A Wearable, Wireless Sensor System for Sports Medicine

by

Michael Tomasz Lapinski

M.A.Sc., Rensselaer Polytechnic Institute (2003)

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements for the degree of

Master of Science in Media Arts and Sciences

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2008

Author ______________________________
Program in Media Arts and Sciences
August 8, 2008

Certified by ______________________________
Joseph A. Paradiso
Sony Career Development Associate Professor of Media Arts and Sciences
MIT Media Lab
Thesis Supervisor

Accepted by ______________________________
Deb Roy
Chair, Department Committee on Graduate Students
Program in Media Arts and Sciences

A Wearable, Wireless Sensor System for Sports Medicine

by

Michael Tomasz Lapinski

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
on August 8, 2008, in partial fulfillment of the
requirements for the degree of
Master of Science in Media Arts and Sciences

Abstract

This thesis describes a compact, wireless, wearable system that measures, for purposes of biomechanical analysis, signals indicative of forces, torques and other descriptive and evaluative features that the human body undergoes during bursts of extreme physical activity (such as during athletic performance). Standard approaches leverage high speed camera systems, which need significant infrastructure and provide limited update rates and dynamic accuracy, to make these measurements. This project uses 6 degree-of-freedom inertial measurement units worn on various segments of an athlete's body to directly make these dynamic measurements. A combination of low and high G sensors enables sensitivity for slow and fast motion, and the addition of a compass helps in tracking joint angles. Data from the battery-powered nodes is acquired using a custom wireless protocol over an RF link. This data, along with rigorous calibration data, is processed on a PC, with an end product being precise angular velocities and accelerations that can be employed during biomechanical analysis to gain a better understanding of what occurs during activity.

The focus of experimentation was baseball pitching and batting at the professional level. Several pitchers and batters were instrumented with the system and data was gathered during several pitches or swings. The data was analyzed, and the results of this analysis are presented in this thesis. The dynamic results are more precise than from other camera based systems and also offer the measurement of metrics that are not available from any other system, providing the opportunity for furthering sports medicine research. System performance and results are evaluated, and ideas for future work and system improvements are presented.

Thesis Supervisor: Joseph A. Paradiso
Title: Sony Career Development Associate Professor of Media Arts and Sciences, MIT Media Lab

A Wearable, Wireless Sensor System for Sports Medicine

by

Michael Tomasz Lapinski

The following people served as readers for this thesis:

Thesis Reader ______________________________

Dr. Paolo Bonato
Assistant Professor of Physical Medicine and Rehabilitation
Harvard Medical School
Director, Motion Analysis Laboratory
Spaulding Rehabilitation Hospital

Thesis Reader ______________________________

Dr. Eric Berkson
Clinical Associate in Orthopedic Surgery
Massachusetts General Hospital

Acknowledgments

In no particular order:

Asia, Dad, Mom, Dan Hyatt, Eric B, Mike R, Joe P, Zach, Jim Rowe and the Boston Red Rox, Leslie from Texas Instruments and **Slayer**

Also the Boston Red Sox and the Things That Think Consortium at the MIT Media Lab which supported this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

SportSemble is a wearable, wireless inertial sensor network designed and built for the purpose of measuring multipoint acceleration and angular rate (and inferring relevant clinical evaluative/descriptive and biomechanical parameters) associated with high speed, high performance athletic activity. Its core is based around high range inertial measurement units and a wireless data gathering infrastructure. Each identical node provides several streams of data at a high sampling rate and the entire network is flexible enough to handle any type of activity. Data sampling occurs at *1000Hz*, and sportSemble's data probably uniquely at these rates provides a nearly direct measurement of the velocities, forces and torques the human body experiences during bouts of physical activity. The system is a tool that can be used by sports medicine doctors to further their understanding of motion, aid in avoiding injury and help in rehabilitation from injury. Other applications for this technology, in addition to sports medicine, are in health monitoring, immersive gaming, and interactive dance ensemble performance.

1.1 Synopsis

This thesis provides detailed documentation of sportSemble's design, implementation, experimentation and data processing engine. The rest of this *Introduction* will describe the motivation behind building sportSemble and cover other systems and work in this area of research. The *Hardware Systems* chapter will explain the requirements that shaped what hardware was used and detail the hardware design. The following chapter, *Software Systems*, gives and in-depth description of the firmware for the basestation and wireless nodes, the PC side software and the RF protocol developed for sportSemble. Next, the *Experiments* chapter will go through initial experimentation and the procedures used in a larger 3-day experiment that was performed with professional athletes. Chapter 5 is *Calibration*, and covers the process of calibrating the sportSemble nodes to ensure that data was reliable. After this the *Results*, chapter will describe the data processing engine and present the biomechanical parameters that resulted from experimentation. The final chapter, *Conclusions*, will asses the system as a whole and suggest what work can be done in the future.

1.2 Motivation

To understand the motivations for this project, let us take a look at some statistics from just one sport, baseball. In 1973, 50% of pitchers reported shoulder or elbow pain sufficient enough to keep them from throwing[3]. By 1999 that number had grown to over 75% [4]. Andrews[5] reports that from the periods of 1995-1999 and 2000-2004, the need for elbow surgery in professionals increased twofold (for college pitchers fourfold and for high-school pitchers sixfold).

Due to the extreme speeds at which sports like pitching occur, there do not exist any off the shelf products that can accurately measure the dynamics (and directly infer the forces and torques) at which the body is moving during peak activity. It is because of this that sports medicine doctors do not have any true understanding of what happens to the body during extreme physical activity, other than data available from optical systems in a laboratory setting. A better understanding of the forces, torques and speeds enables the creation of precise biomechanical model of, for example, the shoulder and elbow. Using these models to understand the difference between proper and improper motion would give athletes the opportunity to avoid getting injured and coaches the ability to train athletes with techniques that avoid injury. Further, if there is a patient history of such motion in existence, it would be possible for physical and occupational therapists to better help athletes by refining rehab protocols that accurately define the forces and torques on a body segment over time.

Even further, a system such as this can recognize that an athlete is beginning to fatigue and is in danger of injuring themselves. If this can be recognized, then the athlete can be advised as to what motion is 'safer' or be advised to stop the activity, thus avoiding injury. Another step further, if athletes can be characterized and compared in a scientific manner, it is then possible to recognize traits which lead to better performance, hence such data could act as an aid in selecting up and coming athletes.

The author sees three different facets to the motivation for creating a system such as this. Being someone who has had several sports injuries, the first is a personal motivation. This is an opportunity to help normal people who participate in sports avoid getting injured and if they do get injured to help them recover, thus aid in improving their quality of life. The second facet of motivation is the one in the area of advancing clinical medicine and the ability to further medical research and understanding of the body. SportSemble will be used as an enabling tool to help sports doctors and therapists do what they do better. The third is a strategic motivation: if a system like sportSemble can do what is described above, professional sports teams could decide earlier which athletes have the potential to be the next star, and concentrate financial resources on them.

A system such as sportSemble should not be limited to only baseball pitching and batting, therefore one of its key design principles has been to make it as generic as possible, such that it can be placed on any athlete at any venue anywhere and easily used.

1.3 Prior Work

1.3.1 Commercial Camera Systems

There are two well-known commercially available camera systems that have been used to attempt to do what sportSemble does. While there are several different optical tracking

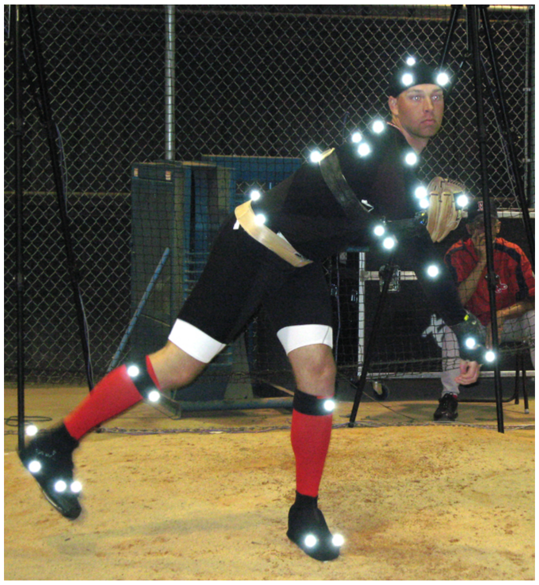


Figure 1-1: Marker Instrumented Player (markers glow due to camera flash)

systems on the market, the author has access to two of these, which appear to be the sports industry standards. These systems are available from XOS Technologies[6] and Motion Analysis[7]. Both of these are passive systems that are based on marker tracking technology and high speed video cameras. A single participant is instrumented with several markers (Figure 1-1) and a set of high speed video cameras records their motion.

Each of these systems has drawbacks with regard to sports medicine applications. From the authors viewpoint (and as observed by his medical collaborators), a primary drawback is that any rotational and acceleration force values that they produce are very rough estimations output from numerical differentiation (typically ±75% accurate [8]). These values are very coarse because they are only based on the instantaneous position of the tracked markers and probably does not properly leverage multi-point estimators like Kalman filters[9] or interpolating differentiators. SportSemble seeks to measure these parameters directly (forces and torques are proportional to acceleration and angular rate) with much smaller error bounds. The second critical shortcoming of these systems is the rate at which they update. Considering the peak acceleration of a baseball pitch lasts about 0.003 seconds and the cameras typically used record at 180Hz maximum (a sampling interval of 0.0056 seconds), it is very likely that the camera will miss or only record a portion of the peak acceleration (higher speed cameras exist [10] [1], but their cost and long setup time in a non-lab environment make them prohibitive). While camera systems are acceptable for generating 3D visualizations of an athlete's motion, they do not offer a reliable way of recording all the kinds of data needed for sports medicine purposes and for biomechanical analysis.

Further, these systems are cumbersome to transport due to the sheer number of cameras necessary. For the XOS system, for example, each marker must be tracked by at least 4 cameras at all times (to maintain accuracy). So a configuration for 1 baseball pitcher required a system of 12 cameras[2]. It also required 2 working days to configure the system at the site where it was deployed, due to the very sensitive nature of these optical systems. SportSemble seeks to sidestep all of these problems by using inertial measurement units to directly measure angular rates and accelerations. This gives sportSemble a typical setup time of less than 5 minutes for any athlete for any sport, since the unit needs only be strapped to the body and their positions measured, as mentioned in Section 4.2. The sportSemble system can easily be used on the mound, in the playing field, or on the court without external infrastructure or limitations on lighting, etc. Further, sportSemble can be left on an individual player and allows for measurement over long periods of time.

Recent work by Raskar and collaborators [11] have developed a very fast multipoint optical motion capture system that instruments subjects with small photodiodes attached to radio transmitters and illuminates them with dynamic structured light coming from an array of masked LEDs. Although this is an extremely promising system for fast, easily set up optical motion capture, it is still uncommercialized.

Even with these drawbacks, the author believes that in its current form at this point in time, that both of these systems are complementary. SportSemble cannot yet generate fluid 3D models of an athletes motion that players and coaches like to look at, although ongoing work integrating the inertial measurements and using the onboard compass data aims to achieve this capability.

[1]Up to 1000Hz or more, however these can be used only in a very specific lab setting.
[2]A 12 camera system also comes with a pricetag of $350,000

1.3.2 Inertial Measurement Unit Based Systems

There have been several inertial measurement unit (IMU) based systems developed for various purposes, but none with the intention of being applied to sports medicine that predate sportSemble's direct predecessor[12, 8] which was used in a small pilot study. There have been many related projects, however with applications in wearable motion analysis via IMUs . Possibly the closest relative is the MIT Media Lab's Responsive Environments Group's gait shoe [13, 14] project. This project used an on-shoe 3-axis gyroscope and 3-axis accelerometer to analyze the gait of a walking human being. Another prior project in the group is an inertial measurement unit for user interfaces[15, 16],which was a wireless six degree-of-freedom IMU capable of recognizing gestures, which were used to control a user interface. Other work of interest is in the area of practical motion capture in everyday surroundings by Vlasic, Popovic et al.[17, 18]. Their system also uses gyroscopes and accelerometers in addition to ultrasonic time of flight sensing in order to drive a graphical model of the human body. The time of flight data is used to record the distance between sensors worn on the body in an attempt to correct for drift in the inertial sensors. Published work on the system is very positive, however its sampling rate of *140Hz* would is too slow for high speed activity. Further, the system depends on a user-worn backpack that holds various support hardware needed to run it. More recent relevant work is [19], a wearable system for providing real time feedback during exercise. The system uses ultrasound and accelerometers to gauge arm position. The granularity of arm position is coarse - on the level of up, middle and down. Further, the arm must be held in a given position for a minimum of 20ms in order for position to be recorded.

In [20], a three axis accelerometer is augmented with a linear encoder and experimentation is done in order to determine joint angles, which are estimated by extracting the gravity vector from the accelerometer to determine tilt and fused with data from the encoder. This is shown to be 98% accurate in a laboratory setting. Also, in [21], raw output from a 2 axis accelerometer being sampled at 25Hz is processed via an *Extended Kalman Filter* [9] in order to calculate thigh flexion. Results are compared to video footage and demonstrate that the Kalman Filter is more precise than using a *median filter* of various orders.

Orient-2 [22] has been shown to estimate body position using nodes equipped with 3 axes of magnetometer, accelerometer and gyroscope. The system processes the sensor data in situ, using a fast complementary quaternion based filter[23], on each node and transmits the orientation data wirelessly. Another system leveraging only accelerometers and magnetometers (again in 3 axes) in order to do similar tracking of posture and orientation is presented in [24]. The results are positive, however the system requires a tether and wiring to be run across the body, again making it difficult to consider for athletics. Sampling rates are not discussed but have the potential to be high due to the use of analog sensors and a wired communication infrastructure.

The IDEEA LifeGait System from Mini-Sun uses wired accelerometers distributed on the body to determine parameters of gait and motion [25].

SHIMMER [26] is a wearable, wireless platform with a 3-axis accelerometer and bluetooth radio. It is unique from many other platforms in that it had a microSD slot that allows for for 2 gigabytes (or more) of data storage. With the ability to store 80 days (in reality this time is limited to 10 days due to battery life) of 50 Hz data from the 3 accelerometers is an enabling technology for allowing rehabilitation professionals do long term classification of various clinical scores. Its drawbacks are that each node is only outfitted with low-G sensing, and it is not clear how node to node time synchronization is done.

Motion capture technology has long been used in the computer graphics community. One example is a flexible fiber-optic exoskeleton named ShapeWrap[27], which senses joint angles is available from the Measurand company. The system is wireless via a beltpack and works well for non-athletic motion, but still requires fiber bundles to be run across a users body. This limits motion and is too bulky for an athlete to wear and perform their natural motion.

Active magnetic systems have also been developed by Polhemus [28] and Ascension [29], and offer an alternative to cameras. Users are required to wear magnetic pickups, but the systems are capable of tracking position in space (a major problem in IMU based systems). A main drawback of magnetic systems are that they have trouble with metal in the surrounding environment and a ”hard-iron” calibration needs to be done for every new location. Further, their sampling rates become limited as the number of sensors increases, and range is limited to the size of the magnetic field that is generated by the system. Most such systems also tend to require the pickups to be tethered to a large beltpack.

Hybrid magnetic and inertial based systems have been developed by Veltink [30] and Xsens [31]. These are comprised of nodes that have 3 axes of gyroscope, 3 axes of accelerometer and, 3 axes of magnetic sensing. The system consists of a belt-worn master controller and a magnetic source, also belt-worn. While this combination of sensors is an excellent approach, it retains some common drawbacks of the other systems presented thus far. In addition to every node having to be tethered, the beltpack and magnetic source are too cumbersome for athletic activity - also the sampling rate is limited to 120Hz.

Goniometers, such as those available from Biometris Ltd. [32], are another approach to measuring joint angles, and are available with 1 or 2 dimensional measuring capability with a ± 2 degree accuracy. These exoskeletons have been used in analyzing golf swings [33] at a sampling rate of 100Hz. They are also used in physical and occupational therapy to measure range of motion on various parts of the body [34] and in computer graphics applications. The sampling rate and accuracy of these is impressive, however each such goiniometer attached to the body requires a belt pack or master module, and an array would require several belt packs making it difficult to perform sports activity. In addition, the exoskeleton needs to be strapped to both sides of each joint, which can be restrictive for athletes.

1.3.3 Clinical Basbeall Studies

In [5], the authors measured several metrics on baseball pitchers using a 6-camera system. Relevant findings for this thesis were the angular velocity of the shoulder's internal rotation. They found a maximum shoulder internal rotation of 6,520 degrees/second, with no margins of errors mentioned. In[1], the same metric was measured with peak values exceeding 10,000 degrees/second, with an average of 8,286 degrees/second. Again error margins are not discussed.

Dr. Thomas Gill, Medical Director of the Boston Red Sox, is intimately involved in the health of professional pitchers and has been involved in two previous studies examining the biomechanics of pitching. In 2001, Dr. Gill and his colleagues assessed shoulder kinematics and kinetics utilizing three-dimensional, high speed video data on 40 minor league pitchers, also published in [1]. In 2002, Dr. Gill examined the factors responsible for variance in the amount of force pushing the elbow toward the midline of the body [35]. This study indicated that statistically, 97.4% of the variance in elbow stress was related to four factors: shoulder abduction angle at stride contact; the angle between the ground and the upper arm at the point of the foot striking the ground during a pitch, elbow angle at instant of peak vulgus stress; the angle between the upper arm and forearm at the point of peak elbow stress, peak shoulder horizontal adduction angular velocity; the rate at which the angle between the ground and the upper arm change; and peak shoulder external rotation torque; and the torque of shoulder motion rotating about an axis parallel to the upper arm. These shoulder findings are the rationale for modeling shoulder kinematics in order to understand and aid in injury prevention and rehabilitation. A system such as sportSemble will lay the ground work to allow for easy, reliable and automated gathering of the data that sports medicine professionals need in order to continue research in this area.

Chapter 2

Hardware Systems

The hardware that this systems runs on is based upon Ryan Aylward's Sensemble project[12] (hence the name sportSemble) of which the fundamentals will be described briefly. Additionally, several modifications were made to the Sensemble hardware in order to use it for sports, and the motivation behind these changes will be described. The hardware can be broken into two parts; wearable battery-powered wireless nodes that are worn by athletes and a USB-powered basestation that they communicate with.

