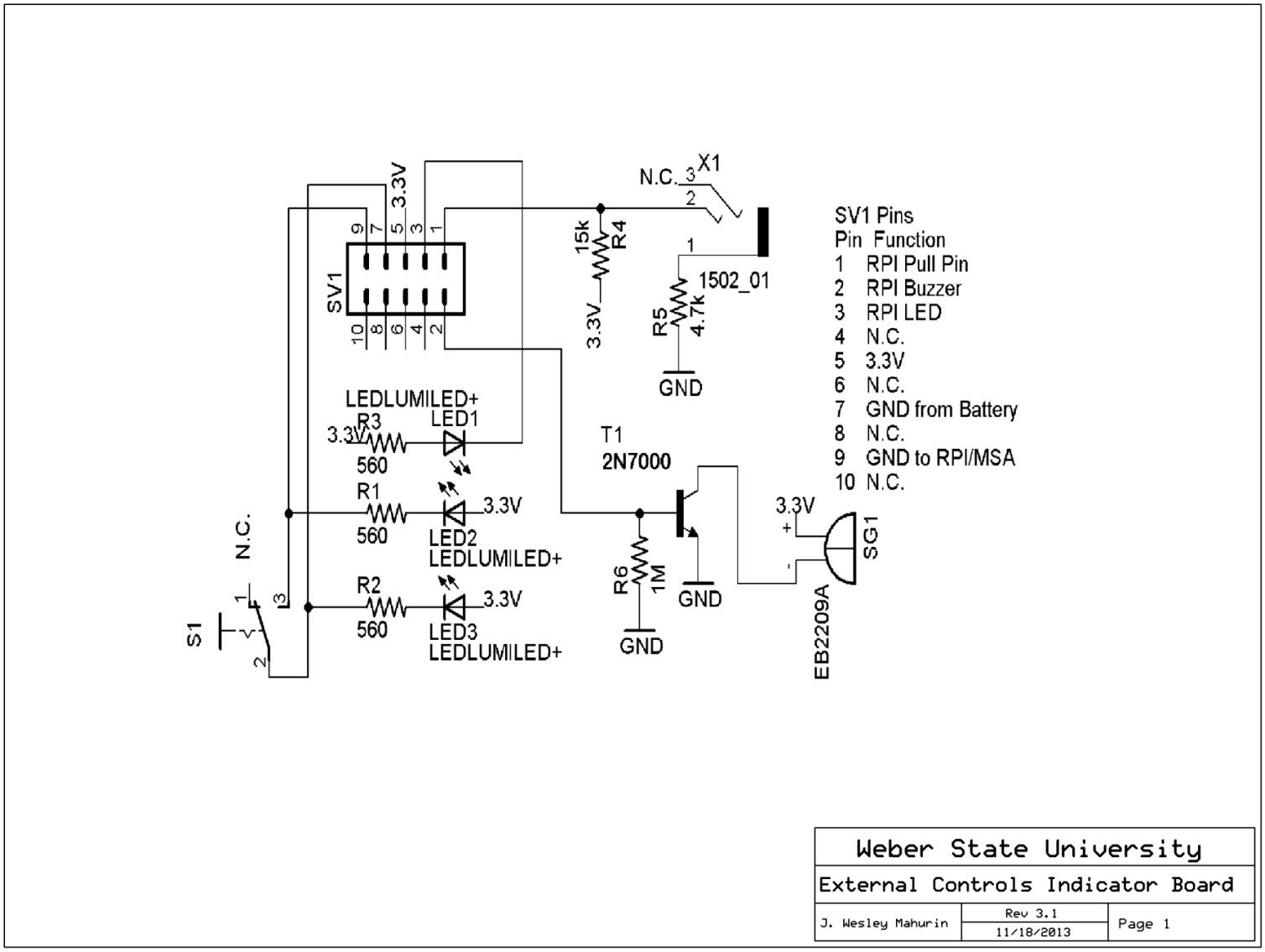

Figure 41: External Controls and Indicators Board Schematic

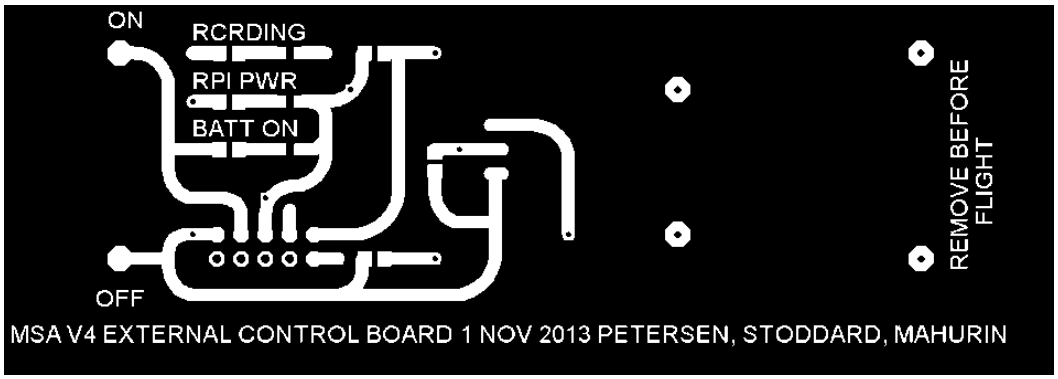

Figure 42: External Controls and Indicators Board Layout (Top View)

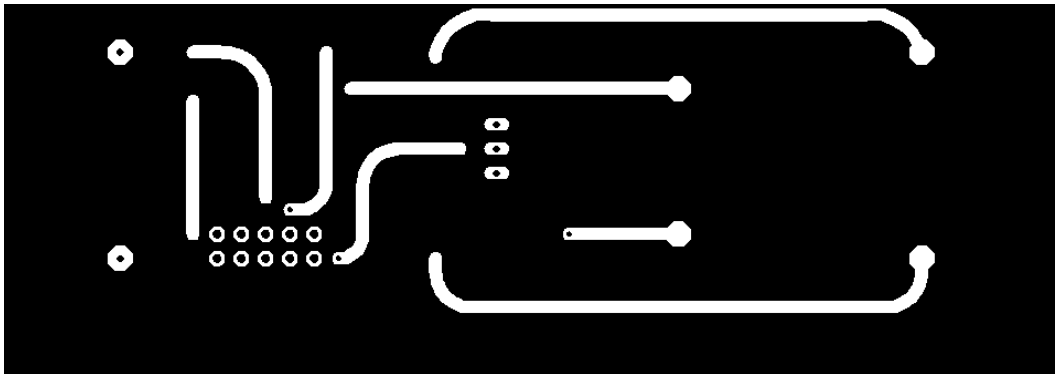

Figure 43: External Controls and Indicators Board Layout (Bottom)

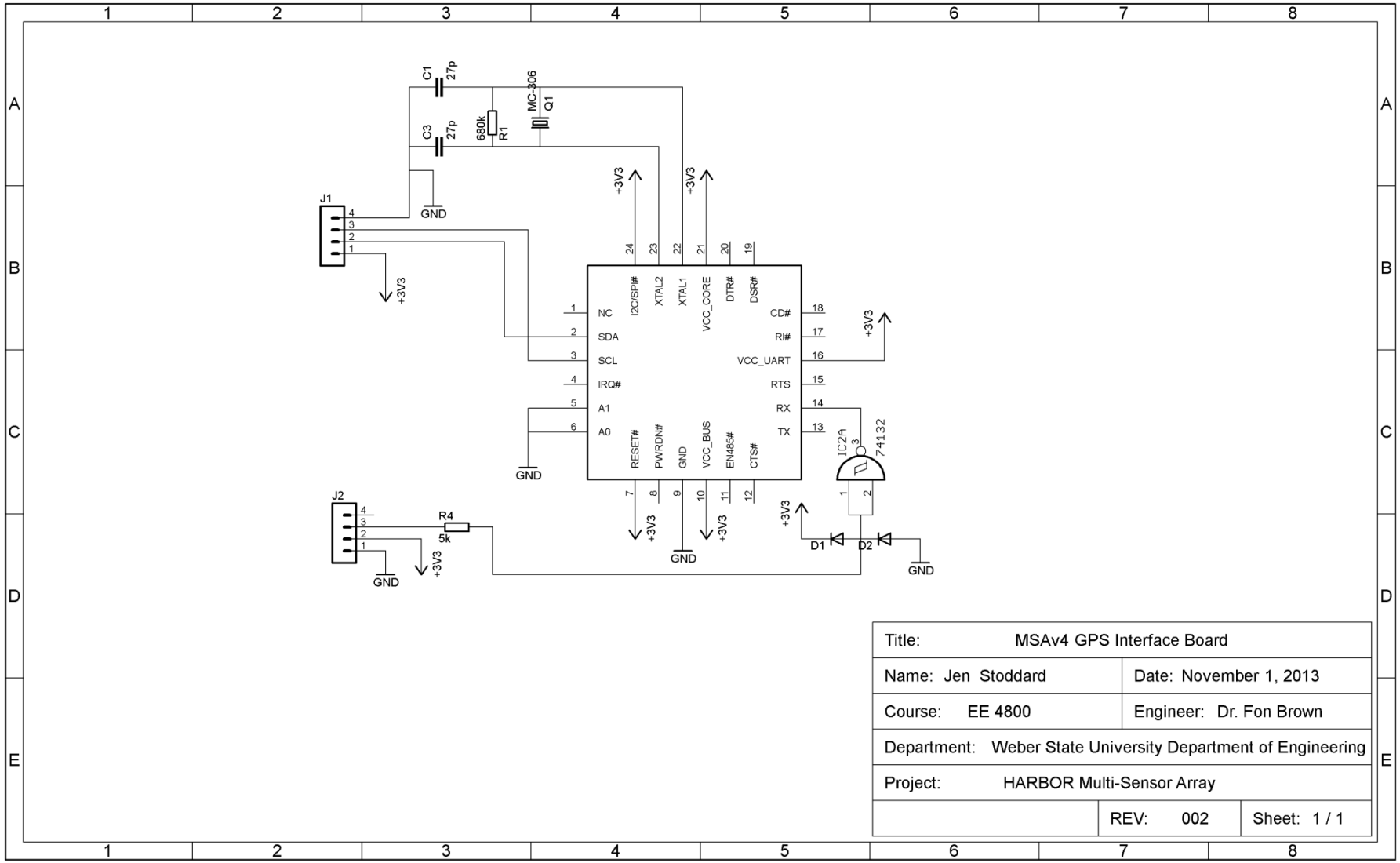

Figure 44: GPS Interface Board Schematic

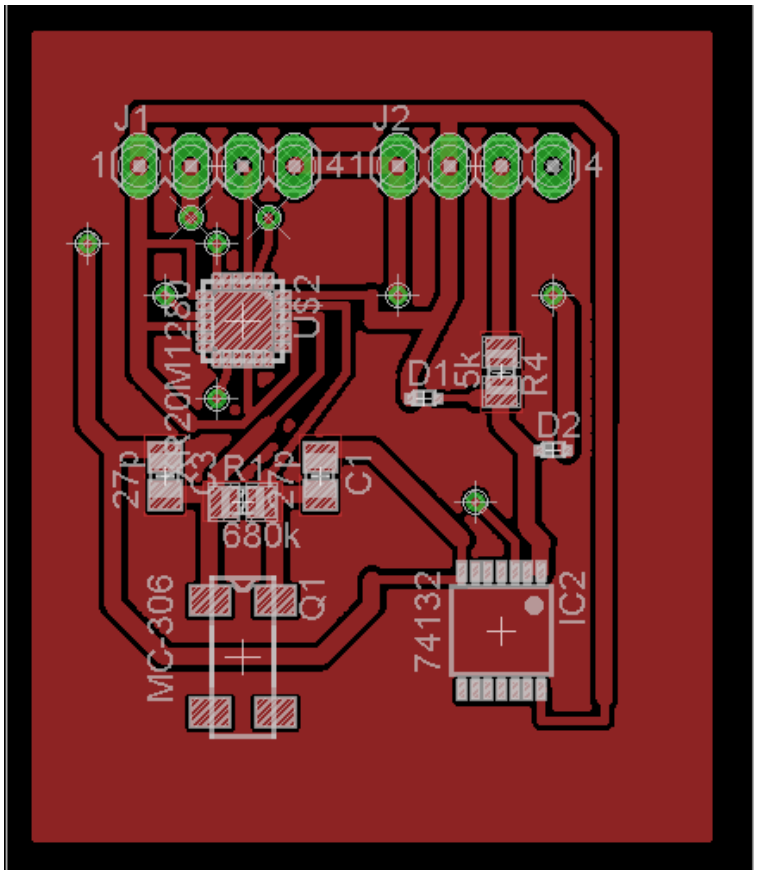

Figure 45: GPS Interface Board Layout (Top View)

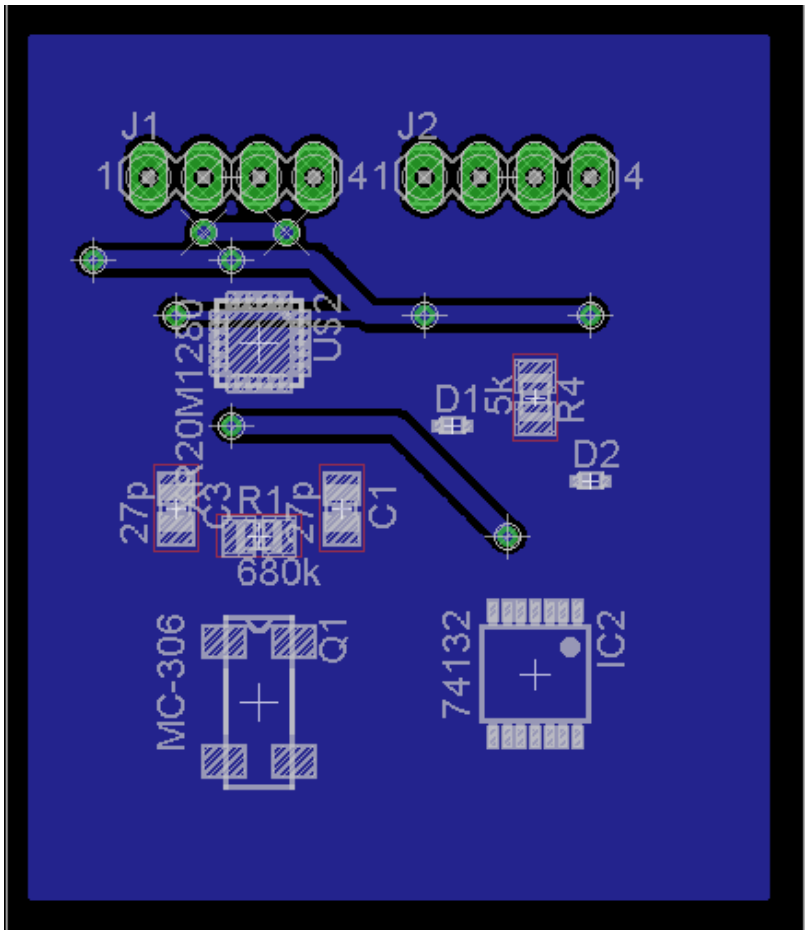

Figure 46: GPS Interface Board Layout (Bottom View)

