All Purpose Mobile GPS

by

James Smith, Matt Weege and Kevin Peters

Senior Project

ELECTRICAL ENGINEERING DEPARTMENT

California Polytechnic State University

San Luis Obispo, CA

2014

Table of Contents

- Acknowledgements - 4
- Abstract - 5
- Introduction - 6
- Background - 6
- Customer Needs Assessment - 9
- Requirements and Specifications - 10
- Design - 11
- Software - 11
  - Inertia -11
  - SD Card -18
  - GPS - 22
- Hardware - 28
  - Microcontroller - 28
  - Inertia - 30
  - SD Card - 33
  - GPS - 34
  - Power System - 35
    - Battery - 35
    - Solar Panel - 35
    - Charger -36
- Testing - 37
- Conclusion - 43
- Bibliography - 44
- Appendix - 45

List of tables

1. Engineering Specifications and Marketing Requirements - 10
2. List of Functions used for the SD card -21
3. MPU-9150 ADC - 30
4. MPU-9150 Filter - 31

List of figures

1. Triangle - 7
2. Vector Rotation - 8
3. Accelerometer Hardware Setup - 11
4. I2C Write - 12
5. DCM Accel Flowchart - 13
6. Matrix Math - 13
7. DCM Flowchart - 15
8. Rotation angle example - 16
9. Rotating a Vector Example - 16
10. Pedometer Flow Chart - 17
11. Shift Register Sum - 18
12. Pedometer conditional - 18
13. SD Card system flow chart - 19
14. Data Structure and Timer Initialization - 19
15. First settings called in the Main Function - 20
16. Excerpt of code showing the process of SD card writing - 22
17. GPS Functionality flowchart - 23
18. Initialization of the UART - 23
19. The System receiving NMEA sentences - 24
20. Instantiation of parsetokengps() - 25
21. Main loop conditionals - 27
22. Tiva C series - 28
23. Available GPIO pins on TM4C123GH6PM - 29
24. Available GPIO pins on TM4C123GH6PM cont. - 29
25. Picture of Sense Hub - 30
26. MPU-9105 block diagram - 32
27. MPU-9150 Circuit - 32
28. SD card breakout board from sparkfun - 33
29. Pin connections between the SD card and the microprocessor - 33
30. GPS module - 34
31. Connections between the GPS breakout board and the microprocessor - 34
32. 2500 mAh Lithium Ion Battery - 35
33. Solar panel used to charge the Lithium Ion Battery when outside - 36
34. Solar battery charger - 36
35. Solar panel V-I chart showing changes with sunlight. - 37
36. Invensense Axis Definition - 38
37. Raw Accel distance - 39
38. DCM Corrected Accel - 40
39. DCM Correction Loop Response - 41
40. Pedometer Output - 42

Acknowledgements

Kevin:

Dr. Bridget Benson: Bridge-dog, You led us down the path of glory. While it was long and treacherous, we made it to the end… Kindof… Thanks for being an awesome advisor and making sure we didn't give up on ourselves when we hit every wall along the way.

Dr. David Braun: Brauntasaurus, thanks for the encouragement and pushing during 460. You truly are a dinosaur man among children.

BP, KP, and SP: Shoutout to the best family I've had so far. Love you fabulous humans.

Rich Thall: Thanks for imbuing a love of science and an interest in questioning everything that happens around me. You're a pretty cool guy.

Matt:

Dr. Bridget Benson: Thank you so much for giving up your time to help us out during this project. We definitely didn't know what we were getting ourselves into but you were confident that we could accomplish the task. Your faith in us truly made this project possible.

Family and friends: Your support and encouragement helped me stay focused on the this project. I am confident that I would not have completed this project without your love.

Kevin & James: Next time when we take on a project I won't leave at the start.

James:

To Steve, Polly & Scott Smith, Dr. Bridget Benson, and the Cal Poly College of Engineering:

Thank you all for the tremendous help, guidance and knowledge you've provided me.
"If I have seen further it is by standing on the shoulders of giants" - Sir Isaac Newton

Abstract

The aim of the All Purpose Mobile GPS (The System) is to create a GPS tracking device that can accurately record location and calculate velocity, distance traveled and elevation change even when the satellite signal is lost. This data can later overlay on existing internet-based maps illustrating a trip to oneself or others. The portable tracker consumes very little power achieving several days' worth of power in a single charge. A solar cell provides a quick battery boost in a pinch. SD Card storage increases compatibility with computers and other forms of data transfer. Data processing, such as viewing the route on a map, occurs on a computer after the completed data logging.

Introduction

All three creators of All Purpose Mobile GPS (The System) came together through a common interest of navigation systems. Looking at the market we noticed that it was entirely flooded with the same kind of product. Satellite communication systems dominated the current market and did not work in all environments. We looked to old dead reckoning technology to develop a network that used both inertial and satellite communications to determine one's location. Dead reckoning was chosen due to versatility. Non-satellite location technology exists using wifi, bluetooth, or optics, however these techniques require a vast infrastructure and do not work outdoors sell. The combination of inertial and satellite measurement allows for accurate locations in more diverse environments.

Often, physically active people like to record details about their activity, whether it is running, biking, hiking or something similar. Typically, they like details such as distance, total time, elevation change and average speed. Many devices currently exist and exhibit these features if not more. One of the most popular examples of this is the Garmin Forerunner watch series. Even today's cell phones have accurate and reliable GPS recording and navigation capabilities. The real purpose of this project aimed to teach us what it takes to create our own fundamental version of this existing technology. To create a device with comparable features to current market devices, we chose specific features we wish to emulate and concentrate on them. These features include low power/long battery life, compact size, simple recharge-ability, consistent and accurate tracking, and most of all durability. By assessing the ideal customer as well as other products, we created a list of the most valuable features (see Table 1).

Background

One goal for The System was to make a very robust device so that the user would not notice the system when he or she was in the middle of an activity. This meant that we wanted to create a system that no matter the orientation of the device it could still know how the user moved. To accomplish this, we used a Direction Cosine Matrix (DCM). The implementation of the DCM is explained in the design section of this report but the math behind it is fundamental in the design of The System.

First off, it is important to define the vector cross product and the vector dot product. A bold character denotes a vector.

Vector Cross Product:

$$\mathbf{a} \times \mathbf{b} = |a|\ |b|\ \sin\theta$$

Vector Dot Product:

$\mathbf{a} \cdot \mathbf{b} = |a|\,|b| \cos\theta$

Where θ is the angle between the two vectors.

In order to rederive the workings of a DCM it is also important to understand the goal of the system. The point of a DCM is to create a global frame to translate a vector in the local or body frame onto. This means that a vector in a body frame can be converted to a vector in the global frame with the use of a global DCM. This derivation can be found at Sterlino Electronics. [1]

Lets first create our global and body frame using vectors with magnitudes of 1.

$\mathbf{I}^G = \{1,0,0\}$ $\mathbf{J}^G = \{0,1,0\}$ $\mathbf{K}^G = \{0,0,1\}$

$\mathbf{i}^B = \{1,0,0\}$ $\mathbf{j}^B = \{0,1,0\}$ $\mathbf{k}^B = \{0,0,1\}$

In order to make the matrix it is important to note that these vectors have been translated to be column vectors. Meaning the matrix that they make would have the vectors written vertically instead of horizontally like in most mathematical descriptions.

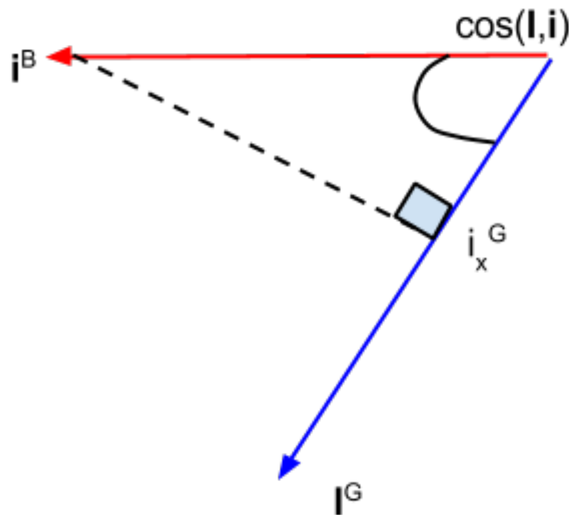

Figure 1: Triangle

Figure 1 shows how it is possible to take the x component of a body frame and translate it to the x component of the global frame.

$$i_x^G = |\mathbf{i}|\cos(\mathbf{I},\mathbf{i})$$

It is possible to repeat this calculation for finding the i^B translated on the y, and z axis.

$$i_y^G = |\mathbf{i}|\cos(\mathbf{J},\mathbf{i})$$
$$i_z^G = |\mathbf{i}|\cos(\mathbf{K},\mathbf{i})$$

With the three equations above it is now possible to translate the $\mathbf{i}^B$ to $\mathbf{i}^G$

$$\mathbf{i}^G = \{i_x^G, i_y^G, i_z^G\} = \{|\mathbf{i}|\cos(\mathbf{I},\mathbf{i}), |\mathbf{i}|\cos(\mathbf{J},\mathbf{i}), |\mathbf{i}|\cos(\mathbf{K},\mathbf{i})\}$$

The vector can be simplified more due to the fact that the lengths of the vectors have been normalized to 1.

$$\mathbf{i}^G = \{\cos(\mathbf{I},\mathbf{i}),\cos(\mathbf{J},\mathbf{i}),\cos(\mathbf{K},\mathbf{i})\}$$

This same procedure can be followed to translate the other body vectors to the global frame.

$$\mathbf{j}^G = \{\cos(\mathbf{I},\mathbf{j}),\cos(\mathbf{J},\mathbf{j}),\cos(\mathbf{K},\mathbf{j})\}$$
$$\mathbf{k}^G = \{\cos(\mathbf{I},\mathbf{k}),\cos(\mathbf{J},\mathbf{k}),\cos(\mathbf{K},\mathbf{k})\}$$

Now it is possible to create DCM^G. However when making the matrix its important to remember that initially the vectors were set up vertically and it need to stay that way for the DCM^G.

$$DCM^G = \{(\cos(\mathbf{I},\mathbf{i}),\cos(\mathbf{I},\mathbf{j}),\cos(\mathbf{I},\mathbf{k})), (\cos(\mathbf{J},\mathbf{i}),\cos(\mathbf{J},\mathbf{j}),\cos(\mathbf{J},\mathbf{k})), (\cos(\mathbf{K},\mathbf{i}),\cos(\mathbf{K},\mathbf{j}),\cos(\mathbf{K},\mathbf{k}))\}$$

The DCM^B can be found by rotating the vectors of DCM^G vertically.

$$DCM^B = \{(\cos(\mathbf{I},\mathbf{i}),\cos(\mathbf{J},\mathbf{i}),\cos(\mathbf{K},\mathbf{i})), (\cos(\mathbf{I},\mathbf{j}),\cos(\mathbf{J},\mathbf{j}),\cos(\mathbf{K}, \mathbf{k})), \cos(\mathbf{I},\mathbf{k}),\cos(\mathbf{J},\mathbf{k}),\cos(\mathbf{K},\mathbf{k}))\}$$
$$DCM^B = \{\mathbf{I}^B, \mathbf{J}^B, \mathbf{K}^B\}$$

In order to make the DCM truly useful three points need to be accepted:

1. Rotating a vector does not change the scale of a vector
2. The angle between two vectors does not change during a rotation if both vectors are subject to the same rotation
3. When the vectors are orthogonal the DCM has a 3x3 identity matrix. This can clearly be seen because the cos(0) = 1 and cos(90) = 0

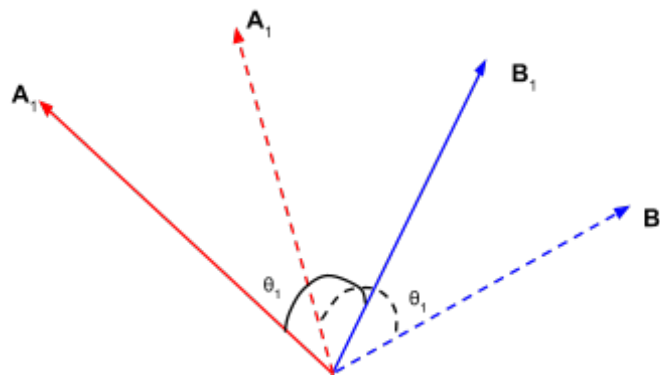

Figure 2: Vector Rotation

Figure 2 shows the two points made in the statement above by rotating two vectors from their initial location (dashed lines) to their current locations (solid lines).

Now lets say we have a vector in the body frame called $\mathbf{A}^B$. Similar to how we found the DCM we can start with x component of **A**. Note: **A** is written as a column vector

$$\mathbf{A}^B = \{A^B_x, A^B_y, A^B_z\}$$
$$A^G_x = |\mathbf{A}^G| \cos(\mathbf{I}^G, \mathbf{A}^G)$$

Using the two properties defined above it is possible to say the following.

$$|\mathbf{A}^B| = |\mathbf{A}^G|$$
$$|\mathbf{I}^B| = |\mathbf{I}^G| = 1 \text{ (due to orthogonality)}$$
$$\cos(\mathbf{I}^G, \mathbf{A}^G) = \cos(\mathbf{I}^B, \mathbf{A}^B)$$

Now it is possible to redefine A^G_X. Note: $\mathbf{I}^B$ is written as a column vector

$$A^G_x = |\mathbf{A}^B|\, |\mathbf{I}^B| \cos(\mathbf{I}^B, \mathbf{A}^B)$$
$$A^G_x = \mathbf{A}^B \cdot \mathbf{I}^B = \{A^B_x, A^B_y, A^B_z\} \cdot \{\cos(\mathbf{I},\mathbf{i}),\cos(\mathbf{I},\mathbf{j}),\cos(\mathbf{I},\mathbf{k})\}$$
$$A^G_x = A^B_x \cos(\mathbf{I},\mathbf{i}) + A^B_y \cos(\mathbf{I},\mathbf{j}) + A^B_z \cos(\mathbf{I},\mathbf{k})$$

Now the same process can be followed for the Y and Z components.

$$A^G_y = A^B_x \cos(\mathbf{J},\mathbf{i}) + A^B_y \cos(\mathbf{J},\mathbf{j}) + A^B_z \cos(\mathbf{J},\mathbf{k})$$
$$A^G_xy= A^B_x \cos(\mathbf{K},\mathbf{i}) + A^B_y \cos(\mathbf{K},\mathbf{j}) + A^B_z \cos(\mathbf{K},\mathbf{k})$$

Finally it is possible to recreate the equations above in the following matrix form.

$$\mathbf{A}^G = \{A^G_x, A^G_y, A^G_z\} = DCM^G\, \mathbf{A}^B$$

The result shows that multiplying a vector in the body frame by the global directional cosine matrix will produce a vector that is represented in the global frame. How this technique was implemented in software is later explained in the inertia software section.

Customer Needs Assessment

To better understand the needs of the average customer, we imagined an ideal customer for The System and created needs that this customer requires. This person enjoys exploring the outdoors but likes to log their locations to share and revisit past explorations. A navigation system that communicates to a satellite at a rate less than 10 Hz suitably accommodates the average customer[11]. The device needs to have a rugged exterior that can handle harsh environments. This requirement comes from personal experience; if the device can't handle a little water or a slight fall it will not do much good out in the real world. Finally, the device should consume minimal power to account for long journeys as well as provide a reliable and useful alternative recharging method for mobile recharging. We wanted to maximize the amount of

time we could get out of a single charge. The use of competing devices, as well as commonly available hardware, aided us in the development of these customer needs.

Requirements and Specifications

We developed the following requirements and specifications (Table 1) using the customer's needs assessment. These specifications set the framework for an innovative new product that can challenge many other current navigation units on the market today.

Table 1: Engineering Specifications and Marketing Requirements

Marketing Requirements	Engineering Specifications	Justification
4	Exports info as CSV or KML file types	This makes it easy for the user to upload their tracking data (time, lat, long, elevation, etc.) to services such as Google Earth or Maps.
2	Total volume of device is smaller than 50mL (=50 cm^3)	To make a location tracker effective, it must be small enough to be portable and easily carried by a single person.
2	Total weight of the device is less than 1lb	Customers traveling with critical weight constraints shouldn't discount the device due to an overwhelming weight issue
1	Waterproof enclosure, capable of depths down to 100 meters	A waterproof enclosure provides more durability and functionality when submerged.
3,4	Rechargeable using a 5V source [12]	Many devices recharge using only a 5V USB interface; This makes the device more universal and easily adaptable.
3,4	Battery life of at least 6 hours[12]	Particularly long and continuous trips require a long lasting battery. without a way to easily recharge the device while away from readily available power, the device should have a fairly long lasting battery
4	Has no more than 5 user interface controls	Limiting the amount of controls on the device gives it a much more user friendly appeal and encourages customer satisfaction
3,4,5	Self-contained solar charging	Ability to charge with solar to keep functionality in remote locations
5,6	Uses a combination of GPS or Accelerometer Data to configure location	By using both GPS and accelerometer Data, the device can pinpoint its location in and out of satellite reception.

5,6	Location accuracy within 30 meters using a combination of GPS and Accelerometer dead reckoning techniques. Only requires GPS update every 10 min to maintain accuracy [11]	A certain level of accuracy needs to be promised to the user. 30 meters is possible using the provided technology. GPS must update at least every ten minutes to achieve this level of accuracy.
Marketing Requirements 1. Durable/Waterproof 2. Portable 3. Long lasting 4. Easy to use 5. Works anywhere 6. Accurate		

Design

The software for the entire system can be broken up into three main parts. The GPS software is responsible for pulling GPS data and parsing it. The SD card data formats and writes to the SD card. Finally, the inertia software reads data from the compass, gyro, and accelerometer (contained inside a motion processor) to produce a distance. Together these software parts produce a distance in any user environment.

The hardware side of The All Purpose Mobile GPS was comprised of five components. A GPS and a motion processing unit were used to read the user's movement. In order to log the user's movement an SD card reader was used to write to an external SD card. A microcontroller was used to manage these devices and output the sensor's data. Finally a half solar half battery power system was used to power all the devices.

Software

Inertia

The first part of the inertia software was to initialize the hardware necessary to interface the microcontroller and the three sensors. In this part the clock frequency was set for the microcontroller and an I2C bus for communication to to the motion processor.

The software flow chart seen in Figure 3 show the initialization of the hardware. First off a 40 MHz clock was set and an I2C bus was set up in Port D. From there, a struct containing information of the motion processing unit (MPU) was created. Next register were set using an I2C write command to the MPU. During this stage, multiple parameters mentioned in the hardware section were set, including, but not limited to: sensitivity of the accelerometer,

gyroscope, and magnetometer. Finally, another struct was created to hold information for the Directional Cosine Matrix (DCM) which was used extensively in the infinite while loop.

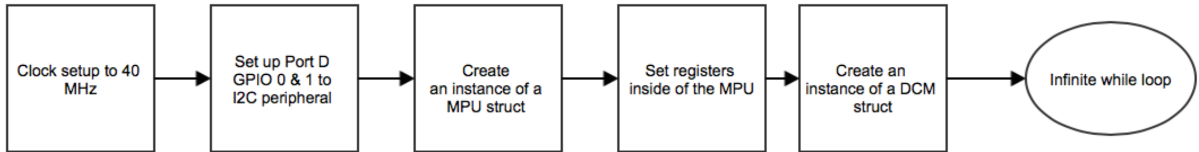

Figure 3: Accelerometer Hardware Setup

Writing to the MPU using the I2C protocol was accomplished using an array called `pui8Data`. Sequential register values were placed into the array. The first register being written to would be placed in the function call `MPU9150Write` and the number of registers that followed. This results in multiple registers being written to in one function call. An example of this can be seen in Figure 4.

```
g_sMPU9150Inst.pui8Data[0] = MPU9150_CONFIG_DLPF_CFG_94_98;
g_sMPU9150Inst.pui8Data[1] = MPU9150_GYRO_CONFIG_FS_SEL_250;
g_sMPU9150Inst.pui8Data[2] = (MPU9150_ACCEL_CONFIG_ACCEL_HPF_5HZ |
                              MPU9150_ACCEL_CONFIG_AFS_SEL_2G);
MPU9150Write(&g_sMPU9150Inst, MPU9150_O_CONFIG, g_sMPU9150Inst.pui8Data, 3,
             MPU9150AppCallback, &g_sMPU9150Inst);
```

Figure 4: I2C Write

The next software flow chart, Figure 5, explains how the DCM corrected accelerometer data was created. All three sensors were read and converted into floats. This data was then used to update the current DCM. Finally the DCM was read and then used for matrix multiplication.