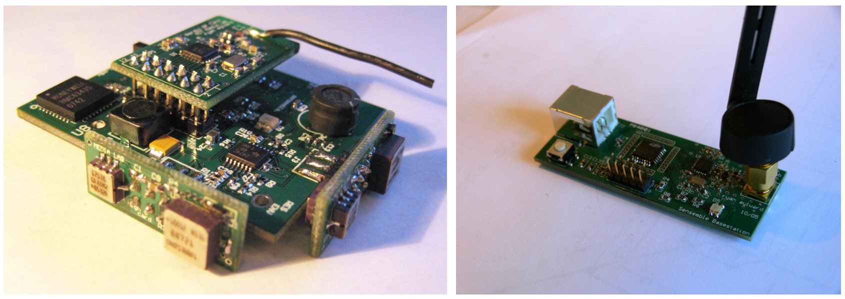


Figure 2-1: sportSemble Node(left) and basestation(right)

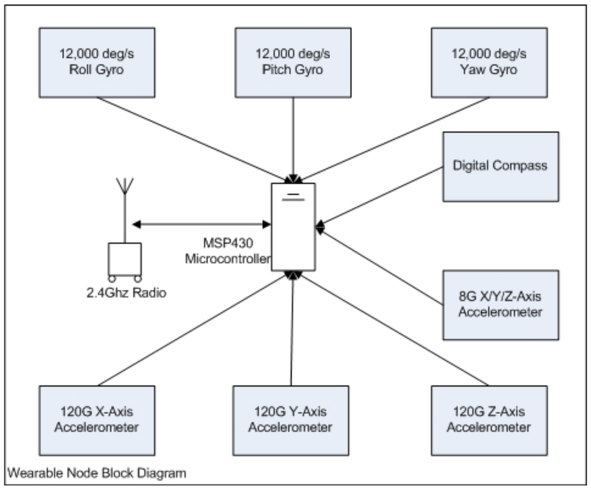

12,000 deg/s Roll Gyro
12,000 deg/s Pitch Gyro
12,000 deg/s Yaw Gyro
Digital Compass
MSP430 Microcontroller
2.4Ghz Radio
8G X/Y/Z-Axis Accelerometer
120G X-Axis Accelerometer
120G Y-Axis Accelerometer
120G Z-Axis Accelerometer
Wearable Node Block Diagram

Figure 2-2: sportSemble Node Block Diagram

2.1 Summary

Each of the wearable battery-powered nodes has three single-axis High-G accelerometers, three single-axis gyroscopes, a 3 axis Low-G accelerometer and a digital compass. They measure about 2.2 inches by 2.0 inches and have a weight of 44 grams with battery and attachment bracelet. The battery is a 145mAh lithium polymer rechargeable that was seen to continuously power a node for up to 3 hours of use. The output from the 3 High-G accelerometers and gyroscopes is analog and is processed by a 12-bit Analog-to-Digital converter[36](A2D) that is onboard the microcontroller. The additional sensors (Low-g accelerometer and compass) have a digital I2C[37] interface and share a common bus. In addition to standard power circuitry, there is also a 805.15.4[38, 39] radio interfacing to the microcontroller directly via a Serial Peripheral Interface(SPI)[40]; a block diagram is in Figure Figure 2-2. Each node has a unique address, and radio communication to the basestation is via a custom frequency hopping[41] protocol. Lastly, full schematics and Bills of material for all hardware are available in Appendix A.

2.2 Node High Level Design

Again, for detailed information regarding the initial design decisions that were made when the foundation hardware for sportSemble was built, refer to Chapter 2 of Ryan Aylwards Thesis[12]. We will concern ourselves here with the requirements and design decisions made in order to extend the Sensemble system and turn it into sportSemble. In general, the sampling rate of Sensemble was too slow for sports applications, and the gyroscopes and accelerometers did not have the necessary range in order to sense the extreme forces that occur during sports activity.

2.3 Requirements

The basic needs for sensing high speed sports actions are the ability to sense at extreme forces and the ability to sample at a very high rate. This is due to the inherent nature of such activity. As mentioned above, a limitation of Sensemble was the range of the accelerometers (± 10 Gs) and the range of the gyroscopes (± 300 degrees/sec). The target applications for sportSemble require a minimum of ± 10000 degrees/sec [42] angular velocity and ± 100 Gs of acceleration to be sensed. Further, Sensemble samples at a rate of 100Hz. This too is not sufficient when considering, for example, that a baseball pitcher's peak acceleration lasts for 3/1000 of a second. The target requirement of sportSemble is to sample at least 3 times during this peak acceleration, which results in a required sampling rate or 1000Hz. There are other hardware issues that arise when attempting to sample at this rate that will be discussed in the following sections. Luckily, many of the problems that arise when increasing range and sampling rate can be addressed in software, these are covered in section Section 3.1.2. As a follow on to Sensemble, Ryan made changes to accommodate some of these needs and they are described in Chapter 7 of his thesis[12]. The main points from his thesis will be re-iterated and the new pieces of hardware will be introduced and discussed.

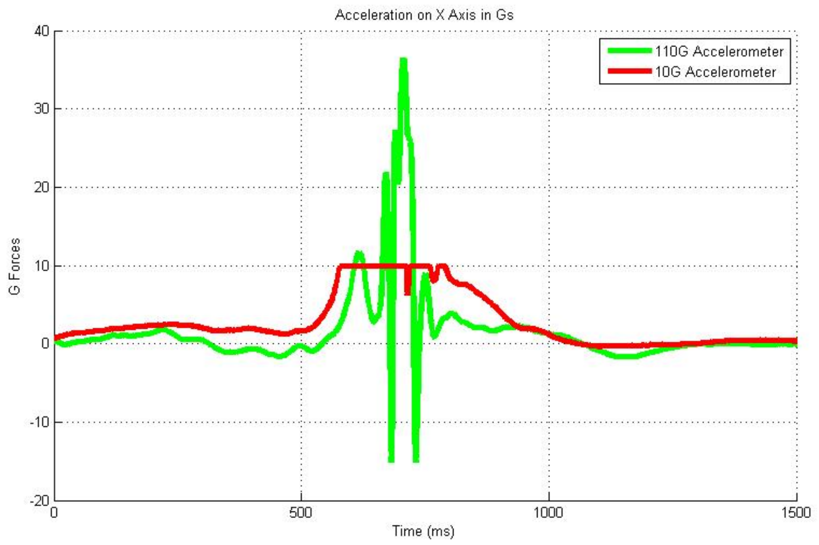


Figure 2-3: Similar motions as recorded by ±10G and ±120G accelerometers, showing the 10G accelerometer being clipped due to lack of range.

2.4 Gyroscopes

The Analog Devices ADXRS300[43] is a ±300 degrees/sec Yaw Rate Gyroscope that was also used in Sensemble. Due to its yaw-sensing-only ability, there was a need for 3 of these to be mounted orthogonally on daughterboards, as shown in Figure 2-1, to cover all three axes. The same design is preserved in sportSemble, as is the same gyroscope. The is possible because the ADRX300 has a range than can be extended to ±12000 degrees/sec, meeting the ±10000 degrees/sec requirement. This is accomplished by lowering the driving voltage of the gyroscope from 12.5V to about 5V. The details of doing this for were provided by an application note[44] from Analog Devices and also through personal contact with Analog Devices engineers, and are described in detail in Section 7 of Ryan's thesis[12].

2.5 High-G Accelerometers

The high-G accelerometers that are used in sportSemble are Analog Devices ADXL193[45] accelerometers that range from ±120 Gs or ±250 Gs; the 120G configuration is used in sportSemble. They are required because the common ±10 G accelerometers, such as used in the Semsemble project, experience clipping at accelerations greater than 10 Gs. This is visible in Figure 2-3; where two similar motions were recorded, and the 10 G accelerometer is obviously clipping at 10 Gs, above which appears as a flat line. The ADXL193 features the same pinout as the Analog Devices ADXL210E[46] that was used in Sensemble, but senses on only one axis. This would have required major rework, had the single axis gyroscopes not also required daughterboards to mount them in orthogonal positions.

2.6 Low-G Accelerometer

Extending the range of the accelerometers created a concern that each node would not have enough resolution to accurately sense low-G motions. This prompted the addition of a STMicroelectronics LIS302DL[47] 3-axis digital output accelerometer. The LIS302DL is capable of a range of ±2 Gs or ±8 Gs and has an I2C[37] output. The range chosen for sportSemble was the ±8 Gs, and it was critical for this accelerometer to have a digital interface, as the chosen microcontroller did not have any more analog input channels available. It did, however, have several unused pins that could be used for digital communication. The drawback of I2C communication is that it is slow in comparison to just sampling an analog channel (the maximum sampling rate of the LIS302DL is about 100Hz). This becomes problematic when sampling at high rates, as sportSemble does, and is discussed further in the Softwares Systems Section 3.1.2.

2.7 Digital Compass

Another addition to the original Sensemble hardware is the Honeywell HMC6343[48] 3 axis digital compass. The technical details of the compass will follow, but first it is important to understand why this compass is needed. The original Sensemble used the acceleration and angular velocity reading from the sensors in order to create or modify a music stream

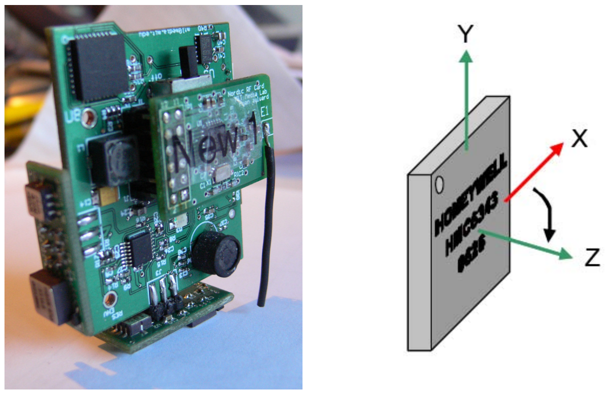


Figure 2-4: HMC6343 Compass orientation with respect to gravity

for dancers, which did not necessarily require knowing anything about the position or orientation of the nodes on the body. SportSemble intends to drive biomechanical models of the body to model athletic motion. In order to do this, an initial body position needs to be established, upon which further positions are derived from integrating gyroscope and accelerometer data, possibly using subsequent compass data as an additional state update. This particular compass provides not only heading information, but also pitch and roll information, as it contains its own hardware to sense tilt. Given a skeletal model with node positions on it, the compass heading, pitch and roll data can be used as a starting point for motion analysis.

The HMC6343[48] has a duty cycle of 10Hz and is by far not capable of keeping up with our desired sampling rate. However, its data can be used to update node orientation, as as needed to null any drift in the gyroscopes and accelerometers. It, like the low-G accelerometer, has a digital I2C interface and shares the same communication bus as the low-G accelerometer. It is important to note the node's orientation with respect to the Earth's magnetic field, as the compass uses this to determine its heading. Figure 2-4 shows the most common orientation that the node is in when worn on the body, and the compass's axis orientation. The force of gravity is in the negative Y axis. Sampling the compass while in the data gathering loop also becomes problematic due to the relatively slow speeds of digital sensor interfaces. Again this is described in detail in section Section 3.1.2.

2.8 802.15.4 Radio

The radio used in sportSemble is a Nordic nRF2401a [49], which has a maximum bandwidth of 1 Mbps and an output power of +4dBm. The radio uses a 1MHz channel spacing with a channel range in the ISM band, from 2400-2527MHz. SportSemble requires a sampling rate of 1000Hz from each of its 6 analog sensors and the maximum sampling rate from the compass(10Hz) and low-G accelerometer ($\tilde{1}$00Hz). The limited bandwidth of the onboard radio also becomes a problem if you calculate the data rate necessary per node. We start with how many bits are needed for a single sample of the analog gyroscopes and accelerometers, and get a *RawAnalogBW*:

$$90 bits/sample = \begin{cases} \text{12 bits Pitch Gyro} \\ \text{12 bits Roll Gyro} \\ \text{12 bits Yaw Gyro} \\ \text{12 bits X Accelerometer} \\ \text{12 bits Y Accelerometer} \\ \text{12 bits Z Accelerometer} \\ \text{16 bits timestamp} \\ \text{2 bits sample type} \end{cases} \tag{2.1}$$

and the number of bits needed for a single low-g accelerometer sample *RawLowGBW*:

$$42 bits/sample = \begin{cases} \text{8 bits Pitch Gyro} \\ \text{8 bits Roll Gyro} \\ \text{8 bits Yaw Gyro} \\ \text{16 bits timestamp} \\ \text{2 bits sample type} \end{cases} \tag{2.2}$$

and finally the number of bits needed for a compass sample $CompassBW$:

$$66 bits/sample = \begin{cases} \text{16 bits heading} \\ \text{16 bits pitch} \\ \text{16 bits roll} \\ \text{16 bits timestamp} \\ \text{2 bits sample type} \end{cases} \tag{2.3}$$

Going on the initial 1000Hz/100Hz/10Hz sampling rates for the analog sensors, low-g accelerometer and compass, respectively, we get a numbers of 90,000bps (bits per second), 420bps and 660bps of raw bandwidth needed per node per second. Each packet (of which there would need to be 1110) would also include communication protocol overhead of:

$$32 bits/packet = \begin{cases} \text{8 bits Packet Type} \\ \text{8 bits Source Address} \\ \text{16 bits Packet Sequence number} \end{cases} \tag{2.4}$$

This bring the bandwidth in bps for each node to 126,600. If we wish to have a 5-node network and ignore radio overhead and ignore any packet loss over the wireless medium, the number grows to 633,000bps. This is more than the 1Mbps that the Nordic [49] radio used in Sensemble offered, hence required a different solution. The option to use a faster radio still does not meet the requirement of data integrity. Sensemble was intended for real time dance performances and any packet loss(over the wireless link) in the data stream was not recoverable (using retransmission) because of the real time nature of live performance. SportSemble's purpose, recording data that could be used in biomotion analysis, requires that all data be present, and any packet loss would be unacceptable. As there is no real-time performance aspect or latency penalty in biomotion analysis, the host node can repeatedly query the nodes for missing or damaged packets. The best solution to this scenario is to store the sampled data locally on the microcontroller's flash memory, and download the data after activity has completed. This concept will be explored further in the Microcontroller section(Section 2.9).

It is important to note that there are only 3 sample types (compass, low range, high range), hence the sample type flag is two bits. However, this would be an accounting nightmare when it came time to manage the byte-addressed flash of any microcontroller. Thus, the sample type requires an additional 6 bits of storage in the flash based approach.

2.9 Microcontroller

Sensemble was built with a Texas Instruments MSP430F148[50, 51, 52] microcontroller. It runs at 8MHz and has several analog and digital input/output options for communication, hence is ideally suited for Sensembles purpose. It also meets sportSemble's needs in terms of the added compass and low-G accelerometer and can handle the 1000Hz sampling rate with flash write (although just barely). The one area in which it is lacking is in flash memory, which is limited to 48k bytes. This was plenty for Sensemble, as it only stored its program code in flash [51, 52] and all other data was transmitted wirelessly in real time. If sportSemble were to store data on board, 48k bytes would only be sufficient space for about 4 seconds of data. Using the numbers from the radio section 2.8, of about 100,000 bits per second of data, which is roughly 10k bytes/second, and allowing for 8k of program code, we obtain the 4 second number. This is insufficient to meet the target number that was set at 10 seconds of data, and led to the requirement that the microcontroller be swapped out for another with more flash memory. The largest available flash memory in this series of microcontroller was 64k bytes, which still did not meet these requirements. Luckily, in Q1 of 2008, Texas Instruments was introducing the MSP430F2618[53], with 116k bytes of flash and the same peripherals as the MSP430F148. Q1 was too long to wait in order to start development, but samples were made available late in 2007, which gave sufficient time. The only caveat was that the chip footprint that Sensemble used was QFN64[54], and the 2618 only came in a QFP64[55] footprint. In short, the footprint of the older chip is smaller than the footprint of the newer chip, hence the hardware had to be rebuilt. Finally, the 116k of flash on 2618 would offer plenty of space to meet the 10 seconds of data requirement.

Chapter 3

Software Systems

The software infrastructure for sportSemble can be broken down into three categories: the firmware running on the nodes that gathers all data and does reading/storage to memory, the RF protocol that facilitates command and control between the basestation and the nodes in the network, and finally the PC side application that communicates with the basestation via USB and dictates the actual command and control of the nodes in the network. A typical system layout for a baseball pitcher is depicted in Figure 3-1 and consists of 5 nodes located on the body and communicates with the basestation. While this is typical for the experiments run, it by no means is mandatory. SportSemble has been designed to allow for any number of nodes located anywhere on the body (or possibly on athletes equipment), with no configuration changes needed to gather biomechanical data.

3.1 Firmware

The firmware running on the nodes at a high level is simple: wait for a beacon via the radio and react based on the command encoded in the beacon packet. Reacting can be one of 3 things: do nothing, begin sampling data to onboard flash, or return a sample to the basestation. The firmware for the basestation also has 3 basic functions; beaconing at