Appendix B

MSA_v4 Operational Flight Program

The following is the main code written for the MSAv4 to perform all of the necessary functions detailed in the requirements.

```
/*********************************************************************/
//       msa_v4.c
/*********************************************************************/

/*********************************************************************/
// Author: Michael A. Petersen
// Copyright (C) 2013 Michael A. Petersen
// Original code date: 5/1/2013
//
// This is the final software for running the MSAv4 on a
// Debian OS installed on a Raspberry Pi (RPi).
// This program is dependent on the bcm 2385 C library, maintained my
// Mike McCauley at http://www.airspayce.com/mikem/bcm2835/
// This program must be compiled on the RPi using the gcc compiler and
// a properly installed version of the bcm3835 1.25 or newer
//
// After installing bcm2835, you can build the MSAv4 program
// with:
// gcc -o msa_v4 msa_v4.c -l bcm2835 -l m -l pthread
// sudo ./msa_v4
//
// This code is written specifically for the RPi, version one, which uses
// the I2C '0' bus, not I2c '1'. The library had to be modified to
// accommodate this. However, RPi V2 and newer are using I2c '1' bus and
// the bcm2835 library will not need to be modified to work correctly.
/*********************************************************************/

/*********************************************************************/
// revision notes
/*********************************************************************/
// Original code: 5/1/2013
// Added tmp112 function : 5/30/13
// Added external temperature sensor: 6/21/13
// Updated particle counter routine: 7/21/13
// Updated with new demo-board: 9/23/13
//       - Moved Particle Counter to channel 2
//       - Moved Pressure Sensor to channel 1
// Added MPU6000 accelerometer/gyroscope routine: 9/27/13
// Added HMC5883L magnetometer and DS3231 rtc routines: 10/1/13
// Replaced Delay function for ADC conversion time with a dynamic read
// function, which reads the mpu6000 and hmc5883 for 150 ms: 10/3/13
// Corrected code for calculating Relative Humidity: 10/13/13
// Modified code with pthread to create paralled processes with a timer
// and a scheduler: 10/21/13
// Included a function to read the battery voltage and initiate a safe
// shutdown in the event that the LIPO battery voltage drops below 6.2 V
// per cell. Also included a function to turn on the buzzer when the
// package gets near the ground on descent: 11/8/2013
/*********************************************************************/

/*********************************************************************/
// includes
/*********************************************************************/

#include <bcm2835.h>        // version 1.25 or newer compile with -l bcm2835
#include <stdio.h>
#include <math.h>           // compile with -l m
#include <pthread.h>        // compile with -l pthread
#include <time.h>           // for nanosleep
#include <semaphore.h>      // compile with -l pthread

/*********************************************************************/
// definitions and constants
/*********************************************************************/
#define PIN_7                       RPI_GPIO_P1_07              // assign bcm pin GPIO 04
#define STATUS_LED                  RPI_GPIO_P1_11              // assign bcm pin GPIO 17
#define BUTTON                      RPI_GPIO_P1_12              // assign bcm pin GPIO 18
#define EXTERNAL_LED        RPI_GPIO_P1_13              // assign bcm pin GPIO 21
#define PULL_PIN            RPI_GPIO_P1_15              // assign bcm pin GPIO 22
```

```
#define DUST_LED            RPI_GPIO_P1_16              // assign bcm pin GPIO 23
#define BUZZER                      RPI_GPIO_P1_18              // assign bcm pin GPIO 24
#define PIN_22                      RPI_GPIO_P1_22              // assign bcm pin GPIO 25
#define PIN_19                      RPI_GPIO_P1_19              // assign bcm pin GPIO
// normally SPI MOSI

#define PIN_21                      RPI_GPIO_P1_21              // assign bcm pin GPIO
// normally SPI MISO

#define PIN_23                      RPI_GPIO_P1_23              // assign bcm pin GPIO
// normally SPI CLK

#define PIN_24                      RPI_GPIO_P1_24              // assign bcm pin GPIO
// normally SPI CEO

#define PIN_26                      RPI_GPIO_P1_26              // assign bcm pin GPIO
// normally SPI CE1

/*************************************************************************/
// global variables
/*************************************************************************/
unsigned char LTC2495       = 0x14;  // LTC2495 slave address
unsigned char HIH5030       = 0x14;  // HIH5030 same slave address as 2495
unsigned char MPXM2102A = 0x14; // MPXM2102A same slave address as 2495
unsigned char TMP112        = 0x48; // TMP112 slave address
unsigned char TMP112_X      = 0x49; // TMP112 external sensor slave address
unsigned char DS3231    = 0x68; // DS3231 RTC slave address
unsigned char MPU6000   = 0x69; // MPU6000 accel/gyro slave address
unsigned char HMC5883       = 0x1e; // HMC5883 magnetometer slave address
unsigned char tmp_ch0[] = {0xa0, 0xc0};
// 0xa0c0 selects the adc on-board temp sensor configuration
unsigned char chan_0[]  = {0xb0, 0x80};
// 0xb080 selects adc channel 0, single ended input, +IN, gain X 1
unsigned char chan_1[]  = {0xb8, 0x80};
// 0xb880 selects adc channel 1, single ended input, +IN, gain X 1
unsigned char chan_2[]  = {0xb1, 0x80};
// 0xb180 selects adc channel 2, single ended input, +IN, gain X 1
unsigned char chan_3[]      = {0xb9, 0x80};
// 0xb980 selects adc channel 3, single ended input, +IN, gain X 1
unsigned char chan_4[]      = {0xb2, 0x80};
// 0xb280 selects adc channel 4, single ended input, +IN, gain X 1
unsigned char chan_5[]      = {0xba, 0x80};
// 0xba80 selects adc channel 5, single ended input, +IN, gain X 1

float h_temp_comp;
// temp from LTC2495 for calibrating humidity sensor
float h_x_temp_comp;
// temp from external tmp112 for calibrating the humidity sensor

// Global File Variables
unsigned int file_num = 0; // for counting number of file names
char ENV_PREFIX[] = "/home/pi/env_";
char DYN_PREFIX[] = "/home/pi/dyn_";
char GPS_PREFIX[] = "/home/pi/gps_";
char POSTFIX[] = ".csv";
char file_name_environmental_data[255];
// max characters in linux file name is 255
char file_name_flight_dynamics_data[255];
char file_name_gps_data[255];

// Global File pointers
FILE *env_fp;      // file pointer for environmental data
FILE *dyn_fp;      // file pointer for flight dynamics data
FILE *gps_fp;      // file pointer for GPS data

// Global semaphore and pthread variables
sem_t sem;                                  // semaphore for scheduling threads
int pshared;                        // shared between the threads of a process
int ret;                            // return value for semaphore
unsigned int value;                 // initial value for semaphore

// Global mission time variables
int mission_timer = 0;
int time_up  = 14400; // Ends mission after 14400 seconds 4.0 hours
                                    // Current batter is 1100mAh
                                    // RPi draws 190mA continuously
                                    // 1100mAh/190ma = 5.7 hours
int save_time = 3000;       // 3000 milliseconds = 5 minutes
                            // closes data-log file and reopens to save data
unsigned int button_status = 1;
unsigned int pin_status = 0;
unsigned int led_status;
```

```
unsigned int buzz_flag = 0;
unsigned int battery_voltage;
unsigned int batt_count_down = 60;
// 60 seconds to determine if battery is critically low

/*********************************************************************/
// prototypes
/*********************************************************************/

void read_tmp112(FILE *);
float read_tmp112_ext(FILE *);
void read_ds3231(FILE *);
void read_mpu6000(FILE *);
void read_hmc5883(FILE *);
void read_dynamics(FILE *);
void set_channel_itemp(void);
void set_channel_press(void);
void set_channel_humid(void);
void set_channel_ext_humid(void);
void set_channel_prtcl(void);
void set_channel_battery(void);
void empty_current_channel(void);
float read_channel_itemp(FILE *);
void read_channel_press(FILE *);
void read_channel_humid(FILE *, float);
void read_channel_ext_humid(FILE *, float);
void read_channel_prtcl(FILE *);
float read_channel_battery(FILE *);
void get_date(FILE *);

/*********************************************************************/
// POSIX threads
/*********************************************************************/
void *schedule_timer(void *arg)
{
         int milisec = 100; // length of time to sleep, in milliseconds
         struct timespec req = {0};
         req.tv_sec = 0;
         req.tv_nsec = milisec * 1000000L; // set delay to 100 milliseconds

         while(1)
         {
                  //nanosleep(&req, (struct timespec *)NULL);
                  // sleep 100 milliseconds
                  nanosleep(&req, NULL);
                  sem_post(&sem); // make semaphore available
         } // end while loop

} // end schedule_timer


void *schedule_tasks(void *arg)
{
         unsigned int conversion = 0;
         unsigned int counter = 0;
         unsigned int channel = 0;

         // schedule sensor reading
         while(1)
         {
                  sem_wait(&sem);    // wait for semaphore to become available


                           conversion ++;
                           if(conversion == 2) // 200ms
                           {
                      switch(counter)
                      {
                                    case 0 :
                                             counter ++;
                                             switch(channel)
                                             {
                                              case 0:
                                                       read_ds3231(env_fp);
                                                       // read time from RTC for environmental data
                                                       read_tmp112(env_fp);
                                                       // read PCB board temperature
                                                  h_x_temp_comp = read_tmp112_ext(env_fp);
                                                  // read external temperature
                                                       set_channel_itemp();
                                                       // set adc to internal temp sensor
                                                       break;
```

```
          case 1:
             set_channel_press();
             // set adc to pressure sensor reading
                   break;
          case 2:
                   set_channel_humid();
                   // set adc to humidity sensor reading
                   break;
          case 3:
                   set_channel_prtcl();
                   // set adc to particle sensor reading
                   break;
          case 4:
                   set_channel_ext_humid();
                   break;
          default:
             set_channel_battery();
         } // end switch
         break;
case 1 :
         counter ++;
         switch(channel)
         // reset adc to ensure proper channel is selected
         {
          case 0:
                    empty_current_channel();
                   //set_channel_itemp();
                   break;
          case 1:
                   set_channel_press();
                   break;
          case 2:
                   set_channel_humid();
                   break;
          case 3:
                   set_channel_prtcl();
                   break;
          case 4:
                   set_channel_ext_humid();
                   break;
          default:
             set_channel_battery();
         } // end switch
         break;
case 2 :
         counter ++;
         empty_current_channel();
         // clear garbage read out of adc
         break;
case 3 :
         counter ++; // I don't think this line is necessary!
    switch(channel)
    // now it's safe to read adc data and write to file
    {
          case 0:
                   h_temp_comp = read_channel_itemp(env_fp);
                   channel ++;
                   break;
          case 1:
                   read_channel_press(env_fp);
                   channel ++;
                   break;
          case 2:
                   read_channel_humid(env_fp, h_temp_comp);
                   channel ++;
                   break;
          case 3:
                   read_channel_prtcl(env_fp);
                   channel ++;
                   break;
          case 4:
             read_channel_ext_humid(env_fp, h_x_temp_comp);
             channel ++;
             break;
          default:
                   battery_voltage = read_channel_battery(env_fp);
                   fprintf(env_fp,"\n");
                   // newline for environmental data
                   channel = 0;
         }// end switch
         counter = 0;
         break;
```

```
                                default :
                                        break;

                        } // end switch
                        conversion = 0;
                        } // end if - conversion

                read_dynamics(dyn_fp);

                // close and save files every 5 minutes
                if(save_time > 0)
                {
                        save_time = save_time - 1;
                } // end if
                else
                {
                        fclose(env_fp);    // close environmental data file
                        fclose(dyn_fp);    // close dynamic flight data file
                        fclose(gps_fp); // close gps file
                        env_fp = fopen(file_name_environmental_data, "a");
                        // file for temp, humidity, pressure, and other environmental data
                        dyn_fp = fopen(file_name_flight_dynamics_data, "a");
                        // file for acclerations, gyroscopic motion, and other dynamic data
                        gps_fp = fopen(file_name_gps_data, "a"); // file for gps tracking data
                        save_time = 3000; // reset save time for another 5 minutes.
                } // end else
        } // end while-loop
} // end schedule_tasks

/*************************************************************************/
// main thread
/*************************************************************************/

int main(void)
{
        // declare variables

        pthread_t clk_thread; // thread to create .1 second reads
        pthread_t sch_thread; // thread with data logging schedule

        //int second = 1000; // length of time to sleep, in milliseconds
        //int tenthsecond = 100; // 100 milliseconds
        //int millisecond = 1; //   1 milliseconds
        struct timespec req = {0};
        req.tv_sec = 1;
        req.tv_nsec = 0;

        unsigned char i;

        // If you call this, it will not actually access the GPIO
        // Use for testing
        // bcm2835_set_debug(1);

        if (!bcm2835_init())
        return 1;
        bcm2835_i2c_begin();        // The default
        bcm2835_i2c_setClockDivider(BCM2835_I2C_CLOCK_DIVIDER_626);
                // Divides the 250Mhz clock
                // to a 400Khz Clock
                // The max speed for i2c sensors
                // is 400Khz
        bcm2835_gpio_fsel(STATUS_LED, BCM2835_GPIO_FSEL_OUTP);
        // On-Board Status LED
        bcm2835_gpio_fsel(EXTERNAL_LED, BCM2835_GPIO_FSEL_OUTP);
        // External Status LED
        bcm2835_gpio_fsel(DUST_LED, BCM2835_GPIO_FSEL_OUTP);
        // Dust Sensor LED
        bcm2835_gpio_fsel(BUTTON, BCM2835_GPIO_FSEL_INPT);
        // Stop Button
        bcm2835_gpio_fsel(PULL_PIN, BCM2835_GPIO_FSEL_INPT);
        // Pull-Pin
        bcm2835_gpio_fsel(BUZZER, BCM2835_GPIO_FSEL_OUTP);
        // Buzzer

        // turn on status LED
        bcm2835_gpio_write(STATUS_LED, LOW);
        bcm2835_gpio_write(EXTERNAL_LED, LOW);

        // Initialize file names for checking
        sprintf(file_name_environmental_data, "%s%d%s",ENV_PREFIX,file_num,POSTFIX);
        sprintf(file_name_flight_dynamics_data, "%s%d%s",DYN_PREFIX,file_num,POSTFIX);
        sprintf(file_name_gps_data, "%s%d%s",GPS_PREFIX,file_num,POSTFIX);
```

```
	// Check if files already exists by attempting to read them

	// generate file name for environmental data
	while((env_fp = fopen(file_name_environmental_data, "r")) != NULL)
	// while the file name already exists
	{
	fclose(env_fp);
	file_num ++;
	sprintf(file_name_environmental_data, "%s%d%s",ENV_PREFIX,file_num,POSTFIX);
	}// stop when a file name has been found that doesn't already exist

	env_fp = fopen(file_name_environmental_data, "a"); // file for environmental data

	file_num = 0;

	// generate file name for flight dynamics data
	while((dyn_fp = fopen(file_name_flight_dynamics_data, "r")) != NULL)
	// while the file name already exists
	{
	fclose(dyn_fp);
	file_num ++;
	sprintf(file_name_flight_dynamics_data, "%s%d%s",DYN_PREFIX,file_num,POSTFIX);
	}// stop when a file name has been found that doesn't already exist

	dyn_fp = fopen(file_name_flight_dynamics_data, "a");
	// file for dynamics data

	file_num = 0;

	// generate file name for gps data
	while((gps_fp = fopen(file_name_gps_data, "r")) != NULL)
	// while the file name already exists
	{
	fclose(gps_fp);
	file_num ++;
	sprintf(file_name_gps_data, "%s%d%s",GPS_PREFIX,file_num,POSTFIX);
	}// stop when a file name has been found that doesn't already exist

	gps_fp = fopen(file_name_gps_data, "a"); // file for gps data

	// insert current date header at the top of all data files
	get_date(env_fp);
	get_date(dyn_fp);
	get_date(gps_fp);

	// install data headers at the top of all data files
	fprintf(env_fp,"time,reg_temp,xtemp,itemp,ipressure,ihumidity,dust_v,xhumidity,batt_voltage,\n");
	// data header external temp, internal temp, internal humidity
	fprintf(dyn_fp,"time,accel_x_2g,accel_y_2g,accel_z_2g,gyro_x_250,gyro_y_250,gyro_z_250,accel_x_8g,accel_y_8g,acc
el_z_8g,gyro_x_1000,gyro_y_1000,gyro_z_1000,mag_x,mag_y,mag_z,\n");

	while(pin_status==0) // wait for mission pin to be pulled
	{
		pin_status = bcm2835_gpio_lev(PULL_PIN);

		if(pin_status == 1)
		// Debounce -- wait half a second and then check the pin status
		// again to make sure it has been pulled for real
		{
			bcm2835_delay(500);
			pin_status = bcm2835_gpio_lev(PULL_PIN);
		} // end if

		bcm2835_gpio_write(STATUS_LED, LOW);
		bcm2835_gpio_write(EXTERNAL_LED, LOW);
		bcm2835_delay(50);
		bcm2835_gpio_write(STATUS_LED, HIGH);
		bcm2835_gpio_write(EXTERNAL_LED, HIGH);
		bcm2835_delay(500);
	} // end while-loop

	// acknowledge pin was pulled
	for(i=0; i<10; i++)
	{
		bcm2835_gpio_write(STATUS_LED, LOW);
		bcm2835_gpio_write(EXTERNAL_LED, LOW);
		bcm2835_gpio_write(BUZZER, HIGH);
		bcm2835_delay(20);
		bcm2835_gpio_write(STATUS_LED, HIGH);
		bcm2835_gpio_write(EXTERNAL_LED, HIGH);
		bcm2835_gpio_write(BUZZER, LOW);
```

```
        bcm2835_delay(100);
}// end for loop

// initialize POSIX threads and semaphores
pshared = 0;       // initial value of 0 -> share between threads
value    = 1;      // initial value of 1 -> do schedule
sem_init(&sem,pshared,value); // initialize semaphore

// start 100 millisecond task timer
pthread_create(&clk_thread, NULL, &schedule_timer, NULL);

// start data logging scheduler
pthread_create(&sch_thread, NULL, &schedule_tasks, NULL);

system("/opt/vc/bin/tvservice -off");
// shut off PAL/HDMI outputs to save power
system("sh -c \"echo 1 > /sys/devices/platform/bcm2708_usb/bussuspend\" ");
// shut off USB chip to save power

// **** START MAIN LOOP ****
while(button_status && (mission_timer < time_up))
// while stop button not pressed and max time not exceeded
{
button_status = bcm2835_gpio_lev(BUTTON);
pin_status = bcm2835_gpio_lev(PULL_PIN);
led_status = bcm2835_gpio_lev(STATUS_LED);

// invert status LED
if(led_status == 0)
{
        bcm2835_gpio_write(STATUS_LED, HIGH); // turn off led
        bcm2835_gpio_write(EXTERNAL_LED, HIGH);        // turn off led
} // end if
else
{
        bcm2835_gpio_write(STATUS_LED, LOW);  // turn on led
        bcm2835_gpio_write(EXTERNAL_LED, LOW);         // turn on led
} // end else

// monitor battery voltage
if(battery_voltage < 6.4) // 6.4V or 3.2 V per Cell;
{
        batt_count_down --;
} // end if
else
{
        batt_count_down = 60;
        // only count 1 consecutive minute of low voltage
}

if(batt_count_down == 0) // Battery is critically low; SHUTDOWN!!!
{
        button_status = 0; // Initiate artificial button press
} // end if

//nanosleep(&req, (struct timespec *)NULL); // sleep 1 second
nanosleep(&req, NULL);

mission_timer ++; // increment mission timer in one second intervals

} // **** END MAIN LOOP ****


printf("Initiating Shutdown!\n");
// acknowledge shutdown initiated
for(i=0; i<10; i++)
{
bcm2835_gpio_write(STATUS_LED, LOW);
bcm2835_gpio_write(EXTERNAL_LED, LOW);
bcm2835_gpio_write(BUZZER, HIGH);
bcm2835_delay(20);
bcm2835_gpio_write(STATUS_LED, HIGH);
bcm2835_gpio_write(EXTERNAL_LED, HIGH);
bcm2835_gpio_write(BUZZER, LOW);
bcm2835_delay(100);
}// end for-loop

fclose(env_fp);    // close environmental data file
fclose(dyn_fp);             // close dynamics data file
fclose(gps_fp);             // close gps data file
bcm2835_i2c_end();          // close i2c port
bcm2835_close();   // close all other bcm2835 ports
```

```
        system("shutdown -h -P now");  // shutdown the pi now

        return 0; // end msa_v4_0
} // end main


/*********************************************************************/
// function definitions
/*********************************************************************/

// The LTC2495 has an average conversion time of 165ms in the 1x speed mode.
// minimum conversion time 144 mS and typical is 150 mS

void set_channel_itemp(void)
{
        bcm2835_i2c_setSlaveAddress(LTC2495);
        bcm2835_i2c_write(tmp_ch0, sizeof(tmp_ch0));
        // set ADC to read the temperature sensor

}// end set_channel_itemp

void empty_current_channel(void)
{
        unsigned char junk2[3];
        bcm2835_i2c_setSlaveAddress(LTC2495);
        bcm2835_i2c_read(junk2, sizeof(junk2)); // junk data