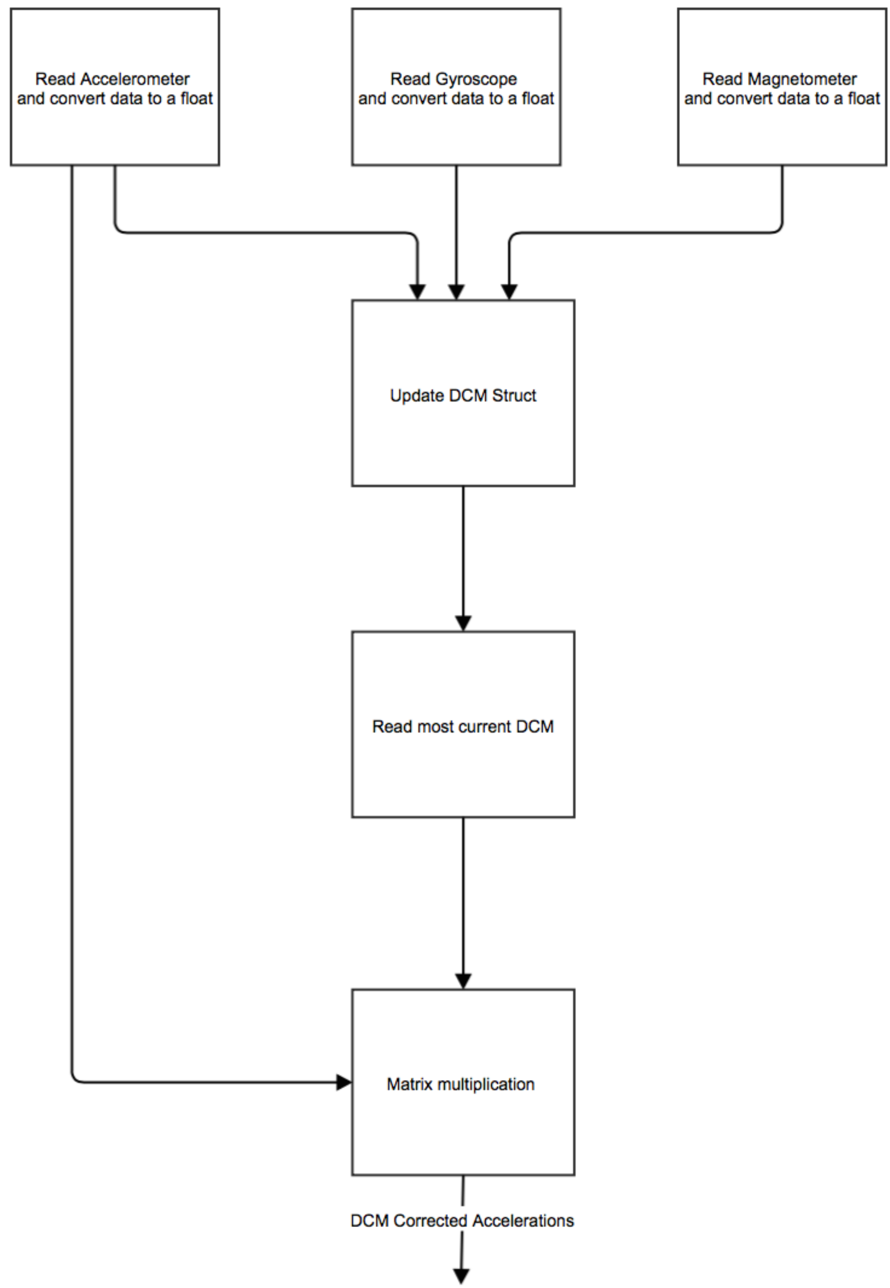

Figure 5: DCM Accel Flowchart

An example of the matrix multiplication can be seen below in Figure 6. Since the DCM produces a 3x3 matrix and its being multiplied by a vector the result is 3x1 matrix representing

another vector. The variables `bodyX`, `bodyY`, and `bodyZ` represent the acceleration on the global axis system.

```
bodyX = (myDCM[0][0] * pfData[0]) + (myDCM[0][1] * pfData[1]) + (myDCM[0][2] * pfData[2]);
bodyY = (myDCM[1][0] * pfData[0]) + (myDCM[1][1] * pfData[1]) + (myDCM[1][2] * pfData[2]);
bodyZ = (myDCM[2][0] * pfData[0]) + (myDCM[2][1] * pfData[1]) + (myDCM[2][2] * pfData[2]);
```

Figure 6: Matrix Math

Figure 7 demonstrates the software loop used to compute the DCM used to find accelerations on the global axis. The DCM code came from example code provided by TI. The accelerometer and magnetometer are used first to create the i and k vectors respectively. After the two vectors are normalized, they are used to find the rotation angle from the previous i and k vectors. This is accomplished by taking the cross product of i and k with their previous values and then scaling the values by an sensor coefficient based on empirical data. A rotation angle is also found using the gyroscope that is scaled and added to the one found by the accelerometer and magnetometer. The i and k vectors are then rotated by the summed rotation angle. Some error reduction takes place by taking the dot product of the rotated i and k vectors. This allows for the system to become more orthogonal. Finally, the j vector is created by taking the cross product of the i and k vectors. All three vectors are then placed in a 3x3 array representing the DCM. These values are used again in the next iteration of the loop.

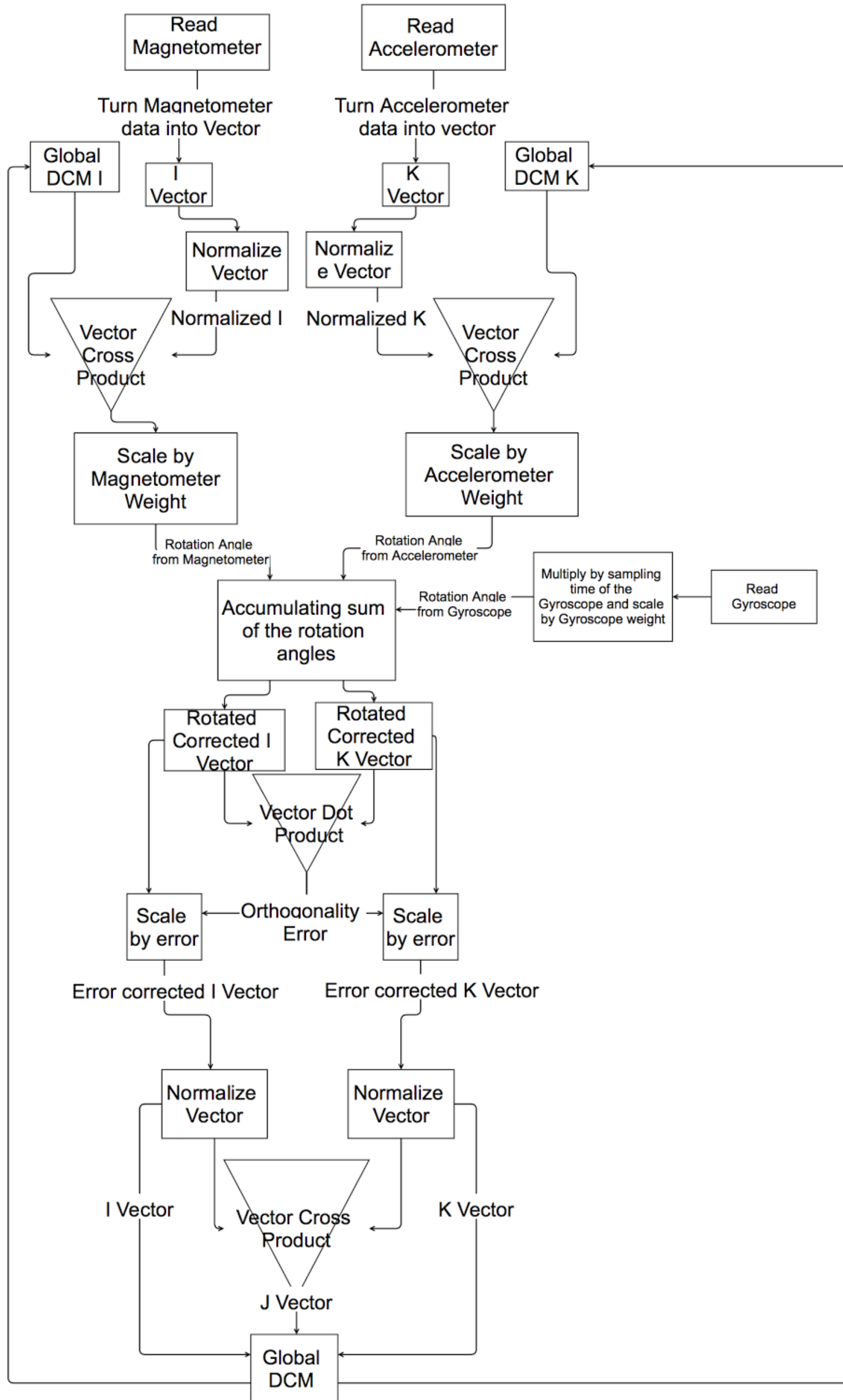

Figure 7: DCM Flowchart

Finding the rotation angle can be seen in Figure 8. First, the cross product of `psDCM->ppfDCM[0]` and `pfI` represents the cross product of the previous i vector and the new i vector read by the magnetometer. The value `pfTemp` holds the cross product result and was then scaled by the magnetometer weight `fScaleM`. Finally the scaled rotation angle is then added to the running angle sum called `pfDelta`.

```
VectorCrossProduct(pfTemp, psDCM->ppfDCM[0], pfI);
VectorScale(pfTemp, pfTemp, psDCM->fScaleM);
VectorAdd(pfDelta, pfDelta, pfTemp);
```

Figure 8: Rotation angle example

Rotating the k vector was accomplished in two steps. First, a cross product was taken between the previous k vector and the running angle sum `pfDelta`. This cross product result was held in the variable `pfK` which was added to the previous k vector. This can be seen in Figure 9.

```
VectorCrossProduct(pfK, pfDelta, psDCM->ppfDCM[2]);
VectorAdd(psDCM->ppfDCM[2], psDCM->ppfDCM[2], pfK);
```

Figure 9: Rotating a Vector Example

Due to the error found when trying to find distance from acceleration, which is explained in the testing section, a pedometer was implemented to calculate the number of steps a person has taken. Using this algorithm, a distance could be calculated but the direction was unknown. Figure 10 shows the algorithm used to find when a user takes a step. First, all three axes are read and placed in a shift register. The values inside the shift register are averaged and then summed with the other averaged axes. This combined average is placed in another shift register. The second shift register has only two indices. Every time a new value is placed inside the second shift register a condition is asked of the two values looking for a steep positive slope indicating a step. If this condition is met then the step count is increased, otherwise the accelerometer is read again.

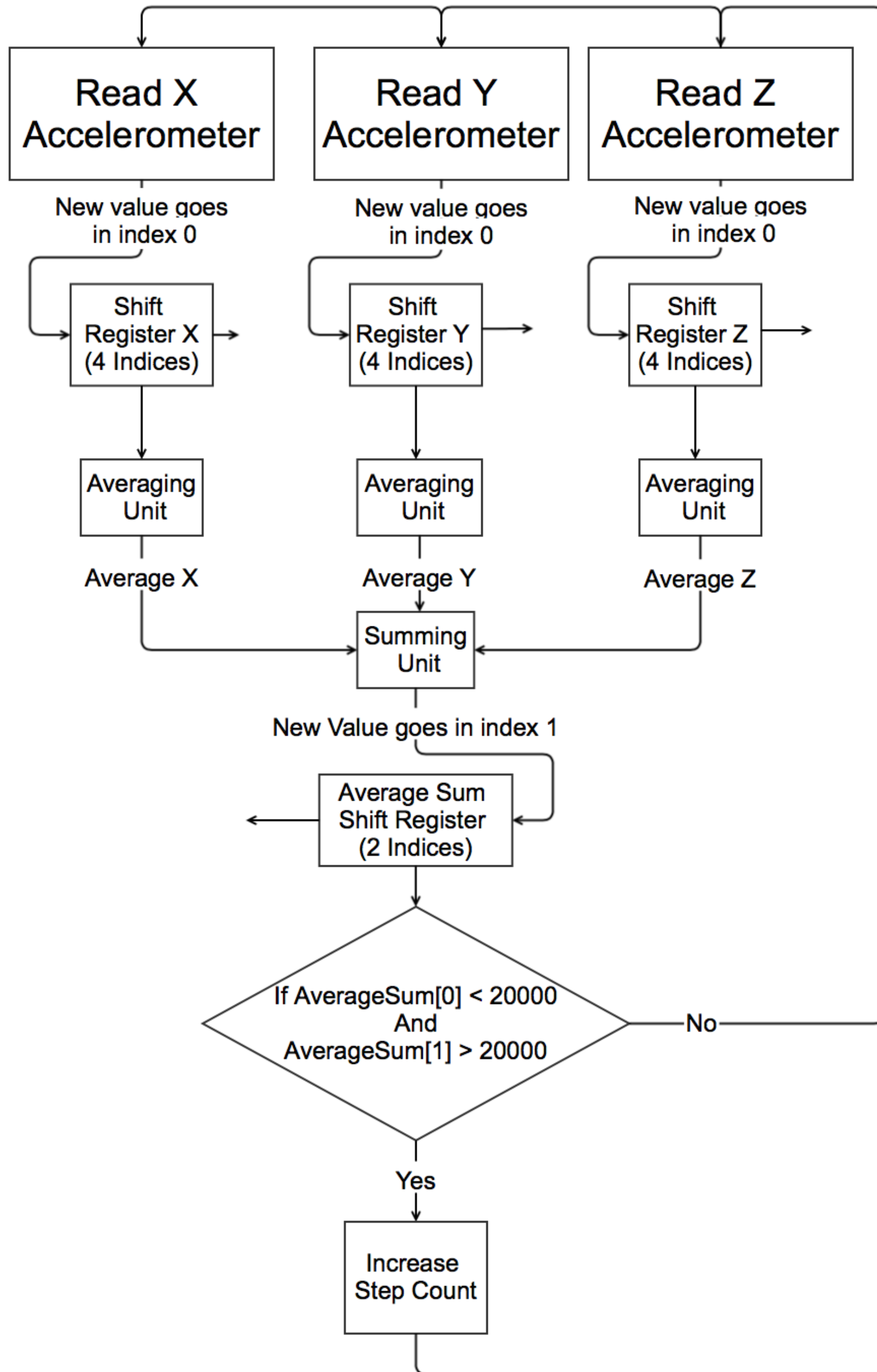

Figure 10: Pedometer Flow Chart

An example of the averaging that takes place in the pedometer algorithm can be seen below in Figure 11. The code below takes place in the **shiftRegisterSum** function. This function requires an int and an array of ints. After shifting the new value in and pushing the oldest value out, the array is summed and divided by four. The averaged value is then returned. A shift register with a size of four was used because any smaller array would have led to under-filtering the signal and a larger array would over-filter the signal.

```
int shiftRegisterSum (int newValue, int array[4])
{
	int sum;

	array[3] = array[2];
	array[2] = array[1];
	array[1] = array[0];
	array[0] = newValue;

	sum = (array[0] + array[1] + array[2] + array [3])/4;
	return sum;
}
```

Figure 11: Shift Register Sum

The conditional mentioned in the pedometer flow chart can be seen in the code below. Figure 12 shows that the algorithm is looking for the threshold of 20000 to be cross by the summed averaged accelerometer measurements. The threshold value was derived from test that are later explained.

```
if(accelArray[0] < 20000 && accelArray[1] >= 20000)
{
	stepCount++;
}
```

Figure 12: Pedometer conditional

SD Card

The SD card code was derived from a TI example program for reading data from an SD card. The libraries used in TI's example come from http://elm-chan.org/fsw/ff/00index_e.html. This library includes a fair amount of the work required to make writing data to the sd card. Below is a basic breakdown of the overall functionality as well as the portions of code used from the example and from the library provided by the link above.

The SD card functionality is relatively simple from a global perspective. Figure 13 below shows the basic functionality of the SD card software. First, the disk is mounted outside of main before any of the data from either the GPS or Inertia is saved to the card.

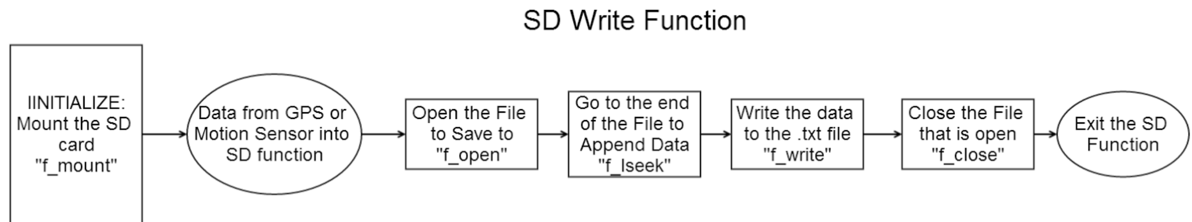

Figure 13: SD Card system flow chart

The first step of the SD card software is initializing all of the correct timers and system structures used by the FAT system. Next, we call the system timer interrupt handler, which is to be used to control the speed that the FAT system uses to read and write data. These should take place before the main function. Figure 14 below shows the initialization of the structures and Timer setting. `g_sFileObject` is the pointer to the next place in the file that data will be read from or written to. `g_sFatFs` is used by the `f_mount()` function to set up the SD card for data manipulation.

```
//*****************************************************************************
// The following are data structures used by FatFs.
//*****************************************************************************
static FATFS g_sFatFs;
static FIL g_sFileObject;

//*****************************************************************************
// This is the handler for this SysTick interrupt.  FatFs requires a timer tick
// every 10 ms for internal timing purposes.
//*****************************************************************************

void
SysTickHandler(void)
{
    //
    // Call the FatFs tick timer.
    //
    disk_timerproc();
}
```

Figure 14: Data Structure and Timer Initialization

Within the main function, we first needed to create an unsigned integer `ui32BytesWrite` to hold the value of the number of bytes written to the SD Card. This is useful mainly for testing but can also be used as a flag to determine whether anything was written or not. Next, we used the lazy stacking feature to avoid latency caused by the unnecessary stacking of floating point registers. After that, the system clock is set to 50 MHz and SPI (SSI0) is enabled for use with the SD Card peripheral. Finally, the master interrupt function and the system timer are enabled (with a period of 10 ms which is what the FAT file structure needs to operate). Figure 15 shows an excerpt of the code used in the project.

```
// Enable lazy stacking for interrupt handlers.  This allows floating-point
// instructions to be used within interrupt handlers, but at the expense of
// extra stack usage.
//
ROM_FPULazyStackingEnable();

//
// Set the system clock to run at 50MHz from the PLL.
//
ROM_SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
                               SYSCTL_XTAL_16MHZ);

//
// Enable the peripherals used by this example.
//
ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);

//
// Configure SysTick for a 100Hz interrupt.  The FatFs driver wants a 10 ms
// tick.
//
ROM_SysTickPeriodSet(ROM_SysCtlClockGet()/100);
ROM_SysTickEnable();
ROM_SysTickIntEnable();

//
// Enable Interrupts
//
ROM_IntMasterEnable();
```

Figure 15: First settings called in the Main Function

Before the final infinite while loop that contains both the GPS and Inertial data, the SD card must first be mounted once as there is no need to continuously mount the card. The function: `f_mount()` is used by the FAT Architecture to register the SD card and begin reading and/or writing data. Within the infinite while loop, the GPS and Inertial acquired data is parsed and sent to the SD to be written. With each new collection of data from the other peripherals, `f_open()` opens the specified file that we want to write data to. In order to continue appending

data instead of constantly overwriting it, we used the `f_lseek()` function to take the pointer to the end of the file and make sure not to write over existing data. After, the `f_write()` function allows text data to be written to the SD card and finally the `f_close()` function closes the file from any further writing or reading. The opening and closing of the file is performed every time to minimize the chance of corrupt data being written to the text file. Table 2 below illustrates the required parameters of each function we use from the FAT library and Figure 16 below shows an excerpt of our code that writes data to the SD card.

Table 2: List of Functions used for the SD card

Function	Parameters	Param Name	Description
f_mount	BYTE	vol	Logical drive number to be mounted/unmounted
	FATFS	*fs	Pointer to new file system object (NULL for unmount)
f_open	FIL	*fp	Pointer to the blank file object
	const TCHAR	*path	Pointer to the file name
	BYTE	mode	Access mode and file open mode flags
f_close	FIL	*fp	Pointer to the file object to be closed
f_write	FIL	*fp	Pointer to the file object
	const void	*buff	Pointer to the data to be written
	UINT	btw	Number of bytes to write
	UINT	*bm	Pointer to number of bytes written
f_lseek	FIL	*fp	Pointer to the file object
	DWORD	ofs	File pointer from top of file

```
// Open SD_Test.txt and begin writing. SD_Test.txt is the permanent output file
// FA_WRITE, FA_OPEN_ALWAYS, and FA_READ are flags indicating what you want to do
once // // the file is open
//
f_open(&g_sFileObject, "SD_Test.txt", FA_WRITE|FA_OPEN_ALWAYS|FA_READ);

//
// set the file pointer to the end of the file, in order to append data
//
f_lseek(&g_sFileObject, f_size(&g_sFileObject));

//
//  Write to the new file pointer location, DATA is any string brought in from
//  either the inertia unit or the GPS
//
f_write(&g_sFileObject, DATA, sizeof(DATA) - 1, &ui32BytesWrite);

//
//  Close The File
//
f_close(&g_sFileObject);
```

Figure 16: Excerpt of code showing the process of SD card writing

GPS

The Global Positioning System (GPS) loop serves as the primary flow for our positioning system system. Only when GPS lock is lost does the system resort to dead reckoning functions. GPS function with regard to top level system flow can be seen in Figure 17.