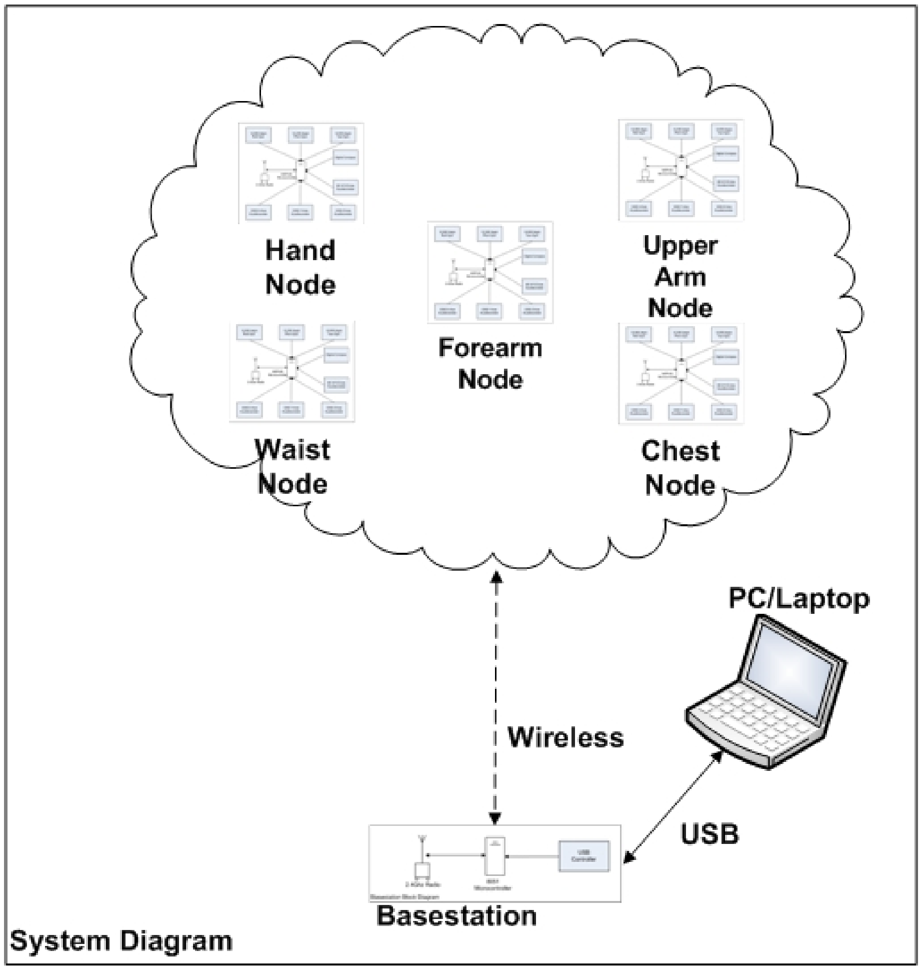


Figure 3-1: sportSemble high level system architecture

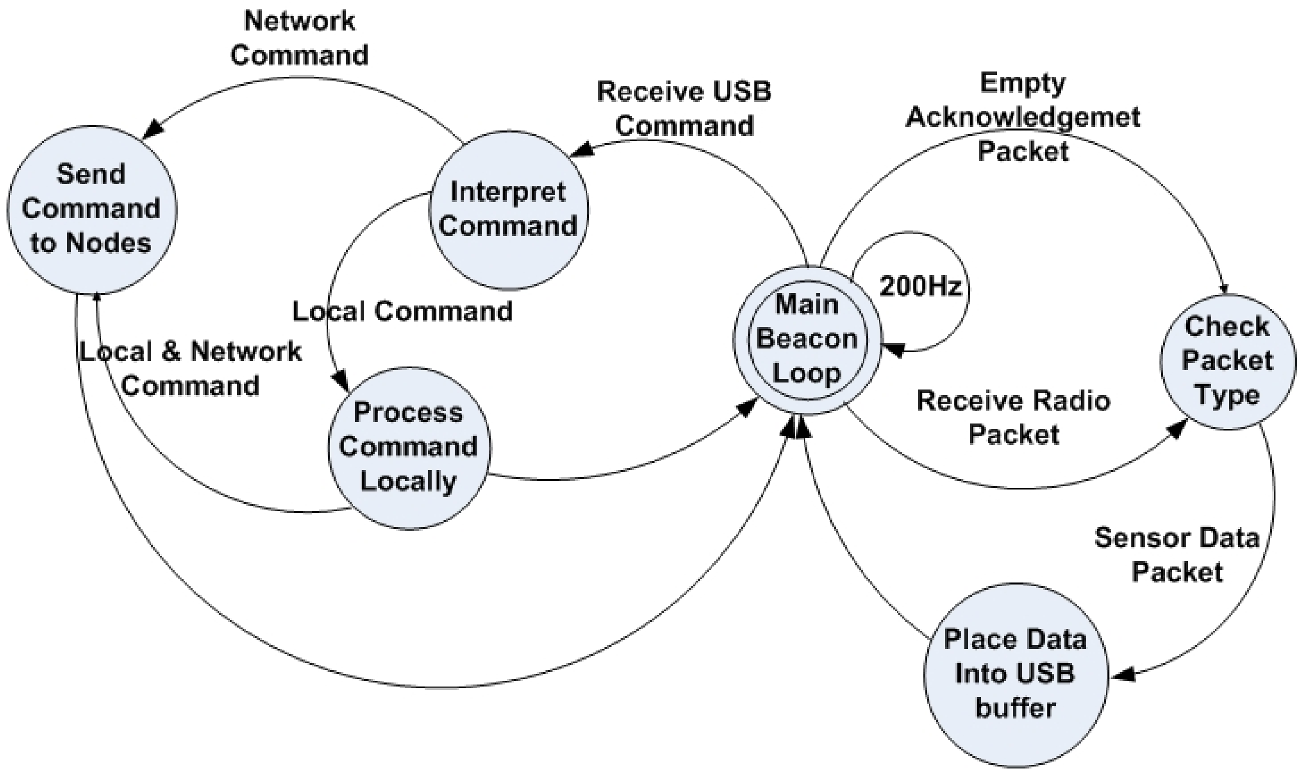


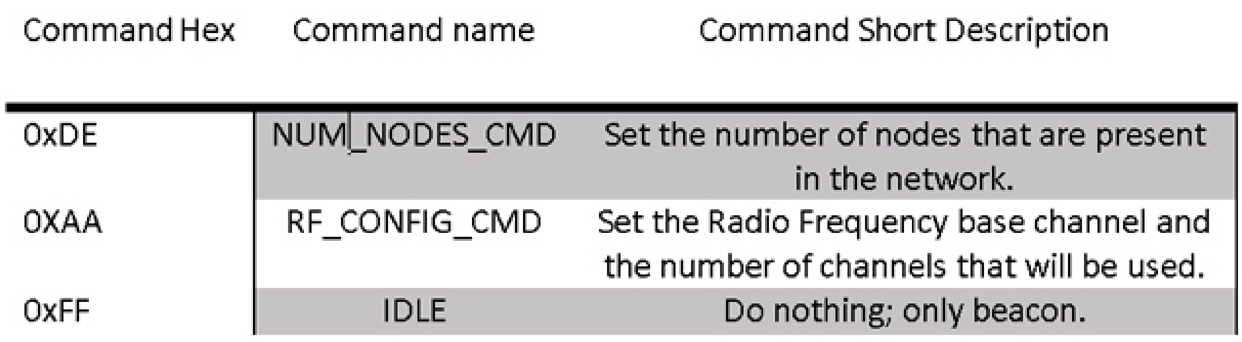
Figure 3-2: Basestation State Machine

200Hz, receiving commands via the USB, transmitting them to the network of nodes when appropriate, and returning the results of commands via USB to the host PC.

3.1.1 Basestation Firmware

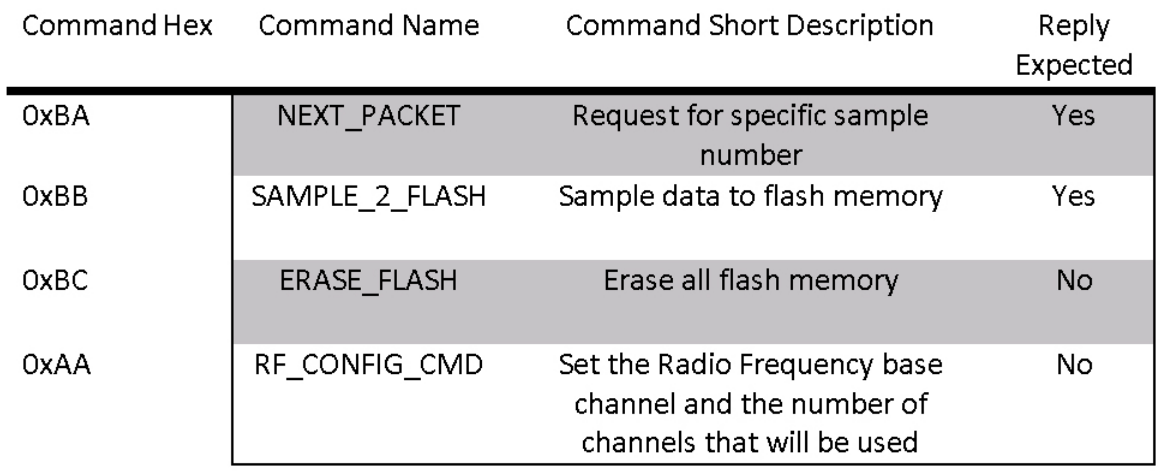
The basestation hardware remained unchanged from the original Sensemble[12]; however, the firmware was changed extensively. A state diagram of the basestation's main operations is in Figure 3-2. The basestation microcontroller clock is configured to run at 24MHz, while the main beaconing loop runs off of *Timer 1*. Every time an interrupt[56] is generated by *Timer 1*, a beacon is sent. Each beacon contains no data, but simply ensures that all nodes keep synchronization.

Things get more interesting when a command is received from the host PC (see Figure 3-1) via an interrupt[56] that indicates USB data has been received - this is considered a USB



Command Hex	Command name	Command Short Description
0xDE	NUM_NODES_CMD	Set the number of nodes that are present in the network.
0XAA	RF_CONFIG_CMD	Set the Radio Frequency base channel and the number of channels that will be used.
0xFF	IDLE	Do nothing; only beacon.

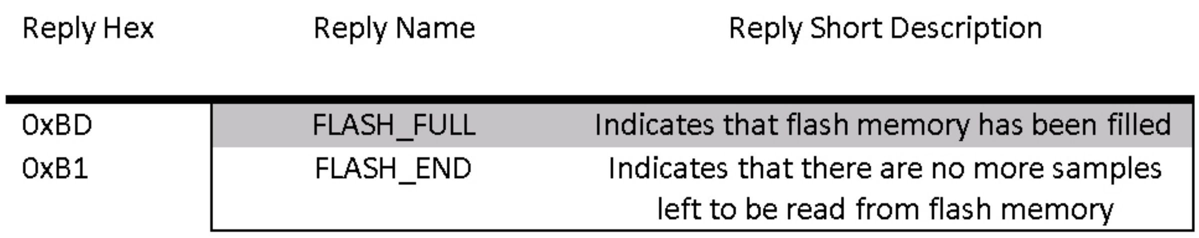
Figure 3-3: Table of local commands between USB Host and basestation



Command Hex	Command Name	Command Short Description	Reply Expected
0xBA	NEXT_PACKET	Request for specific sample number	Yes
0xBB	SAMPLE_2_FLASH	Sample data to flash memory	Yes
0xBC	ERASE_FLASH	Erase all flash memory	No
0xAA	RF_CONFIG_CMD	Set the Radio Frequency base channel and the number of channels that will be used	No

Figure 3-4: Table of network commands between USB Host and nodes

Command. The received command is then interpreted (based on a 2 byte code stored inside of it) to check if it is meant for all nodes in the network or if it is a local command. Local commands are meant only for the basestation and are not broadcast to the network. A complete list of these is available in Figure 3-3. The *RF_CONFIG_CMD* is part of the RF Protocol and will be discussed in section Section 3.3. The *IDLE* command tell the basestation to simply beacon and does nothing. The *NUM_NODES_CMD* command tells the basestation how many nodes are present in the network. This is necessary in order for the basestation to know how many nodes it should wait to receive responses from before commands are sent out into the network and replies are anticipated.



Reply Hex	Reply Name	Reply Short Description
0xBD	FLASH_FULL	Indicates that flash memory has been filled
0xB1	FLASH_END	Indicates that there are no more samples left to be read from flash memory

Figure 3-5: Table of network replies between nodes and USB Host

Commands that are not meant for the basestation (Figure 3-4) are broadcast out to the network. We will start with the simplest, that being the *ERASE_FLASH* command, then move on to the others. *ERASE_FLASH* is sent out over the wireless link, and all nodes erase their flash memory. Once flash has been erased, nodes are ready to receive a *SAMPLE_2_FLASH* command. When the basestation receives and interprets a *SAMPLE_2_FLASH* command, it does two things. First, it zeros out an array of booleans that is of size *NUM_NODES*. It then forwards the command to all nodes in the network. Upon receipt, each node samples until its flash memory is full, then creates a *FLASH_FULL* packet (Figure 3-5, Figure 3-17) and returns it along with its unique ID to the basestation. Upon receipt, the basestation then updates the boolean (that keeps track of which node has reported flash as being full) corresponding to the given to be *true*. This occurs until every boolean in the array is in the true state (indicating that each node has sampled until its flash was full). Once this occurs, the Host PC is notified that the basestation has received a *FLASH_FULL* packet from each node.

The next step is to retrieve the data from each node, which is triggered by the receipt of a *NEXT_PACKET* packet from the USB host. Each of these contains a sample number indicating which sample should be sent back to the host. The basestation sends these 2 pieces of data to the nodes and again initializes the array of booleans. Only this time, the booleans keep track of which node has responded with the requested data. The packets that come back are of type *NEXT_PACKET* and are automatically placed in a buffer that is sent back to the USB host (this includes duplicates). If a requested sample number is greater than the number of samples a nodes has, a *FLASH_END* (Figure 3-18) packet is returned from the node. The basestation continuously asks for the requested packet number until each node has replied with that packet or until a *FLASH_END* packet has been received for each node.

3.1.2 Node Firmware

The nodes operate in a manner that is similar to that of the basestation, in that they wait for packets to be received with commands and react based upon what type of packet is received. Each node is informed of a new packet by the radio via an interrupt[56] on Port 1.4[53], and each received packet is handled by the *nRF_data_waiting()* function in *mainRedSoxNode.c*. (All source code can be found in Appendix B)

Based on the type of packet that is received, the node takes appropriate action. We will cover all packet types listed in Figure 3-4 except the *RF_CONFIG_CMD*, which is discussed in Section 3.3. However, before we can begin discussing sampling and writing to flash, we will discuss the structure of the MSP430F2618's[53] flash memory, as it proved to be quite differently structured from other MSP430 microcontrollers.

Figure 3-6 shows the memory map of the MSP430F2618. All MSP430 series microcontrollers, until the introduction of the 2x1x series, had 64k bytes of flash memory or less. As described in Section 2.9, the reason for using the MSP4302618 was because of its larger flash memory, which enables more data storage, hence a longer sampling time. This added feature came with two problems that needed to be overcome. The first was that the extended memory required an address space of 20 bits (compared to the previous 16 bit address space). The second was that the interrupt vector table (which normally resides at the end of the flash memory) was now stored in the last 64 bytes of the lower 64kb of flash memory, as shown in Figure 3-6. This is a hardwired feature of the chip's microcode that had to be worked around.

In order to read or write the memory in the lower 64kb of flash, standard pointer referencing and dereferencing could be used [1]:

[1]This code does not include the semantics for locking and unlocking the flash memory

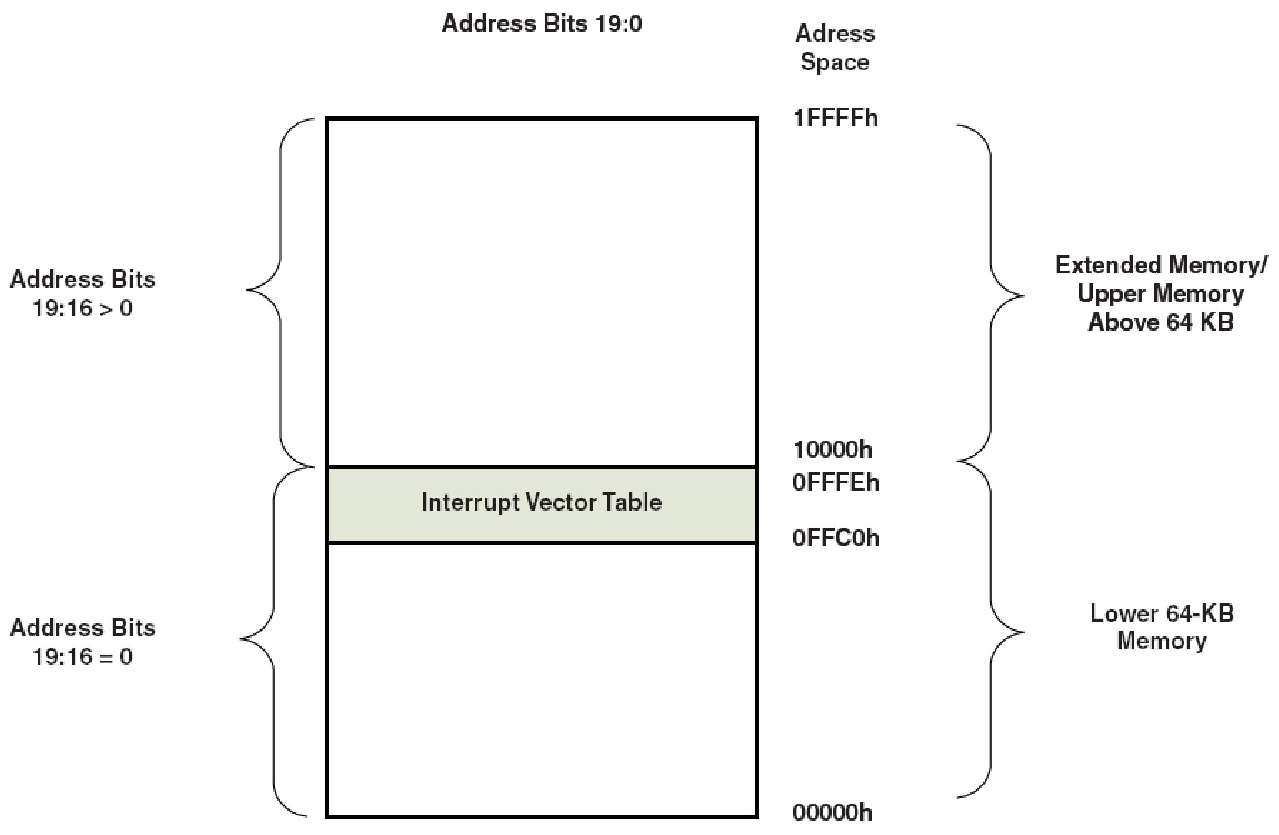


Figure 3-6: Memory Map of MSP430F2618

```
//A Write
unsigned char * address = 0x3322; // destination address in flash
*address = 0xFFFF; //dereference and write to the address



//A read
unsigned char data; //where the data will go
unsigned char * address = 0x3322; //where to read the data from
data = *address; //dereference and read address
```

It is important to note that the mechanism that allows us to do this is that all pointers in the MSP430 are 16 bits. When data needs to be read or stored to an address that is greater than 16 bits, this mechanism cannot be used. In fact, there is no way to do this in C - it must be implemented in assembly. The read of a byte then looks like this:

```
.global __data20_read_char
.text
__data20_read_char:
        push.w  R13         ;upper word
        push.w  R12         ;lower word
        popx.a  R13         ;20-bit address
        ;Move 8-bit char value at the 20-bit addr locn to R12
        movx.b  0x0(R13), R12   ;R12 contains the return value
        reta                              ; return
```

and would be used in standard C code like this:

```
unsigned char __data20_read_char(unsigned long __addr);

unsigned long address = 0x1FFFF; //top byte of flash
unsigned char data = __data20_read_char(address);
```

Similar functions are available for writing a byte to the extended memory, and can be found in Appendix B.

This addresses the issue of mixed 16-bit and 20-bit memory addresses, but it does not address the issue of having to not write into the address space where the interrupt vector is stored. This required every read and write to the flash memory to be checked for which part of memory it was in (upper or lower) and the appropriate methods to access it be used. Now that we have covered how to make use of the flash memory, we can move on to the state machine that governs every node's behavior (Figure 3-7).

