A Multi-Sensor Array for Atmospheric Science

**Part 2: Implementing the Code**

**Introduction**

This two part series describes the design and construction of a Multi-Sensor Array (MSA) for studying atmospheric pollution in an urbanized mountain basin; specifically, the region above Salt Lake City, Utah (USA). The MSA is flown on research balloons by HARBOR, an undergraduate research group at Weber State University in Ogden, Utah. The MSA produces a column of measurements from ground level (approx. 1km ASL) to the lower stratosphere (approx. 35 km ASL).

During flight, the system is exposed to pressures as low as 0.75 mmHg, where heat exchange becomes difficult even at temperatures of -50°C. Jet stream winds can exceed speeds of 200 km/h, applying punishing shock and vibration forces to our electronic equipment. In this extreme environment, the MSA must continue to gather scientific data for a minimum of 4 hours.

The first part of this series focused on the hardware design of the MSA system. This second part describes how the software was designed and implemented.

**The Programming Language**

The MSA software is written completely in C. We chose C primary because we were already familiar with it, and because it is a fairly common language for embedded systems. We made use of a fantastic C library for the BCM2835 ARM chip used on the Raspberry Pi. The Library is maintained by Mike McCauley at: <http://www.airspayce.com/mikem/bcm2835>

The BCM2835 C library provides functions for communicating with I2C devices, accessing GPIO pins, and generating Pulse Width Modulation (PWM) signals to control motors and heaters.

To install the library onto the Raspberry Pi, connect to the internet and enter the following:

wget http://www.airspayce.com/mikem/bcm2835/bcm2835-1.36.tar.gz

The most current version is 1.36. Once downloaded enter:

tar zxvf bcm2835-1.36.tar.gz

cd bcm2835-1.32

Before installing the library, it is necessary to set the library to compile for the correct version of the Raspberry Pi. For Version 2, the library can be compiled as-is. The library, by default, uses the BCS1 I2C bus. For version 1, you must access the source code by entering:

cd src

sudo nano bcm2835.c

Now, go to line 30 and uncomment the following portion:

#define I2C\_V1

This compiles the code to function with the BSC0 I2C bus. Afterwards, save and close the bcm2835.c file and then install the library with:

./configure

make

sudo make check

sudo make install

To include this library when compiling in C, use:

gcc –o name name.c –l bcm2835

**MSA Software Overview**

The MSA operational flight program is broken into three basic threads:

1. Main
2. Sample Rate Timer
3. Data logging scheduler

**Main Thread**

Upon start-up, the Main thread automatically begins and sets up POSIX threads for the timer and scheduler, initiates the I2C bus, initializes the GPIO pins, creates data files, and monitors the mission pull pin. Once the mission pin is pulled, a mission timer begins and the current time is stamped to each line in the data files. The USB, Ethernet, and HDMI chips are disabled to conserve power, and the main loop begins. In the main loop, the MSA simply waits for an event to initiate a safe shutdown; such as when the shutdown button is pressed, the mission countdown timer has elapsed, or the battery voltage drops below a preset threshold for more than one minute.



Figure 1: Main Thread

Shutdown is initiated in C by the following command:

System(“shutdown –h –P now”);

**Sample Rate Timing Thread**

The MSA code is written without any interrupts. It was a challenge to create a timer that could ensure that measurements would be taken at regular intervals. Fortunately, the C compiler that comes standard with Linux distributions is equipped with POSIX thread capabilities. We made use of parallel programming to create the timing and scheduling threads.

The Sample Rate Timing thread is designed to post a semaphore at a given time interval, in our case 100 ms, to set the base data sample rate at 10hz. It makes use of the nanosleep() function to create a 100 millisecond timer for reading sensors.

void \*schedule\_timer(void \*arg)

{

int milisec = 100;

struct timespec req= {0};

req.tv\_sec = 0;

req.tv\_nsec = milisec \* 1000000l;

while(1)

{  
nanosleep(&req, null);

sem\_post(&sem);

} // end while loop

} // end schedule\_timer

After 100 milliseconds, the timer releases the semaphore to the Data Logging Schedule thread.

**Data Logging Schedule Thread**

The Data Logging Schedule thread takes the semaphore, whenever it is available, and begins a read cycle. The MSA has two main reading cycles:

1. Flight dynamics data (accelerometers, gyroscopes, and magnetometer)
2. Environmental data (temperature, humidity, pressure, etc,….)

The scheduling thread also maintains a five minute countdown timer. When the timer elapses, the MSA code saves all of the data files and the timer is reset.



Figure 2: Timer and Scheduling Threads

Flight dynamic sensors are read every cycle at a rate of 10Hz. Environmental data is read much slower due to the relatively long conversion time of the LTC2495 ADC, approximately 160ms. Every 200ms, the scheduling thread determines whether to set, clear, or read an ADC channel. Nested switch statements determine which channel to read (0 through 15) and which function to call ( set, clear, or read).

It takes approximately 10 seconds to read all of the ADC channels:

200 msec \* 3 cycles \* 16 channels = 9.6 sec

which seems like a lot of time. This is fine, however, since most environmental parameters change relatively slowly as a function of altitude. A typical ascent rate is only 4.1 meters per second; this means that the balloon would travel about 40 meters per read cycle. One read every 40 meters is adequate resolution for temperature, pressure, humidity, and even gas composition when viewed over 30,000 meters. If greater resolution is necessary an additional ADC can always be added.