}// end empty_current_channel

float read_channel_itemp(FILE *gen_fp)
{
        bcm2835_i2c_setSlaveAddress(LTC2495);

        // on-board temperature sensor variables
        int temp,sign;  // DATAOUT16 HEX Value for Temperature
        float temp2;       // Kelvin
        //float tslope = 3.712121212;// slope = VRef(+)/12.25 per datasheet
        float tslope = 0.269387755;  // slope = VRef(+)/12.25 per datasheet
                                                                    // VRef(+) = 3.3 Volts
        float temp3;       // Celsius
        float temp4;       // Fahrenheit

        unsigned char t_buf[3];     // buffers for capturing the temperature
/*********************************************************************/
// read on-board temperature sensor
/*********************************************************************/

        bcm2835_i2c_read(t_buf, sizeof(t_buf));
        // read the data register from the LTC2495
        sign = t_buf[0] & 0x40;     // capture sign bit
        t_buf[0] = t_buf[0] & 0x7f;
        // mask the sign bit for number conversion
        // Concatenate the two temp bytes into a single 16bit value
        temp = (t_buf[0]*0x10000) + (t_buf[1]*0x100) + (t_buf[2]);
        // write temperature data in hex
    temp = temp/0x40; // right justify data bits
        temp2 = ((float)temp*tslope);
        // Convert the 16-bit hex value into Kelvin
        temp3 = temp2 - 273.0;
        // Convert the Kelvin value into degrees Celsius
        if(sign == 0x40)
        {
                temp3 = temp3*(-1);
        } // end if statement

        fprintf(gen_fp, "%.2f,",temp3);

        return temp3;

}// end read_channel_itemp

void set_channel_press(void)
{
        bcm2835_i2c_setSlaveAddress(LTC2495);
        bcm2835_i2c_write(chan_0, sizeof(chan_0));
        // set ADC to read the pressure sensor

}// end set_channel_press

void read_channel_press(FILE *gen_fp)
{
```

```
        bcm2835_i2c_setSlaveAddress(LTC2495);

        unsigned char p_buf[3];     // buffers for capturing the pressure

        // pressure sensor variables
        int   pressure;    // DATAOUT16 HEX Value for Pressure Sensor
        float v_press;     // ADC voltage output for pressure value
        float press2;      // Vin from Pressure Sensor
        double c_press;  // Unit calibrated value [Torr]
        float v_low = 0.3;
        // Minimum voltage when sensor reads 0 Torr "y intercept"
        float c_slow = 0.001513;
        // Calculated Slope [Volts/Torr] from calibration test
        double log_press;
         // variable used when sensor is operating in logarithmic region
/*********************************************************************/
// read pressure sensor
/*********************************************************************/

        bcm2835_i2c_read(p_buf, sizeof(p_buf));
        // read the data register from the LTC2495
        p_buf[0] = p_buf[0] & 0x7f; // mask the first bit (unused)
        pressure = (p_buf[0] * 0x10000) + (p_buf[1] * 0x100) + (p_buf[2]);
        pressure = pressure/0x40;
        v_press = pressure * 0.000025177;
        // multiply by step size to get voltage
        if(v_press > 0.319) // sensor operating in linear region above 150 torr
        {
        c_press = 500 * v_press + 37.75;
        // convert sensor voltage to units [Torr]
        }
        else
        {
        log_press = (v_press-0.3089)/0.002;
        c_press = exp(log_press);
        }
        fprintf(gen_fp, "%.2f,",c_press);

}// end read_channel_press

void set_channel_humid(void)
{
        bcm2835_i2c_setSlaveAddress(LTC2495);
        bcm2835_i2c_write(chan_1, sizeof(chan_1));
        // set ADC to read the humidity sensor

}// end set_channel_humid

void read_channel_humid(FILE *gen_fp, float temp3)
{
        bcm2835_i2c_setSlaveAddress(LTC2495);
        char h_buf[3];     // buffers for capturing the humidity
        // humidity sensor variables
        int   humid;       // DATAOUT16 HEX Value for Humidity
        float v_humid;     // ADC voltage output for humidity value
        float humid2;      // Relative Humidity (1st Order Curve)
        float t_humid;     // temperature corrected humidity
        float hslope = 0xfde8;

/*********************************************************************/
// read humidity sensor
/*********************************************************************/

        bcm2835_i2c_read(h_buf, sizeof(h_buf));
        // read the data register from the LTC2495
    h_buf[0] = h_buf[0] & 0x7f;      // mask the first bit (unused)
        humid = (h_buf[0] * 0x10000) + (h_buf[1] * 0x100) + (h_buf[2]);
    humid = humid/0x40;
        v_humid = humid*.000025177; // hex reading multiplied by step size
        // V(out)=V(source)[(0.00636)(RH)+0.1515]
        // V(out) is being divided so that max doesn't exceed ADC input range
        // The datasheet doesn't provide a realistic value compared to measurement
        // Thus:
        // V(out) = 80.806*V(out-to adc)-22.168
        // This accounts for the min (0.274 V) and max (1.51 V) for this sensor
        // as seen by the adc due to the 68K 56K voltage divider.
        humid2 = (80.806*(v_humid) - 22.168);          // convert humidty to RH %
        // The following is suggested by the data sheet
        t_humid = humid2/(1.0546-.00216*temp3); // correct for temperature
        fprintf(gen_fp, "%.2f,", t_humid);  // save humidty value in file

}// end read_channel_humid
```

```
void set_channel_prtcl(void)
{
         char junk[3];
         bcm2835_i2c_setSlaveAddress(LTC2495);
         bcm2835_i2c_write(chan_2, sizeof(chan_2));
         // set ADC to read the dust sensor

}// end set_channel_prtcl

void read_channel_prtcl(FILE *gen_fp)
{
         bcm2835_i2c_setSlaveAddress(LTC2495);
         unsigned char d_buf[3];     // buffers for capturing the dust

         // dust sensor variables
         int    dust;       // DATAOUT16 HEX Value for Dust Sensor
         float  v_dust;     // ADC voltage output for dust value
         float  dust2;      // averaged dust voltage
         double total_dust; // running sum of voltage readings

/*********************************************************************/
// read optical particle monitor
/*********************************************************************/

         bcm2835_i2c_read(d_buf, sizeof(d_buf));        // read data register
         d_buf[0] = d_buf[0] & 0x7f; // mask the first bit
         dust = (d_buf[0] * 0x10000) + (d_buf[1] * 0x100) + (d_buf[2]);
         dust = dust/0x40;  // remove least most 6 bits (unused)
         v_dust = dust * .000025177; // multiply by step size to get voltage
         fprintf(gen_fp, "%.4f,",dust);
         // log dust value (voltage) in datalog file

}// end read_channel_prtcl


void set_channel_ext_humid(void)
{
         char junk[3];
         bcm2835_i2c_setSlaveAddress(LTC2495);
         bcm2835_i2c_write(chan_3, sizeof(chan_3));
         // set ADC to read the humidity sensor

}// end set_channel_ext_humid

void read_channel_ext_humid(FILE *gen_fp, float x_temp3)
{
         bcm2835_i2c_setSlaveAddress(LTC2495);
         char h_x_buf[3];   // buffers for capturing the external humidity

         // humidity sensor variables
         int   humid_x;     // DATAOUT16 HEX Value for Humidity
         float v_humid_x;   // ADC voltage output for humidity value
         float humidx2;     // Relative Humidity (1st Order Curve)
         float t_humid_x;   // temperature corrected humidity
         float hxslope = 0xfde8;

/*********************************************************************/
// read humidity sensor
/*********************************************************************/

         bcm2835_i2c_read(h_x_buf, sizeof(h_x_buf));
         // read the data register from the LTC2495
    h_x_buf[0] = h_x_buf[0] & 0x7f;  // mask the first bit (unused)
         humid_x = (h_x_buf[0] * 0x10000) + (h_x_buf[1] * 0x100) + (h_x_buf[2]);
    humid_x = humid_x/0x40;
         v_humid_x = humid_x*.000025177; // hex reading multiplied by step size
         // V(out)=V(source)[(0.00636)(RH)+0.1515]
         // V(out) is being divided so that max doesn't exceed ADC input range
         // The datasheet doesn't provide a realistic value compared to measurement
         // Thus:
         // V(out) = 80.806*V(out-to adc)-22.168
         // This accounts for the min (0.274 V) and max (1.51 V) for this sensor
         // as seen by the adc due to the 68K 56K voltage divider.
         humidx2 = (80.806*(v_humid_x) - 22.168);       // convert humidty to RH %
         // The following is suggested by the data sheet
         t_humid_x = humidx2/(1.0546-.00216*x_temp3); // correct for temperature
         fprintf(gen_fp, "%.2f,", t_humid_x);  // save humidty value in file

}// end read_channel_humid

void set_channel_battery(void)
```

```
{
         char junk[3];
         bcm2835_i2c_setSlaveAddress(LTC2495);
         bcm2835_i2c_write(chan_5, sizeof(chan_5));
         // set ADC to read the humidity sensor
}

float read_channel_battery(FILE *gen_fp)
{
         bcm2835_i2c_setSlaveAddress(LTC2495);

         unsigned char bat_buf[3];   // buffers for capturing the dust

         // battery voltage variables
         int    battery;    // DATAOUT16 HEX Value for battery voltage
         float  v_battery;  // ADC voltage output for battery value
         float  battery2;   // averaged battery voltage

/*********************************************************************/
// read the battery voltage
/*********************************************************************/

         bcm2835_i2c_read(bat_buf, sizeof(bat_buf));    // read data register
         bat_buf[0] = bat_buf[0] & 0x7f;       // mask the first bit
         battery = (bat_buf[0] * 0x10000) + (bat_buf[1] * 0x100) + (bat_buf[2]);
         battery = battery/0x40;     // remove least most 6 bits (unused)
         v_battery = battery * .000025177; // multiply by step size to get voltage
         // The battery voltage is divided down to meet the ADC input requirements using a
         // 0.175438596 ratio voltage divider with a 1M ohm and 4.7M ohm resistor pair
         battery2 = v_battery / 0.175;
         // re-scale voltage back up to actual battery volage
         fprintf(gen_fp, "%.2f,",battery2);
         // log battery value (voltage) in datalog file
         return battery2;
} // end read_channel_battery

void read_tmp112(FILE *gen_fp)
{

         bcm2835_i2c_setSlaveAddress(TMP112);

         int reg_temp,sign,i;
         int conv_time = 35; // conversion time for tmp112 is 35 mS max
         float reg_temp2;
         float reg_temp3;
         float reg_temp4;
         unsigned char bufreg_temp[] = {0x00, 0x00};

         for(i=0; i<2; i++)
         {
         bcm2835_i2c_read(bufreg_temp, sizeof(bufreg_temp));
         sign = bufreg_temp[0] & 0x40;         // capture sign bit
         if(sign == 0x40)
         {
         bufreg_temp[0] = ~bufreg_temp[0];
         bufreg_temp[1] = ~bufreg_temp[1];
         reg_temp = (bufreg_temp[1]) + (bufreg_temp[0]*0x100);
         reg_temp2 = (float)reg_temp / (float)0x10; // left justify 12 bits
         reg_temp2 = reg_temp2 + 1;  // two's compliment
         reg_temp3 = reg_temp2 * (-1) * 0.0625;         // convert to celsius
         } // end if
         else
         {
         reg_temp = (bufreg_temp[1]) + (bufreg_temp[0]*0x100);
         reg_temp2 = (float)reg_temp / (float)0x10; // left justify 12 bits
         reg_temp3 = reg_temp2 * 0.0625;       // convert to celsius
         } // end else

         bcm2835_delay(conv_time);
         } // end for loop

         fprintf(gen_fp, "%.2f,",reg_temp3);
         // save external temperature in data file

} // end read_tmp112

float read_tmp112_ext(FILE *gen_fp)
{

         bcm2835_i2c_setSlaveAddress(TMP112_X);

         int ext_temp,x_sign,x_i;
```

```
        int x_conv_time = 35; // conversion time for tmp112 is 35 mS max
        float ext_temp2;
        float ext_temp3;
        float ext_temp4;
        unsigned char bufext_temp[] = {0x00, 0x00};

        for(x_i=0; x_i<2; x_i++)
        {
        bcm2835_i2c_read(bufext_temp, sizeof(bufext_temp));
        x_sign = bufext_temp[0] & 0x40;      // capture sign bit
        if(x_sign == 0x40)
        {
        bufext_temp[0] = ~bufext_temp[0];
        bufext_temp[1] = ~bufext_temp[1];
        ext_temp = (bufext_temp[1]) + (bufext_temp[0]*0x100);
        ext_temp2 = (float)ext_temp / (float)0x10; // left justify 12 bits
        ext_temp2 = ext_temp2 + 1;  // two's compliment
        ext_temp3 = ext_temp2 * (-1) * 0.0625;         // convert to celsius
        } // end if
        else
        {
        ext_temp = (bufext_temp[1]) + (bufext_temp[0]*0x100);
        ext_temp2 = (float)ext_temp / (float)0x10; // left justify 12 bits
        ext_temp3 = ext_temp2 * 0.0625;      // convert to celsius
        } // end else

        bcm2835_delay(x_conv_time);
        } // end for loop

        fprintf(gen_fp, "%.2f,",ext_temp3);
        // save external temperature in data file

        return ext_temp3;

} // end read_tmp112


void read_mpu6000(FILE *gen_fp)
{
        bcm2835_i2c_setSlaveAddress(MPU6000);

        // Data Registers on MPU6000 chip

        unsigned char accel_xout_high[]      = {0x3b};
        // x-axis accelerometer measurement [15:8] register
        unsigned char accel_xout_low[]       = {0x3c};
        // x-axis accelerometer measurement [7:0] register
        unsigned char accel_yout_high[]      = {0x3d};
        // y-axis accelerometer measurement [15:8] register
        unsigned char accel_yout_low[]       = {0x3e};
        // y-axis accelerometer measurement [7:0] register
        unsigned char accel_zout_high[]      = {0x3f};
        // z-axis accelerometer measurement [15:8] register
        unsigned char accel_zout_low[]       = {0x40};
        // z-axis accelerometer measurement [7:0] register

        unsigned char gyro_xout_high[]       = {0x43};
        // x-axis gyroscope measurement [15:8] register
        unsigned char gyro_xout_low[]        = {0x44};
        // x-axis gyroscope measurement [7:0] register
        unsigned char gyro_yout_high[]       = {0x45};
        // y-axis gyroscope measurement [15:8] register
        unsigned char gyro_yout_low[]        = {0x46};
        // y_axis gyroscope measurement [7:0] register
        unsigned char gyro_zout_high[]       = {0x47};
        // z-axis gyroscope measurement [15:8] register
        unsigned char gyro_zout_low[]        = {0x48};
        // z-axis gyroscope measurement [7:0] register

        unsigned char temp_out_high[]        = {0x41};
        // temperature measurement [15:8] register
        unsigned char temp_out_low[]= {0x42};
        // temperature measurement [7:0] register

        // Commands

        unsigned char wake_up[]              = {0x6b, 0x03};
        // command to restart power management and use z_gyro reference clock
        unsigned char flscl_2g[]    = {0x1c, 0x00};
        // command to set accelerometer to +/- 2g full scale
        unsigned char flscl_8g[]    = {0x1c, 0x10};
        // command to set accelerometer to +/- 8g full scale
```

```
        unsigned char flscl_250[]   = {0x1b, 0x00};
        // command to set gyrometer to +/- 250 d/s full scale
        unsigned char flscl_1000[]  = {0x1b, 0x10};
        // command set gyrometer to +/- 1000 d/s full scale

        // Data buffers for reading MPU6000 chip

        unsigned char accel_data[6];
        unsigned char gyro_data[6];
        unsigned char temp_data[2];

        unsigned int accel_x_uint32, accel_y_uint32, accel_z_uint32;
        unsigned int gyro_x_uint32, gyro_y_uint32, gyro_z_uint32;
        float accel_x_flt, accel_y_flt, accel_z_flt;
        float gyro_x_flt, gyro_y_flt, gyro_z_flt;

        signed int temp_int;
        float temp_flt;
        int q;


        // restart the mpu6000 and set CLKSEL to 3 to use Z axis gyroscope reference
        bcm2835_i2c_write(wake_up, sizeof(wake_up));

        // set accelerometer configuration to +/- 2g
        bcm2835_i2c_write(flscl_2g, sizeof(flscl_2g));

        // set gyroscope configuration to +/- 250 d/s
        bcm2835_i2c_write(flscl_250, sizeof(flscl_250));

        float acc_sens = 16384;
        float gyr_sens = 131;

        for(q=0;q<2;q++)
        {

        // burst-read accelerometer data starting at 0x3b and ending at 0x40
        bcm2835_i2c_read_register_rs(accel_xout_high, accel_data, sizeof(accel_data));

        // burst-read gyroscope data starting at 0x43 and ending at 0x48
        bcm2835_i2c_read_register_rs(gyro_xout_high, gyro_data, sizeof(gyro_data));

        // burst-read temperature data starting at 0x41 and ending at 0x42
        bcm2835_i2c_read_register_rs(temp_out_high, temp_data, sizeof(temp_data));

        // compute acclerometer reading 16 bit two's compliment values our of register
        accel_x_uint32 = ((unsigned int)accel_data[0] << 8) | (accel_data[1]);
        if (accel_x_uint32 >= 32768)
                 accel_x_uint32 -= 65536;
        accel_x_flt = (int)accel_x_uint32 / acc_sens;

        accel_y_uint32 = ((unsigned int)accel_data[2] << 8) | (accel_data[3]);
        if (accel_y_uint32 >= 32768)
                 accel_y_uint32 -= 65536;
        accel_y_flt = (int)accel_y_uint32 / acc_sens;

        accel_z_uint32 = ((unsigned int)accel_data[4] << 8) | (accel_data[5]);
        if (accel_z_uint32 >= 32768)
                 accel_z_uint32 -= 65536;
        accel_z_flt = (int)accel_z_uint32 / acc_sens;

        // compute gyrometer reading
        gyro_x_uint32 = ((unsigned int)gyro_data[0] << 8) | (gyro_data[1]);
        if (gyro_x_uint32 >= 32768)
                 gyro_x_uint32 -= 65536;
        gyro_x_flt = (int)gyro_x_uint32 / gyr_sens;

        gyro_y_uint32 = ((unsigned int)gyro_data[2] << 8) | (gyro_data[3]);
        if (gyro_y_uint32 >= 32768)
                 gyro_y_uint32 -= 65536;
        gyro_y_flt = (int)gyro_y_uint32 / gyr_sens;

        gyro_z_uint32 = ((unsigned int)gyro_data[4] << 8) | (gyro_data[5]);
        if (gyro_z_uint32 >= 32768)
                 gyro_z_uint32 -= 65536;
        gyro_z_flt = (int)gyro_z_uint32 / gyr_sens;

        fprintf(gen_fp, "%.2f,%.2f,%.2f,%.2f,%.2f,%.2f,", accel_x_flt, accel_y_flt, accel_z_flt, gyro_x_flt, gyro_y_flt,
gyro_z_flt);

        // set acceleromter configureation to +/- 8g
        bcm2835_i2c_write(flscl_8g, sizeof(flscl_8g));
```

```
        // set gyroscope configuration to +/- 1000 d/s
        bcm2835_i2c_write(flscl_1000, sizeof(flscl_1000));

        acc_sens = 4096;
        gyr_sens = 32.8;

        } // end for loop

} // end read_mpu6000

void read_hmc5883(FILE *gen_fp)
{
        bcm2835_i2c_setSlaveAddress(HMC5883);

        unsigned char mag_buf[6];
        // buffer to hold data from magnetometer registers 0x03 to 0x08
        unsigned char mag_reg[] = {0x03};
        // first addresss of magnetomter registers 0x03 to 0x08

        unsigned int mag_x_uint32,mag_y_uint32,mag_z_uint32;
        float mag_x_flt,mag_y_flt,mag_z_flt;

        float mag_sens = 390;
        // for gain = 5 sensitivity = 390 LSb/Gauss

        // commands
        unsigned char samp_rate_8_15[] = {0x00,0x70};
        // 8 averaged samples, 15 Hz, normal measurement
        unsigned char gain_5[]                = {0x01,0xa0};
        // set gain to 5
        unsigned char cont_mode[]    = {0x02,0x00};
        // set to continuous measurement mode
        unsigned char single_mode[] = {0x02,0x01};
        // set to single-measurement mode

        // set sample rate
        bcm2835_i2c_write(samp_rate_8_15, sizeof(samp_rate_8_15));

        // set gain
        bcm2835_i2c_write(gain_5, sizeof(gain_5));

        // set measurement mode
        bcm2835_i2c_write(cont_mode, sizeof(single_mode));

        bcm2835_delay(10);

        // read magnetometer
        bcm2835_i2c_read_register_rs(mag_reg, mag_buf, sizeof(mag_buf));

        // compute readings
        mag_x_uint32 = ((unsigned int)mag_buf[0] << 8) | (mag_buf[1]);
        if (mag_x_uint32 >= 32768)
                mag_x_uint32 -= 65536;
        mag_x_flt = (int)mag_x_uint32 / mag_sens;

        mag_y_uint32 = ((unsigned int)mag_buf[2] << 8) | (mag_buf[3]);
        if (mag_y_uint32 >= 32768)
                mag_y_uint32 -= 65536;
        mag_y_flt = (int)mag_y_uint32 / mag_sens;

        mag_z_uint32 = ((unsigned int)mag_buf[4] << 8) | (mag_buf[5]);
        if (mag_z_uint32 >= 32768)
                mag_z_uint32 -= 65536;
        mag_z_flt = (int)mag_z_uint32 / mag_sens;

        fprintf(gen_fp,"%.2f,%.2f,%.2f,",mag_x_flt,mag_y_flt,mag_z_flt);

}// end read_hmc5883

void read_ds3231(FILE *gen_fp)
{
        bcm2835_i2c_setSlaveAddress(DS3231);

        unsigned char time_buf[3];
        // buffer to hold data from time registers 0x00 to 0x02

        unsigned char time_reg[] = {0x00};
        // first address of time registers 0x00 to 0x02

        unsigned int bcd_sec, bcd_min, bcd_hr;
```

```
        bcm2835_i2c_read_register_rs(time_reg, time_buf, sizeof(time_buf));

        bcd_sec = (((time_buf[0] & 0xf0)/0x10) * 10) + (time_buf[0] & 0x0f);
        bcd_min = (((time_buf[1] & 0xf0)/0x10) * 10) + (time_buf[1] & 0x0f);
        bcd_hr   = (((time_buf[2] & 0xf0)/0x10) * 10) + (time_buf[2] & 0x0f);

        fprintf(gen_fp,"%.2d:%.2d:%.2d,", bcd_hr,bcd_min,bcd_sec);

}//end read_ds3231

void get_date(FILE *gen_fp)
{
        bcm2835_i2c_setSlaveAddress(DS3231);

        unsigned char date_buf[3];
        // buffer to hold data from date registers 0x04 to 0x06

        unsigned char date_reg[] = {0x04};
        // first address of date registers 0x04 to 0x06

        unsigned int bcd_dy, bcd_mnth, bcd_yr;

        bcm2835_i2c_read_register_rs(date_reg, date_buf, sizeof(date_buf));

        bcd_dy = (((date_buf[0] & 0xf0)/0x10) * 10) + (date_buf[0] & 0x0f);
        bcd_mnth = (((date_buf[1] & 0xf0)/0x10) * 10) + (date_buf[1] & 0x0f);
        bcd_yr   = (((date_buf[2] & 0xf0)/0x10) * 10) + (date_buf[2] & 0x0f);

        fprintf(gen_fp,"Date: %.2d/%.2d/%.2d \n",bcd_mnth,bcd_dy,bcd_yr);
}

void read_dynamics(FILE *gen_fp)
{

        read_ds3231(gen_fp);        // read time from RTC for dynamics data
        read_mpu6000(gen_fp);       // read accelerometer/gyrometer
        read_hmc5883(gen_fp);       // read magnetometer
        fprintf(gen_fp,"\n");


}// end read_dynamics
```