When a node boots up, it goes through several configuration steps and then goes into a infinite loop in the *main()* function, then waits for interrupts to occur from incoming radio packets. The notable parts of the configuration are as follows (definitions can be found in the MSP430F2618 datasheet [53]):

- Microcontroller master clock speed - *8MHz*
- *UCB0 in SPI Mode*[40] at *400kbps*; sourced from ACLK
- *UCB1 in I2C Mode*[37] at *100kHz*; sourced from ACLK
- *ADC12[Channel 0-6]* with input on each respective analog channel *A[0-6]*
- Flash controller at *400kHz*; sourced from ACLK
- *PORT 2 pins 1,4,5* as digital output (for LEDs)
- *PORT 2 pins 0,3,6* as digital input (for LEDs)
- *Timer B0* configured to run at *1000Hz*; stopped
- Low-G Accelerometer - powered up running at full scale ± 8 Gs with X,Y,Z axes enabled
- Compass in run mode with an update rate of 10Hz

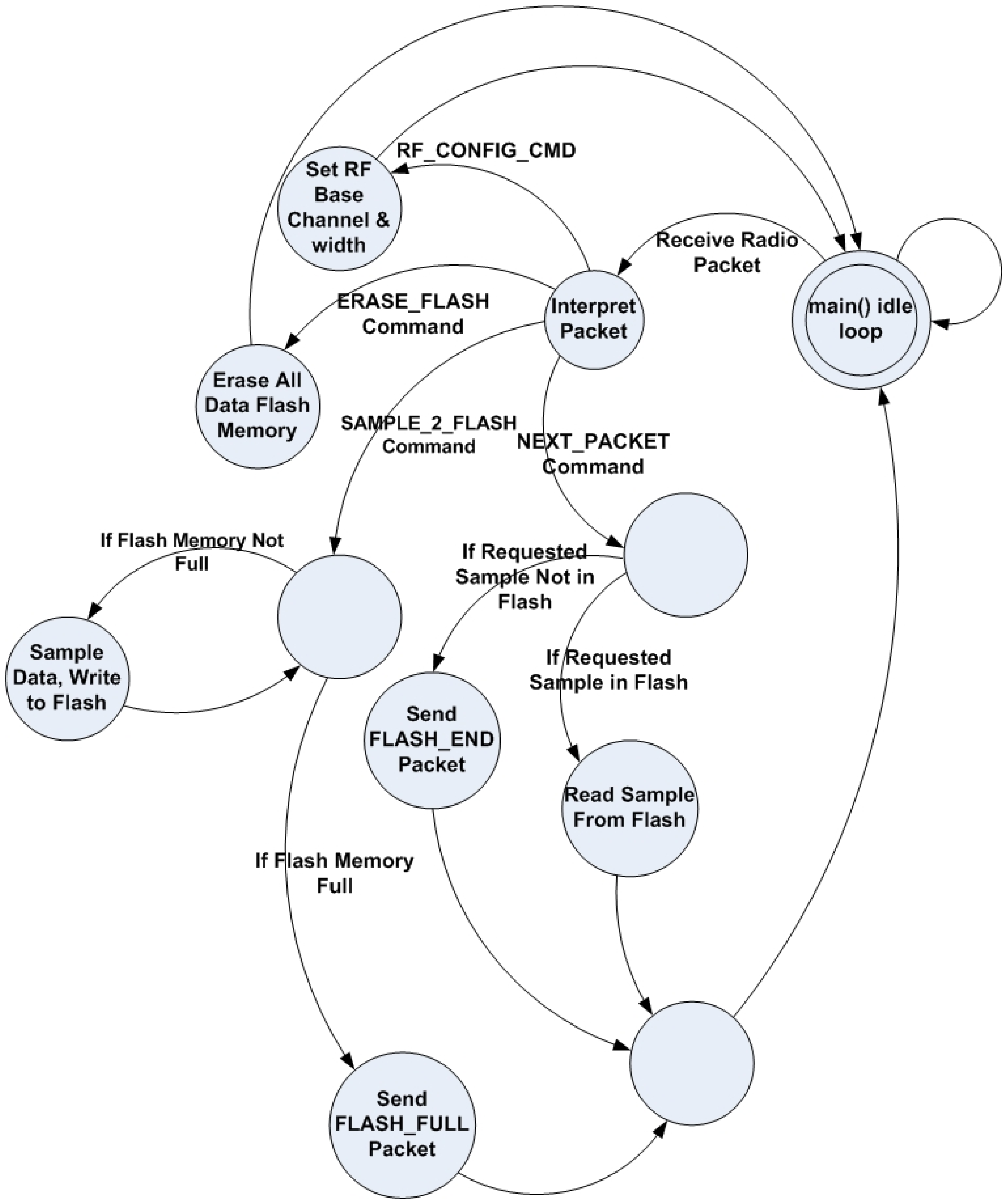


Figure 3-7: Node State Machine

Once all of this has been done, the node enables interrupts and goes into an infinite *while* loop, waiting for an interrupt from the radio once it receives a packet. Each received packet is processed based on its packet type (packet structure and the RF_CONFIG_CMD are covered in Section 3.3).

The simplest command, and one that must be sent to the node before any data sampling can occur, is the *ERASE_FLASH* command. When an *ERASE_FLASH* command is received (as with all other commands, unless noted), interrupts are disabled until the command has been processed. Once this occurs, the flash memory is erased in 512 byte segments from memory location *0x5000* to *0xFDFF* and *0x10000* to *0x1FFFF*[2]. This is dependent on the 16/20 bit addressing that was described previously in this section; 16 bit addresses use the *erase_flash_segment()* function and 20 bit addresses use *erase_upper_flash_segment()* function, both of which can be found in *flash.c* in Appendix B.

Once flash memory has been erased, a *SAMPLE_2_FLASH* command can be received and processed. When a command of this type is received, interrupts are enabled, except those that would be received from the radio. *Timer B* is then enabled, and the 1000Hz sampling loop begins. Sampling just the 6 analog sensors (3 High-G Accelerometers and 3 Gyroscopes) is trivial. The 12 bit A2D[36] is used to do 6 conversions on the appropriate channels. In order to maximize flash memory space, we use 3 bytes (24 bits) to store two 12 bit A2D samples, for example:

$$Byte1 = \begin{cases} \text{Sample 1 First 8 Bits} \end{cases} \tag{3.1}$$

$$Byte2 = \begin{cases} \text{Sample 1 Last 4 Bits} \\ \text{Sample 2 First 4 Bits} \end{cases} \tag{3.2}$$

$$Byte3 = \begin{cases} \text{Sample 2 Last 8 Bits} \end{cases} \tag{3.3}$$

The structure of the analog sensor packet is depicted in Figure 3-8 - it is simply the packed data along with a *SAMPLE_TYPE* to identify what type of data it is. In order to simplify

[2]Note that one range of addresses is 16 bits wide and the other is 20 bits wide.

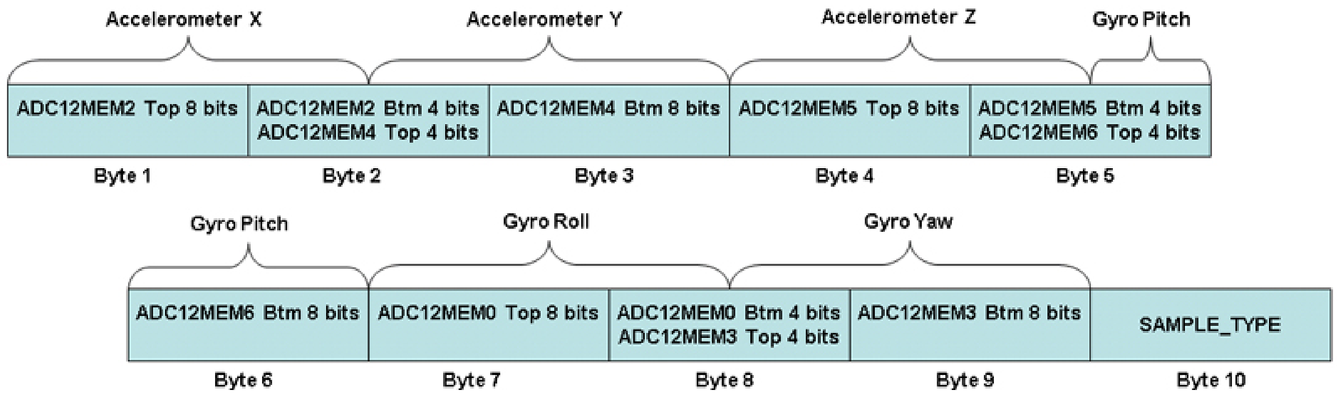


Figure 3-8: Analog Sensor Flash Structure

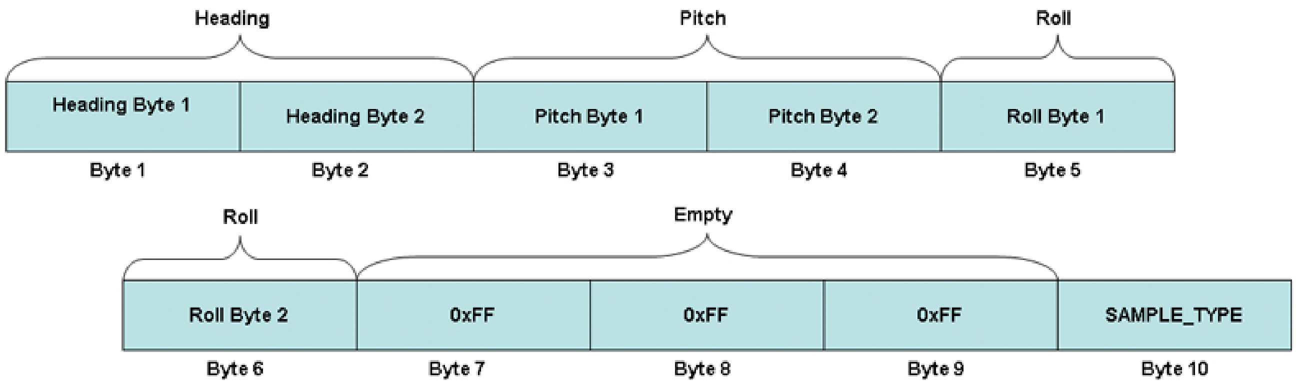


Figure 3-9: Compass Flash Structure

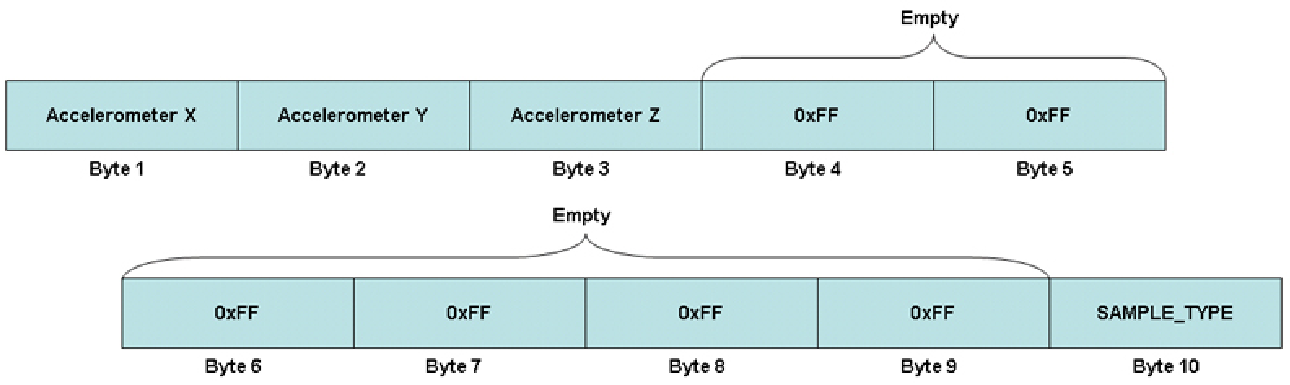


Figure 3-10: Low-G Accelerometer Flash Structure

flash memory management, the same size packet is used to store compass(Figure 3-9) and low-G accelerometer (Figure 3-10) data, just with a different packet type.

As we have covered sampling the analog sensors and how all sensor data is stored, now we will discuss sampling the compass and low-G accelerometer. As mentioned previously, the amount of time it takes to sample a digital I2C[37] sensor is much longer then that of an analog A2D conversion. Running at 1000Hz, we have 100 milliseconds (ms) to complete all sampling and write the data to flash. Writing the 10 bytes to flash takes about 80ms, leaving about 20ms for sampling. The amount of time it takes to get a set of data (X, Y, Z) from the low-g accelerometer is about 35ms and the compass takes about 50ms. Based on this, it was obvious that directly sampling a digital sensor and writing its sample to flash was impossible if the 1000Hz sampling rate is to be preserved. There is a solution to this, thanks to the asynchronous nature of I2C. The approach used is to break the I2C commands and bytes that are sent and received into stages, and execute one stage at every iteration of the sampling loop. For example (each step is an iteration through the loop; in each iteration, unless otherwise noted, the analog sensors are sampled and their data is written to flash):

- send I2C start, place read command in TX Buffer [stage 1 complete]
- check for TX completion; if TX done send I2C stop [stage 2 complete; otherwise stay in stage 2]
- check for I2C stop completion; if done send I2C start [stage 3 complete; otherwise stay in stage 3]
- check for ACK of start; [if ACK'd stage 4 complete; otherwise stay in stage 4]
- check for RX of byte; [if no more bytes expected, stage 5 complete; otherwise stay in stage 5;
- send I2C stop; [stage 6 complete]
- check for I2C stop completion; if done sampling is finished otherwise stay in stage 7

The process is the same for the compass and the low-G accelerometer, the only differences are the I2C addresses used and the number of bytes that are expected to be read back. Additionally, the compass is only polled every 100 iterations (because its maximum sampling rate is 10Hz) and the low-G accelerometer is polled every 10 iterations (because maximum sampling rate is about 100Hz). The sampling rate is governed by the speed at which one can clock the I2C communication bus.

The sampling loop continues until the flash memory is full. When this occurs, a packet of type *FLASH_FULL* is sent back to the basestation, and we return to the *main() idle loop* state (Figure 3-7). The next logical command that can be received by a node is the *NEXT_PACKET* command. This command comes along with a sample number that is being requested. This sample number needs to be translated into a flash memory address, which is simple due to our samples all being the same size. The formula for doing this is:

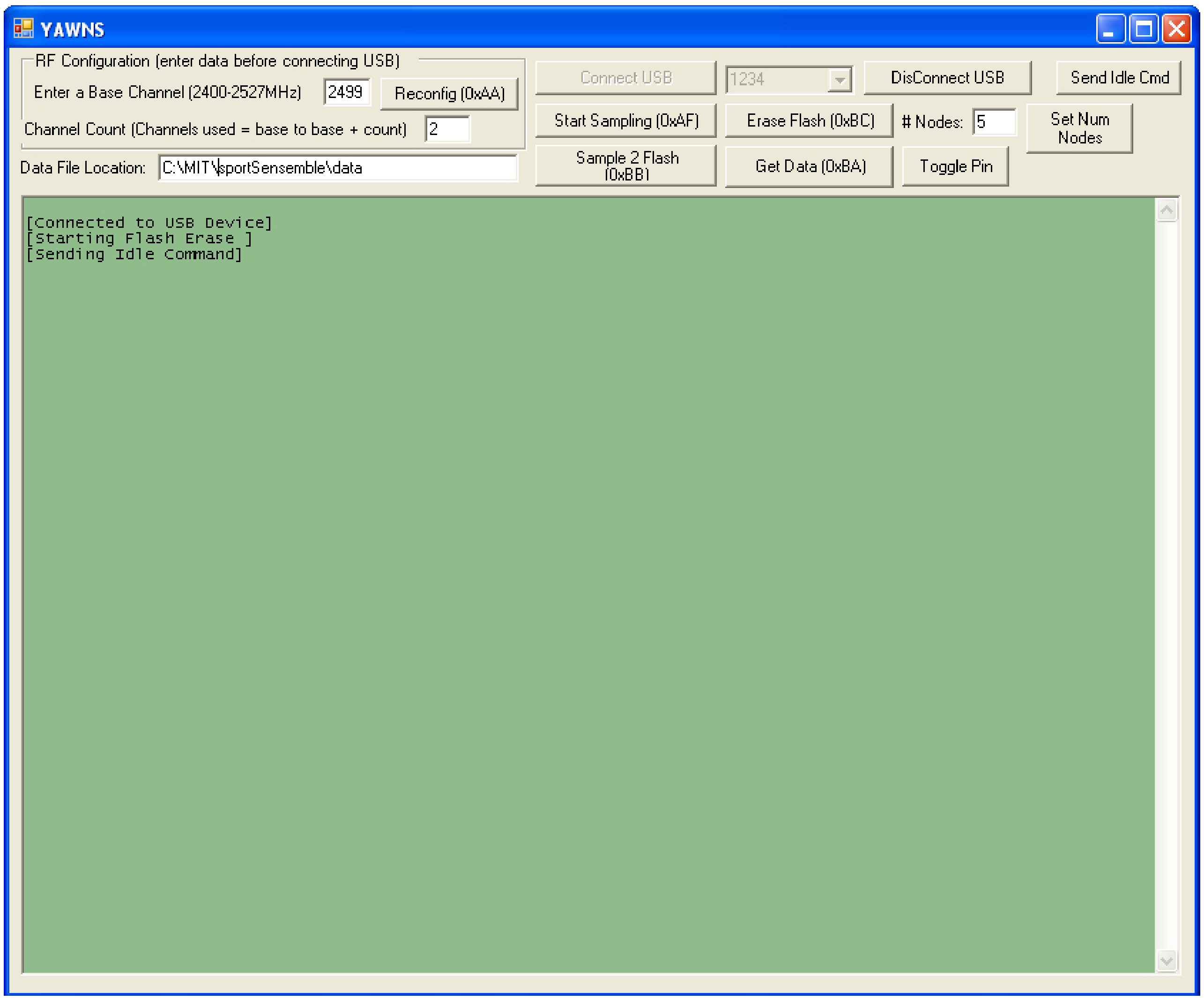
```
Flash Address = BASE_FLASH_ADDRESS + (REQUESTED_SAMPLE_NUMBER * SAMPLE_SIZE)
```

The flash address is then checked to ensure that it is within the appropriate bounds for where data is stored. If it is, then the data is read from flash memory and packed along with the *REQUESTED_SAMPLE_NUMBER* into a packet and sent back to the basestation, and we return to the *main() idle loop* state. If the flash address is not within the bounds (this usually occurs when we have reached the end of flash memory and there are no more samples), a *FLASH_END* packet is created and returned to the basestation[3], then the program returns to the *main() idle loop* state.

3.2 PC Host Application

The PC host application is the source of all command and control for sportSemble. It is a Visual Basic application that is responsible for communication with the basestation, controlling the network, and gathering all data from the network. The best way to describe

[3] *Variables such as BASE_FLASH_ADDRESS* and *SAMPLE_SIZE*, along with the bounds of flash memory, are defined in *flash.h*, which can be found in Appendix B



Enter a Base Channel (2400-2527MHz): 2499

Channel Count (Channels used = base to base + count): 2

Data File Location: C:\MIT\sportSensemble\data

1234

Nodes: 5

```
[Connected to USB Device]
[Starting Flash Erase ]
[Sending Idle Command]
```

Figure 3-11: PC Host Application Screen Capture after connection to the USB basestation

the functionality of the application is to go through the buttons that the GUI has (Figure 3-11) and detail what each does.

The logical place to start is with connecting the application via USB to the basestation. This and all other USB based code uses Silicon Labs USBXpress API[57]. USBXpress is linked into the GUI via a windows DLL[58] that exposes several functions that the visual basic code uses in order to communicate with the basestation. Before the *Connect USB* button is clicked, it is important to ensure that the *RF Configuration* for the network is properly entered (this is part of the RF protocol and is covered in Section 3.3) and that the number of wireless nodes that are in the network is entered into the *# Nodes:* text field. Once the *Connect USB:* button has been clicked, the background of the lower area of the screen will change from gray to green, and the message *[Connected to USB Device]* is displayed. It is important to click the *Set Num Nodes* button after a connect to tell the basestation the size of the network. The default is 5 and *Set Num Nodes* does not need to be clicked if 5 nodes will be in the network.

With a successful connection to the basestation, data can be gathered, erased and downloaded from the nodes. The most logical first step, as always, is to erase the flash memory of the nodes. This is accomplished by clicking the *Erase Flash (0xBC)* button. Clicking this triggers a call to the USBXpress API *SI_Write()* function with an argument of *ERASE_FLASH* (See Figure 3-3). Erased flash can now be sampled to; sampling is triggered by clicking the *Sample 2 Flash (0xBB)* button, this again triggers a call to *SI_Write()*, but with an argument of *SAMPLE_2_FLASH*. The details of what happens once these commands are received by the basestation are covered in Section 3.1.1.

Getting the data back requires another command to be sent, however this time repeatedly. Clicking the *Get Data (0xBA)* triggers a loop that starts calling *SI_Write()* with *NEXT_PACKET* as the command and the value of a counter along with it. The counter specifies which sample is being requested from the nodes and is increased only when the basestation has received data from each node for that specific sample number. The data that the basestation receives from the nodes is retrieved using the *SI_Read()* USBXpress API function call, passing a buffer to be filled.

Since, this is a wireless system, and some packet loss is unavoidable, certain mechanisms to ensure data integrity are in place. It is important to note that the basestation passes all packets received to the host PC regardless of whether they are out of order or not. This approach allows the application to decide what data to discard and what action to take next. The basestation's behavior is described in Section 3.1.1. In general, it handles ensuring that the requested sample number has been received from each node before allowing the next to be requested. This may seem like the management of the process is being spread out in many places, but is necessary because letting the host PC application manage this created too much USB traffic and was very slow. The application continually asks for a given sample until it receives a response from each node. If it gets a sample out of order, that sample is simply discarded.