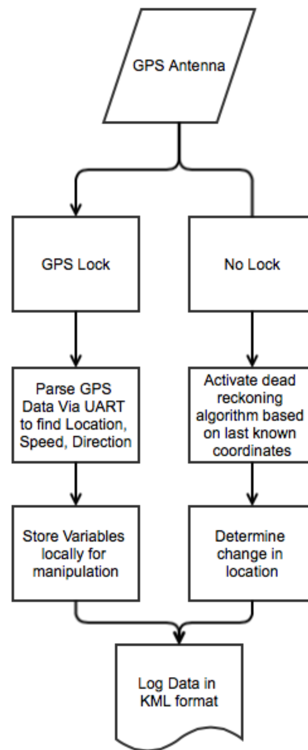

Figure 17: GPS Functionality flowchart

The framework for the GPS code was modeled after the UART Tutorial by TI. This contained the necessary declarations and setup required to initialize UART ports (Port A), pins, and communication speeds (Baud rate = 9600 for GPS to MCU, and 115200 for MCU to Terminal debugging) as shown in Figure 18. TI boards use functions to set up up the active IO rather than bit banging as we were accustomed to in EE 329. Since the GPS communicates via UART, all the data on the bus is in char form.

```
// Set the clocking to run directly from the external crystal/oscillator.

       SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
                      SYSCTL_XTAL_16MHZ);

    // Enable the peripherals
    // The UART itself needs to be enabled, as well as the GPIO port
    // containing the pins that will be used.
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART1);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    // Configure the GPIO pin muxing for the UART function.
    // This is only necessary if your part supports GPIO pin function muxing.
    // Study the data sheet to see which functions are allocated per pin.
```

```
    //fix this later
    GPIOPinConfigure(0x00010001);
    GPIOPinConfigure(0x00010401);

    //uart 0 pin configs
    GPIOPinConfigure(0x00000001);
    GPIOPinConfigure(0x00000401);

    //configure GPIO A0 and A1 for use as a peripheral function instead of GPIO
    GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
    GPIOPinTypeUART(GPIO_PORTB_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    //configure the board UART for 115,200 baud
    //configure the gps UART for 9600 baud
    UARTConfigSetExpClk(UART1_BASE, SysCtlClockGet(), 9600,
                        (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
                         UART_CONFIG_PAR_NONE));

    UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 115200,
                          (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
                           UART_CONFIG_PAR_NONE));
```

Figure 18: Initialization of the UART

The System's antenna outputs five NMEA sentences - comma delimited strings from the GPS antenna containing all the various status, location, estimated speed, and direction data. Because of this, the desired data was separated into various strings and needed to be parsed. This was accomplished by reading in all the strings character by character from the antenna, tokenizing each value in the string, then storing the individually desired values in memory. With the aforementioned status values, The System determined whether or not GPS lock is established. If so, the loop directs it to log those values to the SD Card, and if not, it goes into dead reckoning mode. Figure 19 shows how the software receives the NMEA sentences.

```
while(1)
    {

       gpsdata = UARTCharGet(UART1_BASE);

       //get gps uart data

       i=0;

       if (gpsdata == '$')
       {
```

```
        do
        {
                gpsstring[i] = gpsdata;
                i++;
                gpsdata = UARTCharGet(UART1_BASE); //get next char

        } while(gpsdata != '\n'); //uart doesnt send null char at end of
        transmission, so we're using new line YOLO

        gpsstring[i] = 0;

    }
    //parse gps uart string into tokens

    //use charmin/zero out parsed[i] so no pieces get left behind
     for (i = 0; i < 30; i++) {
        parsed[i] = 0;
     }

    parsetokengps(parsed, gpsstring);
```

Figure 19: The System receiving NMEA sentences

After the NMEA sentence strings are read in char by char, they need to be parsed so The System can glean useful status and location information. This is done in the parsetokengps() called in Figure 19 and instantiated in Figure 20. This function takes in the GPSdata string (which in C can be treated like an array of chars), and separates the comma delimited values into separate elements of an array so The System can access them individually. parsetokengps() also dynamically allocates memory to the exact size of the parsed tokens so as to minimize impact on The System's limited 256KB memory space. Also to note: some nuances with C necessitate double pointers in this scenario. Since the GPSdata string is an array of chars, pointing at the string (*GPSdata) gives the whole string. To parse the string down to tokens, The System turns the string into an array of token strings, or more pedantically, into an array of arrays of chars - hence the need for double pointers.

```
void parsetokengps(char **arr, char *GPSdata)
{
   int i = 0;
   char temp[30];

   while (*GPSdata)
   {
      while (*GPSdata && *GPSdata != ',')
      {
         temp[i++] = *GPSdata++;
      }
```

```
        temp[i] = 0;
        if (*GPSdata == ',')
        {
            GPSdata++;
        }

        *arr = calloc(i + 1, sizeof(char)); //i still hate pointers.jpg

        if (i)
        {
            memcpy(*arr, temp, i); //dynamically size memory for string sizes
        }
        arr++;
        i = 0;
    }

    return;
}
```

Figure 20: Instantiation of parsetokengps()

After the information has been parsed and tokenized, the main loop checks the status values for GPS lock and stores tokenized values locally, or it recognizes that there is no lock and shifts into dead reckoning mode as noted in Figure 17 and Figure 21. Though there is only a comment placeholder there, had the integration of SD Card logging been successful The System would log the GPS data from the local variables, then free the memory for use elsewhere in the next loop iteration. The free() was implemented after we experienced numerous points where the loop would freeze up and no longer process anything. Upon analysis, the dynamically allocated memory in parsetokengps() was being held, thus The System was literally running out of memory with everything it was storing.

```
if(strcmp(parsed[0],"$GPRMC") == 0)
{
        if(strcmp(parsed[2],"V") == 0) //if second value == V, gps data invalid w/ no
        lock
       {
                //-> fix not available -> goto accelerometer<later>
                for(j = 0; j < 18; j++)
                {
                        UARTCharPut(UART0_BASE,nolock[j]);            //uart print no
lock
                }
                UARTCharPut(UART0_BASE, '\r');
                UARTCharPut(UART0_BASE, '\n');
                }else if( strcmp(parsed[2],"A") == 0) //lock established, goto gps time
                and position data to get lat and long
```

```
        {
        for(j = 0; j < 20; j++)
        {
                UARTCharPut(UART0_BASE,lockestablished[j]);   //uart print lock
        established
        }

        UARTCharPut(UART0_BASE, '\r');
        UARTCharPut(UART0_BASE, '\n');

        strcpy(timestamp, parsed[1]);    //first value <hhmmss.ss>
         strcpy(lat,parsed[3]);          //second value <ddmm.mmmm> [definitely
         double], latitude
        strcpy(latdir,parsed[4]); //N or S
        strcpy(lon,parsed[5]);           //4th value <ddmm.mmmm> [definitely
        double], longitude
        strcpy(longdir,parsed[6]);//E or W
        strcpy(speed,parsed[7]);  //knots <x.xx> <- not a huge range (0-9) :/
        strcpy(dir,parsed[8]);           //<ddd.dd>

    }

    UARTCharPut(UART0_BASE, '\r');
    UARTCharPut(UART0_BASE, '\n');


    //     later log gps data to sdcard
    //     get location data into legit format

    for (i = 0; i < 10; i++)
    {
        string[i] = 0;              //word string now cleared
    }

    for (i = 0; i < 30; i++)
    {
         free(parsed[i]);   //parsed[] now full of null so monsters dont come
         out and eat the code
    }
  }

  return(0);
}
```

Figure 21: Main loop conditionals

Hardware

Microcontroller

This project called for a microprocessor that was small, powerful, and easily configurable with an array of hardware. We decided on TI's Tiva C series TM4C123GH6PM[2] because it had the processing power to handle complex floating point math, it had a variety of booster packs that were easily mountable to the board, it had a fairly extensive volume of user and developer created examples, it was relatively small, and because it was well within our budget.

Figure 22: Tiva C series[8]

The Figures 23 and 24 below illustrate the purpose of each pin on the TM4C123. For our project, we mainly used the UART, I2C, and SPI pins.

IO	Pin	Analog Function	Digital Function (GPIOPCTL PMCx Bit Field Encoding)[a]										
			1	2	3	4	5	6	7	8	9	14	15
PA0	17	-	U0Rx	-	-	-	-	-	-	CAN1Rx	-	-	-
PA1	18	-	U0Tx	-	-	-	-	-	-	CAN1Tx	-	-	-
PA2	19	-	-	SSI0Clk	-	-	-	-	-	-	-	-	-
PA3	20	-	-	SSI0Fss	-	-	-	-	-	-	-	-	-
PA4	21	-	-	SSI0Rx	-	-	-	-	-	-	-	-	-
PA5	22	-	-	SSI0Tx	-	-	-	-	-	-	-	-	-
PA6	23	-	-	-	I2C1SCL	-	M1PWM2	-	-	-	-	-	-
PA7	24	-	-	-	I2C1SDA	-	M1PWM3	-	-	-	-	-	-
PB0	45	USB0ID	U1Rx	-	-	-	-	-	T2CCP0	-	-	-	-
PB1	46	USB0VBUS	U1Tx	-	-	-	-	-	T2CCP1	-	-	-	-
PB2	47	-	-	-	I2C0SCL	-	-	-	T3CCP0	-	-	-	-
PB3	48	-	-	-	I2C0SDA	-	-	-	T3CCP1	-	-	-	-
PB4	58	AIN10	-	SSI2Clk	-	M0PWM2	-	-	T1CCP0	CAN0Rx	-	-	-
PB5	57	AIN11	-	SSI2Fss	-	M0PWM3	-	-	T1CCP1	CAN0Tx	-	-	-
PB6	1	-	-	SSI2Rx	-	M0PWM0	-	-	T0CCP0	-	-	-	-
PB7	4	-	-	SSI2Tx	-	M0PWM1	-	-	T0CCP1	-	-	-	-
PC0	52	-	TCK SWCLK	-	-	-	-	-	T4CCP0	-	-	-	-
PC1	51	-	TMS SWDIO	-	-	-	-	-	T4CCP1	-	-	-	-
PC2	50	-	TDI	-	-	-	-	-	T5CCP0	-	-	-	-
PC3	49	-	TDO SWO	-	-	-	-	-	T5CCP1	-	-	-	-
PC4	16	C1-	U4Rx	U1Rx	-	M0PWM6	-	IDX1	WT0CCP0	U1RTS	-	-	-
PC5	15	C1+	U4Tx	U1Tx	-	M0PWM7	-	PhA1	WT0CCP1	U1CTS	-	-	-
PC6	14	C0+	U3Rx	-	-	-	-	PhB1	WT1CCP0	USB0EPEN	-	-	-
PC7	13	C0-	U3Tx	-	-	-	-	-	WT1CCP1	USB0PFLT	-	-	-
PD0	61	AIN7	SSI3Clk	SSI1Clk	I2C3SCL	M0PWM6	M1PWM0	-	WT2CCP0	-	-	-	-
PD1	62	AIN6	SSI3Fss	SSI1Fss	I2C3SDA	M0PWM7	M1PWM1	-	WT2CCP1	-	-	-	-
PD2	63	AIN5	SSI3Rx	SSI1Rx	-	M0FAULT0	-	-	WT3CCP0	USB0EPEN	-	-	-
PD3	64	AIN4	SSI3Tx	SSI1Tx	-	-	-	IDX0	WT3CCP1	USB0PFLT	-	-	-
PD4	43	USB0DM	U6Rx	-	-	-	-	-	WT4CCP0	-	-	-	-
PD5	44	USB0DP	U6Tx	-	-	-	-	-	WT4CCP1	-	-	-	-
PD6	53	-	U2Rx	-	-	M0FAULT0	-	PhA0	WT5CCP0	-	-	-	-
PD7	10	-	U2Tx	-	-	-	-	PhB0	WT5CCP1	NMI	-	-	-
PE0	9	AIN3	U7Rx	-	-	-	-	-	-	-	-	-	-
PE1	8	AIN2	U7Tx	-	-	-	-	-	-	-	-	-	-
PE2	7	AIN1	-	-	-	-	-	-	-	-	-	-	-
PE3	6	AIN0	-	-	-	-	-	-	-	-	-	-	-

Figure 23: Available GPIO pins on TM4C123GH6PM[8]
(Used Pins are boxed)

IO	Pin	Analog Function	Digital Function (GPIOPCTL PMCx Bit Field Encoding)[a]										
			1	2	3	4	5	6	7	8	9	14	15
PE4	59	AIN9	U5Rx	-	I2C2SCL	M0PWM4	M1PWM2	-	-	CAN0Rx	-	-	-
PE5	60	AIN8	U5Tx	-	I2C2SDA	M0PWM5	M1PWM3	-	-	CAN0Tx	-	-	-
PF0	28	-	U1RTS	SSI1Rx	CAN0Rx	-	M1PWM4	PhA0	T0CCP0	NMI	C0o	-	-
PF1	29	-	U1CTS	SSI1Tx	-	-	M1PWM5	PhB0	T0CCP1	-	C1o	TRD1	-
PF2	30	-	-	SSI1Clk	-	M0FAULT0	M1PWM6	-	T1CCP0	-	-	TRD0	-
PF3	31	-	-	SSI1Fss	CAN0Tx	-	M1PWM7	-	T1CCP1	-	-	TRCLK	-
PF4	5	-	-	-	-	-	M1FAULT0	IDX0	T2CCP0	USB0EPEN	-	-	-

Figure 24: Available GPIO pins on TM4C123GH6PM cont.

Inertia

A motion processing unit was responsible for the inertia reference side of The System. A TI BoostXL-Senshub breakout board, seen in Figure 25 was used to detect the motion for the inertial reference system. This breakout board was chosen due to the number of sensors on the board and the custom fitted hardware to interface with the TI Tiva TM4C123GH6PM microcontroller breakout board. Motion processing unit found on the BoostXL-Senshub breakout board[3] was used exclusively for measuring the movement of a user. The BoostXL-Senshub breakout board uses a Invensense MPU-9150 9 axis sensor IC [3].

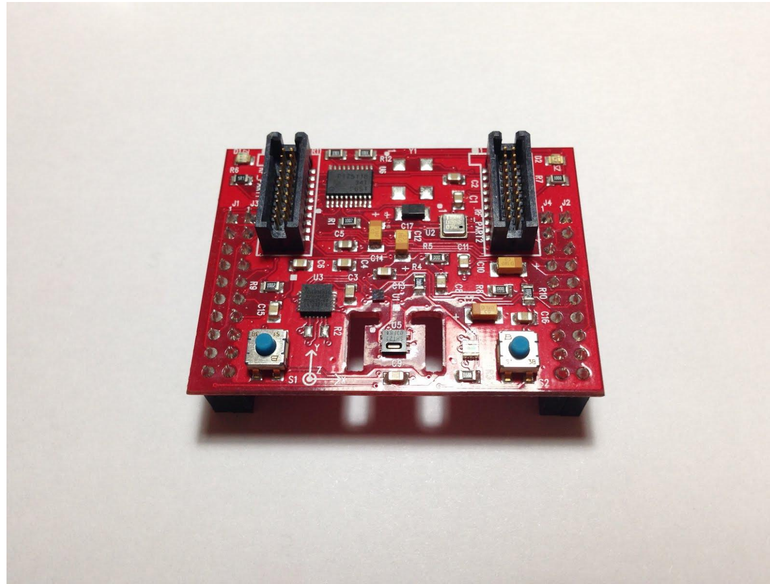

Figure 25: Picture of Sense Hub

Contained in the MPU-9150 is a 3 axis accelerometer, gyroscope, and magnetometer or compass. The MPU-9150 had three ADC for the converting motion into a digital logic value[3]. Table 3 sums up the rated limits for the motion sensing hardware in the MPU-9150. The System had the accelerometer and gyroscope set to their lowest setting to get the most accuracy out of the system (±2g and ±250º/sec). This is due to the fact that the resolution would be the highest for these sensors and we would be able to see the movement and direction of the user. Finally the data outputted by the sensors was a two's complement signed bit value[5].

Device	ADC	Measuring Limits
Accelerometer	16 bit	±2g, ±4g,±8g,±16g
Gyroscope	16 bit	±250,±500,±1000,±2000º/sec
Compass	13 bit	±1200μT

Table 3: MPU-9150 ADC [4]

The communication protocol used between the MPU-9150 and the TM4C123GH6PM microcontroller was I2C. For our application the clock speed was 400 kHz and the MPU-9150 was the only device activated on this I2C bus. Other features of the MPU-9150 that were used for The System included a digitally programmable low pass filter[4] which is characterized in Table 4. For our application, filter setting 3 was used.

Filter Setting	Accelerometer Bandwidth (Hz)	Gyroscope Bandwidth (Hz)
1	200	256
2	184	188
3	94	98
4	44	42
5	21	20
6	10	10
7	5	5

Table 4: MPU-9150 Filter[4]

The MPU-9150 block diagram can be seen below in Figure 26.

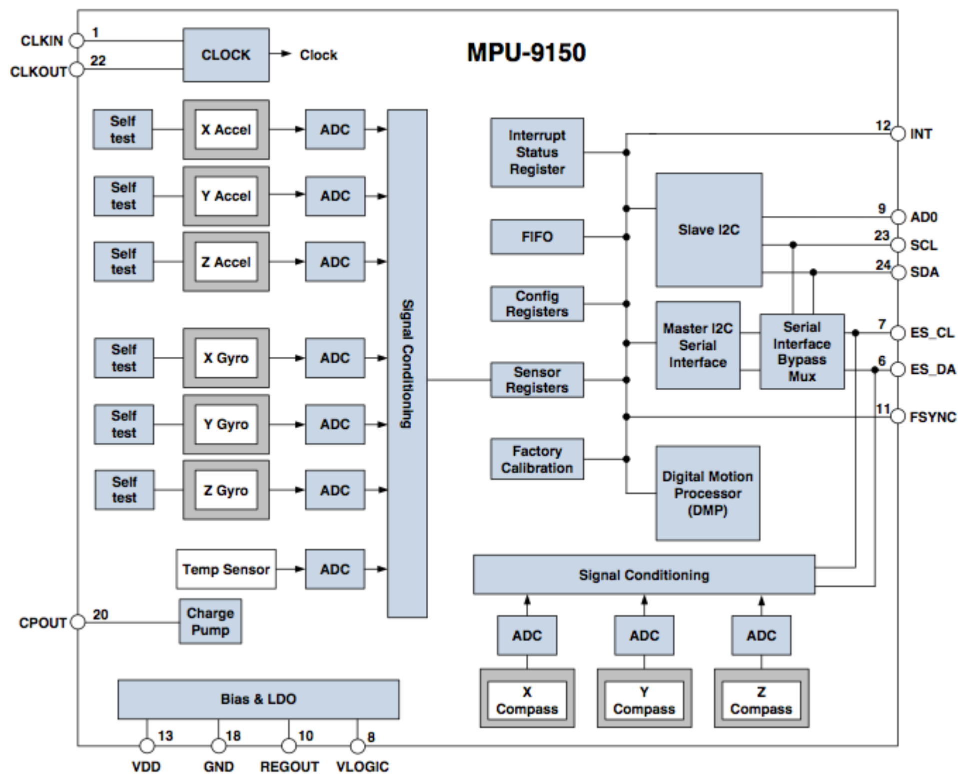

Figure 26: MPU-9105 block diagram[5]

Below in Figure 27, is the circuit diagram used in The System for communication between the motion sensing unit and the microcontroller.

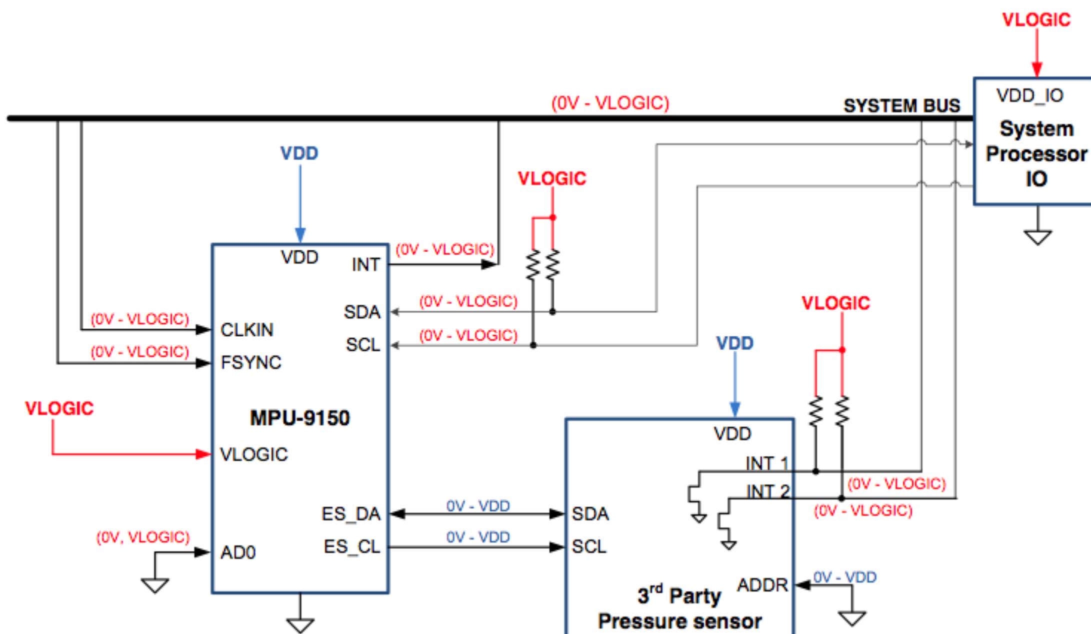

Figure 27: MPU-9150 Circuit [5]

SD Card

The SD card reader breakout board from sparkfun was used as the storage device for our project. The board features Card Detect (CD) pin, Data Out (DO), Ground (GND), Clock (SCK), Voltage supply (VCC), Data In (DI), and Chip Select (CS) pins. Figure 28 shows the physical layout of the SD breakout board and Figure 29 illustrates the connections between the SD breakout board and the microprocessor. For our purposes we left the Card Detect unconnected.

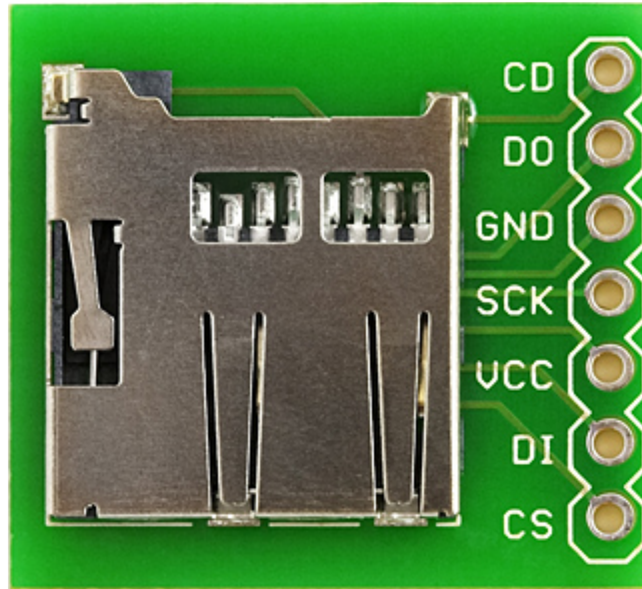

Figure 28: SD card breakout board from sparkfun [18]

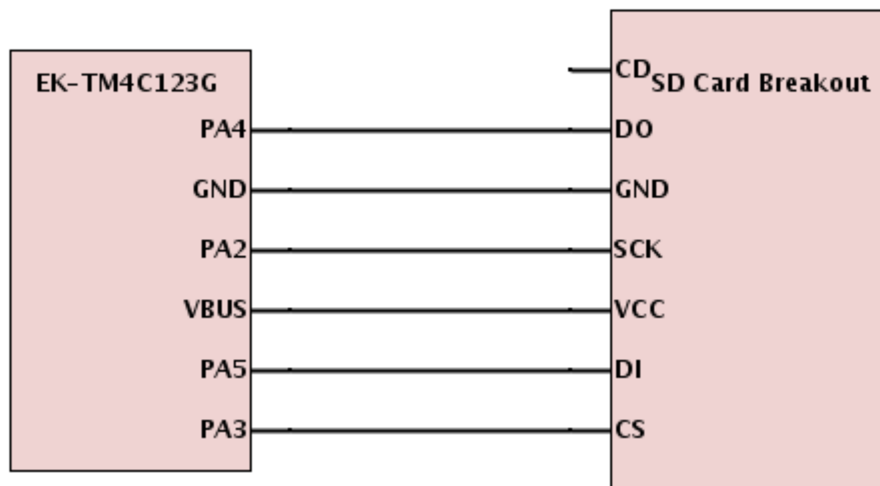

Figure 29: Pin connections between the SD card and the microprocessor

GPS

The GPS peripheral came from Adafruit and is based on the MTK3339 chipset. This chip is capable tracking up to 22 satellites on 66 channels. It has a controllable update rate which can be as fast as 10 hz and requires no more than 20 mA. The board includes an antenna so no external antenna is required. The only necessary pins for our purposes are VIN, GND, RX, and TX. The Breakout board communicates with the microprocessor via UART. Figure 30 shows what the breakout board physically looks out and Figure 31 shows the connections made with the microprocessor.