GPS Code

```
// GPS.c
// Michael Petersen
// 11 October 2013
// This code is written to access the XR20M1280 UART to I2C which is used
// to read the GARMIN 15xL GPS module
// This code sets the baud rate to 4800 to match the GPS module. It also
// sets up the holding registers and begins reading.

#include<stdio.h>
#include<bcm2835.h>

unsigned char UART_MOD          = 0x35;         // address from i2c detect
unsigned char setLCR[]          = {0x18,0x03};  // one stop bit with set 8-bit words
unsigned char clearFIFO[]       = {0x10,0x07};  // reset Tx, Rx, FIFOS
unsigned char setFCR[]          = {0x10,0x01};  // enable FIFOs
unsigned char setLCR_7[]        = {0x18, 0x83}; // set LCR[7] = 1 to set divisor
unsigned char setDLL[]          = {0x00,0x34};  // Set the Divisor LSB register to 52 = 0x34

unsigned char LCR = 0x18;       // Line Control Register
unsigned char RHR = 0x00;       // Receive Holding Register when LCR[7] = 0
unsigned char THR = 0x00;       // Transmit Holding Register when LCR[7] = 0
unsigned char LSR = 0x28;       // Line Status Register when LCR != 0xBF
unsigned char SFR = 0x30;       // Special Function Register when LCR != 0xBF

int main(void)
{
        unsigned char read[1];
    unsigned char write[2];
    unsigned int i;
        unsigned int rx_ready, over_run, parity_error, framing_error, rx_break, thr_empty, thr_tsr_empty,
data_error;

        FILE *gps_fp;

        gps_fp = fopen("/home/pi/gps_data.csv", "a"); // open file to store
        // GPS data

        if (!bcm2835_init())
        return 1;

        bcm2835_i2c_begin();
        bcm2835_i2c_setClockDivider(BCM2835_I2C_CLOCK_DIVIDER_626); // 400 Khz "High Speed"


        bcm2835_i2c_setSlaveAddress(UART_MOD);

        // Set LCR[7] = 1 for changing divisor
        bcm2835_i2c_write(setLCR_7, sizeof(setLCR_7));

        // Set divisor LSB to 52
        bcm2835_i2c_write(setDLL, sizeof(setDLL));

        // Set LCR[7] = 0 for writing to FCR and LSR
        bcm2835_i2c_write(setLCR, sizeof(setLCR));

        // Clear and enable FIFOs
        bcm2835_i2c_write(clearFIFO, sizeof(clearFIFO));

        bcm2835_i2c_read_register_rs(&SFR, read, sizeof(read));

        bcm2835_i2c_read_register_rs(&LSR, read, sizeof(read));

        rx_ready = (read[0] & 0x01);
        thr_empty = (read[0] & 0x20) >> 5;


while(1) // infinite for testing
{

        while(rx_ready == 0) // wait for receive byte to be ready
```

```
        {
        bcm2835_i2c_read_register_rs(&LSR, read, sizeof(read));
        rx_ready = (read[0] & 0x01);
        }

        bcm2835_i2c_read_register_rs(&LSR, read, sizeof(read));
    data_error = (read[0] & 0x80) >> 7;
        thr_tsr_empty = (read[0] & 0x40) >> 6;
        thr_empty = (read[0] & 0x20) >> 5;
        rx_break = (read[0] & 0x10) >> 4;
        framing_error = (read[0] & 0x08) >> 3;
        parity_error = (read[0] & 0x04) >> 2;
        over_run = (read[0] & 0x02) >> 1;
        rx_ready = (read[0] & 0x01);

        if(data_error == 1)
        printf("data_error!\n");

        if(rx_break == 1)
        printf("break error!\n");

        if(framing_error == 1)
        printf("framing error!\n");

        if(parity_error ==1)
        printf("parity error!\n");

        if(over_run == 1)
        printf("over run error!\n");

        // read from RHR register
        bcm2835_i2c_read_register_rs(&RHR, read, sizeof(read));

        fprintf(gps_fp,"%c", read[0]); // print received byte to file

        bcm2835_i2c_read_register_rs(&LSR, read, sizeof(read));
        rx_ready = (read[0] & 0x01);

}

        fclose(gps_fp);

        bcm2835_i2c_end();
        bcm2835_close();
return 0;
}
```