The next important question to answer is where is all of this data being stored? There are 3 files that are created with each set of data that is gathered. The path and filename to which the data should be written is user-entered in the text field labeled *Data File Location:* and defaults to C:\MIT\sportSensemble\data. The data files that are created with default path are:

- data.txt - the main data file that contains sample numbers, node numbers and actual sensor data from the analog sensors
- data.LOWG.txt - a secondary data file that contains low-g accelerometer data and compass data.
- data.CAMERA.txt - external camera synchronization data. (See Appendix C)

Each of the files are comma separated, for easy parsing, and have the following format.

Main data file:

- Column 1 - Basestation TS upon receipt (the basestation keeps its own local timestamp to avoid duplicate packet confusion)
- Column 2 - Node ID - Source node ID

- Column 3 - Sample Number - index of the sample in flash memory
- Column 4 - High-G accelerometer X value
- Column 5 - High-G accelerometer Y value
- Column 6 - High-G accelerometer Z value
- Column 7 - Gyroscope pitch value
- Column 8 - Gyroscope roll value
- Column 9 - Gyroscope yaw value

data.LOWG file (low-g accelerometer sample):

- Column 1 - 0 to indicate low-g data
- Column 2 - Node ID - Source node ID
- Column 3 - Sample Number - index of sample in flash memory
- Column 4 - Low-G accelerometer X value
- Column 5 - Low-G accelerometer Y value
- Column 6 - Low-G accelerometer Z value

data.LOWG file (compass sample):

- Column 1 - 0 to indicate low-g data
- Column 2 - Node ID - Source node ID
- Column 3 - Sample Number - index of sample in flash memory
- Column 4 - Compass heading
- Column 5 - Compass pitch

- Column 6 - Compass roll

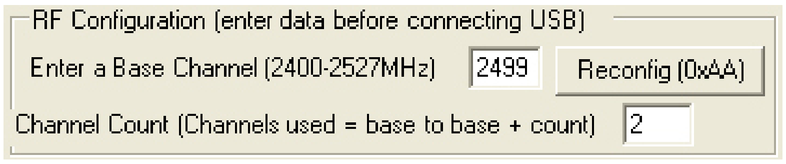
The last two functions of the application that are relevant are the *DisConnect USB* and *Send Idle Cmd*— buttons. *DisConnect USB* is self explanatory and *Send Idle Cmd* is a sort of escape clause for when you want everything to stop what it is doing and set itself to a ”stable” state.

3.3 RF Protocol

Sensemble [12] was designed, built and tested in an environment that was relatively free of RF traffic on the channel that it used. Once it was put into an environment in which it had to share the 2.4GHz ISM Radio band[59] with other things (ubiquitous 802.11a/b/g and bluetooth) its RF protocol quickly fell apart. The reason for this is that Sensemble used a single 1MHz channel that was hardcoded into the basestation and nodes. If anything else was using this channel, especially 802.11 wireless LANs, packet loss would instantly be experienced with losses ranging from 40-95%. In addition, large RF dropout when nodes were occluded by the body were very problematic on stage (where there were no nearby reflective surfaces to bounce occluded signals back to the basestation). This was clearly unacceptable and needed to be addressed.

The best approach for dealing with this is to exploit channel diversity in order to not collide with other devices using this channel. An even better approach is to use channel diversity that is user-tunable to the local RF environment. This is the approach that was taken in sportSemble, and a simple hopping algorithm was implemented that worked very effectively. Additionally, it was possible to reconfigure the network on the fly.

The PC Host Application screen (Figure 3-11) has an area dedicated to configuring and reconfiguring the RF Channels the network would use (Figure 3-12). The protocol takes a *base channel* and a *channel count* as inputs and simply hops channels (incrementing by 1 after every received packet) until it reaches channel *base channel + channel count*. It then starts over again at the *base channel*. This is a trivial algorithm that worked very


RF Configuration (enter data before connecting USB)

Enter a Base Channel (2400-2527MHz) 2499 Reconfig (0xAA)

Channel Count (Channels used = base to base + count) 2

Figure 3-12: RF configuration GUI

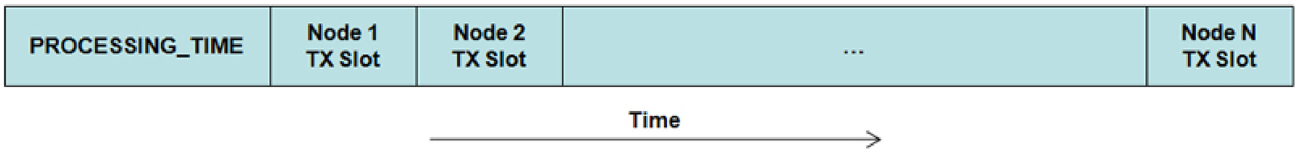


Figure 3-13: RF Protocol Slotting

well in testing, but could easily be expanded to hop in certain sequences for possibly better performance. The mechanism for distributing the configuration is a *RF_CONFIG_CMD* command(Figure 3-14), as described in Section 3.2 and Section 3.1.1, with a configuration passed to *SI_Write()* and then stuffed into the packet sent by the basestation. Upon receipt, the basestation reconfigures itself and floods the packets several times on the current channel that all nodes are on. Once the nodes received this packet, they also reconfigure and begin using the specified channels. In order to test the Quality of Service(QoS) that the new protocol provided, packet loss was used as a metric. Each packet had 4 bytes that were reserved for 2 numbers. The first number is a counter of packets that the basestation had sent. The second is a counter that is kept on the node that kept track of how many packets the node had received. The basestation's number was sent out with each packet and each reply packet contained both the basestation and node numbers. This created an easy way to calculate packet loss and test various base channels and channel widths. In the end, the optimum channel width was 2 channels. The optimal base channel was determined by looking to see what 802.11 channels were being used and choosing a base that avoided them. In the worst-case scenario when all 801.11 channels are in use, the best option is to use base channel 2499Mhz, which is reserved and only used in Japan. This, however, is not fully legitimate and is not advised.

As described above, the protocol hops over 1MHz channels, and it also slotted the time that a node could respond to any packet sent based upon the node's ID Figure 3-13, and be guaranteed that it would not collide with another node. The approach to deciding when to respond was to calculate a hold off time before responding:

$$\mathrm{HOLDOFF} = \mathrm{PROCESSING\_TIME} + (\mathrm{NODE\_ID} * \mathrm{SLOT\_SIZE})$$

SLOT_SIZE is dictated by how long it takes to process a packet and how long it takes to transmit a packet and was determined with some simple math and a bit of trial and error.

As promised in previous sections and chapters, the details of each packet type's structure will be described. The packet length for each packet type is 20 bytes[4], whether or not all 20 are needed. This is because the nRF2401A[49] radio would need to be reconfigured on every node and the basestation every time the packet length changed. This would require massive amounts of bookkeeping and synchronization, and is not a technically feasible approach.

The structure of each packet that is sent by the basestation is very similar. Each packet uses the first byte to specify the command, bytes 2 and 3 are always the *basestation's local timestamp* and the last 2 bytes are always the basestation's transmitted packet count (its use is described in the beginning of this section). The *basestation local timestamp* is a counter that incremented much faster than the rate at which packets are transmitted and was intended to be used for any very fine-grained synchronization, if needed. It has not been needed, and remains as an artifact of over-engineering. The packet that bears mention because it differs from the other is the one that is sent in response to a *NEXT_PACKET* (Figure 3-17) request. Under the normal conditions (as described in Section 3.1.1), the reply is also of type *NEXT_PACKET*. When a requested sample is not in flash, the first byte will be *FLASH_END*. It is important to note that the node received packet count is stuffed into bytes 17 and 18 for QoS purposes, as is also done with the *FLASH_FULL* packet (Figure 3-18).

[4]Sensemble used 16, sportSemble added 4 in order to do QoS calculations

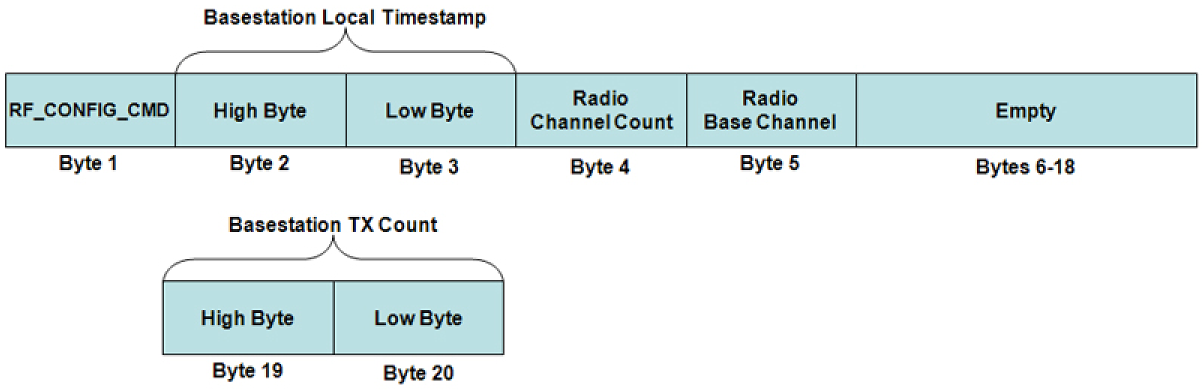


Figure 3-14: RF_CONFIG_CMD Packet

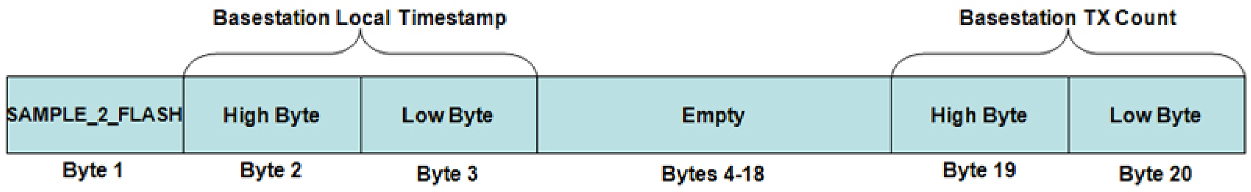


Figure 3-15: SAMPLE_2_FLASH Packet

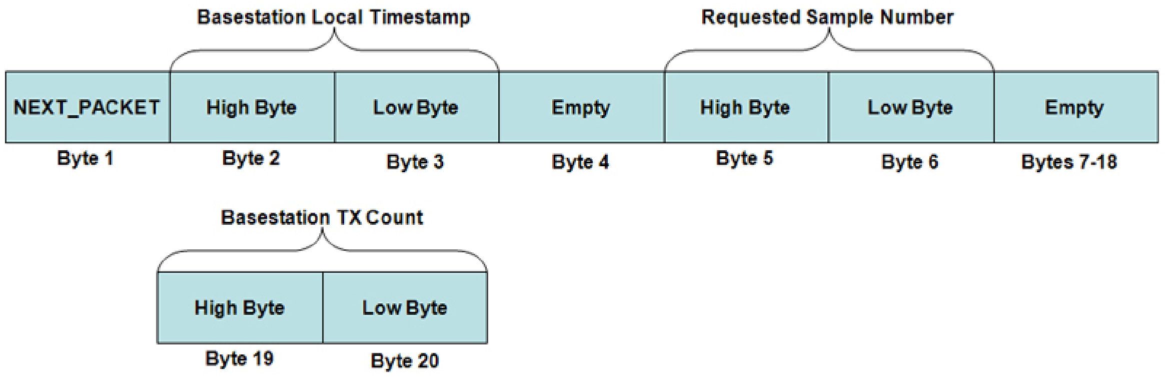


Figure 3-16: NEXT_PACKET Packet

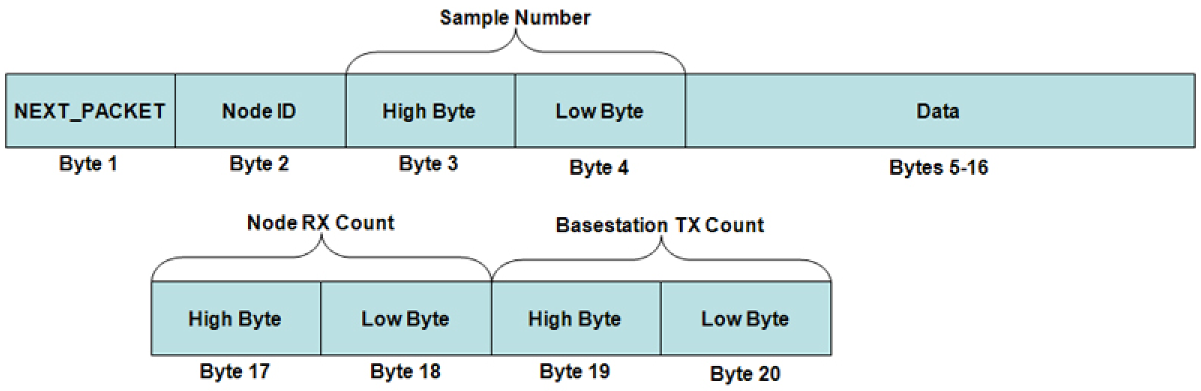


Figure 3-17: Node Response to NEXT_PACKET

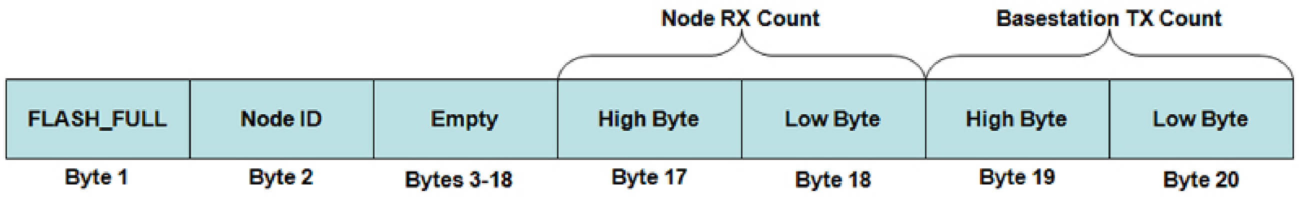


Figure 3-18: Node Response to SAMPLE_2_FLASH

Chapter 4

Experiments

While sportSemble is a very flexible system that is capable of being used at any location on the body and with any sport, it was designed and built for the purposes of being used on baseball pitchers and batters. This design effort culminated in a single 3-day experiment that occurred at the Boston Red Sox Spring Training in Fort Myers, Florida, in which the nodes were attached to several players and data was gathered.

4.1 Initial Testing

Before detailing the spring training experiment, the author would like to thank Mike Vasquez from the MIT baseball team who took some hours out of his life to be the first pitcher to test sportSemble. The first experiments with Mike flushed out many faults in the software infrastructure. Once these were fixed, we focused on attaching the nodes to the body in subsequent trials. During these, it was common for a battery or node to come flying off of Mike's body during the course of a pitch. Thanks to these first trials and Mike's patience, a reliable methodology for attaching the nodes was developed, and the system was tested end-to-end.

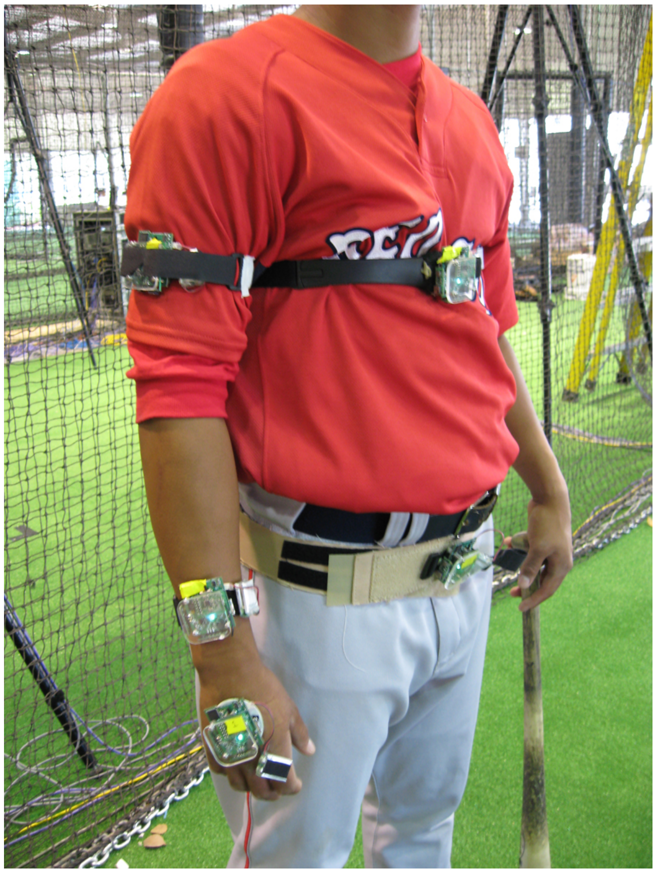


Figure 4-1: sportSemble Pitcher Configuration

4.2 Red Sox Spring Training

The Spring Training experiments took place in Fort Myers, Florida at the Red Sox Training facilities in the Minor League batting cages. They were done in conjunction with XOS Technologies, who were demonstrating their high-speed sportMotion camera system, with which sportSemble was synchronized. The experiments were done over the course of 3 days and involved 8 pitchers and 5 batters, all throwing and/or swinging at the professional level.

The configuration of the nodes on the body for a pitcher is depicted in Figure 4-1. The node numbering scheme for a pitcher is as depicted in Figure 4-3 [1].

Node 1 was originally to be placed inside of a pouch sewn into the top of a batting glove

[1]Node 5 can be on a batter's bat or around a batter's waist - for a pitcher it is always around the waist

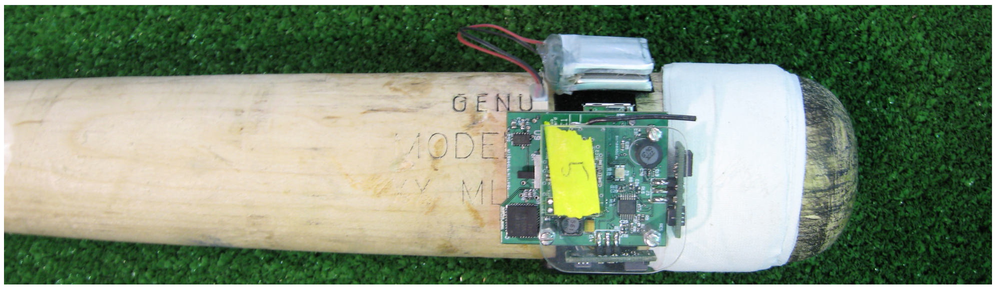


Figure 4-2: sportSemble Instrumented Bat, before applying take to keep it in place

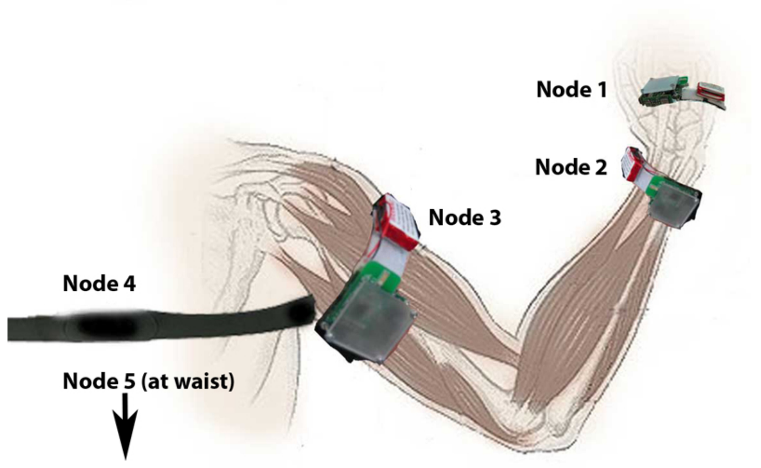


Figure 4-3: Node body placement. Note, the waist node is around the waist or on the bat.