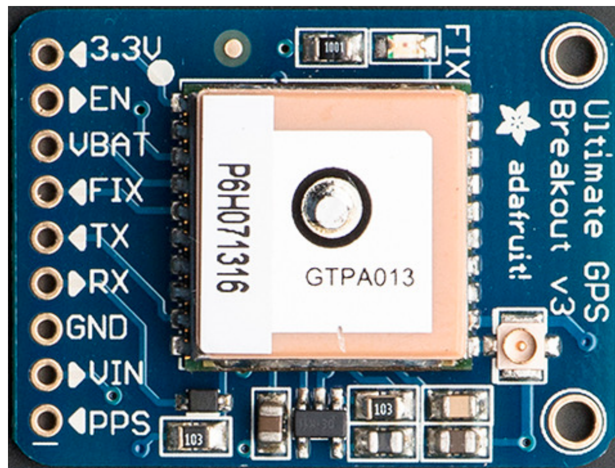

Figure 30: GPS module [14]

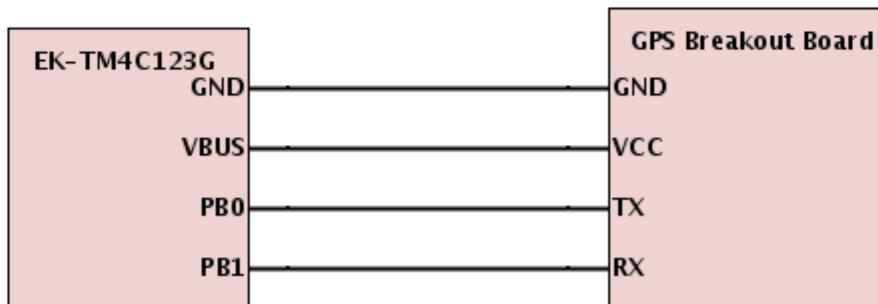

Figure 31: Connections between the GPS breakout board and the microprocessor

Power System

The power system consists of three discrete components: the Battery, Solar Panel, and the Charger.

Battery

The battery came from Adafruit and is Lithium Ion, rated at 3.7 V output (but can be as high as 4.2 V) with a capacity of 2500 mAh. The battery has a low voltage cutout at 3.0 to prevent any harm to the system.

Figure 32: 2500 mAh Lithium Ion Battery [15]

Solar Panel

The solar panel used for our project also came from adafruit. It is a 6 V panel capable of supplying up to 330 mA. It contains 12 cells that each have 0.5 V so that the total voltage is 6 V. Figure 33 shows a picture of the solar panel.

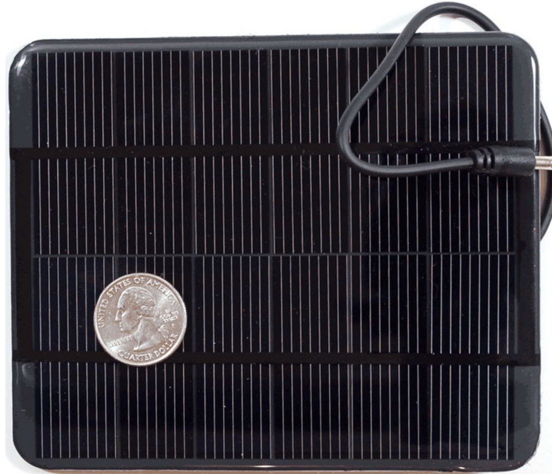

Figure 33: Solar panel used to charge the Lithium Ion Battery when outside [16]

Charger

The battery charger was also from adafruit and consists of three major components. It can be hooked up to a standard barrel plug from a 5-6V wall charger, the barrel plug from a 6V solar panel or a USB mini cable. The BATT plug (lower left of Figure 34) interfaces with the lithium Ion battery and can be used to simultaneously charge and power the project. The LOAD plug (lower right of Figure 34) interfaces with the rest of the system and acts as the source voltage for the microprocessor, GPS and Inertial measurement unit.

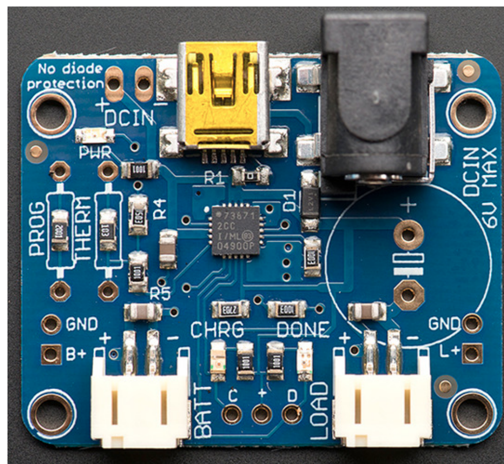

Figure 34: Solar battery charger [17]

Since most battery chargers are used to being connected to a steady DC wall outlet, they are simple and don't require much circuitry to account for the constantly changing V-I characteristics of a solar panel. Adafruit supplies a guide that explains how this charger counteracts the effects of constantly changing V-I characteristics from the solar panel. Below is the link to this description as well as our own summary. (https://learn.adafruit.com/usb-dc-and-solar-lipoly-charger/design-notes)

Figure 35 helps describe the effect of changing sunlight on a single solar cell's output. Observing the top red line (maximum light) it is obvious that at too high of a voltage, the current drops to zero, this would be called the open circuit voltage. Also, with too much current the voltage falls to zero, or otherwise called the short circuit current. As the light changes the current changes and the charger may begin to draw too much current and completely tank the voltage. In order to prevent too much current from being drawn and relinquishing the voltage to zero, adafruit included a chip called the MCP73871. This chip requires a reference voltage (chosen to be ~4.5 which is just above the battery's voltage) to be set and then makes sure to draw as much current from the solar cell as possible without letting the cell's voltage fall below the dictated level.

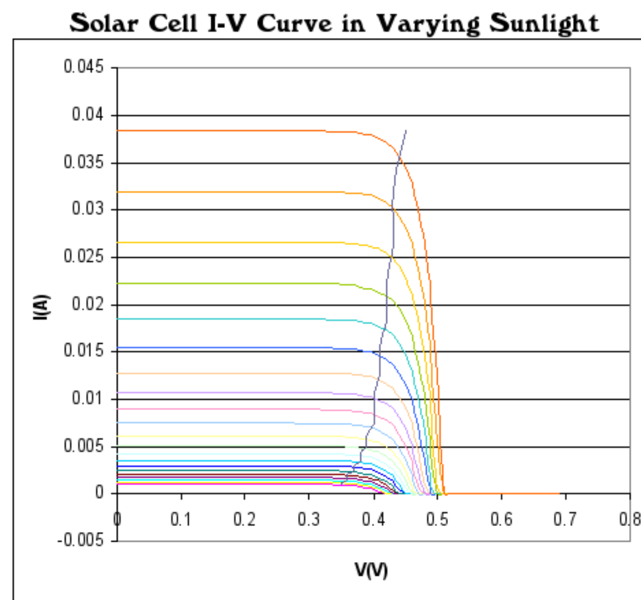

Figure 35: Solar panel V-I chart showing changes with sunlight[16]

The different colored lines indicate the different amounts of light. The red line at the top indicates maximum light and the lines below represent subsequently less light.

Testing

This project is far from finished and the only requirements tested were the following (numbers come from the Marketing Requirements in Table 1):

- Rechargeable using a 5V source (4)
- Battery life of at least 6 hours (3)
- Self-contained solar charging (5)
- Uses a combination of GPS or Accelerometer Data to configure location (5)

The solar panel battery charger used in our project was designed run off of a 5-6 V source or from either a DC wall adapter or from a standard USB hub. The battery was successfully charged using a USB port as the means of supply voltage as well as with the solar panel. We measured the current drawn by the micro while running the IMU, which by far used the most current, and found that the total current consumption ranged between 58 mA - 72 mA, with an average value of 62 mA. Using a battery rated at 2500 mAh, the system could run for approximately 40 hours. This is assuming that the IMU is running the entire time and that it is still searching for a GPS lock, so this is a worst case scenario.

The testing for using the IMU to define the user's location was extensive and was mostly a trial and comparison with previous test iterations. In order to create an accurate and repeatable testing environment the following conditions were set for all tests.

- 5 meters walked in the X axis direction
- The X, Y, and Z acceleration outputted on an excel spreadsheet
- Acceleration values would then be integrated using the rectangular method to get distance

The X, Y, and Z axes are defined by Invensense in Figure 36.

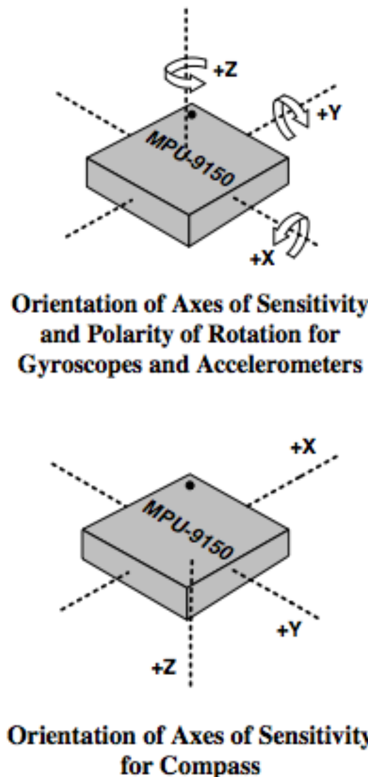

Figure 36: Invensense Axis Definition [5]

We first wanted to start out with a baseline test which would be the worst case scenario for accuracy. This test iteration just read the raw accelerometer data. The unit was kept flat when and moved in the +X direction 5 meters.

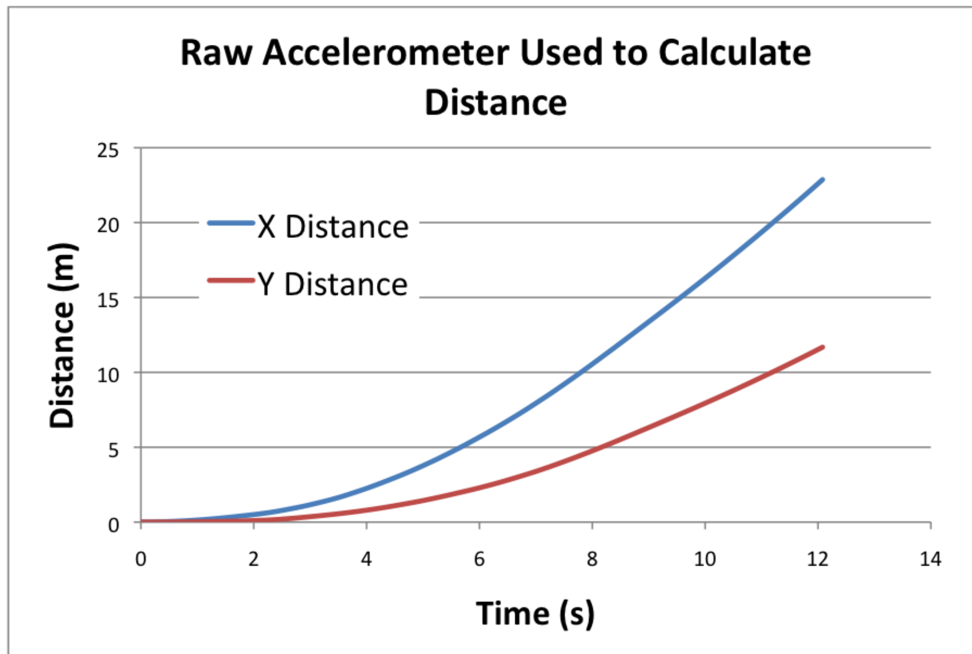

Figure 37: Raw Accel distance

Figure 37 shows the results of the baseline test and it was obvious that more accuracy would be needed. After analyzing the results the following we concluded that the inaccuracy in the results were due to how sensitive the system was. Reading the raw unprocessed acceleration data meant that if we tilted the X axis down then the system would assume that the user would be accelerating in the X direction at Earth's gravitational acceleration. We concluded from these results that we needed to find a way for the system to know which way the user accelerated no matter the orientation of the device.

Based off of the conclusions from the baseline test we determined that we needed to find a way for the system to know how it was moving within a global reference frame rather than its own localized frame. After some research we determined that the best way to do this was using a DCM. This meant that all three sensors would play a role in determining the user's location. After implementing the DCM code explained in the inertia software section, the accuracy in the system was dramatically better.

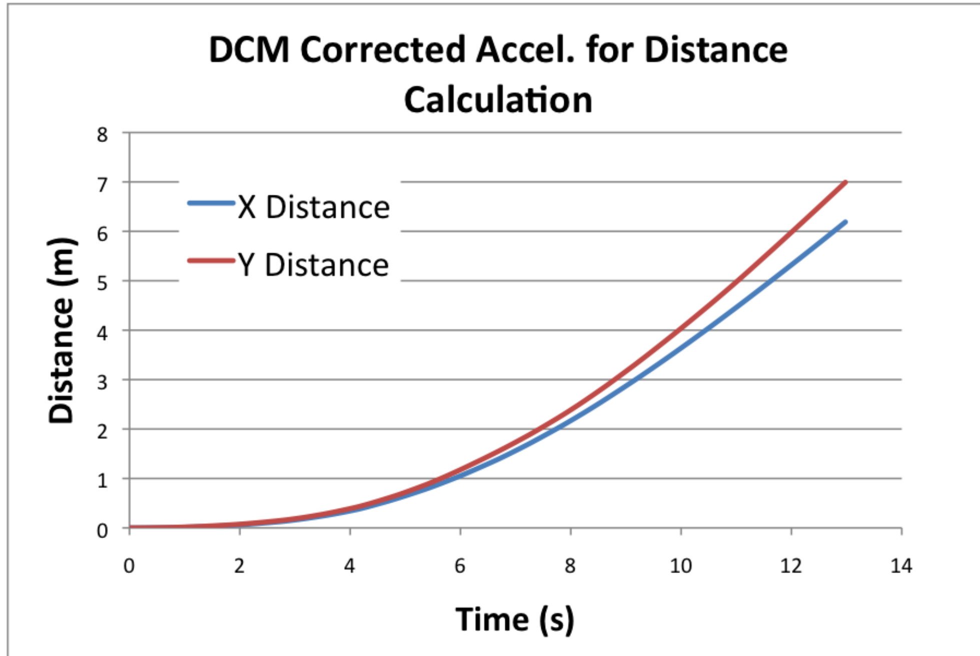

Figure 38: DCM Corrected Accel

The distance results from the DCM corrected accelerometer data show that the x distance is much more accurate than before. The evidence can be seen in Figure 38. This test was conducted the exact same way as the previous test, walking 5 meters in the +X direction. After reviewing this data we noticed that in both test iterations there was a very large drift in the +Y direction. Many types of filters and sampling methods were tried to lower this error, but nothing produced any results that were any better than what we had been getting from the DCM corrected distance.

Due to the large amount of error in the Y direction we figured the next part to check in the system was the DCM software loop. We figured that if we could characterize this section of the code we would be able to find a way to reduce the error in the system. During this test the device was held stationary and then rotated around the Y axis twice.

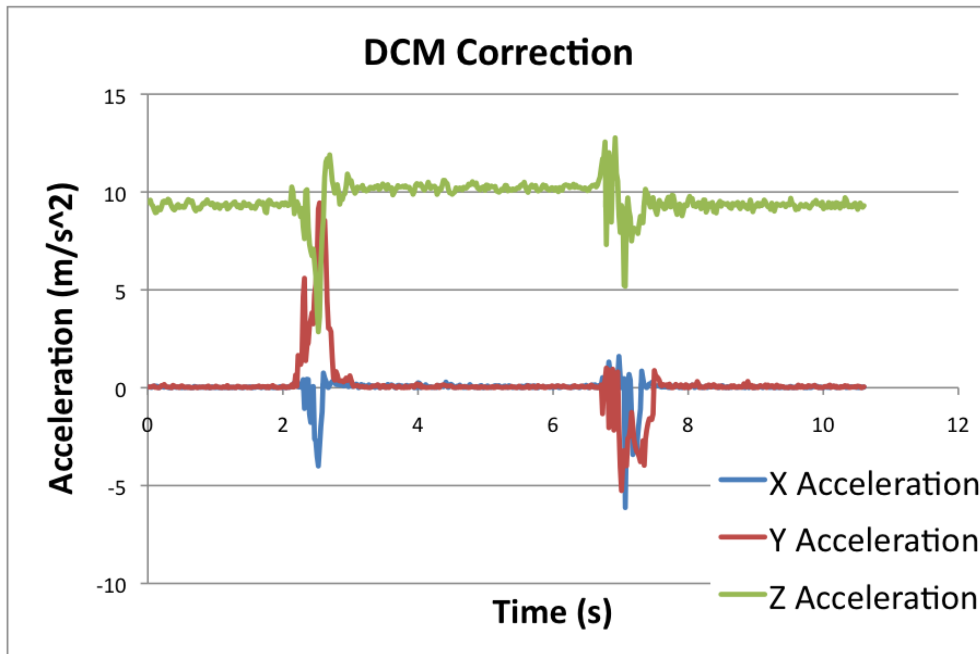

Figure 39: DCM Correction Loop Response

It was expected that there would be some oscillation in the acceleration data before it corrected itself however it was unknown how much the data would be affected. Figure 39 shows that there were acceleration spikes in the X and Y directions that were very large in magnitude. At this point it was clear that the system would not be as accurate as we hoped for due to the amount of error that takes place during movement. Do to the fact that the microcontroller was processing large amounts of data that was coming from three non-ideal sensors, any amount of error and drift in the system would be compounded through all the calculations. Although the system would eventually correct itself over time as seen in Figure 39. The All Purpose Mobile GPS would have to remain in a much more stable and steady environment than the one we had intended our device to be in.

We then changed our focus on finding distance. We determined that trying to find the direction of the user was going to be very inaccurate, so we changed our focus on trying to find the distance by using a pedometer. By multiplying the number of steps by a distance per step coefficient, a more accurate distance could be found. This meant however that the direction of this distance would be unknown.

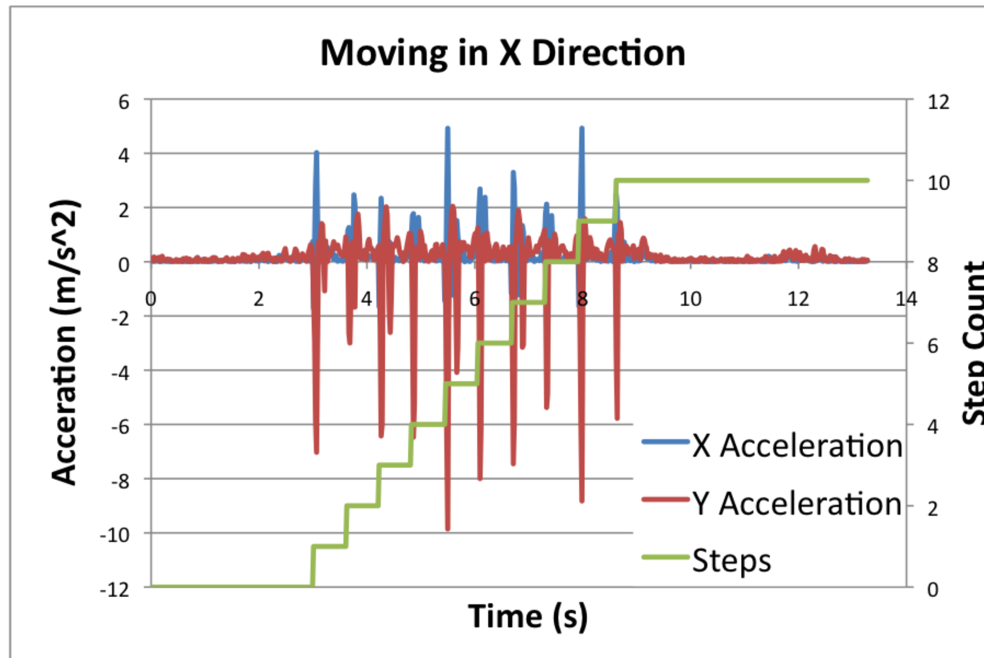

Figure 40: Pedometer Output

The pedometer algorithm shown in the inertia software section produced the output of steps seen in Figure 40. Ten steps were taken and it clearly shows the correct amount of steps with each acceleration pulse. The X and Y acceleration data shown in Figure 40 is the raw acceleration and not the averaged one used in the pedometer algorithm. These two accelerations showed us that it was definitely possible to characterize a user's step using the accelerometer. Also due to the fact that the pedometer algorithm used the absolute value of the X, Y, and Z accelerations, the orientation of the device did not matter as long as there was a spike seen in the acceleration data.

GPS communication functionality was verified with multiple UART to Terminal status and value print statements. Each stage of receiving, parsing, and handling was verified with UART debugging. Once communication was verified, GPS coordinates were verified with Google Maps and were accurate to within 1m when GPS was locked.

The SD card writing function was based off of an example from the Tivaware software package from TI. The original example used a basic terminal that the user would interface with in order to view the contents of an SD card. The original example used a slightly different model of microprocessor that featured a built in SD card. Fortunately, since both microprocessors used the same chip, it was rather straight forward to find the pins that the other board used to talk to the SD card reader and use the same ones on our micro. The example included a UART communication with the computer terminal to accept commands and also output information. This was a useful feature for debugging but it was unnecessary for the final product.

We were able to strain out the useful bit of code that data from the SD card and alter slightly to make it write data instead. Using the documentation from elm-chan.org on the FatFs module

used by TI, we were able to convert the read function into a write function that would successfully write full strings onto the SD card in a .txt file with name SD_Test.txt.

The next thing to do was to make sure that data written to the SD card would not overwrite existing data. Conveniently, there was a command to send the file pointer to the end of a file in order to append data and avoid overwriting it. By this point the main code had been boiled down to a few functions, and could be easily implemented into the GPS or IMU sections to allow the writing of data.

After this it was time to integrate the software. First, we attempted to integrate the GPS and SD software as they were complete and worked well individually. Since each section of software was created using a separate project, so that we could each work on separate parts individually, we were tasked with porting one piece of software into another in order to make a complete system. The SD software project contained a lengthier list of included files and libraries so it seemed reasonable to import the GPS code into the SD project. This proved to be more difficult than originally anticipated.

Creating a single executable C-file was not all that difficult, however it did not behave as expected. When bringing the code modules together, we used the TI onboard emulation to step through the code pedantically. We noted that multiple loop counters and other variables were simple being skipped when the compiler was running through the variable declarations causing their information to be absent from the registers, and being unable to execute as intended. This is the current state of the project as a reason for the unexpected behavior has yet to be found. We suspect there was an error linking the included files and libraries into one project, though this is uncertain as the project compiles perfectly and gives no tell tale errors. Further investigation is required.

Conclusion

Though we didn't get the opportunity to fully complete our objectives, we made big steps toward a functional product that, interestingly enough, is starting to be used and announced in modern technology conferences as a method for indoor mapping[7]. The accelerometer hardware that we used was only really good for small motions and determining which way is down due to the Earth's gravitational pull. This was most obvious when the distance calculations showed that we were drifting significantly in the Y direction. The intended use for this product was to be used for hours and we saw large amounts of error for just a few seconds. One of the largest issues that we ran into was trying to meet the specification of making a very robust system. This required the ability to have The System oriented in any way and still calculate the appropriate distances. However devices that use dead reckoning to determine a location have much more finely tuned systems that do not need to be as rugged as our's needed to be. There is also a lot of room in this project for future advancements. Indoor mapping is an increasing trend and is a challenge being met by some of the most powerful companies in consumer electronics. The addition of a wifi module would add the ability to tap into indoor wifi routers and

use the signal that they output to help calculate a location[9]. Another possibility would be to look into the optics side of indoor tracking. Using IR range sensors could help out with finding changes in distances by reading how much closer or further away you are from an object. The use of optics and computer vision are being researched to help with indoor mapping[10].