Appendix C

C Code Libraries

Several libraries were required to complete the MSAv4 code. The read_channel_press() function requires a logarithmic function for processing pressure data in the non-linear region of the sensor. The -m library for math.h provides the necessary logarithmic function. The timing and scheduling functions are POSIX threads which require the use of pthread_create(), post_sem, and sem_wait(). These functions belong to the -pthread library for the pthread.h and semaphore.h header files. The only non-standard library used for the MSAv4 is the -bcm2835 library, which is used with bcm2835.h.

BCM2835 C Library

This library is written and maintained by Mike McCauley. All of the source code is available on Mike McCauley's website at:

<http://www.airspayce.com/mikem/bcm2835/>.

This library is licensed under Open Source GNU and is available for free use and modification. The following is the main source code for the library. It details the I2C and GPIO functions that were used in the design of the MSAv4. As of November 11, 2013 this library is continually updated as new functionality is developed.

```
// bcm2835.c
// C and C++ support for Broadcom BCM 2835 as used in Raspberry Pi
// http://elinux.org/RPi_Low-level_peripherals
// http://www.raspberrypi.org/wp-content/uploads/2012/02/BCM2835-ARM-Peripherals.pdf
//
// Author: Mike McCauley
// Copyright (C) 2011-2013 Mike McCauley
// $Id: bcm2835.c,v 1.12 2013/09/01 00:56:56 mikem Exp mikem $

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <string.h>
#include <time.h>
#include <unistd.h>

#include "bcm2835.h"

// This define enables a little test program (by default a blinking output on pin RPI_GPIO_PIN_11)
// You can do some safe, non-destructive testing on any platform with:
// gcc bcm2835.c -D BCM2835_TEST
// ./a.out
//#define BCM2835_TEST

// Uncommenting this define compiles alternative I2C code for the version 1 RPi
// The P1 header I2C pins are connected to SDA0 and SCL0 on V1.
// By default I2C code is generated for the V2 RPi which has SDA1 and SCL1 connected.
// #define I2C_V1

// Pointers to the hardware register bases
volatile uint32_t *bcm2835_gpio = MAP_FAILED;
volatile uint32_t *bcm2835_pwm  = MAP_FAILED;
volatile uint32_t *bcm2835_clk  = MAP_FAILED;
volatile uint32_t *bcm2835_pads = MAP_FAILED;
volatile uint32_t *bcm2835_spi0 = MAP_FAILED;
volatile uint32_t *bcm2835_bsc0 = MAP_FAILED;
volatile uint32_t *bcm2835_bsc1 = MAP_FAILED;
```

```
volatile uint32_t *bcm2835_st        = MAP_FAILED;


// This variable allows us to test on hardware other than RPi.
// It prevents access to the kernel memory, and does not do any peripheral access
// Instead it prints out what it _would_ do if debug were 0
static uint8_t debug = 0;

// I2C The time needed to transmit one byte. In microseconds.
static int i2c_byte_wait_us = 0;

//
// Low level register access functions
//

void  bcm2835_set_debug(uint8_t d)
{
    debug = d;
}

// safe read from peripheral
uint32_t bcm2835_peri_read(volatile uint32_t* paddr)
{
    if (debug)
    {
        printf("bcm2835_peri_read  paddr %08X\n", (unsigned) paddr);
         return 0;
    }
    else
    {
         // Make sure we dont return the _last_ read which might get lost
         // if subsequent code changes to a different peripheral
         uint32_t ret = *paddr;
         *paddr; // Read without assigneing to an unused variable
         return ret;
    }
}

// read from peripheral without the read barrier
uint32_t bcm2835_peri_read_nb(volatile uint32_t* paddr)
{
    if (debug)
    {
         printf("bcm2835_peri_read_nb  paddr %08X\n", (unsigned) paddr);
         return 0;
    }
    else
    {
         return *paddr;
    }
}

// safe write to peripheral
void bcm2835_peri_write(volatile uint32_t* paddr, uint32_t value)
{
  if (debug)
    {
         printf("bcm2835_peri_write paddr %08X, value %08X\n", (unsigned) paddr, value);
    }
  else
    {
         // Make sure we don't rely on the first write, which may get
         // lost if the previous access was to a different peripheral.
         *paddr = value;
         *paddr = value;
    }
}

// write to peripheral without the write barrier
void bcm2835_peri_write_nb(volatile uint32_t* paddr, uint32_t value)
{
    if (debug)
    {
         printf("bcm2835_peri_write_nb paddr %08X, value %08X\n",
               (unsigned) paddr, value);
    }
    else
    {
         *paddr = value;
    }
}
```

```
// Set/clear only the bits in value covered by the mask
void bcm2835_peri_set_bits(volatile uint32_t* paddr, uint32_t value, uint32_t mask)
{
    uint32_t v = bcm2835_peri_read(paddr);
    v = (v & ~mask) | (value & mask);
    bcm2835_peri_write(paddr, v);
}

//
// Low level convenience functions
//

// Function select
// pin is a BCM2835 GPIO pin number NOT RPi pin number
//      There are 6 control registers, each control the functions of a block
//      of 10 pins.
//      Each control register has 10 sets of 3 bits per GPIO pin:
//
//      000 = GPIO Pin X is an input
//      001 = GPIO Pin X is an output
//      100 = GPIO Pin X takes alternate function 0
//      101 = GPIO Pin X takes alternate function 1
//      110 = GPIO Pin X takes alternate function 2
//      111 = GPIO Pin X takes alternate function 3
//      011 = GPIO Pin X takes alternate function 4
//      010 = GPIO Pin X takes alternate function 5
//
// So the 3 bits for port X are:
//      X / 10 + ((X % 10) * 3)
void bcm2835_gpio_fsel(uint8_t pin, uint8_t mode)
{
    // Function selects are 10 pins per 32 bit word, 3 bits per pin
    volatile uint32_t* paddr = bcm2835_gpio + BCM2835_GPFSEL0/4 + (pin/10);
    uint8_t   shift = (pin % 10) * 3;
    uint32_t  mask = BCM2835_GPIO_FSEL_MASK << shift;
    uint32_t  value = mode << shift;
    bcm2835_peri_set_bits(paddr, value, mask);
}

// Set output pin
void bcm2835_gpio_set(uint8_t pin)
{
    volatile uint32_t* paddr = bcm2835_gpio + BCM2835_GPSET0/4 + pin/32;
    uint8_t shift = pin % 32;
    bcm2835_peri_write(paddr, 1 << shift);
}

// Clear output pin
void bcm2835_gpio_clr(uint8_t pin)
{
    volatile uint32_t* paddr = bcm2835_gpio + BCM2835_GPCLR0/4 + pin/32;
    uint8_t shift = pin % 32;
    bcm2835_peri_write(paddr, 1 << shift);
}

// Set all output pins in the mask
void bcm2835_gpio_set_multi(uint32_t mask)
{
    volatile uint32_t* paddr = bcm2835_gpio + BCM2835_GPSET0/4;
    bcm2835_peri_write(paddr, mask);
}

// Clear all output pins in the mask
void bcm2835_gpio_clr_multi(uint32_t mask)
{
    volatile uint32_t* paddr = bcm2835_gpio + BCM2835_GPCLR0/4;
    bcm2835_peri_write(paddr, mask);
}

// Read input pin
uint8_t bcm2835_gpio_lev(uint8_t pin)
{
    volatile uint32_t* paddr = bcm2835_gpio + BCM2835_GPLEV0/4 + pin/32;
    uint8_t shift = pin % 32;
    uint32_t value = bcm2835_peri_read(paddr);
    return (value & (1 << shift)) ? HIGH : LOW;
}

// See if an event detection bit is set
// Sigh cant support interrupts yet
uint8_t bcm2835_gpio_eds(uint8_t pin)
{
```

```
    volatile uint32_t* paddr = bcm2835_gpio + BCM2835_GPEDS0/4 + pin/32;
    uint8_t shift = pin % 32;
    uint32_t value = bcm2835_peri_read(paddr);
    return (value & (1 << shift)) ? HIGH : LOW;
}

// Write a 1 to clear the bit in EDS
void bcm2835_gpio_set_eds(uint8_t pin)
{
    volatile uint32_t* paddr = bcm2835_gpio + BCM2835_GPEDS0/4 + pin/32;
    uint8_t shift = pin % 32;
    uint32_t value = 1 << shift;
    bcm2835_peri_write(paddr, value);
}

// Rising edge detect enable
void bcm2835_gpio_ren(uint8_t pin)
{
    volatile uint32_t* paddr = bcm2835_gpio + BCM2835_GPREN0/4 + pin/32;
    uint8_t shift = pin % 32;
    uint32_t value = 1 << shift;
    bcm2835_peri_set_bits(paddr, value, value);
}
void bcm2835_gpio_clr_ren(uint8_t pin)
{
    volatile uint32_t* paddr = bcm2835_gpio + BCM2835_GPREN0/4 + pin/32;
    uint8_t shift = pin % 32;
    uint32_t value = 1 << shift;
    bcm2835_peri_set_bits(paddr, 0, value);
}

// Falling edge detect enable
void bcm2835_gpio_fen(uint8_t pin)
{
    volatile uint32_t* paddr = bcm2835_gpio + BCM2835_GPFEN0/4 + pin/32;
    uint8_t shift = pin % 32;
    uint32_t value = 1 << shift;
    bcm2835_peri_set_bits(paddr, value, value);
}
void bcm2835_gpio_clr_fen(uint8_t pin)
{
    volatile uint32_t* paddr = bcm2835_gpio + BCM2835_GPFEN0/4 + pin/32;
    uint8_t shift = pin % 32;
    uint32_t value = 1 << shift;
    bcm2835_peri_set_bits(paddr, 0, value);
}

// High detect enable
void bcm2835_gpio_hen(uint8_t pin)
{
    volatile uint32_t* paddr = bcm2835_gpio + BCM2835_GPHEN0/4 + pin/32;
    uint8_t shift = pin % 32;
    uint32_t value = 1 << shift;
    bcm2835_peri_set_bits(paddr, value, value);
}
void bcm2835_gpio_clr_hen(uint8_t pin)
{
    volatile uint32_t* paddr = bcm2835_gpio + BCM2835_GPHEN0/4 + pin/32;
    uint8_t shift = pin % 32;
    uint32_t value = 1 << shift;
    bcm2835_peri_set_bits(paddr, 0, value);
}

// Low detect enable
void bcm2835_gpio_len(uint8_t pin)
{
    volatile uint32_t* paddr = bcm2835_gpio + BCM2835_GPLEN0/4 + pin/32;
    uint8_t shift = pin % 32;
    uint32_t value = 1 << shift;
    bcm2835_peri_set_bits(paddr, value, value);
}
void bcm2835_gpio_clr_len(uint8_t pin)
{
    volatile uint32_t* paddr = bcm2835_gpio + BCM2835_GPLEN0/4 + pin/32;
    uint8_t shift = pin % 32;
    uint32_t value = 1 << shift;
    bcm2835_peri_set_bits(paddr, 0, value);
}

// Async rising edge detect enable
void bcm2835_gpio_aren(uint8_t pin)
{
```

```
    volatile uint32_t* paddr = bcm2835_gpio + BCM2835_GPAREN0/4 + pin/32;
    uint8_t shift = pin % 32;
    uint32_t value = 1 << shift;
    bcm2835_peri_set_bits(paddr, value, value);
}
void bcm2835_gpio_clr_aren(uint8_t pin)
{
    volatile uint32_t* paddr = bcm2835_gpio + BCM2835_GPAREN0/4 + pin/32;
    uint8_t shift = pin % 32;
    uint32_t value = 1 << shift;
    bcm2835_peri_set_bits(paddr, 0, value);
}

// Async falling edge detect enable
void bcm2835_gpio_afen(uint8_t pin)
{
    volatile uint32_t* paddr = bcm2835_gpio + BCM2835_GPAFEN0/4 + pin/32;
    uint8_t shift = pin % 32;
    uint32_t value = 1 << shift;
    bcm2835_peri_set_bits(paddr, value, value);
}
void bcm2835_gpio_clr_afen(uint8_t pin)
{
    volatile uint32_t* paddr = bcm2835_gpio + BCM2835_GPAFEN0/4 + pin/32;
    uint8_t shift = pin % 32;
    uint32_t value = 1 << shift;
    bcm2835_peri_set_bits(paddr, 0, value);
}

// Set pullup/down
void bcm2835_gpio_pud(uint8_t pud)
{
    volatile uint32_t* paddr = bcm2835_gpio + BCM2835_GPPUD/4;
    bcm2835_peri_write(paddr, pud);
}

// Pullup/down clock
// Clocks the value of pud into the GPIO pin
void bcm2835_gpio_pudclk(uint8_t pin, uint8_t on)
{
    volatile uint32_t* paddr = bcm2835_gpio + BCM2835_GPPUDCLK0/4 + pin/32;
    uint8_t shift = pin % 32;
    bcm2835_peri_write(paddr, (on ? 1 : 0) << shift);
}

// Read GPIO pad behaviour for groups of GPIOs
uint32_t bcm2835_gpio_pad(uint8_t group)
{
    volatile uint32_t* paddr = bcm2835_pads + BCM2835_PADS_GPIO_0_27/4 + group*2;
    return bcm2835_peri_read(paddr);
}

// Set GPIO pad behaviour for groups of GPIOs
// powerup value for al pads is
// BCM2835_PAD_SLEW_RATE_UNLIMITED | BCM2835_PAD_HYSTERESIS_ENABLED | BCM2835_PAD_DRIVE_8mA
void bcm2835_gpio_set_pad(uint8_t group, uint32_t control)
{
    volatile uint32_t* paddr = bcm2835_pads + BCM2835_PADS_GPIO_0_27/4 + group*2;
    bcm2835_peri_write(paddr, control | BCM2835_PAD_PASSWRD);
}

// Some convenient arduino-like functions
// milliseconds
void bcm2835_delay(unsigned int millis)
{
    struct timespec sleeper;

    sleeper.tv_sec  = (time_t)(millis / 1000);
    sleeper.tv_nsec = (long)(millis % 1000) * 1000000;
    nanosleep(&sleeper, NULL);
}

// microseconds
void bcm2835_delayMicroseconds(uint64_t micros)
{
    struct timespec t1;
    uint64_t        start;

    // Calling nanosleep() takes at least 100-200 us, so use it for
    // long waits and use a busy wait on the System Timer for the rest.
    start =  bcm2835_st_read();
```

```
    if (micros > 450)
    {
         t1.tv_sec = 0;
         t1.tv_nsec = 1000 * (long)(micros - 200);
         nanosleep(&t1, NULL);
    }

    bcm2835_st_delay(start, micros);
}

//
// Higher level convenience functions
//

// Set the state of an output
void bcm2835_gpio_write(uint8_t pin, uint8_t on)
{
    if (on)
         bcm2835_gpio_set(pin);
    else
         bcm2835_gpio_clr(pin);
}

// Set the state of a all 32 outputs in the mask to on or off
void bcm2835_gpio_write_multi(uint32_t mask, uint8_t on)
{
    if (on)
         bcm2835_gpio_set_multi(mask);
    else
         bcm2835_gpio_clr_multi(mask);
}

// Set the state of a all 32 outputs in the mask to the values in value
void bcm2835_gpio_write_mask(uint32_t value, uint32_t mask)
{
    bcm2835_gpio_set_multi(value & mask);
    bcm2835_gpio_clr_multi((~value) & mask);
}

// Set the pullup/down resistor for a pin
//
// The GPIO Pull-up/down Clock Registers control the actuation of internal pull-downs on