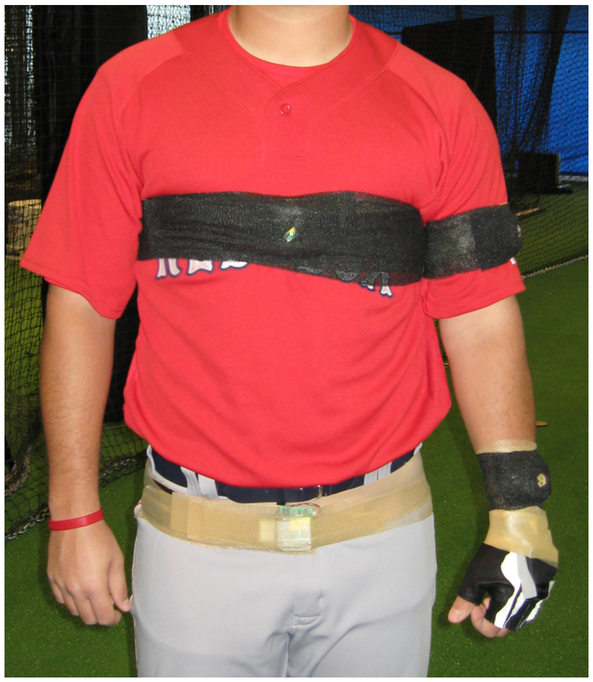


Figure 4-4: sportSemble Nodes Ready for Action

with the fingers cut off(visible in figure 4-4). However this was deemed uncomfortable by the pitchers, and a second option was used. This option was to place sticky velcro on the bottom of the node, and then sticky the mating sticky velcro to a physical therapy stimulation electrode[2]. Nodes 2 and 3 were attached to the player using velcro straps. Node 4 was attached with the same sticky velcro to a heart rate monitor chest strap. Node 5 was attached to a girdle that was cut down, and also had sticky velcro applied to it. Once this was done, every node was wrapped with either sports pre-wrap or some other elastic tape in order to keep them from flying off of the body(figure 4-4).

After being suited up with the sportSemble nodes the protocol for gathering data from a pitcher was as follows:

- Pitches 1 through 10 - fastballs thrown are medium speed
- Pitch 11 - a changeup (slower ball)

[2]The electrodes used were Empi brand 2 inch round StimCare Carbon FM Electrodes

- As sportSemble was downloading data, keep throwing easy pitches to stay warmed up
- No more than 11 pitches to avoid player fatigue
- Players were scheduled every 30 minutes
- Players cannot wait
- No player could be 'kept' for more than 30 minutes, otherwise the next player would be waiting (see previous)

In addition to this protocol, each player had a variety of physical and anatomical data gathered before starting any activity. This data was:

- Name
- Age
- Height
- Weight
- Player Number
- Bats Left or Right
- Throws Left or Right
- Whether or not they had experienced any previous shoulder pain
- Whether or not they had had previous shoulder surgery

The anatomical data (other then height and weight) was a series of measurements of the player's body and measurements of node positions on the body, taken with a measuring tape. The measurements are first presented in medical terms then translated to lay English below:

- Radiocarpal Joint to Node - Wrist Joint to Node on Hand

- Lateral Epicondyle to Radial Styloid - Length of Forearm
- Lateral Epicondyle to Node - Elbow to Node on Forearm
- Lateral Acromiom to Lateral Epycondyle - Length of Upper Arm
- Lateral Acromiom to Node - Shoulder Joint to Node
- Sternal Notch to Node
- Vertical Distance Between Chest Node and Waist Node
- Waist Node height Off of The ground

Batters also had some additional data gathered:

- Bat Length
- Distance From Hand Node to Bat Node
- Distance From Bat Tip to Bat Node

Once we arrived in Florida, it was decided to add one more piece of data: as each pitcher pitched and each batter swung, the speed of the ball or bat was measured with a standard baseball radar gun. The gun that used was a Stalker Sport 24.15GHz Digital Sports Radar (Serial # SS79607) and is pictured in Figure 4-5. From our experiences, the radar worked very well for recording the pitched ball speed. However, it did not work very well for recording the batspeed and is one of the reasons why batspeed is not commonly measured and used in baseball.

This was the planned experimental procedure. However, the adage that the best laid plans go awry often proved to be correct. Being pressed for time, mistakes were made, things were forgotten, and in general, especially at the beginning, things took longer than expected. For some pitchers, less than 10 pitches of data were gathered due to lack of time. In some instances, nodes broke, probably because of the extreme forces they were put under together with potential circuit board warping from our lashdown strap. This resulted in data sets

Figure 4-5: Stalker Sport Radar Gun

that were sometimes incomplete or that had certain individual sensors outputting bad data. In the worst case, an entire node had to be replaced, which cost time because the node had to be programmed with a specific ID. Broken nodes also cause problems when it comes to calibration (more on this in section 5.4). However, thanks to the patience of players, coaches and support staff, several sets of good data was gathered and brought back to Boston for analysis.

Chapter 5

Calibration

During field experimentation, data was viewed in an ad-hoc manner to ensure that it appeared to make sense and that data was actually being gathered. Before any further analysis of the data could be done and reliable physical quantities obtained, a calibration had to be performed on each node. Calibration ensures the node output accurately reflects input from the real physical world. One obvious lesson learned in this project is that calibration of systems that could possibly be damaged during their use should be done before experimentation begins. As mentioned in section 4, some nodes broke during the process of gathering player data. In some cases, the inertial components had been seen to be damaged when it came time to calibrate, hence other means of using their data had to be found. These will be discussed later in this chapter.

Section 2.4 describes the gyroscopes used in each sportSemble node. Each type of device has a manufacturer's datasheet associated with it [43], which, among other things, specifies how output voltages map proportionally onto angular velocity (degrees/second). In general, these numbers are not highly accurate, as they vary from device to device. Also the sportSemble gyroscopes are amplified to give a cleaner signal, and the gyroscopes are extremely biased to give wider dynamic range [44]. This requires a calibration to be run on each gyro by applying a known physical angular velocity and measuring the devices output. The same

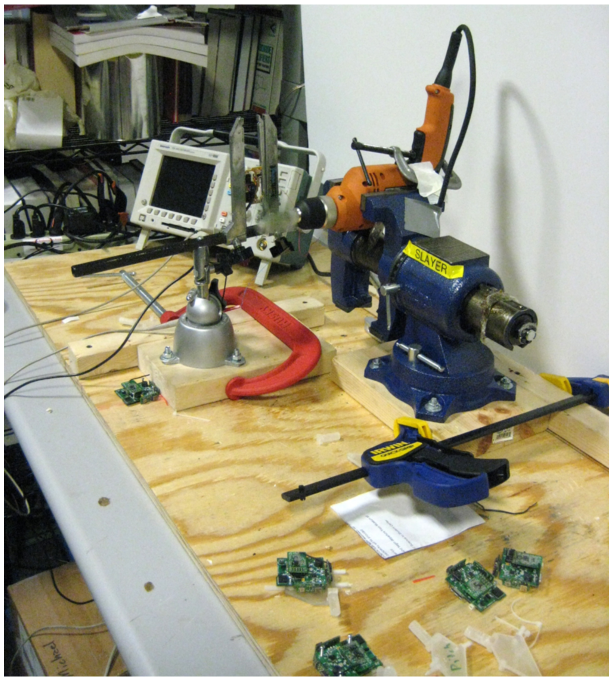


Figure 5-1: Calibration Table for Sensemble Nodes

concept applies to the high-g accelerometers; apply a known acceleration and measure the device output.

Accordingly, the process for calibrating gyroscopes is to spin them at a known rate and gather data, and the process for calibrating accelerometers is to move them at a known acceleration and gather data. This approach was taken by Stacy Morris to calibrate the IMU used in her gait shoe [14], although here we ran at much higher rates. Nonetheless, the rotation for gyroscopes is feasible as the revolutions per minute (RPMs) are easily achievable. The accelerometers are a bit more troublesome, as it is very difficult to linearly accelerate an object at a constant 110Gs for a given period of time. The solution for this is also rotation, and will be described in the following sections.

5.1 Calibration Table and Node Brackets

It was necessary to spin the nodes at a know rate for a given period of time, which was easily enabled by a common variable speed electric drill that could spin at ±2500 RPM.

A calibration table was constructed(Figure 5-1) using the following materials:

- A high quality drill (which was capable of ±2500 RPM)
- A vise, to hold the drill
- An analog (ball based) mouse, to use as a beam breaker to time revolution period
- A Panavise circuit board holding vise, to hold the circuit board of the dissected mouse
- A few clamps, to keep things from vibrating out of position
- Some 2x4's
- A 2'x4' sheet of 3/4" plywood
- Several custom fabricated node brackets
- An oscilloscope

The first problem to be tackled was that of reliably measuring the rotational velocity of whatever it was that would be spun. The solution to this was simple and cheap. A standard analog mouse (which uses IR optical encoders internally) was taken apart and an oscilloscope was connected to the output of one of the IR photodiode receivers(Figure 5-2). What was left of the mouse was plugged into a desktop PC's PS2 port, in order to power it. Once powered, every time the beam between the photodiode and it receiver was broken, there was a voltage drop that could be easily see on the oscilloscope. Using the oscilloscope to to measure the period of between voltage drops indicated the rate at which the beam was being broken. This constituted a poor man's tachometer.

The next issue was to mount the drill in a manner that would keep it secure as it spun at high speeds, and the heavy vise was an ideal candidate. From the beginning, it was obvious

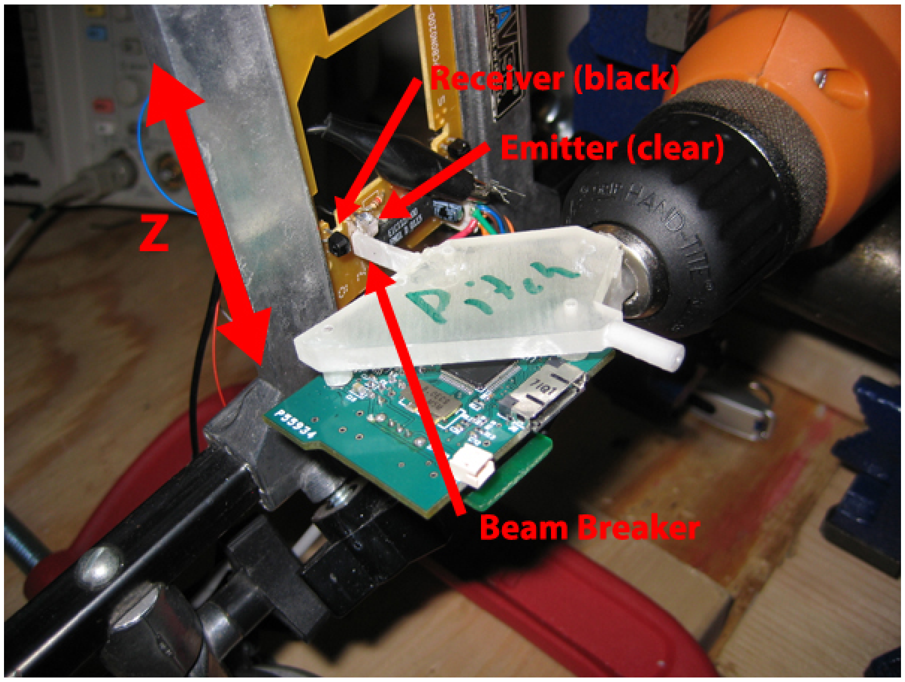


Figure 5-2: Bracket Beam Breaker and IR Photodiode with Z axis denoted.

that custom brackets for the nodes would need to be fabricated to hold the nodes as they rotated. The requirements for these brackets were that for the gyroscopes, the center of the gyroscope be directly at the center of rotation and, for the accelerometers it was necessary that the accelerometer's sensing axis be orthogonal to the center of rotation and, that it was mounted at a specific radius from the center of rotation. These requirements mandated that a custom bracket be designed for gyroscopes and accelerometers. Further, each individual axis of gyroscope and accelerometer would need it own bracket because the sportSemble nodes were not symmetric in any way. Attempts to create an all-in-one bracket were tried, but were not successful. The brackets were designed in Solidworks [60] and several iterations were required; some of the failed brackets are shown in Figure 5-4. Initially, it was desirable to have the brackets CNC machined from aluminum, but this approach was not taken due to the high cost of CNC machining and the long turnaround time on getting parts back from a machine shop. The best option was to use the Media Lab's Invision SI^2 3D printer [61], which prints using photocurable acrylic. While photocurable acrylic is weaker than aluminum, the right design proved that it could get the job done. Also, because the printer

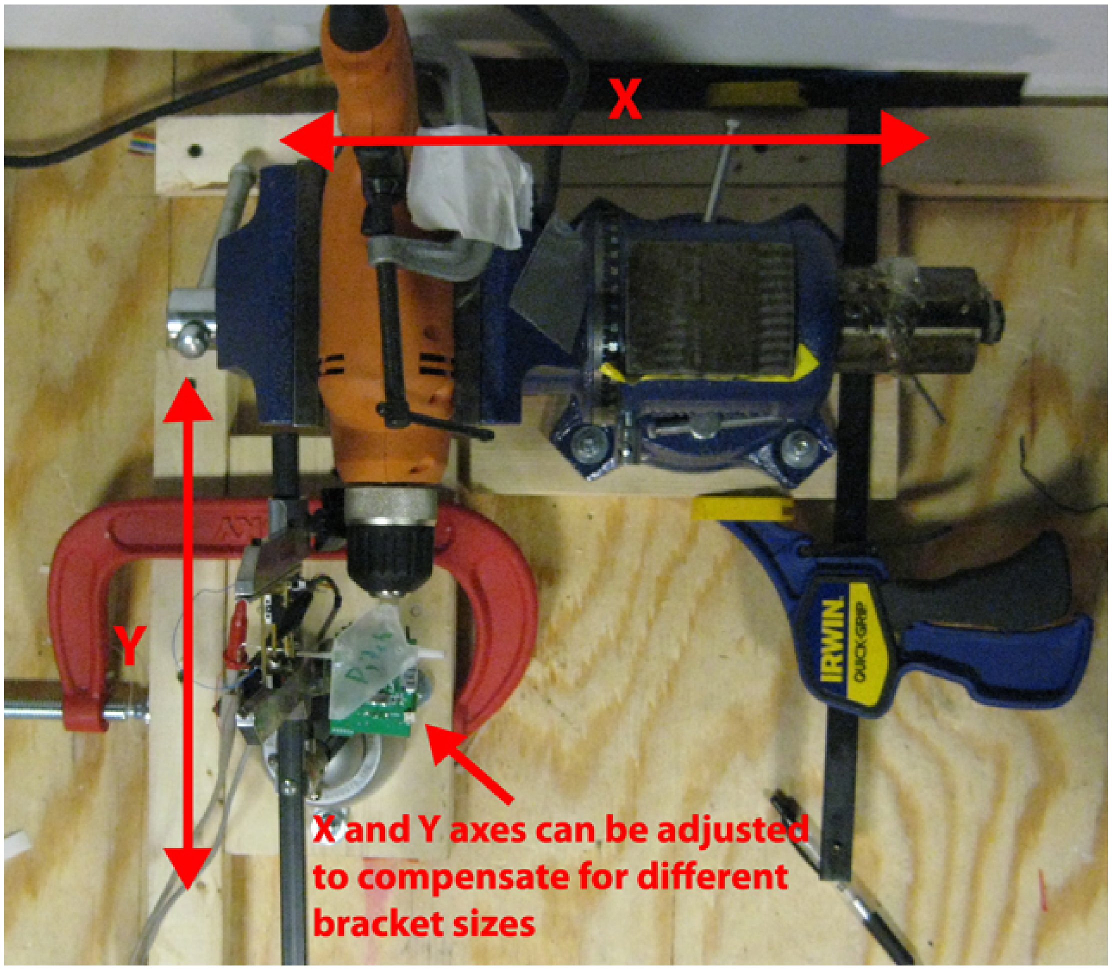

X
Y
X and Y axes can be adjusted to compensate for different bracket sizes
IRWIN
QUICK-GRIP

Figure 5-3: Calibration Table with X and Y axes denoted.

was in-house at the Media Lab, turnaround time for new bracket prototypes was only 24 hours. In addition to brackets being custom for each axis of sensing, each had a beam breaker built into it in order to count RPMs (Figure 5-2).

Due to the irregular shape of the nodes, each bracket had different dimensions and different parts of the node protruding as it rotated. The required that the calibration table have 3 degrees of freedom for bracket mounting, because each bracket had the beam breaker in different locations, and each bracket had different dimensions. With this in mind, the vise and the Panavise were each mounted to wooden blocks that slid along wooden rails screwed to the plywood sheet. Each was secured to the rails with a clamp, in order to

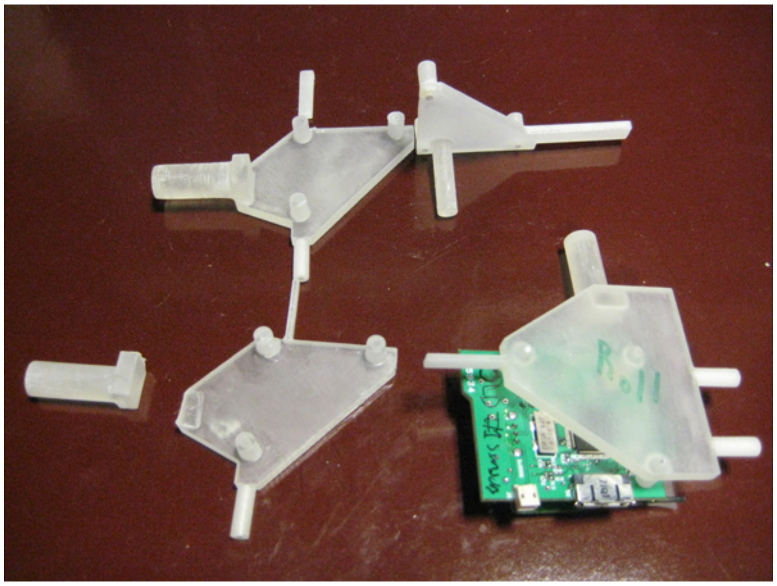


Figure 5-4: Prototype Brackets

keep the assembly from moving (Figure 5-1). This provided an X and Y axis of movement (Figure 5-3); the third axis of freedom was the ability to slide the circuit board from the mouse up and down in the Panavise (visible in Figure 5-2). This constituted the Z axis and was used to precisely adjust the beam breakers location, and also ensured that it was properly splitting the path between the photodiode emitter-receiver pair. In hindsight, the beam breaker could have been mounted on the chuck of the drill, since rotational rate is independent of radius, and the table may have not needed so many variable axes.