Another obstacle was the integration of different code modules to bring the functions of our project together. We verified the system did not run out of memory, and eliminated extraneous library dependencies, and even started from scratch, all to no avail. We surmise there may be some preprocessor directives or other optimizations that the compiler is performing that were causing some of our iteration variables to "disappear," but further testing is required.

Bibliography

[1] Starlino Electronics. "DCM Tutorial - An Introduction to Orientation Kinematics". Internet: http://www.starlino.com/dcm_tutorial.html, May 27, 2011[March 15, 2014]

[2] Texas Instruments. "TivaTM TM4C123GH6PM Microcontroller." Internet: http://www.ti.com/lit/ds/spms376d/spms376d.pdf. July 2013 [November 2013]

[3] Texas Instruments. "BoostXL-Sensehub Sensor Hub Booster Pack". Sensor Hub User manual, April 2013

[4] Invensense. "MPU-9150 Register Map and Descriptions". MPU-9150 Register Map. June 2011[September 2013]

[5] Invensense. "MPU-9150 Product Specification". MPU-9150 Datasheet. May 2011 [September 2013]

[6] ChaN. FatFs - Generic FAT File System Module. Internet: http://elm-chan.org/fsw/ff/00index_e.html May 2014 [May 2014]

[7] Jordan Kahn. "Apple taps into M7 & motion sensors for indoor positioning in iOS 8, signing up venues to contribute". Internet: http://9to5mac.com/2014/06/05/apple-taps-into-m7-motion-sensors-for-indoor-positioning-in-ios-8-signing-up-venues-to-contribute/#more-327149

[8] "TivaWare™ for C Series (Complete)". Internet: http://www.ti.com/tool/sw-tm4c

[9] Apple Inc. "iOS 5 : Understanding Location Services". Internet: http://support.apple.com/kb/HT4995. September 2013[June 5, 2014]

[10] Google. "ATAP Project Tango". Internet: https://www.google.com/atap/projecttango/#project.

[11] Spark Fun. "GPS Buying Guide". Internet: https://www.sparkfun.com/pages/GPS_Guide.

[12] K. Ozawa, *Lithium Ion Rechargeable Batteries*, Weinheim Wiley. New York. 2009

[13] R. Ford and C. Coulston, *Design for Electrical and Computer Engineers*, 1st edition: McGraw Hill, New York. 2008

[14] Adafruit. "Adafruit Ultimate GPS Breakout - 66 channel w/10 Hz updates" Internet: http://www.adafruit.com/products/746. 2014

[15] Adafruit. "Lithium Ion Polymer Battery - 3.7v 2500mAh". Internet: http://www.adafruit.com/products/328 2014

[16] Adafruit. "Medium 6V 2W Solar panel". Internet: http://www.adafruit.com/products/200 2014

[17] Adafruit. "USB / DC / Solar Lithium Ion/Polymer charger". Internet: http://www.adafruit.com/products/390 2014

[18] Sparkfun. "Breakout Board for microSD". Internet: https://www.sparkfun.com/products/544 2014

Appendix

Senior Project Analysis

• 1. Summary of Functional Requirements

The All Purpose Mobile GPS provides location points to the user without dependency on satellite communication. The mobile device includes solar charging capabilities and a compact durable package.

• 2. Primary Constraints

Size, weight, and power consumption provide the main design constraints for the All Purpose Mobile GPS. Those three factors provide the user with the best experience. Accuracy limits the device the device by the largest margin. A device that does not depend on satellite communication for positioning must have a design that retains high accuracy.

• 3. Economic[13]

The production of the All Purpose Mobile GPS results in various types of economic impacts.

Human Capital: A demand for manufacturers to build the device for sale. The lack of dependency on satellite communication creates a surplus of satellite manufacturers and

engineers. The device promotes exploration and travel, thus the movement of humans between nations increases.

Financial Capital: Manufacturing increases financial growth in the area where the manufacturing takes place. Manufacturing creates more money in the area surrounding the manufacturer. Taxation in these regions provides a stronger infrastructure in these neighborhoods. Investing in newer technologies related to the All Purpose GPS provides economic opportunities to innovators.

Manufactured Capital: Increase in the number of All Purpose Mobile GPS devices in the market. The market sees a large increase in automated manufacturing machinery and testing equipment. A decline in the number of other similar GPS systems may occur due to the increase competition of mobile navigation devices.

Natural Capital: All raw materials needed for production of the All Purpose Mobile GPS come from the Earth's natural resources. The device promotes exploration and an increase of National park admission could provide revenue for environmental sustainability.

The product's life cycle produces costs in different areas globally. The gathering of raw materials increases economic growth in the regions where they are gathered and manufactured. The sale of the device provides a local financial stimulus to the retailer. The use of the product might lead to an increase in travel that positively affects local economies.

The production of the device also causes damage locally. Possible corruption in local economies and governments could hinder the quality of the device. When the device promotes more economic damage than gain then the manufacturing of the All Purpose Mobile GPS ends.

We estimate the project costing around $250. Once we complete prototyping and testing then the cost per device decreases to provide room for profit. One member of the team funds the project thus creating a chief financial officer who becomes responsible for financial decisions in the future.

• 4. If manufactured on a commercial basis:

The number of devices sold per year increases over time. Device sales start low due to the low recognition of the product. over time, through advertising, product recognition rises along with the number of sold devices. We estimate selling 100 All Purpose Mobile GPS devices with a 10% increase every following year. The manufacturing cost of the device, when in full production, amounts to $70. Retailers are provided with a MSRP of $200 and a $120 wholesale price. We project a $5000 profit after one calendar year. We expect profits increasing 20% annually. The user experiences an increase in travel cost throughout the lifetime of the device.

• 5. Environmental[13]

The All Purpose Mobile GPS results in two environmental impacts. The first impact comes from manufacturing the device and delivering the product to the customer. The raw goods used to make the product come from the natural resources found across the globe. These raw goods need to ship to various manufacturing plants in other places across the globe as well thus producing more pollution. Once the raw goods are processed into the device components, they ship to the main manufacturing facility in the US. Once assembled, they ship to retailers and as a result more pollution takes place. Shipping causes the largest environmental impact. Once the user has the device then the environmental impact shifts to the user. The device design promotes exploration. Thus, the user could damage the environments that they visit.

• **6. Manufacturability[13]**

Initial low volume production slows manufacturing. as demand increases then the need for automated manufacturing increases. over time the cost of manufacturing the device decreases over time, but the jump to automated manufacturing necessitates a large financial investment. Testing the enclosure of the device to ensure that it remains waterproof and durable consumes the most time during the manufacturing process.

• **7. Sustainability**

Maintaining the device over the product's lifespan depends on how the user uses the device. The electronics can handle harsh environments so long as the housing holds to the specs. over time the possibility for making the device more environmentally sustainable increases. When in full production we can use recycled materials to make the housing and maximize the efficiency of production. Current infrastructure could limit modification of the design in the future.

• **8. Ethical**

The All Purpose Mobile GPS faces many ethical decisions. During the design phase, number seven of the IEEE code of ethics comes into play. It states "to seek, accept, and offer honest criticism of technical work, to acknowledge and correct errors, and to credit properly the contributions of others." Industry already contains similar designs to the All Purpose Mobile GPS. We must ensure the originality of the design and give credit to previous designs. The manufacturing phase needs to follow the "Platinum Rule". It states that others should be treated how they wish to be treated. Full production requires safe working conditions. Misuse of the product brings the utilitarianism ethical framework into question. It is unethical for a third party to use the All Purpose Mobile GPS to track someone even if for the benefit of the greater good. Finally the use of the device aims to meet the ethical egoism framework. It is within a user's best interest to know where they have been for their own records or to share using their free will.

• **9. Health and Safety[13]**

Health and safety is a primary concern for the whole lifecycle of the device. During the design phase testing the device in various conditions could endanger the tester. Taking safety precautions avoids most harm. Employees require a safe manufacturing environment. The working environment must have clean and well-labeled work conditions. Distraction creates the largest safety concern when using the All Purpose GPS. The design of the product centers around keeping the device not noticeable so that the user can focus on forging a trail ahead.

• **10. Social and Political**[13]

Retaining a safe work environment during the design phase avoids social and political issues. The manufacturing phase includes a few more social and political problems. A sharp rise in the local economy occurs in the raw goods' purchasing and processing locations. Stability in these locations ensures that corruption does not impede worker's rights or quality of product. Retaining a safe work environment and keeping competitive pay based on job responsibility keeps social and political issues away during the rest of the manufacturing phase. The use of the product faces the largest social issue. The device ensures that the user and the user alone can view where they have traveled. Governments and other third parties might want to use the device to keep track of others.

• **11. Development**

The process of developing a complete project plan is standard in industry but unique to the classroom experience. Other useful tools include learning new development suites for embedded processing and PCB fabrication.

Appendix A

```
//*****************************************************************************
//
// compdcm_mpu9150.c - Example use of the SensorLib with the MPU9150
//
// Copyright (c) 2013 Texas Instruments Incorporated.  All rights reserved.
// Software License Agreement
//
// Texas Instruments (TI) is supplying this software for use solely and
// exclusively on TI's microcontroller products. The software is owned by
// TI and/or its suppliers, and is protected under applicable copyright
// laws. You may not combine this software with "viral" open-source
// software in order to form a larger program.
//
// THIS SOFTWARE IS PROVIDED "AS IS" AND WITH ALL FAULTS.
// NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT
// NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. TI SHALL NOT, UNDER ANY
// CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL, OR CONSEQUENTIAL
// DAMAGES, FOR ANY REASON WHATSOEVER.
//
// This is part of revision 2.0.1.11577 of the EK-TM4C123GXL Firmware Package.
//
//*****************************************************************************

#include <stdint.h>
#include <stdbool.h>
#include <stdio.h>
#include <math.h>
#include "inc/hw_memmap.h"
#include "inc/hw_ints.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"
#include "sensorlib/hw_mpu9150.h"
#include "sensorlib/hw_ak8975.h"
#include "sensorlib/i2cm_drv.h"
#include "sensorlib/ak8975.h"
#include "sensorlib/mpu9150.h"
#include "sensorlib/comp_dcm.h"
#include "drivers/rgb.h"




//*****************************************************************************
//
//! \addtogroup example_list
//! <h1>Nine Axis Sensor Fusion with the MPU9150 and Complimentary-Filtered
//! DCM (compdcm_mpu9150)</h1>
```

```
//!
//! This example demonstrates the basic use of the Sensor Library, TM4C123G
//! LaunchPad and SensHub BoosterPack to obtain nine axis motion measurements
//! from the MPU9150.  The example fuses the nine axis measurements into a set
//! of Euler angles: roll, pitch and yaw.  It also produces the rotation
//! quaternions.  The fusion mechanism demonstrated is complimentary-filtered
//! direct cosine matrix (DCM) algorithm is provided as part of the Sensor
//! Library.
//!
//! Connect a serial terminal program to the LaunchPad's ICDI virtual serial
//! port at 115,200 baud.  Use eight bits per byte, no parity and one stop bit.
//! The raw sensor measurements, Euler angles and quaternions are printed to
//! the terminal.  The RGB LED begins to blink at 1Hz after initialization is
//! completed and the example application is running.
//
//*****************************************************************************

//*****************************************************************************
//
// Define MPU9150 I2C Address.
//
//*****************************************************************************
#define MPU9150_I2C_ADDRESS     0x68

//*****************************************************************************
//
// Global array for holding the color values for the RGB.
//
//*****************************************************************************
uint32_t g_pui32Colors[3];

//*****************************************************************************
//
// Global instance structure for the I2C master driver.
//
//*****************************************************************************
tI2CMInstance g_sI2CInst;

//*****************************************************************************
//
// Global instance structure for the ISL29023 sensor driver.
//
//*****************************************************************************
tMPU9150 g_sMPU9150Inst;

//*****************************************************************************
//
// Global Instance structure to manage the DCM state.
//
//*****************************************************************************
tCompDCM g_sCompDCMInst;



//*****************************************************************************
//
```

```
// Global flags to alert main that MPU9150 I2C transaction is complete
//
//*****************************************************************************
volatile uint_fast8_t g_vui8I2CDoneFlag;

//*****************************************************************************
//
// Global flags to alert main that MPU9150 I2C transaction error has occurred.
//
//*****************************************************************************
volatile uint_fast8_t g_vui8ErrorFlag;

//*****************************************************************************
//
// Global flags to alert main that MPU9150 data is ready to be retrieved.
//
//*****************************************************************************
volatile uint_fast8_t g_vui8DataFlag;

//*****************************************************************************
//
// Global counter to control and slow down the rate of data to the terminal.
//
//*****************************************************************************
#define PRINT_SKIP_COUNT        10


uint32_t g_ui32PrintSkipCounter;

//*****************************************************************************
//
// The error routine that is called if the driver library encounters an error.
//
//*****************************************************************************
#ifdef DEBUG
void
__error__(char *pcFilename, uint32_t ui32Line)
{
}
#endif

//*****************************************************************************
//
// MPU9150 Sensor callback function.  Called at the end of MPU9150 sensor
// driver transactions. This is called from I2C interrupt context. Therefore,
// we just set a flag and let main do the bulk of the computations and display.
//
//*****************************************************************************
void
MPU9150AppCallback(void *pvCallbackData, uint_fast8_t ui8Status)
{
    //
    // If the transaction succeeded set the data flag to indicate to
    // application that this transaction is complete and data may be ready.
    //
    if(ui8Status == I2CM_STATUS_SUCCESS)
```

```
    {
        g_vui8I2CDoneFlag = 1;
    }

    //
    // Store the most recent status in case it was an error condition
    //
    g_vui8ErrorFlag = ui8Status;
}

//*****************************************************************************
//
// Called by the NVIC as a result of GPIO port B interrupt event. For this
// application GPIO port B pin 2 is the interrupt line for the MPU9150
//
//*****************************************************************************
void
IntGPIOb(void)
{
    unsigned long ulStatus;

    ulStatus = GPIOIntStatus(GPIO_PORTB_BASE, true);

    //
    // Clear all the pin interrupts that are set
    //
    GPIOIntClear(GPIO_PORTB_BASE, ulStatus);

    if(ulStatus & GPIO_PIN_2)
    {
        //
        // MPU9150 Data is ready for retrieval and processing.
        //
        MPU9150DataRead(&g_sMPU9150Inst, MPU9150AppCallback, &g_sMPU9150Inst);
    }
}

//*****************************************************************************
//
// Called by the NVIC as a result of I2C3 Interrupt. I2C3 is the I2C connection
// to the MPU9150.
//
//*****************************************************************************
void
MPU9150I2CIntHandler(void)
{
    //
    // Pass through to the I2CM interrupt handler provided by sensor library.
    // This is required to be at application level so that I2CMIntHandler can
    // receive the instance structure pointer as an argument.
    //
    I2CMIntHandler(&g_sI2CInst);
}

//*****************************************************************************
//
```

```
// MPU9150 Application error handler. Show the user if we have encountered an
// I2C error.
//
//*****************************************************************************
void
MPU9150AppErrorHandler(char *pcFilename, uint_fast32_t ui32Line)
{
    //
    // Set terminal color to red and print error status and locations
    //
    UARTprintf("\033[31;1m");
    UARTprintf("Error: %d, File: %s, Line: %d\n"
               "See I2C status definitions in sensorlib\\i2cm_drv.h\n",
               g_vui8ErrorFlag, pcFilename, ui32Line);

    //
    // Return terminal color to normal
    //
    UARTprintf("\033[0m");

    //
    // Set RGB Color to RED
    //
    g_pui32Colors[0] = 0xFFFF;
    g_pui32Colors[1] = 0;
    g_pui32Colors[2] = 0;
    RGBColorSet(g_pui32Colors);

    //
    // Increase blink rate to get attention
    //
    RGBBlinkRateSet(10.0f);

    //
    // Go to sleep wait for interventions.  A more robust application could
    // attempt corrective actions here.
    //
    while(1)
    {
        //
        // Do Nothing
        //
    }
}

//*****************************************************************************
//
// Function to wait for the MPU9150 transactions to complete. Use this to spin
// wait on the I2C bus.
//
//*****************************************************************************
void
MPU9150AppI2CWait(char *pcFilename, uint_fast32_t ui32Line)
{
    //
    // Put the processor to sleep while we wait for the I2C driver to
```

```
    // indicate that the transaction is complete.
    //
    while((g_vui8I2CDoneFlag == 0) && (g_vui8ErrorFlag == 0))
    {
        //
        // Do Nothing
        //
    }

    //
    // If an error occurred call the error handler immediately.
    //
    if(g_vui8ErrorFlag)
    {
        MPU9150AppErrorHandler(pcFilename, ui32Line);
    }

    //
    // clear the data flag for next use.
    //
    g_vui8I2CDoneFlag = 0;
}

//*****************************************************************************
//
// Configure the UART and its pins.  This must be called before UARTprintf().
//
//*****************************************************************************
void
ConfigureUART(void)
{
    //
    // Enable the GPIO Peripheral used by the UART.
    //
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    //
    // Enable UART0
    //
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);

    //
    // Configure GPIO Pins for UART mode.
    //
    ROM_GPIOPinConfigure(GPIO_PA0_U0RX);
    ROM_GPIOPinConfigure(GPIO_PA1_U0TX);
    ROM_GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    //
    // Use the internal 16MHz oscillator as the UART clock source.
    //
    UARTClockSourceSet(UART0_BASE, UART_CLOCK_PIOSC);

    //
    // Initialize the UART for console I/O.
    //
```

```
        UARTStdioConfig(0, 115200, 16000000);
    }

    //Returns the integer part of the float in an integer form
    int32_t getInteger (float number)
    {
        int32_t temp;
        temp = (int32_t)number;
        return temp;
    }

    //Returns the decimal part of the float in an integer form
    int32_t getFraction (float number)
    {
        int32_t integer, fraction;
        integer = getInteger(number);
        fraction = (int32_t) (number * 1000.0f);

        fraction = fraction - (integer * 1000);

        if(fraction < 0)
        {
            fraction *= -1;
        }
        return fraction;

    }

    //Shifts the 4 int array, adds the new value, and averages the entire array
    int shiftRegisterSum (int newValue, int array[4])
    {
        int sum;

        array[3] = array[2];
        array[2] = array[1];
        array[1] = array[0];
        array[0] = newValue;

        sum = (array[0] + array[1] + array[2] + array [3])/4;
        return sum;
    }


    //Time cycle
    //delay calculation: desired sampling time = sysdelay count * 3 / Clock frequency
    #define delay 1600000
    #define time 0.12f

    //Metric used to convert a step in do a meaningful distance in meters
    #define stepDistance 2



    //*****************************************************************************
    //*****************************************************************************
    //*****************************************************************************
```

```
//*****************************************************************************
//*****************************************************************************
//*****************************************************************************
//*****************************************************************************
//*****************************************************************************
//*****************************************************************************
//*****************************************************************************
//*****************************************************************************
//*****************************************************************************
//
// Main application entry point.
//
//*****************************************************************************
//*****************************************************************************
//*****************************************************************************
//*****************************************************************************
//*****************************************************************************
//*****************************************************************************
//*****************************************************************************
//*****************************************************************************
//*****************************************************************************
int
main(void)
{
    uint_fast32_t ui32CompDCMStarted;
    int32_t accelSum = 0;

    float myDCM[3][3] = {{0,0,0},{0,0,0},{0,0,0}};
    float bodyX, bodyY, bodyZ;

    uint_fast16_t *rawAccel, *rawGyro, *rawMag;
    uint_fast16_t rawData[9];
    int rawX, rawY, rawZ;
    int xArray[4] = {0,0,0,0}, yArray[4] = {0,0,0,0}, zArray[4] = {0,0,0,0};
    int xSum, ySum, zSum;

    int stepCount = 0;
    int distance;
    int accelArray[2] = {0,0};


    rawAccel = rawData;
    rawGyro = rawData + 3;
    rawMag = rawData + 6;


    float pfData[16];
    float *pfAccel, *pfGyro, *pfMag, *pfEulers, *pfQuaternion;


    //
    // Initialize convenience pointers that clean up and clarify the code
    // meaning. We want all the data in a single contiguous array so that
    // we can make our pretty printing easier later.
    //
    pfAccel = pfData;
```

```
    pfGyro = pfData + 3;
    pfMag = pfData + 6;
    pfEulers = pfData + 9;
    pfQuaternion = pfData + 12;

    //
    // Setup the system clock to run at 40 Mhz from PLL with crystal reference
    //
    ROM_SysCtlClockSet(SYSCTL_SYSDIV_5 | SYSCTL_USE_PLL | SYSCTL_XTAL_16MHZ |
                       SYSCTL_OSC_MAIN);

    //
    // Enable port B used for motion interrupt.
    //
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);

    //
    // Initialize the UART.
    //
    ConfigureUART();

    //
    // Print the welcome message to the terminal.
    //
    //UARTprintf("\033[2JMPU9150 Raw Example\n");

    //
    // Set the color to a purple approximation.
    //
    g_pui32Colors[RED] = 0x8000;
    g_pui32Colors[BLUE] = 0x8000;
    g_pui32Colors[GREEN] = 0x0000;

    //
    // Initialize RGB driver.
    //
    RGBInit(0);
    RGBColorSet(g_pui32Colors);
    RGBIntensitySet(0.5f);
    RGBEnable();

    //
    // The I2C3 peripheral must be enabled before use.
    //
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C3);
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);

    //
    // Configure the pin muxing for I2C3 functions on port D0 and D1.
    //
    ROM_GPIOPinConfigure(GPIO_PD0_I2C3SCL);
    ROM_GPIOPinConfigure(GPIO_PD1_I2C3SDA);

    //
    // Select the I2C function for these pins.  This function will also
    // configure the GPIO pins pins for I2C operation, setting them to
```

```
    // open-drain operation with weak pull-ups.  Consult the data sheet
    // to see which functions are allocated per pin.
    //
    GPIOPinTypeI2CSCL(GPIO_PORTD_BASE, GPIO_PIN_0);
    ROM_GPIOPinTypeI2C(GPIO_PORTD_BASE, GPIO_PIN_1);

    //
    // Configure and Enable the GPIO interrupt. Used for INT signal from the
    // MPU9150
    //
    ROM_GPIOPinTypeGPIOInput(GPIO_PORTB_BASE, GPIO_PIN_2);
    GPIOIntEnable(GPIO_PORTB_BASE, GPIO_PIN_2);
    ROM_GPIOIntTypeSet(GPIO_PORTB_BASE, GPIO_PIN_2, GPIO_FALLING_EDGE);
    ROM_IntEnable(INT_GPIOB);

    //
    // Keep only some parts of the systems running while in sleep mode.
    // GPIOB is for the MPU9150 interrupt pin.
    // UART0 is the virtual serial port
    // TIMER0, TIMER1 and WTIMER5 are used by the RGB driver
    // I2C3 is the I2C interface to the ISL29023
    //
    ROM_SysCtlPeripheralClockGating(true);
    ROM_SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_GPIOB);
    ROM_SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_UART0);
    ROM_SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_TIMER0);
    ROM_SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_TIMER1);
    ROM_SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_I2C3);
    ROM_SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_WTIMER5);

    //
    // Enable interrupts to the processor.
    //
    ROM_IntMasterEnable();

    //
    // Initialize I2C3 peripheral.
    //
    I2CMInit(&g_sI2CInst, I2C3_BASE, INT_I2C3, 0xff, 0xff,
             ROM_SysCtlClockGet());

    //
    // Initialize the MPU9150 Driver.
    //
    MPU9150Init(&g_sMPU9150Inst, &g_sI2CInst, MPU9150_I2C_ADDRESS,
                MPU9150AppCallback, &g_sMPU9150Inst);

    //
    // Wait for transaction to complete
    //
    MPU9150AppI2CWait(__FILE__, __LINE__);

    //
    // Write application specific sensor configuration such as filter settings
    // and sensor range settings.
    //
```

```
    g_sMPU9150Inst.pui8Data[0] = MPU9150_CONFIG_DLPF_CFG_94_98;
    g_sMPU9150Inst.pui8Data[1] = MPU9150_GYRO_CONFIG_FS_SEL_250;
    g_sMPU9150Inst.pui8Data[2] = (MPU9150_ACCEL_CONFIG_ACCEL_HPF_5HZ |
                                  MPU9150_ACCEL_CONFIG_AFS_SEL_2G);
    MPU9150Write(&g_sMPU9150Inst, MPU9150_O_CONFIG, g_sMPU9150Inst.pui8Data, 3,
                 MPU9150AppCallback, &g_sMPU9150Inst);

    //
    // Wait for transaction to complete
    //
    MPU9150AppI2CWait(__FILE__, __LINE__);

    //
    // Configure the data ready interrupt pin output of the MPU9150.
    //
    g_sMPU9150Inst.pui8Data[0] = MPU9150_INT_PIN_CFG_INT_LEVEL |
                                    MPU9150_INT_PIN_CFG_INT_RD_CLEAR |
                                    MPU9150_INT_PIN_CFG_LATCH_INT_EN;
    g_sMPU9150Inst.pui8Data[1] = MPU9150_INT_ENABLE_DATA_RDY_EN;
    MPU9150Write(&g_sMPU9150Inst, MPU9150_O_INT_PIN_CFG,
                 g_sMPU9150Inst.pui8Data, 2, MPU9150AppCallback,
                 &g_sMPU9150Inst);

    //
    // Wait for transaction to complete
    //
    MPU9150AppI2CWait(__FILE__, __LINE__);

    //
    // Initialize the DCM system. 50 hz sample rate.
    // accel weight = .2, gyro weight = .8, mag weight = .2
    //
    CompDCMInit(&g_sCompDCMInst, 1.0f / 500.0f, 0.2f, 0.6f, 0.2f);


    //
    // Enable blinking indicates config finished successfully
    //
    RGBBlinkRateSet(1.0f);

    ui32CompDCMStarted = 0;



    while(1)
    {
        //
        // Go to sleep mode while waiting for data ready.
        //
        while(!g_vui8I2CDoneFlag)
        {
            ROM_SysCtlSleep();
        }

        //
        // Clear the flag
```

```
        //
        g_vui8I2CDoneFlag = 0;

        //
        // Get floating point version of the Accel Data in m/s^2.
        //
        MPU9150DataAccelGetFloat(&g_sMPU9150Inst, pfAccel, pfAccel + 1,
                                 pfAccel + 2);

        //
        // Get floating point version of angular velocities in rad/sec
        //
        MPU9150DataGyroGetFloat(&g_sMPU9150Inst, pfGyro, pfGyro + 1,
                                pfGyro + 2);

        //
        // Get floating point version of magnetic fields strength in tesla
        //
        MPU9150DataMagnetoGetFloat(&g_sMPU9150Inst, pfMag, pfMag + 1,
                                   pfMag + 2);

        MPU9150DataAccelGetRaw(&g_sMPU9150Inst, rawAccel, rawAccel + 1, rawAccel
            + 2);
        MPU9150DataGyroGetRaw(&g_sMPU9150Inst, rawGyro, rawGyro + 1, rawGyro + 2)
            ;
        MPU9150DataMagnetoGetRaw(&g_sMPU9150Inst, rawMag, rawMag + 1, rawMag + 2)
            ;


        //
        // Check if this is our first data ever.
        //
        if(ui32CompDCMStarted == 0)
        {
            //
            // Set flag indicating that DCM is started.
            // Perform the seeding of the DCM with the first data set.
            //
            ui32CompDCMStarted = 1;
            CompDCMMagnetoUpdate(&g_sCompDCMInst, pfMag[0], pfMag[1],
                                 pfMag[2]);
            CompDCMAccelUpdate(&g_sCompDCMInst, pfAccel[0], pfAccel[1],
                               pfAccel[2]);
            CompDCMGyroUpdate(&g_sCompDCMInst, pfGyro[0], pfGyro[1],
                              pfGyro[2]);
            CompDCMStart(&g_sCompDCMInst);
        }
        else
        {
            //
            // DCM Is already started.  Perform the incremental update.
            //
            CompDCMMagnetoUpdate(&g_sCompDCMInst, pfMag[0], pfMag[1],
                                 pfMag[2]);
            CompDCMAccelUpdate(&g_sCompDCMInst, pfAccel[0], pfAccel[1],
                               pfAccel[2]);
```

```
            CompDCMGyroUpdate(&g_sCompDCMInst, -pfGyro[0], -pfGyro[1],
                              -pfGyro[2]);
            CompDCMUpdate(&g_sCompDCMInst);
        }

        //sampling period time
        //SysCtlDelay(delay);


        //Used to print off raw Accelerometer data as a float
        //UARTprintf("%3d.%03d\t", getInteger(pfData[0]),
            getFraction(pfData[0]));
        //UARTprintf("%3d.%03d\t", getInteger(pfData[1]),
            getFraction(pfData[1]));
        //UARTprintf("%3d.%03d\n", getInteger(pfData[2]),
            getFraction(pfData[2]));

        //Reads accelerometer data as a 16 bit 2's complement value
        rawX = (int16_t)rawData[0];
        rawY = (int16_t)rawData[1];
        rawZ = (int16_t)rawData[2];

        //Finds the average of the 4 samples
        xSum = shiftRegisterSum(rawX, xArray);
        ySum = shiftRegisterSum(rawY, yArray);
        zSum = shiftRegisterSum(rawZ, zArray);

        //Finds the magnitude of the 3 accelerations
        accelSum = sqrt(abs(xSum)*abs(xSum) + abs(ySum)*abs(ySum) + abs(zSum)*abs
            (zSum));

        //Shifts the magnitude of the acceleration data
        accelArray[0] = accelArray[1];
        accelArray[1] = accelSum;

        //Conditional statement looking for step
        if(accelArray[0] < 20000 && accelArray[1] >= 20000)
        {
            stepCount++;
        }

        distance = stepCount * stepDistance;

        //Print statements used for outputting characteristics of a step
        //UARTprintf("%d,%d,%d \n", xSum, ySum, zSum);
        //UARTprintf("%d\n", accelSum);
        //UARTprintf("%d\n", stepCount);
        //UARTprintf("%d\n", distance);

        //Returns the global Directional cosine matrix
        CompDCMMatrixGet(&g_sCompDCMInst, myDCM);

        //Transformed vector math
        bodyX = (myDCM[0][0] * pfData[0]) + (myDCM[0][1] * pfData[1]) + (myDCM[0]
            [2] * pfData[2]);
        bodyY = (myDCM[1][0] * pfData[0]) + (myDCM[1][1] * pfData[1]) + (myDCM[1]
```

```
            [2] * pfData[2]);
        bodyZ = (myDCM[2][0] * pfData[0]) + (myDCM[2][1] * pfData[1]) + (myDCM[2]
            [2] * pfData[2]);


        //Used to print off the global DCM
        /*UARTprintf("%3d.%03d\t", getInteger(myDCM[0][0]), getFraction(myDCM[0]
            [0]));
        UARTprintf("%3d.%03d\t", getInteger(myDCM[0][1]), getFraction(myDCM[0]
            [1]));
        UARTprintf("%3d.%03d\n", getInteger(myDCM[0][2]), getFraction(myDCM[0]
            [2]));
        UARTprintf("%3d.%03d\t", getInteger(myDCM[1][0]), getFraction(myDCM[1]
            [0]));
        UARTprintf("%3d.%03d\t", getInteger(myDCM[1][1]), getFraction(myDCM[1]
            [1]));
        UARTprintf("%3d.%03d\n", getInteger(myDCM[1][2]), getFraction(myDCM[1]
            [2]));
        UARTprintf("%3d.%03d\t", getInteger(myDCM[2][0]), getFraction(myDCM[2]
            [0]));
        UARTprintf("%3d.%03d\t", getInteger(myDCM[2][1]), getFraction(myDCM[2]
            [1]));
        UARTprintf("%3d.%03d\n", getInteger(myDCM[2][2]), getFraction(myDCM[2]
            [2]));
        UARTprintf("\n");
        UARTprintf("\n");
        UARTprintf("\n");
        UARTprintf("\n");
        UARTprintf("\n");*/


        //Used to print off the DCM corrected Acceleration
        //UARTprintf("%3d.%03d\t", getInteger(bodyX), getFraction(bodyX));
        //UARTprintf("%3d.%03d\t", getInteger(bodyY), getFraction(bodyY));
        //UARTprintf("%3d.%03d\n", getInteger(bodyZ), getFraction(bodyZ));





    }
}
```