// the respective GPIO pins. These registers must be used in conjunction with the GPPUD
// register to effect GPIO Pull-up/down changes. The following sequence of events is
// required:
// 1. Write to GPPUD to set the required control signal (i.e. Pull-up or Pull-Down or neither
// to remove the current Pull-up/down)
// 2. Wait 150 cycles ? this provides the required set-up time for the control signal
// 3. Write to GPPUDCLK0/1 to clock the control signal into the GPIO pads you wish to
// modify ? NOTE only the pads which receive a clock will be modified, all others will
// retain their previous state.
// 4. Wait 150 cycles ? this provides the required hold time for the control signal
// 5. Write to GPPUD to remove the control signal
// 6. Write to GPPUDCLK0/1 to remove the clock
//
// RPi has P1-03 and P1-05 with 1k8 pullup resistor
void bcm2835_gpio_set_pud(uint8_t pin, uint8_t pud)
{
    bcm2835_gpio_pud(pud);
    delayMicroseconds(10);
    bcm2835_gpio_pudclk(pin, 1);
    delayMicroseconds(10);
    bcm2835_gpio_pud(BCM2835_GPIO_PUD_OFF);
    bcm2835_gpio_pudclk(pin, 0);
}

void bcm2835_spi_begin(void)
{
    // Set the SPI0 pins to the Alt 0 function to enable SPI0 access on them
    bcm2835_gpio_fsel(RPI_GPIO_P1_26, BCM2835_GPIO_FSEL_ALT0); // CE1
    bcm2835_gpio_fsel(RPI_GPIO_P1_24, BCM2835_GPIO_FSEL_ALT0); // CE0
    bcm2835_gpio_fsel(RPI_GPIO_P1_21, BCM2835_GPIO_FSEL_ALT0); // MISO
    bcm2835_gpio_fsel(RPI_GPIO_P1_19, BCM2835_GPIO_FSEL_ALT0); // MOSI
    bcm2835_gpio_fsel(RPI_GPIO_P1_23, BCM2835_GPIO_FSEL_ALT0); // CLK

    // Set the SPI CS register to the some sensible defaults
    volatile uint32_t* paddr = bcm2835_spi0 + BCM2835_SPI0_CS/4;
    bcm2835_peri_write(paddr, 0); // All 0s

    // Clear TX and RX fifos
    bcm2835_peri_write_nb(paddr, BCM2835_SPI0_CS_CLEAR);
}
```

```
void bcm2835_spi_end(void)
{
    // Set all the SPI0 pins back to input
    bcm2835_gpio_fsel(RPI_GPIO_P1_26, BCM2835_GPIO_FSEL_INPT); // CE1
    bcm2835_gpio_fsel(RPI_GPIO_P1_24, BCM2835_GPIO_FSEL_INPT); // CE0
    bcm2835_gpio_fsel(RPI_GPIO_P1_21, BCM2835_GPIO_FSEL_INPT); // MISO
    bcm2835_gpio_fsel(RPI_GPIO_P1_19, BCM2835_GPIO_FSEL_INPT); // MOSI
    bcm2835_gpio_fsel(RPI_GPIO_P1_23, BCM2835_GPIO_FSEL_INPT); // CLK
}

void bcm2835_spi_setBitOrder(uint8_t order)
{
    // BCM2835_SPI_BIT_ORDER_MSBFIRST is the only one suported by SPI0
}

// defaults to 0, which means a divider of 65536.
// The divisor must be a power of 2. Odd numbers
// rounded down. The maximum SPI clock rate is
// of the APB clock
void bcm2835_spi_setClockDivider(uint16_t divider)
{
    volatile uint32_t* paddr = bcm2835_spi0 + BCM2835_SPI0_CLK/4;
    bcm2835_peri_write(paddr, divider);
}

void bcm2835_spi_setDataMode(uint8_t mode)
{
    volatile uint32_t* paddr = bcm2835_spi0 + BCM2835_SPI0_CS/4;
    // Mask in the CPO and CPHA bits of CS
    bcm2835_peri_set_bits(paddr, mode << 2, BCM2835_SPI0_CS_CPOL | BCM2835_SPI0_CS_CPHA);
}

// Writes (and reads) a single byte to SPI
uint8_t bcm2835_spi_transfer(uint8_t value)
{
    volatile uint32_t* paddr = bcm2835_spi0 + BCM2835_SPI0_CS/4;
    volatile uint32_t* fifo = bcm2835_spi0 + BCM2835_SPI0_FIFO/4;

    // This is Polled transfer as per section 10.6.1
    // BUG ALERT: what happens if we get interupted in this section, and someone else
    // accesses a different peripheral?
    // Clear TX and RX fifos
    bcm2835_peri_set_bits(paddr, BCM2835_SPI0_CS_CLEAR, BCM2835_SPI0_CS_CLEAR);

    // Set TA = 1
    bcm2835_peri_set_bits(paddr, BCM2835_SPI0_CS_TA, BCM2835_SPI0_CS_TA);

    // Maybe wait for TXD
    while (!(bcm2835_peri_read(paddr) & BCM2835_SPI0_CS_TXD))
         ;

    // Write to FIFO, no barrier
    bcm2835_peri_write_nb(fifo, value);

    // Wait for DONE to be set
    while (!(bcm2835_peri_read_nb(paddr) & BCM2835_SPI0_CS_DONE))
         ;

    // Read any byte that was sent back by the slave while we sere sending to it
    uint32_t ret = bcm2835_peri_read_nb(fifo);

    // Set TA = 0, and also set the barrier
    bcm2835_peri_set_bits(paddr, 0, BCM2835_SPI0_CS_TA);

    return ret;
}

// Writes (and reads) an number of bytes to SPI
void bcm2835_spi_transfernb(char* tbuf, char* rbuf, uint32_t len)
{
    volatile uint32_t* paddr = bcm2835_spi0 + BCM2835_SPI0_CS/4;
    volatile uint32_t* fifo = bcm2835_spi0 + BCM2835_SPI0_FIFO/4;

    // This is Polled transfer as per section 10.6.1
    // BUG ALERT: what happens if we get interupted in this section, and someone else
    // accesses a different peripheral?

    // Clear TX and RX fifos
    bcm2835_peri_set_bits(paddr, BCM2835_SPI0_CS_CLEAR, BCM2835_SPI0_CS_CLEAR);

    // Set TA = 1
```

```
    bcm2835_peri_set_bits(paddr, BCM2835_SPI0_CS_TA, BCM2835_SPI0_CS_TA);

    uint32_t i;
    for (i = 0; i < len; i++)
    {
         // Maybe wait for TXD
         while (!(bcm2835_peri_read(paddr) & BCM2835_SPI0_CS_TXD))
             ;

         // Write to FIFO, no barrier
         bcm2835_peri_write_nb(fifo, tbuf[i]);

         // Wait for RXD
         while (!(bcm2835_peri_read(paddr) & BCM2835_SPI0_CS_RXD))
             ;

         // then read the data byte
         rbuf[i] = bcm2835_peri_read_nb(fifo);
    }
    // Wait for DONE to be set
    while (!(bcm2835_peri_read_nb(paddr) & BCM2835_SPI0_CS_DONE))
         ;

    // Set TA = 0, and also set the barrier
    bcm2835_peri_set_bits(paddr, 0, BCM2835_SPI0_CS_TA);
}

// Writes an number of bytes to SPI
void bcm2835_spi_writenb(char* tbuf, uint32_t len)
{
    volatile uint32_t* paddr = bcm2835_spi0 + BCM2835_SPI0_CS/4;
    volatile uint32_t* fifo = bcm2835_spi0 + BCM2835_SPI0_FIFO/4;

    // This is Polled transfer as per section 10.6.1
    // BUG ALERT: what happens if we get interupted in this section, and someone else
    // accesses a different peripheral?

    // Clear TX and RX fifos
    bcm2835_peri_set_bits(paddr, BCM2835_SPI0_CS_CLEAR, BCM2835_SPI0_CS_CLEAR);

    // Set TA = 1
    bcm2835_peri_set_bits(paddr, BCM2835_SPI0_CS_TA, BCM2835_SPI0_CS_TA);

    uint32_t i;
    for (i = 0; i < len; i++)
    {
         // Maybe wait for TXD
         while (!(bcm2835_peri_read(paddr) & BCM2835_SPI0_CS_TXD))
             ;

         // Write to FIFO, no barrier
         bcm2835_peri_write_nb(fifo, tbuf[i]);

         // Read from FIFO to prevent stalling
         while (bcm2835_peri_read(paddr) & BCM2835_SPI0_CS_RXD)
             (void) bcm2835_peri_read_nb(fifo);
    }

    // Wait for DONE to be set
    while (!(bcm2835_peri_read_nb(paddr) & BCM2835_SPI0_CS_DONE)) {
         while (bcm2835_peri_read(paddr) & BCM2835_SPI0_CS_RXD)
                   (void) bcm2835_peri_read_nb(fifo);
    };

    // Set TA = 0, and also set the barrier
    bcm2835_peri_set_bits(paddr, 0, BCM2835_SPI0_CS_TA);
}

// Writes (and reads) an number of bytes to SPI
// Read bytes are copied over onto the transmit buffer
void bcm2835_spi_transfern(char* buf, uint32_t len)
{
    bcm2835_spi_transfernb(buf, buf, len);
}

void bcm2835_spi_chipSelect(uint8_t cs)
{
    volatile uint32_t* paddr = bcm2835_spi0 + BCM2835_SPI0_CS/4;
    // Mask in the CS bits of CS
    bcm2835_peri_set_bits(paddr, cs, BCM2835_SPI0_CS_CS);
}
```

```
void bcm2835_spi_setChipSelectPolarity(uint8_t cs, uint8_t active)
{
    volatile uint32_t* paddr = bcm2835_spi0 + BCM2835_SPI0_CS/4;
    uint8_t shift = 21 + cs;
    // Mask in the appropriate CSPOLn bit
    bcm2835_peri_set_bits(paddr, active << shift, 1 << shift);
}

void bcm2835_i2c_begin(void)
{
#ifdef I2C_V1
    volatile uint32_t* paddr = bcm2835_bsc0 + BCM2835_BSC_DIV/4;
    // Set the I2C/BSC0 pins to the Alt 0 function to enable I2C access on them
    bcm2835_gpio_fsel(RPI_GPIO_P1_03, BCM2835_GPIO_FSEL_ALT0); // SDA
    bcm2835_gpio_fsel(RPI_GPIO_P1_05, BCM2835_GPIO_FSEL_ALT0); // SCL
#else
    volatile uint32_t* paddr = bcm2835_bsc1 + BCM2835_BSC_DIV/4;
    // Set the I2C/BSC1 pins to the Alt 0 function to enable I2C access on them
    bcm2835_gpio_fsel(RPI_V2_GPIO_P1_03, BCM2835_GPIO_FSEL_ALT0); // SDA
    bcm2835_gpio_fsel(RPI_V2_GPIO_P1_05, BCM2835_GPIO_FSEL_ALT0); // SCL
#endif

    // Read the clock divider register
    uint16_t cdiv = bcm2835_peri_read(paddr);
    // Calculate time for transmitting one byte
    // 1000000 = micros seconds in a second
    // 9 = Clocks per byte : 8 bits + ACK
    i2c_byte_wait_us = ((float)cdiv / BCM2835_CORE_CLK_HZ) * 1000000 * 9;
}

void bcm2835_i2c_end(void)
{
#ifdef I2C_V1
    // Set all the I2C/BSC0 pins back to input
    bcm2835_gpio_fsel(RPI_GPIO_P1_03, BCM2835_GPIO_FSEL_INPT); // SDA
    bcm2835_gpio_fsel(RPI_GPIO_P1_05, BCM2835_GPIO_FSEL_INPT); // SCL
#else
    // Set all the I2C/BSC1 pins back to input
    bcm2835_gpio_fsel(RPI_V2_GPIO_P1_03, BCM2835_GPIO_FSEL_INPT); // SDA
    bcm2835_gpio_fsel(RPI_V2_GPIO_P1_05, BCM2835_GPIO_FSEL_INPT); // SCL
#endif
}

void bcm2835_i2c_setSlaveAddress(uint8_t addr)
{
        // Set I2C Device Address
#ifdef I2C_V1
        volatile uint32_t* paddr = bcm2835_bsc0 + BCM2835_BSC_A/4;
#else
        volatile uint32_t* paddr = bcm2835_bsc1 + BCM2835_BSC_A/4;
#endif
        bcm2835_peri_write(paddr, addr);
}

// defaults to 0x5dc, should result in a 166.666 kHz I2C clock frequency.
// The divisor must be a power of 2. Odd numbers
// rounded down.
void bcm2835_i2c_setClockDivider(uint16_t divider)
{
#ifdef I2C_V1
    volatile uint32_t* paddr = bcm2835_bsc0 + BCM2835_BSC_DIV/4;
#else
    volatile uint32_t* paddr = bcm2835_bsc1 + BCM2835_BSC_DIV/4;
#endif
    bcm2835_peri_write(paddr, divider);
    // Calculate time for transmitting one byte
    // 1000000 = micros seconds in a second
    // 9 = Clocks per byte : 8 bits + ACK
    i2c_byte_wait_us = ((float)divider / BCM2835_CORE_CLK_HZ) * 1000000 * 9;
}

// set I2C clock divider by means of a baudrate number
void bcm2835_i2c_set_baudrate(uint32_t baudrate)
{
        uint32_t divider;
        // use 0xFFFE mask to limit a max value and round down any odd number
        divider = (BCM2835_CORE_CLK_HZ / baudrate) & 0xFFFE;
        bcm2835_i2c_setClockDivider( (uint16_t)divider );
}

// Writes an number of bytes to I2C
uint8_t bcm2835_i2c_write(const char * buf, uint32_t len)
```

```
{
#ifdef I2C_V1
    volatile uint32_t* dlen    = bcm2835_bsc0 + BCM2835_BSC_DLEN/4;
    volatile uint32_t* fifo    = bcm2835_bsc0 + BCM2835_BSC_FIFO/4;
    volatile uint32_t* status  = bcm2835_bsc0 + BCM2835_BSC_S/4;
    volatile uint32_t* control = bcm2835_bsc0 + BCM2835_BSC_C/4;
#else
    volatile uint32_t* dlen    = bcm2835_bsc1 + BCM2835_BSC_DLEN/4;
    volatile uint32_t* fifo    = bcm2835_bsc1 + BCM2835_BSC_FIFO/4;
    volatile uint32_t* status  = bcm2835_bsc1 + BCM2835_BSC_S/4;
    volatile uint32_t* control = bcm2835_bsc1 + BCM2835_BSC_C/4;
#endif

    uint32_t remaining = len;
    uint32_t i = 0;
    uint8_t reason = BCM2835_I2C_REASON_OK;

    // Clear FIFO
    bcm2835_peri_set_bits(control, BCM2835_BSC_C_CLEAR_1 , BCM2835_BSC_C_CLEAR_1 );
    // Clear Status
         bcm2835_peri_write_nb(status, BCM2835_BSC_S_CLKT | BCM2835_BSC_S_ERR | BCM2835_BSC_S_DONE);
         // Set Data Length
    bcm2835_peri_write_nb(dlen, len);
    // pre populate FIFO with max buffer
    while( remaining && ( i < BCM2835_BSC_FIFO_SIZE ) )
    {
        bcm2835_peri_write_nb(fifo, buf[i]);
        i++;
        remaining--;
    }

    // Enable device and start transfer
    bcm2835_peri_write_nb(control, BCM2835_BSC_C_I2CEN | BCM2835_BSC_C_ST);

    // Transfer is over when BCM2835_BSC_S_DONE
    while(!(bcm2835_peri_read_nb(status) & BCM2835_BSC_S_DONE ))
    {
        while ( remaining && (bcm2835_peri_read_nb(status) & BCM2835_BSC_S_TXD ))
         {
         // Write to FIFO, no barrier
         bcm2835_peri_write_nb(fifo, buf[i]);
         i++;
         remaining--;
         }
    }

    // Received a NACK
    if (bcm2835_peri_read(status) & BCM2835_BSC_S_ERR)
    {
                reason = BCM2835_I2C_REASON_ERROR_NACK;
    }

    // Received Clock Stretch Timeout
    else if (bcm2835_peri_read(status) & BCM2835_BSC_S_CLKT)
    {
                reason = BCM2835_I2C_REASON_ERROR_CLKT;
    }

    // Not all data are sent
    else if (remaining)
    {
                reason = BCM2835_I2C_REASON_ERROR_DATA;
    }

    bcm2835_peri_set_bits(control, BCM2835_BSC_S_DONE , BCM2835_BSC_S_DONE);

    return reason;
}

// Read an number of bytes from I2C
uint8_t bcm2835_i2c_read(char* buf, uint32_t len)
{
#ifdef I2C_V1
    volatile uint32_t* dlen    = bcm2835_bsc0 + BCM2835_BSC_DLEN/4;
    volatile uint32_t* fifo    = bcm2835_bsc0 + BCM2835_BSC_FIFO/4;
    volatile uint32_t* status  = bcm2835_bsc0 + BCM2835_BSC_S/4;
    volatile uint32_t* control = bcm2835_bsc0 + BCM2835_BSC_C/4;
#else
    volatile uint32_t* dlen    = bcm2835_bsc1 + BCM2835_BSC_DLEN/4;
    volatile uint32_t* fifo    = bcm2835_bsc1 + BCM2835_BSC_FIFO/4;
    volatile uint32_t* status  = bcm2835_bsc1 + BCM2835_BSC_S/4;
    volatile uint32_t* control = bcm2835_bsc1 + BCM2835_BSC_C/4;
```

```
#endif

    uint32_t remaining = len;