5.2 Gyroscope Calibration

Reiterating the previous section; each gyroscope available commercially has published formulas that convert an output voltage to angular velocity. These published formulas are known to be of limited accuracy (with no published statistics on error margins). In addition, any amplification, signal conditioning, and bias change to the device will also have an effect on the output that is not reflected in the datasheet. Based on this, a calibration is necessary to have confidence in the data gathered. Each node has 3 separate gyroscopes on

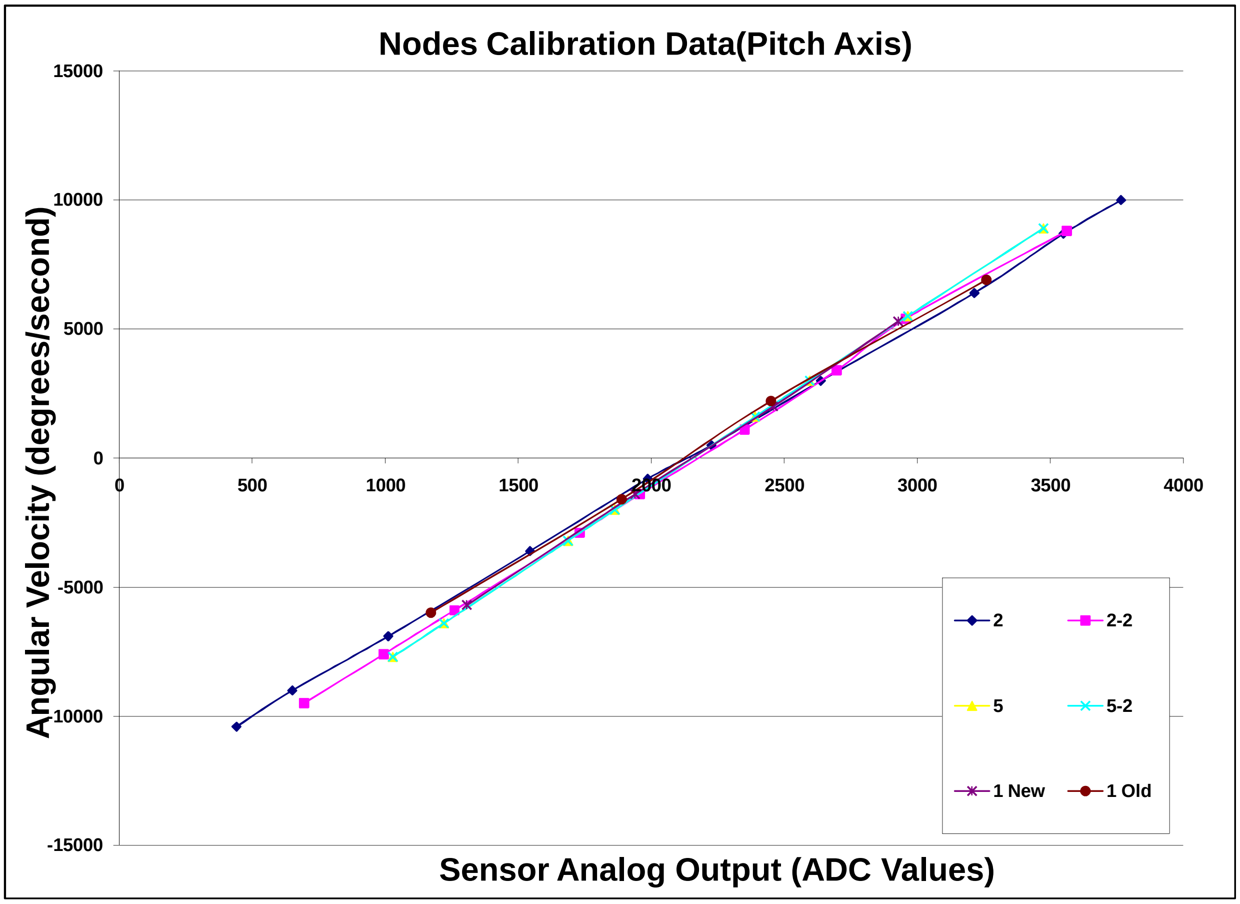
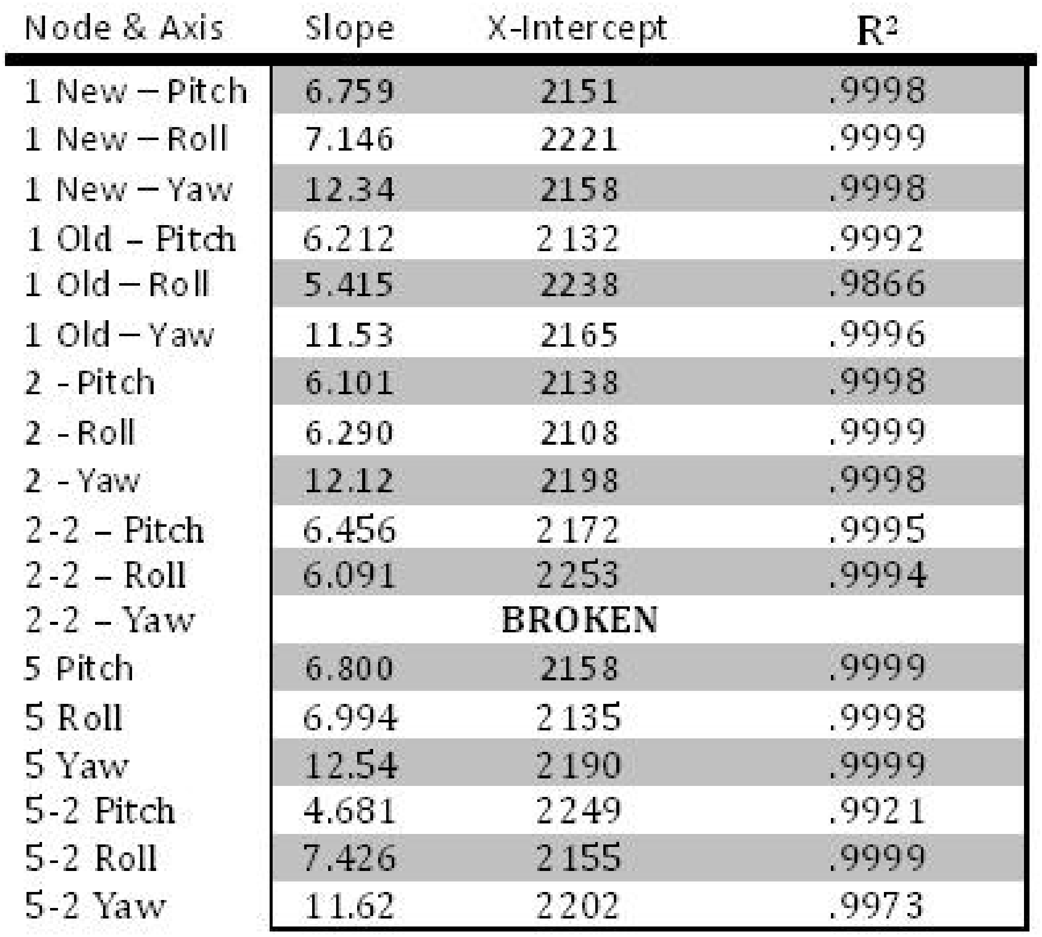


Figure 5-5: Node Calibration Data For All Gyroscopes

orthogonal axes; pitch, roll and yaw, and each required calibration. As mentioned previously, the calibration procedure required that the node be rotated about the center of each gyroscope on each axis at a known controlled rate, and data be gathered. In order to do this, a calibration table needed to be built along with specialized brackets for the nodes. The calibration table is described in depth in section 5.1.

The calibration involved 6 nodes. Each node was rotated at the following rates (mapped to RPMs for the calibration table):

- ±10,000 degrees/second - 1,667 RPM



Node & Axis	Slope	X-Intercept	R^2
1 New – Pitch	6.759	2151	.9998
1 New – Roll	7.146	2221	.9999
1 New – Yaw	12.34	2158	.9998
1 Old – Pitch	6.212	2132	.9992
1 Old – Roll	5.415	2238	.9866
1 Old – Yaw	11.53	2165	.9996
2 - Pitch	6.101	2138	.9998
2 - Roll	6.290	2108	.9999
2 - Yaw	12.12	2198	.9998
2-2 – Pitch	6.456	2172	.9995
2-2 – Roll	6.091	2253	.9994
2-2 – Yaw		**BROKEN**	
5 Pitch	6.800	2158	.9999
5 Roll	6.994	2135	.9998
5 Yaw	12.54	2190	.9999
5-2 Pitch	4.681	2249	.9921
5-2 Roll	7.426	2155	.9999
5-2 Yaw	11.62	2202	.9973

Figure 5-6: Table of Node Calibration Data For 6 Node for Gyroscopes

- ±9,000 degrees/second - 1500 RPM
- ±6,000 degrees/second - 1000 RPM
- ±3,000 degrees/second - 500 RPM
- ±1,000 degrees/second - 167 RPM
- 0 degrees/second - 0 RPM

During the constant rotation, the nodes gathered a 10 second set of data. This data (Figure 5-5) was averaged for each speed and each node's data created a line. This indicated that the gyroscope's output is linear, as expected, and that further gyroscopes only needed 2 or 3 sets of data to be gathered instead of the 11 that were initially needed. These 2 or 3 sets of data would lie along a line with a given slope and offset when graphed, which is exploited to determine the angular velocity of any gyroscope output. There are, however, two caveats. First, each of the lines from a given gyroscope have a different slope and, second, each of the lines have different 0 *degrees/second* intercept (See Figure 5-6). Although the lines in Figure 5-5 look similar, the differences are significant (especially in applications that repeatedly integrate the gyroscope signal to get angular displacement), hence each gyro is calibrated separately.

While Figure 5-5 appears to show very similar slopes and 0 *degrees/second* intercepts is it important to note that if there is an error of *.5 degrees* it is not a major issue in calculating a single *0.001 second* sample. However, if you make a *.5 degree* mistake and sum 1 second of data, you are off by 500 degrees. This turns into a major issue and requires any analysis to use individual gyroscope-specific slopes and 0 intercepts in any calculations.

5.3 High-G Accelerometer Calibration

The high-g accelerometers have the same issues as described above, and also needed calibration. To do this, we spin each accelerometer with its sensing axis orthogonal to the center of rotation, producing centripetal acceleration. Since a rotating calibration table (Section 5.1) had already been built to calibrate the gyroscopes, it was prudent to attempt to use it for the accelerometers also. The question was whether or not the table could match the outputs of the accelerometer (±110Gs), in order to test across the full range.

The formula for determining the desired rotational velocity is:

$$RPM = (\frac{30}{\pi})\sqrt{(\frac{9.8 * G}{r})}$$

where G is the number of desired Gs and r is the radius of the center of the given accelerometer from the center of rotation. This formula is derived using some elementary physics [62] and is based on calculating the formula for the instantaneous acceleration vector for a particle in uniform circular motion.

An attempt was made to use existing gyroscope brackets in order to achieve the desired G forces, but it failed because the rotational velocity necessary exceeded the capabilities of the calibration table. As a result, new brackets had to be designed that would provide a larger radius of rotation. The final radius that was used was *29.5 millimeters*, which provided 110 Gs at 1,825 RPM. The new brackets have not yet been tested, as time did not allow, therefore only calibration data for the X-axis is presented here (Figure 5-7 and Figure 5-8). The Y and Z axes are trivial to calibrate now that a good process is in place to do so.

Indeed, as expected, the accelerometers were also seen to be linear, hence only 2 or 3 sets of data were needed from each accelerometer to determine slope and 0 G intercept. Figure 5-7 shows some low-G departures from the calculated slope, this is attributed to the drill not spinning consistently at very low RPMs.

One mistake that was made, that the author realized as he wrote this document, was the mounting of the drill and brackets in the calibration table for this accelerometer calibration were not parallel to the Earth's surface. Thus, the Earth's gravitational field would be present in the data that was gathered. Considering that 10,000 data points were gathered, over 10 seconds, and averaged in each calibration step, gravity becomes statically insignificant in this case, as positive and negative inclinations cancel. At opposing points in the

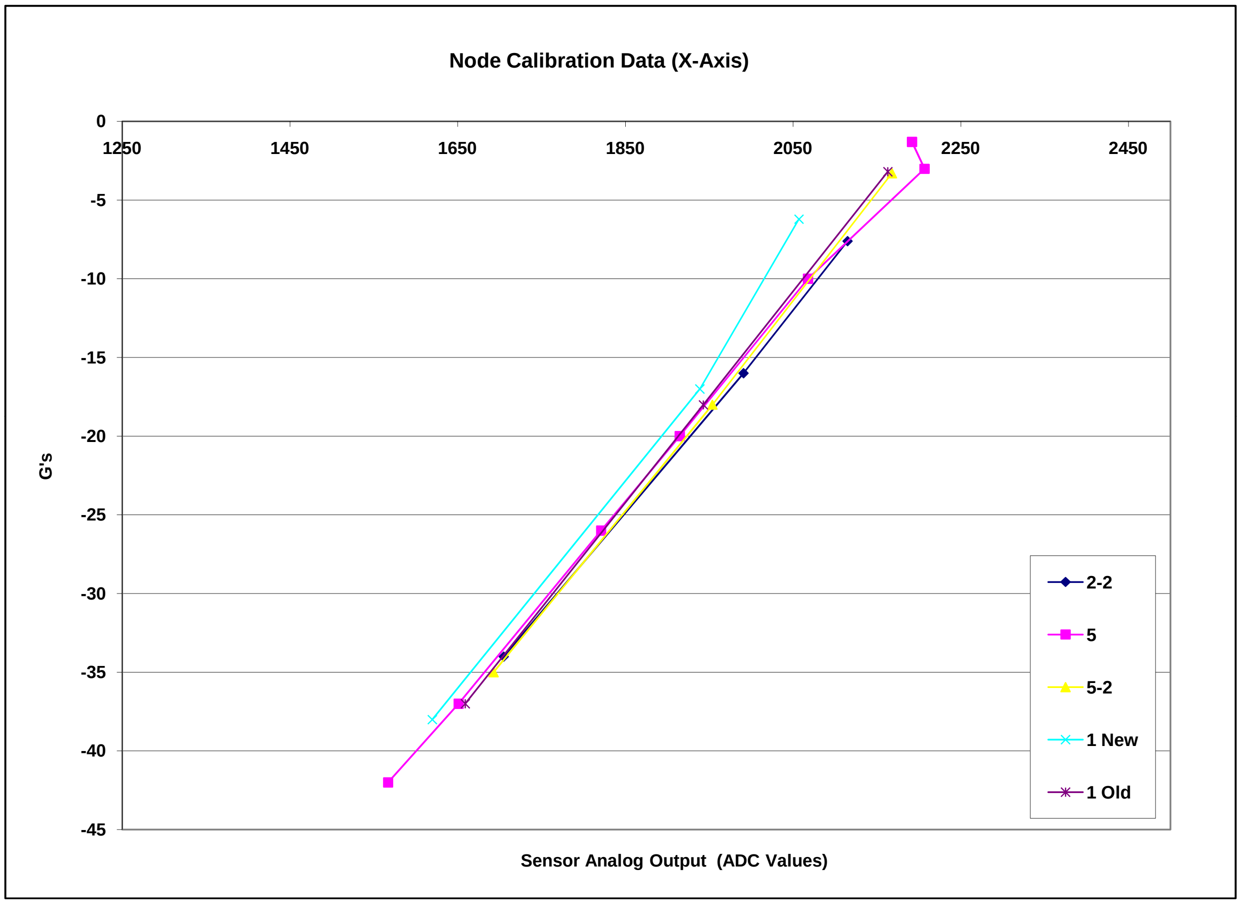
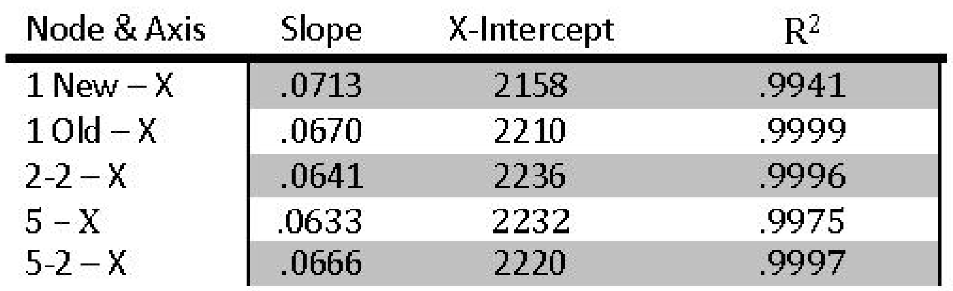


Figure 5-7: Node Calibration Data For X Axis Accelerometers



Node & Axis	Slope	X-Intercept	R^2
1 New – X	.0713	2158	.9941
1 Old – X	.0670	2210	.9999
2-2 – X	.0641	2236	.9996
5 – X	.0633	2232	.9975
5-2 – X	.0666	2220	.9997

Figure 5-8: Table of Node Calibration Data For X Axis Accelerometers

circular rotation, one would have +1G added to its reading and the other point would have -1G added to its reading. Thus, they would cancel each other when averaged.

5.4 Broken Node Calibration

Four nodes had individual sensors broken or were rendered completely unusable, because of mechanical failures, during experimentation. Unfortunately, the nodes were not calibrated before experimentation. This data could not simply be discarded because it is critical. It did not have to be ignored, thanks to the linearity of the gyroscopes and accelerometers. The approach for reliably determining the characteristic output of a given sensor was quite simple. Given that the slopes of the output of Pitch, Roll, Yaw and X, Y, Z sensors were quite similar, the slopes of all calibrated nodes were averaged. This average slope was used as the slope of what is called an *averaged node*, which denotes any node that lacked calibration data. The variance in the slope data was only ± 2 percent. The 0 intercepts (discussed in paragraph 2 of section 5.2) were determined by taking a minimum 2,000 (normally 4,000-10,000) hand-selected data points of the node in question when it was at rest, to determine the gyroscopes output at rest. This resulted in a process of using a generic (average) slope and a gyroscope specific 0 intercept to simulate each broken gyroscope's output.

Chapter 6

Results

Data gathered during experiments (Section 4) that has been married with calibration(Section 5) data produces real world angular velocities and accelerations. It is now of interest to do some analysis and extract features from this data that are relevant to the sports medicine community. The author, not being a sports medicine doctor, turned to medical professionals, and with their guidance determined interesting questions to be asked of this data. These questions and the resulting answers are divided into the two following sections: Batting and Pitching.

Before continuing, it is important to review what potential results are not present in this thesis. Accelerometer data was only minimally used, due to a rigorous calibration not being available at the time result data was being created. This calibration is now complete, and the calibration data is available in Section 5.3, enabling ongoing work to employ the accelerometer information.