**Reading I2C Sensors**

The MSA sensors can be divided into two different groups:

1. Analog Sensors
2. Digital Sensors

The analog sensors are all routed through an I2C ready LTC2495 ADC. The digital sensors for the MSA were all chosen to be I2C accessible. With all of the sensors communicating over the I2C bus, we were able to free up the remaining GPIOs for other purposes.

**Reading Analog Sensors**

Before reading any I2C sensors, the bus must be initialized and set to a desired speed. In our case, we designed the bus to run at high-speed (400Khz). The following code shows how we set up the I2C bus. Note that function calls prefaced with bcm2835 are part of the bcm2835 C library previously mentioned.

bcm2835\_i2c\_begin();

bcm2835\_i2c\_setclockdivider(bcm2835\_i2c\_clock\_divider\_626);

To read data from the ADC it is first necessary to select an input channel, set the gain, and choose whether it is single ended or differential input. We used only single ended sensors for the MSA, so setup was the same for each channel. The following function sets the channel to read the pressure sensor on the ADC which is located at the I2C address 0x14.

unsigned char chan\_0[] = {0xb0, 0x80};

// 0xb080 selects adc channel 0, single ended input, +in, gain x 1

unsigned char ltc2495 = 0x14;

bcm2835\_i2c\_setslaveaddress(ltc2495);

bcm2835\_i2c\_write(chan\_0, sizeof(chan\_0));

The first read from the ADC is generally unusable and has to be cleared; this is because the initial data has leftover information from a previous sample or a different sensor. The following function clears the channel by storing it a junk register and ignoring it:

unsigned char junk[3];

bcm2835\_i2c\_setslaveaddress(ltc2495);

bcm2835\_i2c\_read(junk, sizeof(junk));

We can now read the ADC. The following example code reads the pressure sensor on channel 0:

bcm2835\_i2c\_setslaveaddress(ltc2495);

unsigned char p\_buf[3];

bcm2835\_i2c\_read(p\_buf, sizeof(p\_buf));

p\_buf[0] = p\_buf & 0x7f; // mask the first bit (unused)

pressure = (p\_buf[0] \* 0x10000) + (p\_buf[1] \* 0x100) + (p\_buf[2]);

pressure = pressure/0x40;

v\_pressure = pressure \* 0.000025177;

// multiply by the step size to get voltage

The actual value is then processed from the signal voltage according the datasheet for a particular sensor.

**Reading Digital Sensors**

The digital sensors are a little more straight forward, because they don’t have to be reset every read cycle. However, some of them require burst reads (I2C reads with repeated starts and no stop) to send the address of a specific register to be read and an address to the destination where the data will be saved. Burst reads are also used to ensure that multiple axis data comes from the same sample.

Sensors like the HMC5883 3-axis magnetometer must be set up initially, but keep their settings for successive reads. The following code sets the sample rate of an HMC5883 to 8 averaged samples at rate of 15Hz. It is also set to measure continuously with a gain of 5:

unsigned char hmc5883 = 0x1e;

bcm2835\_i2c\_setslaveaddress(hmc5883);

Unsigned char gain\_5[] = {0x01,0xa0};

Unsigned char cont\_mode[] = {0x02, 0x00};

Unsigned char samp\_rate\_8\_15[] = {0x00, 0x70};

Bcm2835\_i2c\_write(samp\_rate\_8\_15, sizeof(samp\_rate\_8\_15));

Bcm2835\_i2c\_write(gain\_5, sizeof(gain\_5));

Bcm2835\_i2c\_write(cont\_mode, sizeof(cont\_mode));

A read is completed as follows:

unsigned char mag\_reg[] = {0x03};

unsigned char mag\_buf[6];

bcm2835\_i2c\_read\_register\_rs(mag\_reg, mag\_buf, sizeof(mag\_buf));

The data buffer now holds 6 bytes of data, 2 bytes for each axis, which are stored as two’s compliment values. As before, the actual value is then processed according to the datasheet for the sensor being used.

**Data Outputs**

The MSA program stores all of the data in .CSV files that are easy to use in LibreOffice or Excel. A typical output looks something like the following:

Date: 11/09/13

time,reg\_temp,xtemp,itemp,ipressure,ihumidity,dust\_v,xhumidity,batt\_voltage,

12:59:18,37.12,23.81,33.02,768.89,22.78,0.0000,41.57,7.53,

12:59:23,37.00,23.81,33.02,768.85,22.55,0.0000,38.78,7.53,

12:59:28,36.88,23.88,33.29,768.83,22.32,0.0000,40.37,7.53,

. . . . . .

. . . . . .

. . . . . .

The chart below was created using MSA data from a test flight last July above Duchesne, Utah. The mission lasted 4.5 hours and reached an altitude of just over 30 km.



Figure 3: Temperature and Altitude

Notice that the relatively low sample rate of the temperature sensor (0.2 Hz) still provided adequate resolution over the course of the mission.

**Coming Up**

We are currently working on a balloon borne optical sensor for measuring PM2.5 aerosols. The new sensor will require a vacuum pump to maintain a steady flow rate of 3LPM. A feedback loop will be designed to control a heater for maintaining a relative humidity of less than 50%RH. The Raspberry Pi will generate a PWM signal to regulate both temperature and flow rate. We have already begun prototyping critical components. The first flight tests are scheduled for late April of this year.