Appendix B

```
//*****************************************************************************
//
// comp_dcm.c - Complementary filter algorithm on a Direction Cosine Matrix for
//              fusing sensor data from an accelerometer, gyroscope, and
//              magnetometer.
//
// Copyright (c) 2012-2013 Texas Instruments Incorporated.  All rights reserved.
// Software License Agreement
//
// Texas Instruments (TI) is supplying this software for use solely and
// exclusively on TI's microcontroller products. The software is owned by
// TI and/or its suppliers, and is protected under applicable copyright
// laws. You may not combine this software with "viral" open-source
// software in order to form a larger program.
//
// THIS SOFTWARE IS PROVIDED "AS IS" AND WITH ALL FAULTS.
// NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT
// NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. TI SHALL NOT, UNDER ANY
// CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL, OR CONSEQUENTIAL
// DAMAGES, FOR ANY REASON WHATSOEVER.
//
// This is part of revision 2.0.1.11577 of the Tiva Firmware Development Package.
//
//*****************************************************************************

#include <math.h>
#include <stdbool.h>
#include <stdint.h>
#include "driverlib/debug.h"
#include "sensorlib/comp_dcm.h"
#include "sensorlib/vector.h"

//*****************************************************************************
//
//! \addtogroup comp_dcm_api
//! @{
//
//*****************************************************************************

//*****************************************************************************
//
// If M_PI has not been defined by the system headers, define it here.
//
//*****************************************************************************
#ifndef M_PI
#define M_PI                    3.14159265358979323846
#endif

//*****************************************************************************
//
//! Initializes the complementary filter DCM attitude estimation state.
//!
```

```
//! \param psDCM is a pointer to the DCM state structure.
//! \param fDeltaT is the amount of time between DCM updates, in seconds.
//! \param fScaleA is the weight of the accelerometer reading in determining
//! the updated attitude estimation.
//! \param fScaleG is the weight of the gyroscope reading in determining the
//! updated attitude estimation.
//! \param fScaleM is the weight of the magnetometer reading in determining the
//! updated attitude estimation.
//!
//! This function initializes the complementary filter DCM attitude estimation
//! state, and must be called prior to performing any attitude estimation.
//!
//! New readings must be supplied to the complementary filter DCM attitude
//! estimation algorithm at the rate specified by the \e fDeltaT parameter.
//! Failure to provide new readings at this rate results in inaccuracies in the
//! attitude estimation.
//!
//! The \e fScaleA, \e fScaleG, and \e fScaleM weights must sum to one.
//!
//! \return None.
//
//*****************************************************************************
void
CompDCMInit(tCompDCM *psDCM, float fDeltaT, float fScaleA, float fScaleG,
            float fScaleM)
{
    //
    // Initialize the DCM matrix to the identity matrix.
    //
    psDCM->ppfDCM[0][0] = 1.0;
    psDCM->ppfDCM[0][1] = 0.0;
    psDCM->ppfDCM[0][2] = 0.0;
    psDCM->ppfDCM[1][0] = 0.0;
    psDCM->ppfDCM[1][1] = 1.0;
    psDCM->ppfDCM[1][2] = 0.0;
    psDCM->ppfDCM[2][0] = 0.0;
    psDCM->ppfDCM[2][1] = 0.0;
    psDCM->ppfDCM[2][2] = 1.0;

    //
    // Save the time delta between DCM updates.
    //
    psDCM->fDeltaT = fDeltaT;

    //
    // Save the scaling factors that are applied to the accelerometer,
    // gyroscope, and magnetometer readings.
    //
    psDCM->fScaleA = fScaleA;
    psDCM->fScaleG = fScaleG;
    psDCM->fScaleM = fScaleM;
}

//*****************************************************************************
//
//! Updates the accelerometer reading used by the complementary filter DCM
```

```
//! algorithm.
//!
//! \param psDCM is a pointer to the DCM state structure.
//! \param fAccelX is the accelerometer reading in the X body axis.
//! \param fAccelY is the accelerometer reading in the Y body axis.
//! \param fAccelZ is the accelerometer reading in the Z body axis.
//!
//! This function updates the accelerometer reading used by the complementary
//! filter DCM algorithm.  The accelerometer readings provided to this function
//! are used by subsequent calls to CompDCMStart() and CompDCMUpdate() to
//! compute the attitude estimate.
//!
//! \return None.
//
//*****************************************************************************
void
CompDCMAccelUpdate(tCompDCM *psDCM, float fAccelX, float fAccelY,
                   float fAccelZ)
{
    //
    // The user should never pass in values that are not-a-number
    //
    ASSERT(!isnan(fAccelX));
    ASSERT(!isnan(fAccelY));
    ASSERT(!isnan(fAccelZ));

    //
    // Save the new accelerometer reading.
    //
    psDCM->pfAccel[0] = fAccelX;
    psDCM->pfAccel[1] = fAccelY;
    psDCM->pfAccel[2] = fAccelZ;
}

//*****************************************************************************
//
//! Updates the gyroscope reading used by the complementary filter DCM
//! algorithm.
//!
//! \param psDCM is a pointer to the DCM state structure.
//! \param fGyroX is the gyroscope reading in the X body axis.
//! \param fGyroY is the gyroscope reading in the Y body axis.
//! \param fGyroZ is the gyroscope reading in the Z body axis.
//!
//! This function updates the gyroscope reading used by the complementary
//! filter DCM algorithm.  The gyroscope readings provided to this function are
//! used by subsequent calls to CompDCMUpdate() to compute the attitude
//! estimate.
//!
//! \return None.
//
//*****************************************************************************
void
CompDCMGyroUpdate(tCompDCM *psDCM, float fGyroX, float fGyroY, float fGyroZ)
{
    //
```

```
    // The user should never pass in values that are not-a-number
    //
    ASSERT(!isnan(fGyroX));
    ASSERT(!isnan(fGyroY));
    ASSERT(!isnan(fGyroZ));

    //
    // Save the new gyroscope reading.
    //
    psDCM->pfGyro[0] = fGyroX;
    psDCM->pfGyro[1] = fGyroY;
    psDCM->pfGyro[2] = fGyroZ;
}

//*****************************************************************************
//
//! Updates the magnetometer reading used by the complementary filter DCM
//! algorithm.
//!
//! \param psDCM is a pointer to the DCM state structure.
//! \param fMagnetoX is the magnetometer reading in the X body axis.
//! \param fMagnetoY is the magnetometer reading in the Y body axis.
//! \param fMagnetoZ is the magnetometer reading in the Z body axis.
//!
//! This function updates the magnetometer reading used by the complementary
//! filter DCM algorithm.  The magnetometer readings provided to this function
//! are used by subsequent calls to CompDCMStart() and CompDCMUpdate() to
//! compute the attitude estimate.
//!
//! \return None.
//
//*****************************************************************************
void
CompDCMMagnetoUpdate(tCompDCM *psDCM, float fMagnetoX, float fMagnetoY,
                     float fMagnetoZ)
{
    //
    // The user should never pass in values that are not-a-number
    //
    ASSERT(!isnan(fMagnetoX));
    ASSERT(!isnan(fMagnetoY));
    ASSERT(!isnan(fMagnetoZ));

    //
    // Save the new magnetometer reading.
    //
    psDCM->pfMagneto[0] = fMagnetoX;
    psDCM->pfMagneto[1] = fMagnetoY;
    psDCM->pfMagneto[2] = fMagnetoZ;
}

//*****************************************************************************
//
//! Starts the complementary filter DCM attitude estimation from an initial
//! sensor reading.
//!
```

```
//! \param psDCM is a pointer to the DCM state structure.
//!
//! This function computes the initial complementary filter DCM attitude
//! estimation state based on the initial accelerometer and magnetometer
//! reading.  While not necessary for the attitude estimation to converge,
//! using an initial state based on sensor readings results in quicker
//! convergence.
//!
//! \return None.
//
//*****************************************************************************
void
CompDCMStart(tCompDCM *psDCM)
{
    float pfI[3], pfJ[3], pfK[3];

    //
    // The magnetometer reading forms the initial I vector, pointing north.
    //
    pfI[0] = psDCM->pfMagneto[0];
    pfI[1] = psDCM->pfMagneto[1];
    pfI[2] = psDCM->pfMagneto[2];

    //
    // The accelerometer reading forms the initial K vector, pointing down.
    //
    pfK[0] = psDCM->pfAccel[0];
    pfK[1] = psDCM->pfAccel[1];
    pfK[2] = psDCM->pfAccel[2];

    //
    // Compute the initial J vector, which is the cross product of the K and I
    // vectors.
    //
    VectorCrossProduct(pfJ, pfK, pfI);

    //
    // Recompute the I vector from the cross product of the J and K vectors.
    // This makes it fully orthogonal, which it wasn't before since magnetic
    // north points inside the Earth in many places.
    //
    VectorCrossProduct(pfI, pfJ, pfK);

    //
    // Normalize the I, J, and K vectors.
    //
    VectorScale(pfI, pfI, 1 / sqrtf(VectorDotProduct(pfI, pfI)));
    VectorScale(pfJ, pfJ, 1 / sqrtf(VectorDotProduct(pfJ, pfJ)));
    VectorScale(pfK, pfK, 1 / sqrtf(VectorDotProduct(pfK, pfK)));

    //
    // Initialize the DCM matrix from the I, J, and K vectors.
    //
    psDCM->ppfDCM[0][0] = pfI[0];
    psDCM->ppfDCM[0][1] = pfI[1];
    psDCM->ppfDCM[0][2] = pfI[2];
```

```
    psDCM->ppfDCM[1][0] = pfJ[0];
    psDCM->ppfDCM[1][1] = pfJ[1];
    psDCM->ppfDCM[1][2] = pfJ[2];
    psDCM->ppfDCM[2][0] = pfK[0];
    psDCM->ppfDCM[2][1] = pfK[1];
    psDCM->ppfDCM[2][2] = pfK[2];
}

//*****************************************************************************
//
//! Updates the complementary filter DCM attitude estimation based on an
//! updated set of sensor readings.
//!
//! \param psDCM is a pointer to the DCM state structure.
//!
//! This function updates the complementary filter DCM attitude estimation
//! state based on the current sensor readings.  This function must be called
//! at the rate specified to CompDCMInit(), with new readings supplied at an
//! appropriate rate (for example, magnetometers typically sample at a much
//! slower rate than accelerometers and gyroscopes).
//!
//! \return None.
//
//*****************************************************************************
void
CompDCMUpdate(tCompDCM *psDCM)
{
    float pfI[3], pfJ[3], pfK[3], pfDelta[3], pfTemp[3], fError;
    bool bNAN;

    //
    // The magnetometer reading forms the new Im vector, pointing north.
    //
    pfI[0] = psDCM->pfMagneto[0];
    pfI[1] = psDCM->pfMagneto[1];
    pfI[2] = psDCM->pfMagneto[2];

    //
    // The accelerometer reading forms the new Ka vector, pointing down.
    //
    pfK[0] = psDCM->pfAccel[0];
    pfK[1] = psDCM->pfAccel[1];
    pfK[2] = psDCM->pfAccel[2];

    //
    // Compute the new J vector, which is the cross product of the Ka and Im
    // vectors.
    //
    VectorCrossProduct(pfJ, pfK, pfI);

    //
    // Recompute the Im vector from the cross product of the J and Ka vectors.
    // This makes it fully orthogonal, which it wasn't before since magnetic
    // north points inside the Earth in many places.
    //
    VectorCrossProduct(pfI, pfJ, pfK);
```

```
    //
    // Normalize the Im and Ka vectors.
    //
    VectorScale(pfI, pfI, 1 / sqrtf(VectorDotProduct(pfI, pfI)));
    VectorScale(pfK, pfK, 1 / sqrtf(VectorDotProduct(pfK, pfK)));

    //
    // Compute and scale the rotation as inferred from the accelerometer,
    // storing it in the rotation accumulator.
    //
    VectorCrossProduct(pfTemp, psDCM->ppfDCM[2], pfK);
    VectorScale(pfDelta, pfTemp, psDCM->fScaleA);

    //
    // Compute and scale the rotation as measured by the gyroscope, adding it
    // to the rotation accumulator.
    //
    pfTemp[0] = psDCM->pfGyro[0] * psDCM->fDeltaT * psDCM->fScaleG;
    pfTemp[1] = psDCM->pfGyro[1] * psDCM->fDeltaT * psDCM->fScaleG;
    pfTemp[2] = psDCM->pfGyro[2] * psDCM->fDeltaT * psDCM->fScaleG;
    VectorAdd(pfDelta, pfDelta, pfTemp);

    //
    // Compute and scale the rotation as inferred from the magnetometer, adding
    // it to the rotation accumulator.
    //
    VectorCrossProduct(pfTemp, psDCM->ppfDCM[0], pfI);
    VectorScale(pfTemp, pfTemp, psDCM->fScaleM);
    VectorAdd(pfDelta, pfDelta, pfTemp);

    //
    // Rotate the I vector from the DCM matrix by the scaled rotation.
    //
    VectorCrossProduct(pfI, pfDelta, psDCM->ppfDCM[0]);
    VectorAdd(psDCM->ppfDCM[0], psDCM->ppfDCM[0], pfI);

    //
    // Rotate the K vector from the DCM matrix by the scaled rotation.
    //
    VectorCrossProduct(pfK, pfDelta, psDCM->ppfDCM[2]);
    VectorAdd(psDCM->ppfDCM[2], psDCM->ppfDCM[2], pfK);

    //
    // Compute the orthogonality error between the rotated I and K vectors and
    // adjust each by half the error, bringing them closer to orthogonality.
    //
    fError = VectorDotProduct(psDCM->ppfDCM[0], psDCM->ppfDCM[2]) / -2.0;
    VectorScale(pfI, psDCM->ppfDCM[0], fError);
    VectorScale(pfK, psDCM->ppfDCM[2], fError);
    VectorAdd(psDCM->ppfDCM[0], psDCM->ppfDCM[0], pfK);
    VectorAdd(psDCM->ppfDCM[2], psDCM->ppfDCM[2], pfI);

    //
    // Normalize the I and K vectors.
    //
```

```
    VectorScale(psDCM->ppfDCM[0], psDCM->ppfDCM[0],
                0.5 * (3.0 - VectorDotProduct(psDCM->ppfDCM[0],
                                              psDCM->ppfDCM[0])));
    VectorScale(psDCM->ppfDCM[2], psDCM->ppfDCM[2],
                0.5 * (3.0 - VectorDotProduct(psDCM->ppfDCM[2],
                                              psDCM->ppfDCM[2])));

    //
    // Compute the rotated J vector from the cross product of the rotated,
    // corrected K and I vectors.
    //
    VectorCrossProduct(psDCM->ppfDCM[1], psDCM->ppfDCM[2], psDCM->ppfDCM[0]);

    //
    // Determine if the newly updated DCM contains any invalid (in other words,
    // NaN) values.
    //
    bNAN = (isnan(psDCM->ppfDCM[0][0]) ||
            isnan(psDCM->ppfDCM[0][1]) ||
            isnan(psDCM->ppfDCM[0][2]) ||
            isnan(psDCM->ppfDCM[1][0]) ||
            isnan(psDCM->ppfDCM[1][1]) ||
            isnan(psDCM->ppfDCM[1][2]) ||
            isnan(psDCM->ppfDCM[2][0]) ||
            isnan(psDCM->ppfDCM[2][1]) ||
            isnan(psDCM->ppfDCM[2][2]));

    //
    // As a debug measure, we check for NaN in the DCM.  The user can trap
    // this event depending on their implementation of __error__.  Should they
    // choose to disable interrupts and loop forever then they will have
    // preserved the stack and can analyze how they arrived at NaN.
    //
    ASSERT(!bNAN);

    //
    // If any part of the matrix is not-a-number then reset the DCM back to the
    // identity matrix.
    //
    if(bNAN)
    {
        psDCM->ppfDCM[0][0] = 1.0;
        psDCM->ppfDCM[0][1] = 0.0;
        psDCM->ppfDCM[0][2] = 0.0;
        psDCM->ppfDCM[1][0] = 0.0;
        psDCM->ppfDCM[1][1] = 1.0;
        psDCM->ppfDCM[1][2] = 0.0;
        psDCM->ppfDCM[2][0] = 0.0;
        psDCM->ppfDCM[2][1] = 0.0;
        psDCM->ppfDCM[2][2] = 1.0;
    }
}

//*****************************************************************************
//
//! Returns the current DCM attitude estimation matrix.
```

```
//!
//! \param psDCM is a pointer to the DCM state structure.
//! \param ppfDCM is a pointer to the array into which to store the DCM matrix
//! values.
//!
//! This function returns the current value of the DCM matrix.
//!
//! \return None.
//
//*****************************************************************************
void
CompDCMMatrixGet(tCompDCM *psDCM, float ppfDCM[3][3])
{
    //
    // Return the current DCM matrix.
    //
    ppfDCM[0][0] = psDCM->ppfDCM[0][0];
    ppfDCM[0][1] = psDCM->ppfDCM[0][1];
    ppfDCM[0][2] = psDCM->ppfDCM[0][2];
    ppfDCM[1][0] = psDCM->ppfDCM[1][0];
    ppfDCM[1][1] = psDCM->ppfDCM[1][1];
    ppfDCM[1][2] = psDCM->ppfDCM[1][2];
    ppfDCM[2][0] = psDCM->ppfDCM[2][0];
    ppfDCM[2][1] = psDCM->ppfDCM[2][1];
    ppfDCM[2][2] = psDCM->ppfDCM[2][2];
}

//*****************************************************************************
//
//! Computes the Euler angles from the DCM attitude estimation matrix.
//!
//! \param psDCM is a pointer to the DCM state structure.
//! \param pfRoll is a pointer to the value into which the roll is stored.
//! \param pfPitch is a pointer to the value into which the pitch is stored.
//! \param pfYaw is a pointer to the value into which the yaw is stored.
//!
//! This function computes the Euler angles that are represented by the DCM
//! attitude estimation matrix.  If any of the Euler angles is not required,
//! the corresponding parameter can be \b NULL.
//!
//! \return None.
//
//*****************************************************************************
void
CompDCMComputeEulers(tCompDCM *psDCM, float *pfRoll, float *pfPitch,
                     float *pfYaw)
{
    //
    // Compute the roll, pitch, and yaw as required.
    //
    if(pfRoll)
    {
        *pfRoll = atan2f(psDCM->ppfDCM[2][1], psDCM->ppfDCM[2][2]);
    }
    if(pfPitch)
    {
```

```
        *pfPitch = -asinf(psDCM->ppfDCM[2][0]);
    }
    if(pfYaw)
    {
        *pfYaw = atan2f(psDCM->ppfDCM[1][0], psDCM->ppfDCM[0][0]);
    }
}

//*****************************************************************************
//
//! Computes the quaternion from the DCM attitude estimation matrix.
//!
//! \param psDCM is a pointer to the DCM state structure.
//! \param pfQuaternion is an array into which the quaternion is stored.
//!
//! This function computes the quaternion that is represented by the DCM
//! attitude estimation matrix.
//!
//! \return None.
//
//*****************************************************************************
void
CompDCMComputeQuaternion(tCompDCM *psDCM, float pfQuaternion[4])
{
    float fQs, fQx, fQy, fQz;

    //
    // Partially compute Qs, Qx, Qy, and Qz based on the DCM diagonals.  The
    // square root, an expensive operation, is computed for only one of these
    // as determined later.
    //
    fQs = 1 + psDCM->ppfDCM[0][0] + psDCM->ppfDCM[1][1] + psDCM->ppfDCM[2][2];
    fQx = 1 + psDCM->ppfDCM[0][0] - psDCM->ppfDCM[1][1] - psDCM->ppfDCM[2][2];
    fQy = 1 - psDCM->ppfDCM[0][0] + psDCM->ppfDCM[1][1] - psDCM->ppfDCM[2][2];
    fQz = 1 - psDCM->ppfDCM[0][0] - psDCM->ppfDCM[1][1] + psDCM->ppfDCM[2][2];

    //
    // See if Qs is the largest of the diagonal values.
    //
    if((fQs > fQx) && (fQs > fQy) && (fQs > fQz))
    {
        //
        // Finish the computation of Qs.
        //
        fQs = sqrtf(fQs) / 2;

        //
        // Compute the values of the quaternion based on Qs.
        //
        pfQuaternion[0] = fQs;
        pfQuaternion[1] = ((psDCM->ppfDCM[2][1] - psDCM->ppfDCM[1][2]) /
                           (4 * fQs));
        pfQuaternion[2] = ((psDCM->ppfDCM[0][2] - psDCM->ppfDCM[2][0]) /
                           (4 * fQs));
        pfQuaternion[3] = ((psDCM->ppfDCM[1][0] - psDCM->ppfDCM[0][1]) /
                           (4 * fQs));
```

```
    }

    //
    // Qs is not the largest, so see if Qx is the largest remaining diagonal
    // value.
    //
    else if((fQx > fQy) && (fQx > fQz))
    {
        //
        // Finish the computation of Qx.
        //
        fQx = sqrtf(fQx) / 2;

        //
        // Compute the values of the quaternion based on Qx.
        //
        pfQuaternion[0] = ((psDCM->ppfDCM[2][1] - psDCM->ppfDCM[1][2]) /
                           (4 * fQx));
        pfQuaternion[1] = fQx;
        pfQuaternion[2] = ((psDCM->ppfDCM[1][0] + psDCM->ppfDCM[0][1]) /
                           (4 * fQx));
        pfQuaternion[3] = ((psDCM->ppfDCM[0][2] + psDCM->ppfDCM[2][0]) /
                           (4 * fQx));
    }

    //
    // Qs and Qx are not the largest, so see if Qy is the largest remaining
    // diagonal value.
    //
    else if(fQy > fQz)
    {
        //
        // Finish the computation of Qy.
        //
        fQy = sqrtf(fQy) / 2;

        //
        // Compute the values of the quaternion based on Qy.
        //
        pfQuaternion[0] = ((psDCM->ppfDCM[0][2] - psDCM->ppfDCM[2][0]) /
                           (4 * fQy));
        pfQuaternion[1] = ((psDCM->ppfDCM[1][0] + psDCM->ppfDCM[0][1]) /
                           (4 * fQy));
        pfQuaternion[2] = fQy;
        pfQuaternion[3] = ((psDCM->ppfDCM[2][1] + psDCM->ppfDCM[1][2]) /
                           (4 * fQy));
    }

    //
    // Qz is the largest diagonal value.
    //
    else
    {
        //
        // Finish the computation of Qz.
        //
```

```
        fQz = sqrtf(fQz) / 2;

        //
        // Compute the values of the quaternion based on Qz.
        //
        pfQuaternion[0] = ((psDCM->ppfDCM[1][0] - psDCM->ppfDCM[0][1]) /
                           (4 * fQz));
        pfQuaternion[1] = ((psDCM->ppfDCM[0][2] + psDCM->ppfDCM[2][0]) /
                           (4 * fQz));
        pfQuaternion[2] = ((psDCM->ppfDCM[2][1] + psDCM->ppfDCM[1][2]) /
                           (4 * fQz));
        pfQuaternion[3] = fQz;
    }
}

//*****************************************************************************
//
// Close the Doxygen group.
//! @}
//
//*****************************************************************************
```