    uint32_t i = 0;
    uint8_t reason = BCM2835_I2C_REASON_OK;

    // Clear FIFO
    bcm2835_peri_set_bits(control, BCM2835_BSC_C_CLEAR_1 , BCM2835_BSC_C_CLEAR_1 );
    // Clear Status
         bcm2835_peri_write_nb(status, BCM2835_BSC_S_CLKT | BCM2835_BSC_S_ERR | BCM2835_BSC_S_DONE);
         // Set Data Length
    bcm2835_peri_write_nb(dlen, len);
    // Start read
    bcm2835_peri_write_nb(control, BCM2835_BSC_C_I2CEN | BCM2835_BSC_C_ST | BCM2835_BSC_C_READ);

    // wait for transfer to complete
    while (!(bcm2835_peri_read_nb(status) & BCM2835_BSC_S_DONE))
    {
        // we must empty the FIFO as it is populated and not use any delay
        while (bcm2835_peri_read_nb(status) & BCM2835_BSC_S_RXD)
         {
                  // Read from FIFO, no barrier
                  buf[i] = bcm2835_peri_read_nb(fifo);
         i++;
         remaining--;
         }
    }

    // transfer has finished - grab any remaining stuff in FIFO
    while (remaining && (bcm2835_peri_read_nb(status) & BCM2835_BSC_S_RXD))
    {
        // Read from FIFO, no barrier
        buf[i] = bcm2835_peri_read_nb(fifo);
        i++;
        remaining--;
    }

    // Received a NACK
    if (bcm2835_peri_read(status) & BCM2835_BSC_S_ERR)
    {
                  reason = BCM2835_I2C_REASON_ERROR_NACK;
    }

    // Received Clock Stretch Timeout
    else if (bcm2835_peri_read(status) & BCM2835_BSC_S_CLKT)
    {
                  reason = BCM2835_I2C_REASON_ERROR_CLKT;
    }

    // Not all data are received
    else if (remaining)
    {
                  reason = BCM2835_I2C_REASON_ERROR_DATA;
    }

    bcm2835_peri_set_bits(control, BCM2835_BSC_S_DONE , BCM2835_BSC_S_DONE);

    return reason;
}

// Read an number of bytes from I2C sending a repeated start after writing
// the required register. Only works if your device supports this mode
uint8_t bcm2835_i2c_read_register_rs(char* regaddr, char* buf, uint32_t len)
{
#ifdef I2C_V1
    volatile uint32_t* dlen    = bcm2835_bsc0 + BCM2835_BSC_DLEN/4;
    volatile uint32_t* fifo    = bcm2835_bsc0 + BCM2835_BSC_FIFO/4;
    volatile uint32_t* status  = bcm2835_bsc0 + BCM2835_BSC_S/4;
    volatile uint32_t* control = bcm2835_bsc0 + BCM2835_BSC_C/4;
#else
    volatile uint32_t* dlen    = bcm2835_bsc1 + BCM2835_BSC_DLEN/4;
    volatile uint32_t* fifo    = bcm2835_bsc1 + BCM2835_BSC_FIFO/4;
    volatile uint32_t* status  = bcm2835_bsc1 + BCM2835_BSC_S/4;
    volatile uint32_t* control = bcm2835_bsc1 + BCM2835_BSC_C/4;
#endif
         uint32_t remaining = len;
    uint32_t i = 0;
    uint8_t reason = BCM2835_I2C_REASON_OK;

    // Clear FIFO
    bcm2835_peri_set_bits(control, BCM2835_BSC_C_CLEAR_1 , BCM2835_BSC_C_CLEAR_1 );
    // Clear Status
```

```
          bcm2835_peri_write_nb(status, BCM2835_BSC_S_CLKT | BCM2835_BSC_S_ERR | BCM2835_BSC_S_DONE);
          // Set Data Length
    bcm2835_peri_write_nb(dlen, 1);
    // Enable device and start transfer
    bcm2835_peri_write_nb(control, BCM2835_BSC_C_I2CEN);
    bcm2835_peri_write_nb(fifo, regaddr[0]);
    bcm2835_peri_write_nb(control, BCM2835_BSC_C_I2CEN | BCM2835_BSC_C_ST);

    // poll for transfer has started
    while ( !( bcm2835_peri_read_nb(status) & BCM2835_BSC_S_TA ) )
    {
        // Linux may cause us to miss entire transfer stage
        if(bcm2835_peri_read(status) & BCM2835_BSC_S_DONE)
            break;
    }

    // Send a repeated start with read bit set in address
    bcm2835_peri_write_nb(dlen, len);
    bcm2835_peri_write_nb(control, BCM2835_BSC_C_I2CEN | BCM2835_BSC_C_ST  | BCM2835_BSC_C_READ );

    // Wait for write to complete and first byte back.
    bcm2835_delayMicroseconds(i2c_byte_wait_us * 3);

    // wait for transfer to complete
    while (!(bcm2835_peri_read_nb(status) & BCM2835_BSC_S_DONE))
    {
        // we must empty the FIFO as it is populated and not use any delay
        while (remaining && bcm2835_peri_read_nb(status) & BCM2835_BSC_S_RXD)
          {
                   // Read from FIFO, no barrier
                   buf[i] = bcm2835_peri_read_nb(fifo);
          i++;
          remaining--;
          }
    }

    // transfer has finished - grab any remaining stuff in FIFO
    while (remaining && (bcm2835_peri_read_nb(status) & BCM2835_BSC_S_RXD))
    {
        // Read from FIFO, no barrier
        buf[i] = bcm2835_peri_read_nb(fifo);
        i++;
        remaining--;
    }

    // Received a NACK
    if (bcm2835_peri_read(status) & BCM2835_BSC_S_ERR)
    {
                   reason = BCM2835_I2C_REASON_ERROR_NACK;
    }

    // Received Clock Stretch Timeout
    else if (bcm2835_peri_read(status) & BCM2835_BSC_S_CLKT)
    {
                   reason = BCM2835_I2C_REASON_ERROR_CLKT;
    }

    // Not all data are sent
    else if (remaining)
    {
                   reason = BCM2835_I2C_REASON_ERROR_DATA;
    }

    bcm2835_peri_set_bits(control, BCM2835_BSC_S_DONE , BCM2835_BSC_S_DONE);

    return reason;
}

// Read the System Timer Counter (64-bits)
uint64_t bcm2835_st_read(void)
{
    volatile uint32_t* paddr;
    uint64_t st;
    paddr = bcm2835_st + BCM2835_ST_CHI/4;
    st = bcm2835_peri_read(paddr);
    st <<= 32;
    paddr = bcm2835_st + BCM2835_ST_CLO/4;
    st += bcm2835_peri_read(paddr);
    return st;
}

// Delays for the specified number of microseconds with offset
```

```
void bcm2835_st_delay(uint64_t offset_micros, uint64_t micros)
{
    uint64_t compare = offset_micros + micros;

    while(bcm2835_st_read() < compare)
         ;
}

// PWM

void bcm2835_pwm_set_clock(uint32_t divisor)
{
  // From Gerts code
  divisor &= 0xfff;
  // Stop PWM clock
  bcm2835_peri_write(bcm2835_clk + BCM2835_PWMCLK_CNTL, BCM2835_PWM_PASSWRD | 0x01);
  bcm2835_delay(110); // Prevents clock going slow
  // Wait for the clock to be not busy
  while ((bcm2835_peri_read(bcm2835_clk + BCM2835_PWMCLK_CNTL) & 0x80) != 0)
    bcm2835_delay(1);
  // set the clock divider and enable PWM clock
  bcm2835_peri_write(bcm2835_clk + BCM2835_PWMCLK_DIV, BCM2835_PWM_PASSWRD | (divisor << 12));
  bcm2835_peri_write(bcm2835_clk + BCM2835_PWMCLK_CNTL, BCM2835_PWM_PASSWRD | 0x11); // Source=osc and enable
}

void bcm2835_pwm_set_mode(uint8_t channel, uint8_t markspace, uint8_t enabled)
{
  uint32_t control = bcm2835_peri_read(bcm2835_pwm + BCM2835_PWM_CONTROL);

  if (channel == 0)
    {
      if (markspace)
          control |= BCM2835_PWM0_MS_MODE;
      else
          control &= ~BCM2835_PWM0_MS_MODE;
      if (enabled)
          control |= BCM2835_PWM0_ENABLE;
      else
          control &= ~BCM2835_PWM0_ENABLE;
    }
  else if (channel == 1)
    {
      if (markspace)
          control |= BCM2835_PWM1_MS_MODE;
      else
          control &= ~BCM2835_PWM1_MS_MODE;
      if (enabled)
          control |= BCM2835_PWM1_ENABLE;
      else
          control &= ~BCM2835_PWM1_ENABLE;
    }

  // If you use the barrier here, wierd things happen, and the commands dont work
  bcm2835_peri_write_nb(bcm2835_pwm + BCM2835_PWM_CONTROL, control);
  //  bcm2835_peri_write_nb(bcm2835_pwm + BCM2835_PWM_CONTROL, BCM2835_PWM0_ENABLE | BCM2835_PWM1_ENABLE |
BCM2835_PWM0_MS_MODE | BCM2835_PWM1_MS_MODE);

}

void bcm2835_pwm_set_range(uint8_t channel, uint32_t range)
{
  if (channel == 0)
      bcm2835_peri_write_nb(bcm2835_pwm + BCM2835_PWM0_RANGE, range);
  else if (channel == 1)
      bcm2835_peri_write_nb(bcm2835_pwm + BCM2835_PWM1_RANGE, range);
}

void bcm2835_pwm_set_data(uint8_t channel, uint32_t data)
{
  if (channel == 0)
      bcm2835_peri_write_nb(bcm2835_pwm + BCM2835_PWM0_DATA, data);
  else if (channel == 1)
      bcm2835_peri_write_nb(bcm2835_pwm + BCM2835_PWM1_DATA, data);
}

// Allocate page-aligned memory.
void *malloc_aligned(size_t size)
{
    void *mem;
    errno = posix_memalign(&mem, BCM2835_PAGE_SIZE, size);
    return (errno ? NULL : mem);
}
```

```
// Map 'size' bytes starting at 'off' in file 'fd' to memory.
// Return mapped address on success, MAP_FAILED otherwise.
// On error print message.
static void *mapmem(const char *msg, size_t size, int fd, off_t off)
{
    void *map = mmap(NULL, size, (PROT_READ | PROT_WRITE), MAP_SHARED, fd, off);
    if (MAP_FAILED == map)
         fprintf(stderr, "bcm2835_init: %s mmap failed: %s\n", msg, strerror(errno));
    return map;
}

static void unmapmem(void **pmem, size_t size)
{
    if (*pmem == MAP_FAILED) return;
    munmap(*pmem, size);
    *pmem = MAP_FAILED;
}

// Initialise this library.
int bcm2835_init(void)
{
    if (debug)
    {
         bcm2835_pads = (uint32_t*)BCM2835_GPIO_PADS;
         bcm2835_clk  = (uint32_t*)BCM2835_CLOCK_BASE;
         bcm2835_gpio = (uint32_t*)BCM2835_GPIO_BASE;
         bcm2835_pwm  = (uint32_t*)BCM2835_GPIO_PWM;
         bcm2835_spi0 = (uint32_t*)BCM2835_SPI0_BASE;
         bcm2835_bsc0 = (uint32_t*)BCM2835_BSC0_BASE;
         bcm2835_bsc1 = (uint32_t*)BCM2835_BSC1_BASE;
         bcm2835_st   = (uint32_t*)BCM2835_ST_BASE;
         return 1; // Success
    }
    int memfd = -1;
    int ok = 0;
    // Open the master /dev/memory device
    if ((memfd = open("/dev/mem", O_RDWR | O_SYNC) ) < 0)
    {
         fprintf(stderr, "bcm2835_init: Unable to open /dev/mem: %s\n",
                   strerror(errno)) ;
         goto exit;
    }

    // GPIO:
    bcm2835_gpio = mapmem("gpio", BCM2835_BLOCK_SIZE, memfd, BCM2835_GPIO_BASE);
    if (bcm2835_gpio == MAP_FAILED) goto exit;

    // PWM
    bcm2835_pwm = mapmem("pwm", BCM2835_BLOCK_SIZE, memfd, BCM2835_GPIO_PWM);
    if (bcm2835_pwm == MAP_FAILED) goto exit;

    // Clock control (needed for PWM)
    bcm2835_clk = mapmem("clk", BCM2835_BLOCK_SIZE, memfd, BCM2835_CLOCK_BASE);
    if (bcm2835_clk == MAP_FAILED) goto exit;

    bcm2835_pads = mapmem("pads", BCM2835_BLOCK_SIZE, memfd, BCM2835_GPIO_PADS);
    if (bcm2835_pads == MAP_FAILED) goto exit;

    bcm2835_spi0 = mapmem("spi0", BCM2835_BLOCK_SIZE, memfd, BCM2835_SPI0_BASE);
    if (bcm2835_spi0 == MAP_FAILED) goto exit;

    // I2C
    bcm2835_bsc0 = mapmem("bsc0", BCM2835_BLOCK_SIZE, memfd, BCM2835_BSC0_BASE);
    if (bcm2835_bsc0 == MAP_FAILED) goto exit;

    bcm2835_bsc1 = mapmem("bsc1", BCM2835_BLOCK_SIZE, memfd, BCM2835_BSC1_BASE);
    if (bcm2835_bsc1 == MAP_FAILED) goto exit;

    // ST
    bcm2835_st = mapmem("st", BCM2835_BLOCK_SIZE, memfd, BCM2835_ST_BASE);
    if (bcm2835_st == MAP_FAILED) goto exit;

    ok = 1;

exit:
    if (memfd >= 0)
        close(memfd);

    if (!ok)
         bcm2835_close();
```

```
    return ok;
}

// Close this library and deallocate everything
int bcm2835_close(void)
{
    if (debug) return 1; // Success
    unmapmem((void**) &bcm2835_gpio, BCM2835_BLOCK_SIZE);
    unmapmem((void**) &bcm2835_pwm,  BCM2835_BLOCK_SIZE);
    unmapmem((void**) &bcm2835_clk,  BCM2835_BLOCK_SIZE);
    unmapmem((void**) &bcm2835_spi0, BCM2835_BLOCK_SIZE);
    unmapmem((void**) &bcm2835_bsc0, BCM2835_BLOCK_SIZE);
    unmapmem((void**) &bcm2835_bsc1, BCM2835_BLOCK_SIZE);
    unmapmem((void**) &bcm2835_st,   BCM2835_BLOCK_SIZE);
    unmapmem((void**) &bcm2835_pads, BCM2835_BLOCK_SIZE);
    return 1; // Success
}

#ifdef BCM2835_TEST
// this is a simple test program that prints out what it will do rather than
// actually doing it
int main(int argc, char **argv)
{
    // Be non-destructive
    bcm2835_set_debug(1);

    if (!bcm2835_init())
         return 1;

    // Configure some GPIO pins fo some testing
    // Set RPI pin P1-11 to be an output
    bcm2835_gpio_fsel(RPI_GPIO_P1_11, BCM2835_GPIO_FSEL_OUTP);
    // Set RPI pin P1-15 to be an input
    bcm2835_gpio_fsel(RPI_GPIO_P1_15, BCM2835_GPIO_FSEL_INPT);
    //  with a pullup
    bcm2835_gpio_set_pud(RPI_GPIO_P1_15, BCM2835_GPIO_PUD_UP);
    // And a low detect enable
    bcm2835_gpio_len(RPI_GPIO_P1_15);
    // and input hysteresis disabled on GPIOs 0 to 27
    bcm2835_gpio_set_pad(BCM2835_PAD_GROUP_GPIO_0_27, BCM2835_PAD_SLEW_RATE_UNLIMITED|BCM2835_PAD_DRIVE_8mA);

#if 1
    // Blink
    while (1)
    {
         // Turn it on
         bcm2835_gpio_write(RPI_GPIO_P1_11, HIGH);

         // wait a bit
         bcm2835_delay(500);

         // turn it off
         bcm2835_gpio_write(RPI_GPIO_P1_11, LOW);

         // wait a bit
         bcm2835_delay(500);
    }
#endif

#if 0
    // Read input
    while (1)
    {
         // Read some data
         uint8_t value = bcm2835_gpio_lev(RPI_GPIO_P1_15);
         printf("read from pin 15: %d\n", value);

         // wait a bit
         bcm2835_delay(500);
    }
#endif

#if 0
    // Look for a low event detection
    // eds will be set whenever pin 15 goes low
    while (1)
    {
         if (bcm2835_gpio_eds(RPI_GPIO_P1_15))
         {
             // Now clear the eds flag by setting it to 1
             bcm2835_gpio_set_eds(RPI_GPIO_P1_15);
             printf("low event detect for pin 15\n");
```

```
        }

        // wait a bit
        bcm2835_delay(500);
    }
#endif

    if (!bcm2835_close())
        return 1;

    return 0;
}
#endif
```