Appendix C

```
/*
 * Final Project
 * GPS Interfacing and Parsing Code
 *
 * Kevin Peters
 * Matt Weege
 * James Smith
 */


#include <stdbool.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include "inc/hw_memmap.h"
#include "driverlib/gpio.h"
#include "driverlib/pin_map.h"
#include "driverlib/sysctl.h"
#include <driverlib/uart.h>
#include "pin_map.h"
#include <string.h>

//*****************************************************************************
// Configure the UART and perform reads and writes using polled I/O.
//*****************************************************************************

void parsetokengps(char **arr, char *GPSdata); //function: parse gps into tokens
    strings

int main(void)
{
    // Set the clocking to run directly from the external crystal/oscillator.

    SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN |
                   SYSCTL_XTAL_16MHZ);

    // Enable the peripherals
    // The UART itself needs to be enabled, as well as the GPIO port
    // containing the pins that will be used.

    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART1);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    // Configure the GPIO pin muxing for the UART function.
    // This is only necessary if your part supports GPIO pin function muxing.
    // Study the data sheet to see which functions are allocated per pin.

    //fix this later
    GPIOPinConfigure(0x00010001);
    GPIOPinConfigure(0x00010401);
```

```
//uart 0 pin configs
GPIOPinConfigure(0x00000001);
GPIOPinConfigure(0x00000401);

//configure GPIO A0 and A1 for use as a peripheral function instead of GPIO

GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
GPIOPinTypeUART(GPIO_PORTB_BASE, GPIO_PIN_0 | GPIO_PIN_1);

//configure the board UART for 115,200 baud
//configure the gps UART for 9600 baud

UARTConfigSetExpClk(UART1_BASE, SysCtlClockGet(), 9600,
                  (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
                   UART_CONFIG_PAR_NONE));

UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 115200,
                    (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
                     UART_CONFIG_PAR_NONE));

int i = 0;  //primary counter
int j = 0;  //secondary counter
char gpsstring[100] = ""; //imported gps chars, initialized to empty
char *parsed[30]; //pointer array containing starting locations of parsed
    tokens
char lockestablished[20] = "GPS Lock Established";
char nolock[18] = "Unable To GPS Lock";
char gpsdata;   //temp holder for uart imported char, will be put in
    gpsstring[i]
char string[15] = "";       //word to print to uart

//gps variables
char timestamp[9];
char lat[9];
char latdir[2];
char lon[9];
char longdir[2];
char speed[4];
char dir[6];

/*
//accell variables
char xacc;
char xvel;
char xpos;
char yacc;
char yvel;
char ypos;
char zacc;
char zvel;
char zpos;
double xaccn;
double xveln;
double xposn;
double yaccn;
double yveln;
```

```
    double yposn;
    double zaccn;
    double zveln;
    double zposn;
    */

    //conversion of string to number for maths
    //sscanf(string,%d,stringn);

    // Get the party started forever
    while(1)
    {

        gpsdata = UARTCharGet(UART1_BASE);

        //get gps uart data

        i=0;

        if (gpsdata == '$')
        {
            do
            {
                gpsstring[i] = gpsdata;
                i++;
                gpsdata = UARTCharGet(UART1_BASE); //get next char

            } while(gpsdata != '\n');   //uart doesnt send null char at end of
                transmission, so we're using new line YOLO

            gpsstring[i] = 0;

        }

/*
        for(i = 0; i < strlen(gpsstring); i++)                  //print gathered
            big string from gps uart
        {
            UARTCharPut(UART0_BASE,gpsstring[i]);
        }

        UARTCharPut(UART0_BASE, '\r');
        UARTCharPut(UART0_BASE, '\n');
*/
        //parse gps uart string into tokens

        //use charmin/zero out parsed[i] so no pieces get left behind
        for (i = 0; i < 30; i++) {
           parsed[i] = 0;
        }

        parsetokengps(parsed, gpsstring);
/*
        //print each token

        for(i = 0; parsed[i] ; i++) //stops when parsed gets null
```

```
        {
            for(j = 0; j < strlen(parsed[i]); j++)
            {
                UARTCharPut(UART0_BASE,parsed[i][j]);       //uart print
                    tokenized gps data
            }

            UARTCharPut(UART0_BASE,'\r');
            UARTCharPut(UART0_BASE,'\n');
        }
*/
        if(strcmp(parsed[0],"$GPRMC") == 0)
        {
            if(strcmp(parsed[2],"V") == 0) //if second value == V, gps data
                invalid w/ no lock
            {
                //-> fix not available -> goto accelerometer<later>
                for(j = 0; j < 18; j++)
                {
                    UARTCharPut(UART0_BASE,nolock[j]);      //uart print no lock
                }

                UARTCharPut(UART0_BASE, '\r');
                UARTCharPut(UART0_BASE, '\n');

            }else if( strcmp(parsed[2],"A") == 0) //lock established, goto gps
                time and position data to get lat and long
                {
                for(j = 0; j < 20; j++)
                {
                    UARTCharPut(UART0_BASE,lockestablished[j]);             //
                        uart print lock established
                }

                UARTCharPut(UART0_BASE, '\r');
                UARTCharPut(UART0_BASE, '\n');

                strcpy(timestamp, parsed[1]);       //first value <hhmmss.ss> or
                    disregard .ss and just use [int]
                strcpy(lat,parsed[3]);                      //second value
                    <ddmm.mmmm> [definitely double], latitude value
                strcpy(latdir,parsed[4]);                       //N or S
                strcpy(lon,parsed[5]);                      //4th value
                    <ddmm.mmmm> [definitely double], longitude value
                strcpy(longdir,parsed[6]);                  //E or W
                strcpy(speed,parsed[7]);                        //knots <x.xx> <-
                    not a huge range (0-9) :/
                strcpy(dir,parsed[8]);                      //<ddd.dd>

                //now to print the tasty treats to the uart
                strcpy(string,"Time (UTC): ");
                for(i=0; string[i]; i++)
                {
                    UARTCharPut(UART0_BASE,string[i]);      //print le title
                }
```

```
        for(i=0; timestamp[i]; i++)
        {
            UARTCharPut(UART0_BASE,timestamp[i]);       //print le value
        }
        UARTCharPut(UART0_BASE, '\r');
        UARTCharPut(UART0_BASE, '\n');

        strcpy(string,"Latitude: ");
        for(i=0; string[i]; i++)
        {
            UARTCharPut(UART0_BASE,string[i]);      //print le title
        }
        for(i=0; lat[i]; i++)
        {
        UARTCharPut(UART0_BASE,lat[i]);     //print le value
        }

        UARTCharPut(UART0_BASE,' ');

        UARTCharPut(UART0_BASE,latdir[0]);      //print lat direction
        UARTCharPut(UART0_BASE, '\r');
        UARTCharPut(UART0_BASE, '\n');

        strcpy(string,"Longitude: ");
        for(i=0; string[i]; i++)
        {
            UARTCharPut(UART0_BASE,string[i]);      //print le title
        }
        for(i=0; lon[i]; i++)
        {
        UARTCharPut(UART0_BASE,lon[i]);     //print le value
        }

        UARTCharPut(UART0_BASE,' ');

        UARTCharPut(UART0_BASE,longdir[0]);     //print lat direction
        UARTCharPut(UART0_BASE, '\r');
        UARTCharPut(UART0_BASE, '\n');

        strcpy(string,"Est Speed: ");
        for(i=0; string[i]; i++)
        {
            UARTCharPut(UART0_BASE,string[i]);      //print le title
        }

        for(i=0; i<4; i++)
        {
            UARTCharPut(UART0_BASE,speed[i]);       //print le value
        }
        UARTCharPut(UART0_BASE, '\r');
        UARTCharPut(UART0_BASE, '\n');

        strcpy(string,"Est Angle: ");
        for(i=0; string[i]; i++)
        {
            UARTCharPut(UART0_BASE,string[i]);      //print le title
```

```
                }

                for(i=0; dir[i]; i++)
                {
                    UARTCharPut(UART0_BASE,dir[i]);     //print le value
                }
                UARTCharPut(UART0_BASE, '\r');
                UARTCharPut(UART0_BASE, '\n');

                }
        }

        UARTCharPut(UART0_BASE, '\r');
        UARTCharPut(UART0_BASE, '\n');


            //  later log gps data to sdcard
            //  get location data into legit format

        for (i = 0; i < 10; i++)
        {
            string[i] = 0;              //word string now cleared
        }

        for (i = 0; i < 30; i++)
        {
            free(parsed[i]);            //parsed[] now full of null so monsters
                dont come out and eat the code
        }
    }

    return(0);
}

void parsetokengps(char **arr, char *GPSdata)
{
   int i = 0;
   char temp[30];

   while (*GPSdata)
   {
      while (*GPSdata && *GPSdata != ',')
      {
         temp[i++] = *GPSdata++;
      }

      temp[i] = 0;
      if (*GPSdata == ',')
      {
         GPSdata++;
      }

      *arr = calloc(i + 1, sizeof(char)); //i still hate pointers.jpg

      if (i)
      {
```

```
            memcpy(*arr, temp, i); //dynamically size memory for string sizes
        }
        arr++;
        i = 0;
    }

    return;
}
```

Appendix D

```
//*****************************************************************************
//
// sd_card.c - Example program for reading files from an SD card.
//
// Copyright (c) 2011-2013 Texas Instruments Incorporated.  All rights reserved.
// Software License Agreement
//
// Texas Instruments (TI) is supplying this software for use solely and
// exclusively on TI's microcontroller products. The software is owned by
// TI and/or its suppliers, and is protected under applicable copyright
// laws. You may not combine this software with "viral" open-source
// software in order to form a larger program.
//
// THIS SOFTWARE IS PROVIDED "AS IS" AND WITH ALL FAULTS.
// NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT
// NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. TI SHALL NOT, UNDER ANY
// CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL, OR CONSEQUENTIAL
// DAMAGES, FOR ANY REASON WHATSOEVER.
//
// This is part of revision 2.0.1.11577 of the EK-LM4F232 Firmware Package.
//
//*****************************************************************************

#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include "inc/hw_memmap.h"
#include "driverlib/fpu.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/sysctl.h"
#include "driverlib/systick.h"
#include "driverlib/uart.h"
#include "grlib/grlib.h"
#include "utils/cmdline.h"
#include "utils/uartstdio.h"
#include "fatfs/src/ff.h"
#include "fatfs/src/diskio.h"
#include "drivers/cfal96x64x16.h"


//*****************************************************************************
//
//! \addtogroup example_list
//! <h1>SD card using FAT file system (sd_card)</h1>
//!
//! This example application demonstrates reading a file system from an SD
//! card.  It makes use of FatFs, a FAT file system driver.  It provides a
//! simple command console via a serial port for issuing commands to view and
```

```
//! navigate the file system on the SD card.
//!
//! The first UART, which is connected to the USB debug virtual serial port on
//! the evaluation board, is configured for 115,200 bits per second, and 8-N-1
//! mode.  When the program is started a message will be printed to the
//! terminal.  Type ``help'' for command help.
//!
//! For additional details about FatFs, see the following site:
//! http://elm-chan.org/fsw/ff/00index_e.html
//
//*****************************************************************************

//*****************************************************************************
//
// Defines the size of the buffers that hold the path, or temporary data from
// the SD card.  There are two buffers allocated of this size.  The buffer size
// must be large enough to hold the longest expected full path name, including
// the file name, and a trailing null character.
//
//*****************************************************************************
#define PATH_BUF_SIZE           80

//*****************************************************************************
//
// Defines the size of the buffer that holds the command line.
//
//*****************************************************************************
#define CMD_BUF_SIZE            64

//*****************************************************************************
//
// This buffer holds the full path to the current working directory.  Initially
// it is root ("/").
//
//*****************************************************************************
static char g_pcCwdBuf[PATH_BUF_SIZE] = "/";

//*****************************************************************************
//
// A temporary data buffer used when manipulating file paths, or reading data
// from the SD card.
//
//*****************************************************************************
static char g_pcTmpBuf[PATH_BUF_SIZE];

//*****************************************************************************
//
// The buffer that holds the command line.
//
//*****************************************************************************
static char g_pcCmdBuf[CMD_BUF_SIZE];

//*****************************************************************************
//
// The following are data structures used by FatFs.
//
```

```
//*****************************************************************************
static FATFS g_sFatFs;
static DIR g_sDirObject;
static FILINFO g_sFileInfo;
static FIL g_sFileObject;

//*****************************************************************************
//
// A structure that holds a mapping between an FRESULT numerical code, and a
// string representation.  FRESULT codes are returned from the FatFs FAT file
// system driver.
//
//*****************************************************************************
typedef struct
{
    FRESULT iFResult;
    char *pcResultStr;
}
tFResultString;

//*****************************************************************************
//
// A macro to make it easy to add result codes to the table.
//
//*****************************************************************************
#define FRESULT_ENTRY(f)        { (f), (#f) }

//*****************************************************************************
//
// A table that holds a mapping between the numerical FRESULT code and it's
// name as a string.  This is used for looking up error codes for printing to
// the console.
//
//*****************************************************************************
tFResultString g_psFResultStrings[] =
{
    FRESULT_ENTRY(FR_OK),
    FRESULT_ENTRY(FR_DISK_ERR),
    FRESULT_ENTRY(FR_INT_ERR),
    FRESULT_ENTRY(FR_NOT_READY),
    FRESULT_ENTRY(FR_NO_FILE),
    FRESULT_ENTRY(FR_NO_PATH),
    FRESULT_ENTRY(FR_INVALID_NAME),
    FRESULT_ENTRY(FR_DENIED),
    FRESULT_ENTRY(FR_EXIST),
    FRESULT_ENTRY(FR_INVALID_OBJECT),
    FRESULT_ENTRY(FR_WRITE_PROTECTED),
    FRESULT_ENTRY(FR_INVALID_DRIVE),
    FRESULT_ENTRY(FR_NOT_ENABLED),
    FRESULT_ENTRY(FR_NO_FILESYSTEM),
    FRESULT_ENTRY(FR_MKFS_ABORTED),
    FRESULT_ENTRY(FR_TIMEOUT),
    FRESULT_ENTRY(FR_LOCKED),
    FRESULT_ENTRY(FR_NOT_ENOUGH_CORE),
    FRESULT_ENTRY(FR_TOO_MANY_OPEN_FILES),
    FRESULT_ENTRY(FR_INVALID_PARAMETER),
```

```
};

//*****************************************************************************
//
// A macro that holds the number of result codes.
//
//*****************************************************************************
#define NUM_FRESULT_CODES       (sizeof(g_psFResultStrings) /                 \
                                 sizeof(tFResultString))

//*****************************************************************************
//
// Graphics context used to show text on the CSTN display.
//
//*****************************************************************************
tContext g_sContext;


//*****************************************************************************
//
// This function returns a string representation of an error code that was
// returned from a function call to FatFs.  It can be used for printing human
// readable error messages.
//
//*****************************************************************************
const char *
StringFromFResult(FRESULT iFResult)
{
    uint_fast8_t ui8Idx;

    //
    // Enter a loop to search the error code table for a matching error code.
    //
    for(ui8Idx = 0; ui8Idx < NUM_FRESULT_CODES; ui8Idx++)
    {
        //
        // If a match is found, then return the string name of the error code.
        //
        if(g_psFResultStrings[ui8Idx].iFResult == iFResult)
        {
            return(g_psFResultStrings[ui8Idx].pcResultStr);
        }
    }

    //
    // At this point no matching code was found, so return a string indicating
    // an unknown error.
    //
    return("UNKNOWN ERROR CODE");
}

//*****************************************************************************
//
// This is the handler for this SysTick interrupt.  FatFs requires a timer tick
// every 10 ms for internal timing purposes.
//
```

```
//*****************************************************************************
void
SysTickHandler(void)
{
    //
    // Call the FatFs tick timer.
    //
    disk_timerproc();
}

//*****************************************************************************
//
// This function implements the "ls" command.  It opens the current directory
// and enumerates through the contents, and prints a line for each item it
// finds.  It shows details such as file attributes, time and date, and the
// file size, along with the name.  It shows a summary of file sizes at the end
// along with free space.
//
//*****************************************************************************
int
Cmd_ls(int argc, char *argv[])
{
    uint32_t ui32TotalSize;
    uint32_t ui32FileCount;
    uint32_t ui32DirCount;
    FRESULT iFResult;
    FATFS *psFatFs;
    char *pcFileName;
#if _USE_LFN
    char pucLfn[_MAX_LFN + 1];
    g_sFileInfo.lfname = pucLfn;
    g_sFileInfo.lfsize = sizeof(pucLfn);
#endif


    //
    // Open the current directory for access.
    //
    iFResult = f_opendir(&g_sDirObject, g_pcCwdBuf);

    //
    // Check for error and return if there is a problem.
    //
    if(iFResult != FR_OK)
    {
        return((int)iFResult);
    }

    ui32TotalSize = 0;
    ui32FileCount = 0;
    ui32DirCount = 0;

    //
    // Give an extra blank line before the listing.
    //
    UARTprintf("\n");
```

```
    //
    // Enter loop to enumerate through all directory entries.
    //
    for(;;)
    {
        //
        // Read an entry from the directory.
        //
        iFResult = f_readdir(&g_sDirObject, &g_sFileInfo);

        //
        // Check for error and return if there is a problem.
        //
        if(iFResult != FR_OK)
        {
            return((int)iFResult);
        }

        //
        // If the file name is blank, then this is the end of the listing.
        //
        if(!g_sFileInfo.fname[0])
        {
            break;
        }

        //
        // If the attribue is directory, then increment the directory count.
        //
        if(g_sFileInfo.fattrib & AM_DIR)
        {
            ui32DirCount++;
        }

        //
        // Otherwise, it is a file.  Increment the file count, and add in the
        // file size to the total.
        //
        else
        {
            ui32FileCount++;
            ui32TotalSize += g_sFileInfo.fsize;
        }

#if _USE_LFN
        pcFileName = ((*g_sFileInfo.lfname)?g_sFileInfo.lfname:g_sFileInfo.fname)
            ;
#else
        pcFileName = g_sFileInfo.fname;
#endif
        //
        // Print the entry information on a single line with formatting to show
        // the attributes, date, time, size, and name.
        //
        UARTprintf("%c%c%c%c%c %u/%02u/%02u %02u:%02u %9u  %s\n",
```

```
                    (g_sFileInfo.fattrib & AM_DIR) ? 'D' : '-',
                    (g_sFileInfo.fattrib & AM_RDO) ? 'R' : '-',
                    (g_sFileInfo.fattrib & AM_HID) ? 'H' : '-',
                    (g_sFileInfo.fattrib & AM_SYS) ? 'S' : '-',
                    (g_sFileInfo.fattrib & AM_ARC) ? 'A' : '-',
                    (g_sFileInfo.fdate >> 9) + 1980,
                    (g_sFileInfo.fdate >> 5) & 15,
                    g_sFileInfo.fdate & 31,
                    (g_sFileInfo.ftime >> 11),
                    (g_sFileInfo.ftime >> 5) & 63,
                    g_sFileInfo.fsize,
                    pcFileName);
    }

    //
    // Print summary lines showing the file, dir, and size totals.
    //
    UARTprintf("\n%4u File(s),%10u bytes total\n%4u Dir(s)",
                ui32FileCount, ui32TotalSize, ui32DirCount);

    //
    // Get the free space.
    //
    iFResult = f_getfree("/", (DWORD *)&ui32TotalSize, &psFatFs);

    //
    // Check for error and return if there is a problem.
    //
    if(iFResult != FR_OK)
    {
        return((int)iFResult);
    }

    //
    // Display the amount of free space that was calculated.
    //
    UARTprintf(", %10uK bytes free\n", (ui32TotalSize *
                                        psFatFs->free_clust / 2));

    //
    // Made it to here, return with no errors.
    //
    return(0);
}

//*****************************************************************************
//
// This function implements the "cd" command.  It takes an argument that
// specifies the directory to make the current working directory.  Path
// separators must use a forward slash "/".  The argument to cd can be one of
// the following:
//
// * root ("/")
// * a fully specified path ("/my/path/to/mydir")
// * a single directory name that is in the current directory ("mydir")
// * parent directory ("..")
```

```
//
// It does not understand relative paths, so dont try something like this:
// ("../my/new/path")
//
// Once the new directory is specified, it attempts to open the directory to
// make sure it exists.  If the new path is opened successfully, then the
// current working directory (cwd) is changed to the new path.
//
//*****************************************************************************
int
Cmd_cd(int argc, char *argv[])
{
    uint_fast8_t ui8Idx;
    FRESULT iFResult;

    //
    // Copy the current working path into a temporary buffer so it can be
    // manipulated.
    //
    strcpy(g_pcTmpBuf, g_pcCwdBuf);

    //
    // If the first character is /, then this is a fully specified path, and it
    // should just be used as-is.
    //
    if(argv[1][0] == '/')
    {
        //
        // Make sure the new path is not bigger than the cwd buffer.
        //
        if(strlen(argv[1]) + 1 > sizeof(g_pcCwdBuf))
        {
            UARTprintf("Resulting path name is too long\n");
            return(0);
        }

        //
        // If the new path name (in argv[1])  is not too long, then copy it
        // into the temporary buffer so it can be checked.
        //
        else
        {
            strncpy(g_pcTmpBuf, argv[1], sizeof(g_pcTmpBuf));
        }
    }

    //
    // If the argument is .. then attempt to remove the lowest level on the
    // CWD.
    //
    else if(!strcmp(argv[1], ".."))
    {
        //
        // Get the index to the last character in the current path.
        //
        ui8Idx = strlen(g_pcTmpBuf) - 1;
```

```
        //
        // Back up from the end of the path name until a separator (/) is
        // found, or until we bump up to the start of the path.
        //
        while((g_pcTmpBuf[ui8Idx] != '/') && (ui8Idx > 1))
        {
            //
            // Back up one character.
            //
            ui8Idx--;
        }

        //
        // Now we are either at the lowest level separator in the current path,
        // or at the beginning of the string (root).  So set the new end of
        // string here, effectively removing that last part of the path.
        //
        g_pcTmpBuf[ui8Idx] = 0;
    }

    //
    // Otherwise this is just a normal path name from the current directory,
    // and it needs to be appended to the current path.
    //
    else
    {
        //
        // Test to make sure that when the new additional path is added on to
        // the current path, there is room in the buffer for the full new path.
        // It needs to include a new separator, and a trailing null character.
        //
        if(strlen(g_pcTmpBuf) + strlen(argv[1]) + 1 + 1 > sizeof(g_pcCwdBuf))
        {
            UARTprintf("Resulting path name is too long\n");
            return(0);
        }

        //
        // The new path is okay, so add the separator and then append the new
        // directory to the path.
        //
        else
        {
            //
            // If not already at the root level, then append a /
            //
            if(strcmp(g_pcTmpBuf, "/"))
            {
                strcat(g_pcTmpBuf, "/");
            }

            //
            // Append the new directory to the path.
            //
            strcat(g_pcTmpBuf, argv[1]);
```

```
        }
    }

    //
    // At this point, a candidate new directory path is in chTmpBuf.  Try to
    // open it to make sure it is valid.
    //
    iFResult = f_opendir(&g_sDirObject, g_pcTmpBuf);

    //
    // If it can't be opened, then it is a bad path.  Inform the user and
    // return.
    //
    if(iFResult != FR_OK)
    {
        UARTprintf("cd: %s\n", g_pcTmpBuf);
        return((int)iFResult);
    }

    //
    // Otherwise, it is a valid new path, so copy it into the CWD.
    //
    else
    {
        strncpy(g_pcCwdBuf, g_pcTmpBuf, sizeof(g_pcCwdBuf));
    }

    //
    // Return success.
    //
    return(0);
}

//*****************************************************************************
//
// This function implements the "pwd" command.  It simply prints the current
// working directory.
//
//*****************************************************************************
int
Cmd_pwd(int argc, char *argv[])
{
    //
    // Print the CWD to the console.
    //
    UARTprintf("%s\n", g_pcCwdBuf);

    //
    // Return success.
    //
    return(0);
}

//*****************************************************************************
//
// This function implements the "cat" command.  It reads the contents of a file
```

```
// and prints it to the console.  This should only be used on text files.  If
// it is used on a binary file, then a bunch of garbage is likely to printed on
// the console.
//
//*****************************************************************************
int
Cmd_cat(int argc, char *argv[])
{
    FRESULT iFResult;
    uint32_t ui32BytesRead;

    //
    // First, check to make sure that the current path (CWD), plus the file
    // name, plus a separator and trailing null, will all fit in the temporary
    // buffer that will be used to hold the file name.  The file name must be
    // fully specified, with path, to FatFs.
    //
    if(strlen(g_pcCwdBuf) + strlen(argv[1]) + 1 + 1 > sizeof(g_pcTmpBuf))
    {
        UARTprintf("Resulting path name is too long\n");
        return(0);
    }

    //
    // Copy the current path to the temporary buffer so it can be manipulated.
    //
    strcpy(g_pcTmpBuf, g_pcCwdBuf);

    //
    // If not already at the root level, then append a separator.
    //
    if(strcmp("/", g_pcCwdBuf))
    {
        strcat(g_pcTmpBuf, "/");
    }

    //
    // Now finally, append the file name to result in a fully specified file.
    //
    strcat(g_pcTmpBuf, argv[1]);

    //
    // Open the file for reading.
    //
    iFResult = f_open(&g_sFileObject, g_pcTmpBuf, FA_READ);

    //
    // If there was some problem opening the file, then return an error.
    //
    if(iFResult != FR_OK)
    {
        return((int)iFResult);
    }

    //
    // Enter a loop to repeatedly read data from the file and display it, until
```

```
    // the end of the file is reached.
    //
    do
    {
        //
        // Read a block of data from the file.  Read as much as can fit in the
        // temporary buffer, including a space for the trailing null.
        //
        iFResult = f_read(&g_sFileObject, g_pcTmpBuf, sizeof(g_pcTmpBuf) - 1,
                          &ui32BytesRead);

        //
        // If there was an error reading, then print a newline and return the
        // error to the user.
        //
        if(iFResult != FR_OK)
        {
            UARTprintf("\n");
            return((int)iFResult);
        }

        //
        // Null terminate the last block that was read to make it a null
        // terminated string that can be used with printf.
        //
        g_pcTmpBuf[ui32BytesRead] = 0;

        //
        // Print the last chunk of the file that was received.
        //
        UARTprintf("%s", g_pcTmpBuf);
    }
    while(ui32BytesRead == sizeof(g_pcTmpBuf) - 1);

    //
    // Return success.
    //
    return(0);
}

//*****************************************************************************
//
// This function implements the "help" command.  It prints a simple list of the
// available commands with a brief description.
//
//*****************************************************************************
int
Cmd_help(int argc, char *argv[])
{
    tCmdLineEntry *psEntry;

    //
    // Print some header text.
    //
    UARTprintf("\nAvailable commands\n");
    UARTprintf("------------------\n");
```

```
    //
    // Point at the beginning of the command table.
    //
    psEntry = &g_psCmdTable[0];

    //
    // Enter a loop to read each entry from the command table.  The end of the
    // table has been reached when the command name is NULL.
    //
    while(psEntry->pcCmd)
    {
        //
        // Print the command name and the brief description.
        //
        UARTprintf("%6s: %s\n", psEntry->pcCmd, psEntry->pcHelp);

        //
        // Advance to the next entry in the table.
        //
        psEntry++;
    }

    //
    // Return success.
    //
    return(0);
}
//
    ****************************************************************************
    ****************************************************************************
    ***************
//
    ****************************************************************************
    *******************JAMES WRITE
    FUNCTION*****************************************************
//
    ****************************************************************************
    ****************************************************************************
    ***************

//
// An example of data being written to the card -James
// As well as the length of the data being written
// Ideally updated with GPS coordinates later on
//

int
Cmd_wr(int argc, char *argv[])  // Write command
{
    FRESULT iFResult;
    uint32_t ui32BytesWrite;


    //
    // First, check to make sure that the current path (CWD), plus the file
```

```
    // name, plus a separator and trailing null, will all fit in the temporary
    // buffer that will be used to hold the file name.  The file name must be
    // fully specified, with path, to FatFs.
    //
    if(strlen(g_pcCwdBuf) + strlen(argv[1]) + 1 + 1 > sizeof(g_pcTmpBuf))
    {
        UARTprintf("Resulting path name is too long\n");
        return(0);
    }

    //
    // Copy the current path to the temporary buffer so it can be manipulated.
    //
    strcpy(g_pcTmpBuf, g_pcCwdBuf);

    //
    // If not already at the root level, then append a separator.
    //
    if(strcmp("/", g_pcCwdBuf))
    {
        strcat(g_pcTmpBuf, "/");
    }

    //
    // Now finally, append the file name to result in a fully specified file.
    //
    strcat(g_pcTmpBuf, argv[1]);

    //
    // Open the file for writing.
    //
    iFResult = f_open(&g_sFileObject, g_pcTmpBuf, FA_WRITE|FA_OPEN_ALWAYS|FA_READ
        );

    //
    // Move to end of existing File
    // Testing ability to move R/W pointer to last written
    //
    //iFResult = f_lseek(&g_sFileObject, f_size(&g_sFileObject));

    //
    // If there was some problem opening the file, then return an error.
    //
    if(iFResult != FR_OK)
    {
        return((int)iFResult);
    }


    //
    // Print a prompt to the console.  Show the CWD.
    //
    UARTprintf("\n%s> ", g_pcCwdBuf);

    //
    // Get a line of text from the user.
```

```
    //
    UARTgets(g_pcTmpBuf, sizeof(g_pcTmpBuf));


    //
    // Enter a loop to repeatedly write data from the g_pcTmpBuf and display it,
        until
    // the end of the input is reached.
    //
    do
    {
        //
        // Read a block of data from the file.  Read as much as can fit in the
        // temporary buffer, including a space for the trailing null.
        //
        iFResult = f_write(&g_sFileObject, g_pcTmpBuf, sizeof(g_pcTmpBuf) - 1,
                         &ui32BytesWrite);

        //
        // If there was an error reading, then print a newline and return the
        // error to the user.
        //
        if(iFResult != FR_OK)
        {
            UARTprintf("\n");
            return((int)iFResult);
        }

        //
        // Null terminate the last block that was read to make it a null
        // terminated string that can be used with printf.
        //
       g_pcTmpBuf[ui32BytesWrite] = 0;

        //
        // Print the last chunk of the file that was received.
        //
       // UARTprintf("%s", g_pcTmpBuf);

    }
    while(ui32BytesWrite == sizeof(g_pcTmpBuf)-1);

    //
    // Return success.
    //
    return(0);
}




//*****************************************************************************
//
// This is the table that holds the command names, implementing functions, and
// brief description.
//
```

```
//*****************************************************************************
tCmdLineEntry g_psCmdTable[] =
{
    { "help",   Cmd_help,   "Display list of commands" },
    { "h",      Cmd_help,   "alias for help" },
    { "?",      Cmd_help,   "alias for help" },
    { "ls",     Cmd_ls,     "Display list of files" },
    { "chdir",  Cmd_cd,     "Change directory" },
    { "cd",     Cmd_cd,     "alias for chdir" },
    { "pwd",    Cmd_pwd,    "Show current working directory" },
    { "cat",    Cmd_cat,    "Show contents of a text file" },
    { "wr",     Cmd_wr,     "Write content to a text file" }, // Addition write
        command
    { 0, 0, 0 }
};

//*****************************************************************************
//
// The error routine that is called if the driver library encounters an error.
//
//*****************************************************************************
#ifdef DEBUG
void
__error__(char *pcFilename, uint32_t ui32Line)
{
}
#endif

//*****************************************************************************
//
// Configure the UART and its pins.  This must be called before UARTprintf().
//
//*****************************************************************************
void
ConfigureUART(void)
{
    //
    // Enable the GPIO Peripheral used by the UART.
    //
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    //
    // Enable UART0
    //
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);

    //
    // Configure GPIO Pins for UART mode.
    //
    ROM_GPIOPinConfigure(GPIO_PA0_U0RX);
    ROM_GPIOPinConfigure(GPIO_PA1_U0TX);
    ROM_GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    //
    // Use the internal 16MHz oscillator as the UART clock source.
    //
```

```
    UARTClockSourceSet(UART0_BASE, UART_CLOCK_PIOSC);

    //
    // Initialize the UART for console I/O.
    //
    UARTStdioConfig(0, 115200, 16000000);
    //
    UARTStdioConfig(0, 115200, 16000000);
}

//*****************************************************************************
//
// The program main function.  It performs initialization, then runs a command
// processing loop to read commands from the console.
//
//*****************************************************************************


//
// Streamlined main function ************************* THIS IS NOT THE ORIGINAL
    MAIN FUNCTION****************************************
//                           ***************************** SCROLL DOWN TO SEE
    THE ORIGINAL*******************************************

int
main(void)
{
    FRESULT iFResult;
    char Letters[10] = {'a','b','c','d','e','f','g','h','i','j'};
    char Value[1];
    int i = 0;
    uint32_t ui32BytesWrite;

    // Enable lazy stacking for interrupt handlers.  This allows floating-point
    // instructions to be used within interrupt handlers, but at the expense of
    // extra stack usage.
    //
    ROM_FPULazyStackingEnable();

    //
    // Set the system clock to run at 50MHz from the PLL.
    //
    ROM_SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
                       SYSCTL_XTAL_16MHZ);

    //
    // Enable the peripherals used by this example.
    //
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);

    //
    // Configure SysTick for a 100Hz interrupt.  The FatFs driver wants a 10 ms
    // tick.
    //
    ROM_SysTickPeriodSet(ROM_SysCtlClockGet()/100);
    ROM_SysTickEnable();
```

```
    ROM_SysTickIntEnable();

    //
    // Enable Interrupts
    //
    ROM_IntMasterEnable();

    //
    // Mount the file system, using logical disk 0.
    //
    iFResult = f_mount(0, &g_sFatFs);
        if(iFResult != FR_OK)
        {
            UARTprintf("f_mount error: %s\n", StringFromFResult(iFResult));
            return(1);
        }

    //
    // Open SD_Test.txt and begin writing
    //
    iFResult = f_open(&g_sFileObject, "SD_Test.txt", FA_WRITE|FA_OPEN_ALWAYS|
        FA_READ);


    //
    // Count up to 9 and write to file
    //
    for(i=0; i<10; i++)
    {
        //
        // Put the value of 'i' into the string 'Value'
        //
        Value[0] = (char)i;

        //
        // set the file pointer to the end of the file, in order to append data
        //
        iFResult = f_lseek(&g_sFileObject, f_size(&g_sFileObject));

        //
        //  Write to the new file pointer location, attempt to append data
        //
        iFResult = f_write(&g_sFileObject, Value, sizeof(Value) - 1, &
            ui32BytesWrite);
    }
    iFResult = f_close(&g_sFileObject);
}
```

This report is now officially 100 pages long. You’re welcome.

Love,
Kevin, Matt, and James.