Appendix D

Parts Lists

This section contains three lists of parts for the MSAv4 system. Each list represents one of the primary components of the MSAv4; including the RPi, SID, ECIB, GPS, and External Sensors. The cost of the entire system was covered by the O.U.R. (Ralph Nye Charitable Foundation) research grant and funds from the AIAA and Val A. Browning Foundation; totaling $898. The following parts lists account for $315. There is a remaining balance of $339.74 on the research grant; this money will be spent to purchase another balloon and a stock of replacement parts for the MSAv4. $243.26 were spent on development supplies; such as breakout boards, experimental sensors and IC (for compatibility testing) and replacement parts (two of each compatible IC and batteries).

Table 14: MSAv4 Parts List Core Build with SID and Sensor Board

Description	Part Number	Vendor	Unit Price	Qty.	Total
Raspberry Pi	MODB-512M	Element 14	$39.95	1	$39.95
Pushbutton	COM-08720	SparkFun	$0.95	1	$0.95
Coin Cell Battery	PRT-00337	SparkFun	$1.95	1	$1.95
Coin Cell Battery Holder	PRT-07948	SparkFun	$1.50	1	$1.50
Female Headers	PRT-00115	SparkFun	$1.50	6	$9.00
Heat Sink	PRT-11510	SparkFun	$1.95	2	$1.95
MAX4208	MAX4208AUA+-ND	Digi-Key	$3.76	1	$3.76
TMP112	296-24621-1-ND	Digi-Key	$2.64	2	$5.28
HMC5883L	342-1082-1-ND	Digi-Key	$3.30	1	$3.30
MPU6000	1428-1005-1-ND	Digi-Key	$10.72	1	$10.72
MIC2937A-3.3V	576-1131-ND	Digi-Key	$2.82	1	$2.82
MIC2937A-5.0V	576-1133-ND	Digi-Key	$2.97	1	$2.97
LTC2495	LTC2495IUHF#PBF-ND	Digi-Key	$7.04	1	$7.04
DS3231M	DS3231MZ+-ND	Digi-Key	$8.74	1	$8.74
6x9 double sided copper clad board	590-660	Mouser	$18.75	1	$18.75
HIH5030	480-3284-1-ND	Digi-Key	$15.53	2	$31.06
2x13 Female Stackable Header	1112	Adafruit	$1.95	1	$1.95
1x8 Female Stackable Header	CCH01085C	Elecrow	$0.35	1	$0.35
8GB SD Card Class 10	SF8UY/TQMN	Walmart	$11.88	1	$11.88
MPX2102	841-MPXM2102A	Mouser	$6.81	1	$6.81
SMD-805 Resistors	Assorted	Digi-Key	$0.10	15	$1.50
SMD-805 Capacitors	Assorted	Digi-Key	$0.50	14	$7.00
2-SMD LED	Red	Digi-Key	$2.00	1	$2.00
Anderson Power Poles	Red and Black	Amazon	$0.85	2	$1.70
SMD fuse block and fuse 1.5A	576-015401.5DRT	Mouser	$2.60	1	$2.60
Venom 7.4V LIPO Battery 1600mAh	1320 i3c flight pack	Amazon	$23.99	1	$23.99
Total					**$217.31**

Table 15: MSAv4 Parts List External Controls and Indicators Board

Description	Part Number	Vendor	Unit Price	Qty.	Total
6x9 double sided copper clad board	590-660	Mouser	$18.75	1	$18.75
Buzzer	668-1204-ND	Digi-Key	$3.55	1	$3.55
Switch	275-032	Radio Shack	$1.50	1	$1.50
Phone Jack -1/8 inch	274-251	Radio Shack	$0.50	1	$0.50
2N7000 NMOS Transistor	512-2N7000	Mouser	$0.42	1	$0.42
2-SMD LED	Assorted	Digi-Key	$2.00	3	$6.00
SMD-805 Resistors	Assorted	Digi-Key	$0.10	5	$0.50
2x5 Shrouded Header	PRT-08506	SparkFun	$1.50	1	$1.50
Total					**$32.72**

Table 16: MSAv4 Parts List GPS Interface Board

Description	Part Number	Vendor	Unit Price	Qty.	Total
XR20M1280 UART-I2C	701-XR20M1280IL24-F	Mouser	$5.23	1	$5.23
4MHz Crystal	XC588CT-ND	Digi-Key	$2.96	1	$2.96
74HC132 Schmitt Trigger NAND Gate	MM74HC132MTCXCT-ND	Digi-Key	$0.45	1	$0.45
Schottky Diode 0.1A	DB2731600LCT-ND	Digi-Key	$0.26	2	$0.52
Male Header	PRT-00116	SparkFun	$1.50	1	$1.50
SMD-805 Resistors	Assorted	Digi-Key	$0.10	2	$0.20
SMD-805 Capacitor	27pF	Digi-Key	$0.50	2	$1.00
GPS Module 15xH/15xL	010-00240-22	Garmin	$53.30	1	$53.30
Total					**$65.16**

Appendix E

Concept Document

HARBOR MSA v4 Design Concepts, John Sohl, 30 November 2012

Primary Project Goals:

1. Easy operation on the flight line with external controls and indicators.
2. Light weight, ideally, less than the current weight which is 0.90 lbs not counting the gas and dust sensor assembly.
3. Compact, total size with foam insulation box no larger than about 12” square, 8” would be better. I have some sample boxes and a way to make custom foam boxes.
4. On board data memory storage that is easy to access via either an external USB connection or internal SD card or similar.
5. Ability to have “guest” packages attached with programming that is easy to modify (add a subroutine and a function call and good to go). External connections.

Secondary Project Goals for new features:

1. Live downlink telemetry (XBee, WiFi, whatever) with a 40 mile line-of-sight range.
2. External LCD display of status (2 or 4 line display or a graphical display)

On Board Sensors, I have almost all of these, in stock:

<u>Existing</u>

1. Time stamp, not real time but just counting up in seconds from zero from when it gets turned on
2. 3-axis acceleration in the ±3g range
3. 3-axis acceleration in the ±10g or higher range
4. 3-axis gyroscope (By the way, check out the InvenSense MPU-6000 chip)
5. 3-axis magnetic field measurement
6. GPS (must be a high altitude chip, we have a list)
7. Internal system temperature inside box at electronics (°C)
8. External ambient atmosphere temperature (°C), two sensors would be better than one

9. Atmospheric pressure, absolute not gauge, must go 1mbar to 1000mbar
10. External relative humidity (internal might be interesting too, but not critical)
11. Battery voltage

New

1. Particle sensor with flow meter
2. Gas sensors, temperature adjusted, CO, CO2, NOx (The CO2 currently exists on the MSA but is not thermally stabilized; we have several ideas that we are working on for that.)
3. Motion and altitude triggered external “siren” (turns on for last few thousand feet then off upon landing on the ground or, at least, a less frequent chirp).

Option for Guest Sensors, including 5V power. Communication (at least one), here is my preferred order:

1. USB
2. I^2C
3. analog 0-5V
4. SPI
5. RS-232
6. Wireless