There was hope to use a stick figure model to generate joint angles over time. This was not possible for two reasons. First, there was no ability to define starting geometry, from which to begin using gyroscope data to drive a model of joint angles. The starting geometry was supposed to be provided from the XOS Camera system, which was running in parallel with sportSemble during experimentation. A synchronization system was built (See Appendix

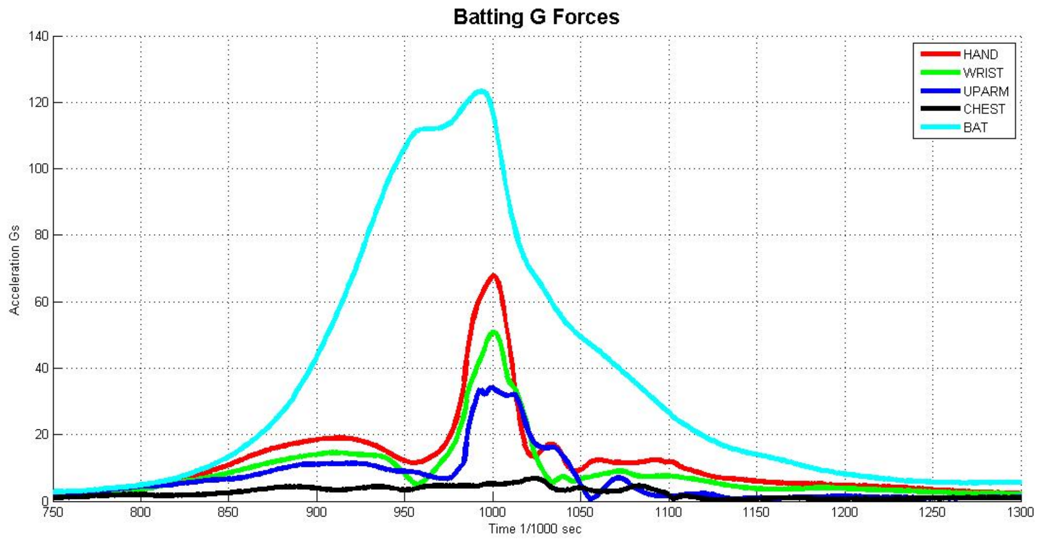


Figure 6-1: Accelerometer vector magnitudes for a single batter's swing, with no ball contact.

C), however the gathered data from the XOS System was not available in reasonable time for the completion of this thesis. The compass data could have also been used, however it was not properly functioning at the time of experimentation. The second reason why joint angle determination was not feasible is that the gyroscope's sensitivity at low angular velocity was poor, and any slower rotations were buried in noise, leading to lost data.

These ideas are now used as part of ongoing analysis and refactoring of the hardware platform for the purposes of future experimentation.

6.1 Batters

From the batting experiments, there are three major questions that lead to many minor ones. Most of them focus around the metric of bat speed, which, in baseball, is thought to be indicative of how far, and where, the ball will go.

The first, considering that it is undesirable to mount a node on the bat (because the massive G forces at impact would most likely destroy the inertial hardware as was seen in a previous

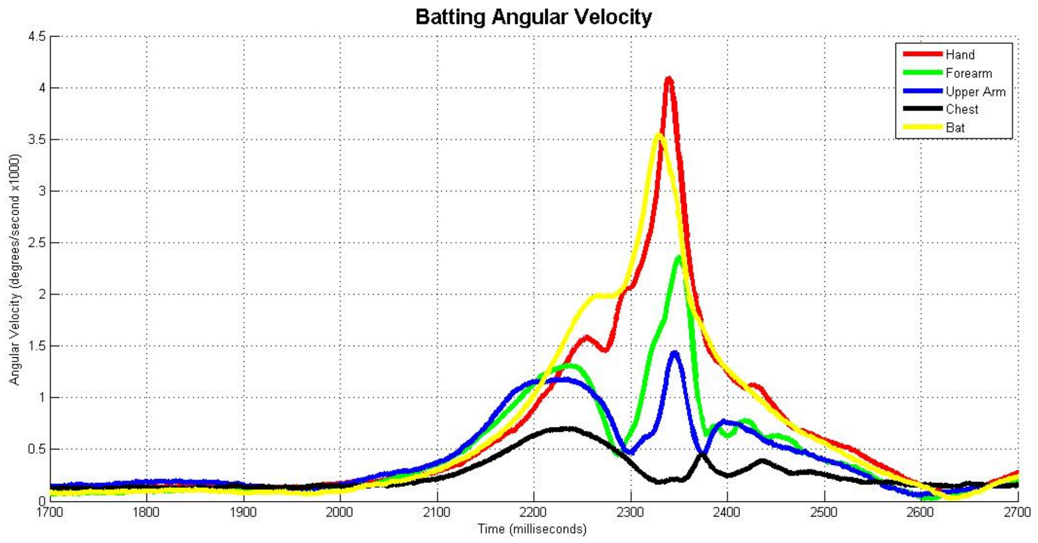


Figure 6-2: Gyroscope vector magnitudes for a single batter's swing, with no ball contact.

Media Lab study [63]), to get bat speed, is can the node be removed from the bat, and can the data from other nodes be used to get bat speed? Is this data consistent per player? Is it consistent enough across all players to allow for generalization to a single generic player? How scattered is the data?

A second question is what are the time differences between the peak velocities of different body segments? A sub-question here is are these differences similar per player? Are they similar across all players? Can they be generalized to distinguish between a good swing and a bad swing? These questions are currently unanswered in the baseball community, but there are indications that these factors affect the end product, i.e., where the ball goes.

The third question, for data that was gathered from a batter hitting a ball off of a tee, is can the point of impact in relation to the peak speeds be determined? If so what does it mean when the ball is struck before the peak is reached? What does it mean if the ball is struck after or at the peak? While there is some speculation on the author's part that this may determine where (left, right, center field) the ball goes, this is a research question posed by his medical collaborators as, until now, no system has been capable of attempting to measure this.

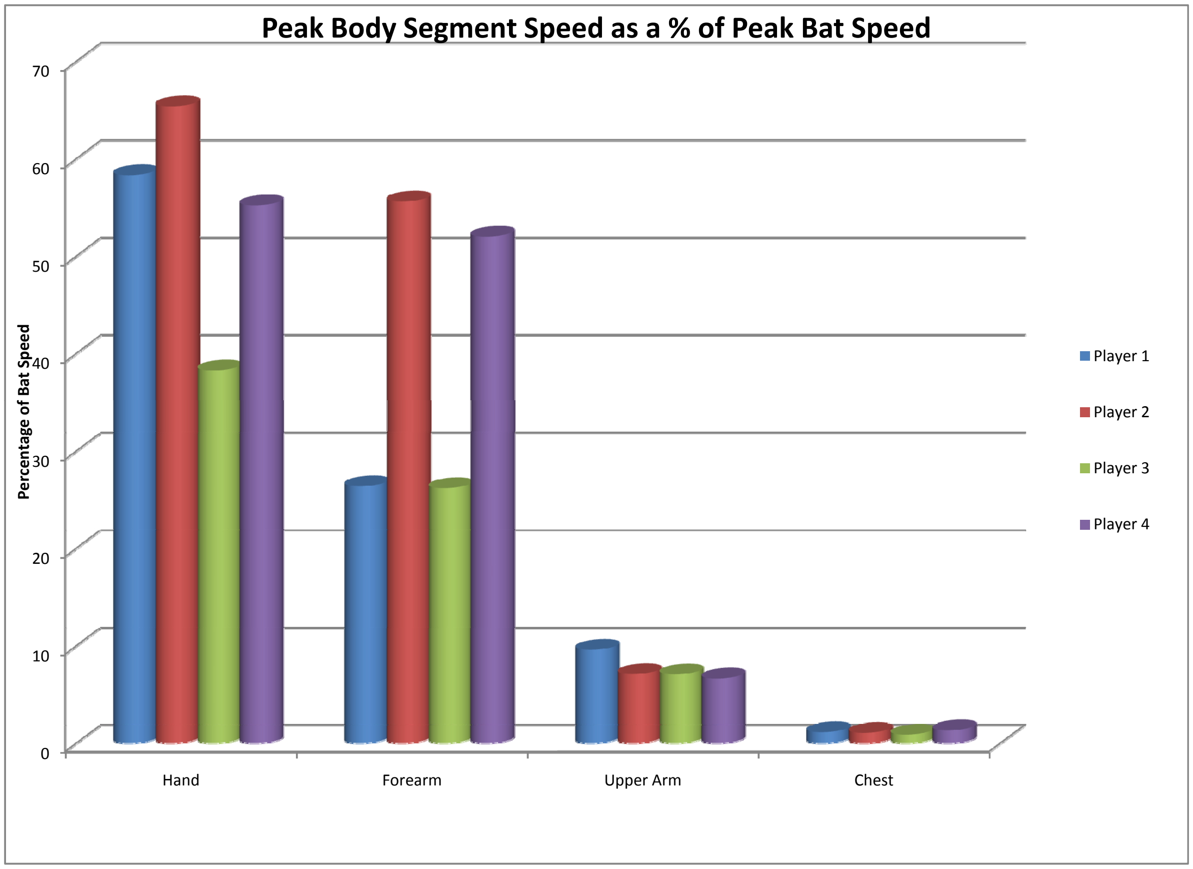


Figure 6-3: Peak Body Segment Speed as a Percentage of Peak Bat Speed Per Player

6.1.1 Bat Speed Estimation

As a refresher we will review the experiment; details are in Section 4. Five batters were instrumented with sportSemble nodes, each took 6 swings with a node on the bat, and the data was recorded. This data was analyzed and bat, hand, forearm, upper arm and chest speeds were extracted. The assumption that was made was that the arms are fully extended at the peak of the swing and all calculations were based on anthropometric data gathered on each player (Section 4.2) and the momentary angular velocity of each body segment. The calculation is done by multiplying the known radius(from the player's body measurement data) at which the node is placed by the gyroscope's angular velocity and converting it to

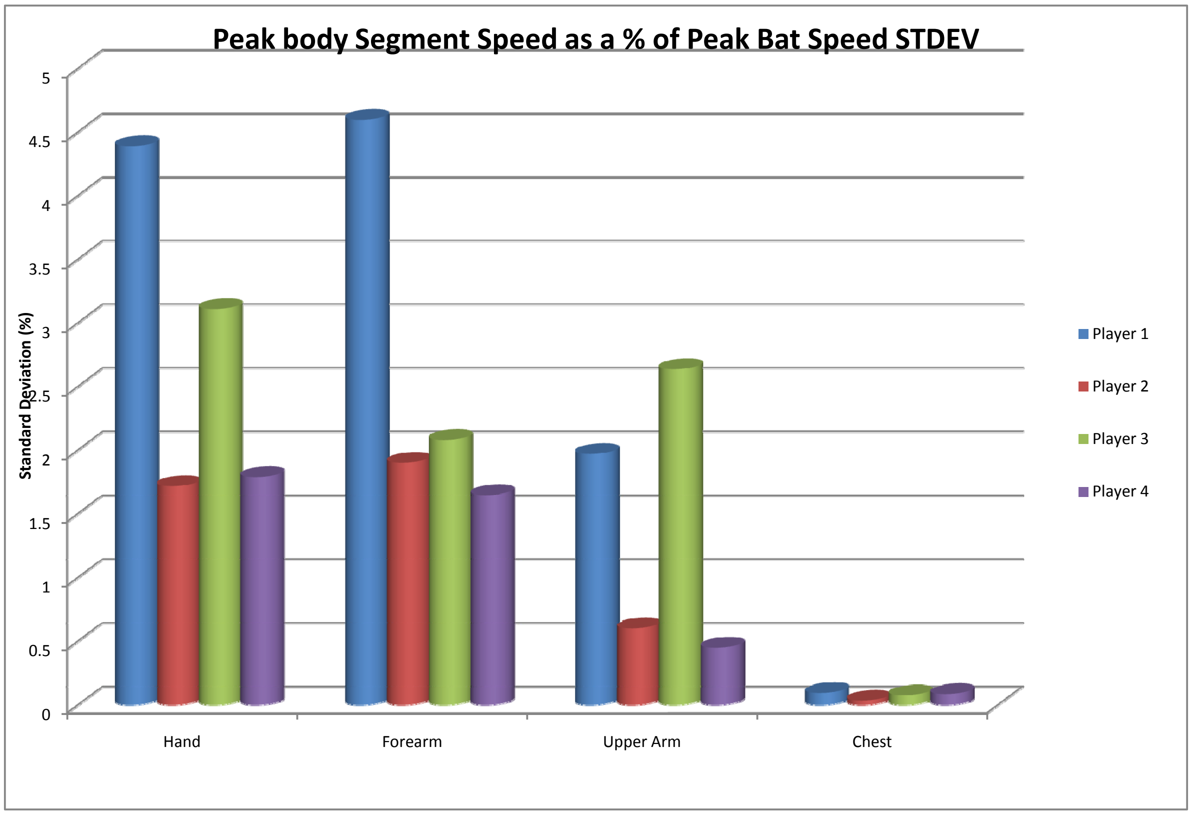


Figure 6-4: Peak Body Segment Speed as a Percentage of Peak Bat Speed Standard Deviations Per Player

miles per hour (MPH).

$$Speed = AngularVelocityinRadians * Radius$$

The author is fully aware that there is a margin of error that this introduces due to players not being fully extended throughout the duration of the swing. In order to avoid this error, joint angle data would need to be available, which unfortunately is not reliably calculated from the data set of these tests. This will be left as a part of future work to be performed.

We will now move onto actually looking at some of our data. An example swing with speeds in MPH is shown in Figure 6-7, where it is interesting to note that while the bat speed is

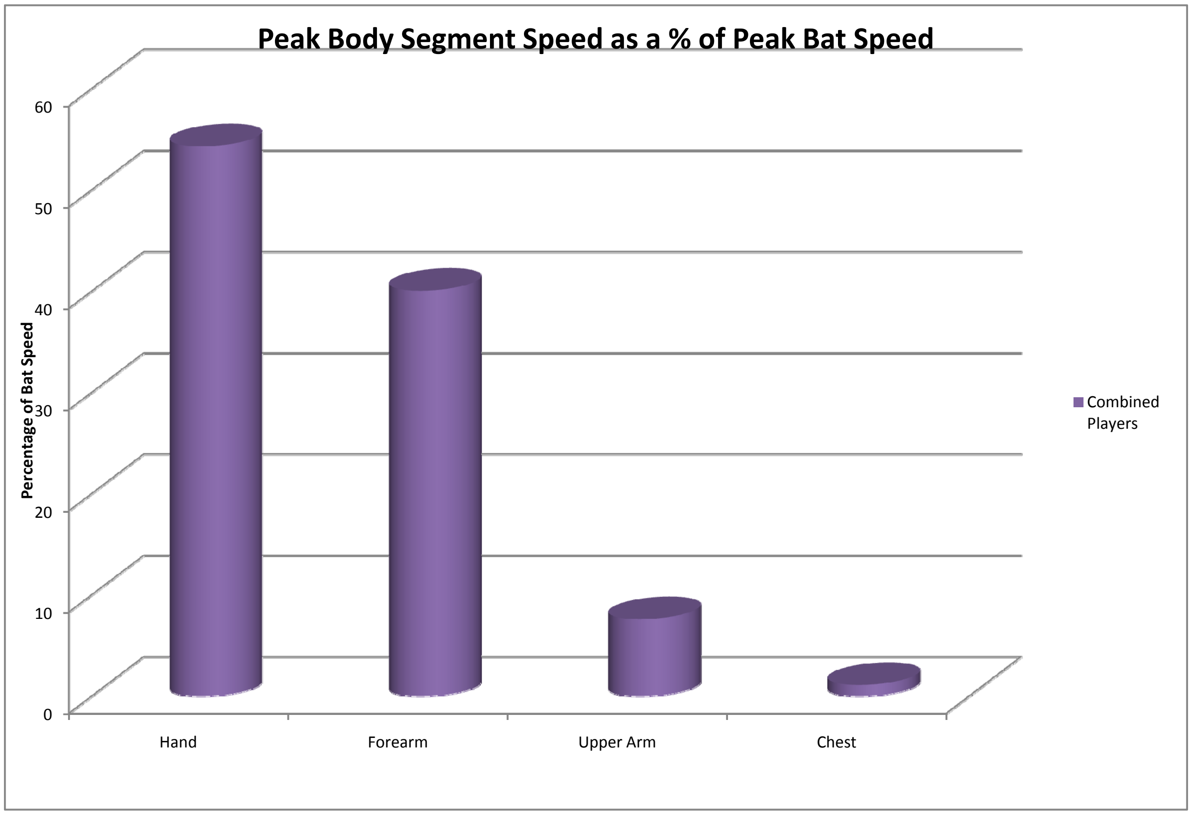


Figure 6-5: Peak Body Segment Speed for All Players as a Percentage of Peak Bat Speed

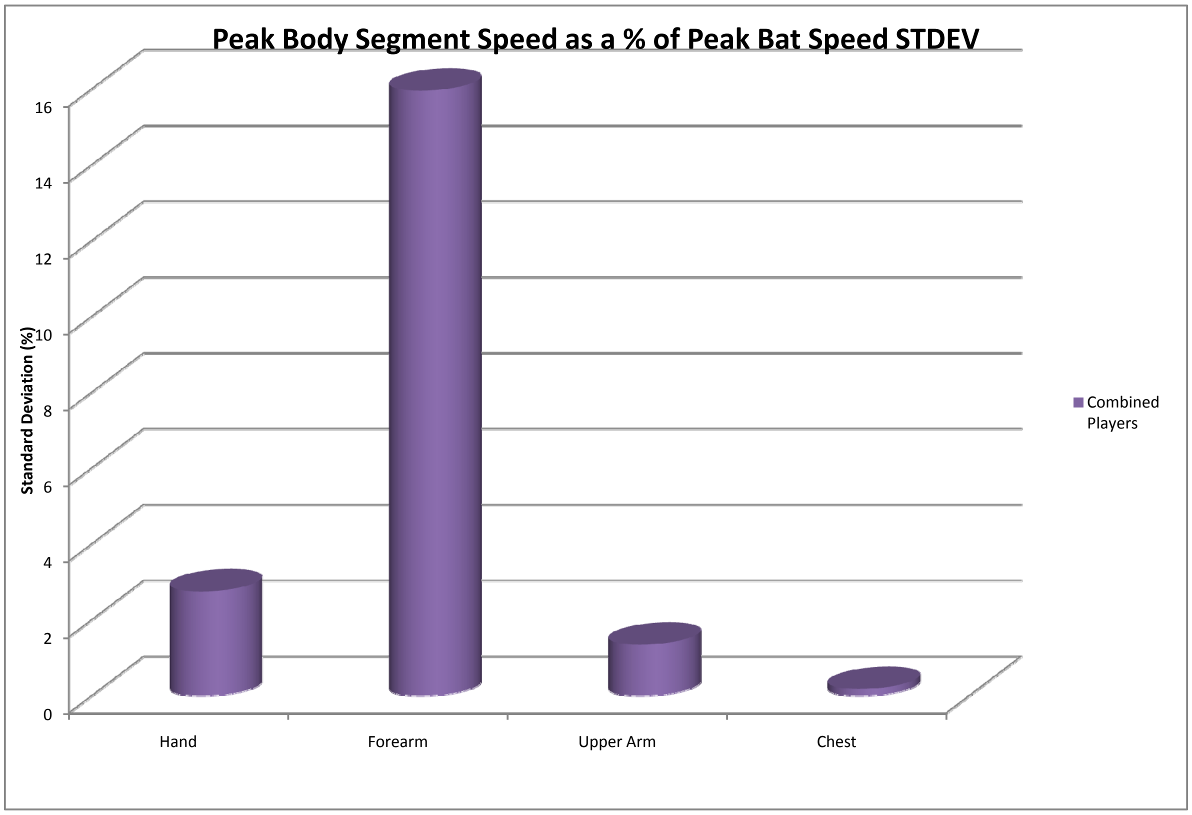


Figure 6-6: Peak Body Segment Speed for All Players as a Percentage of Peak Bat Speed Standard Deviations

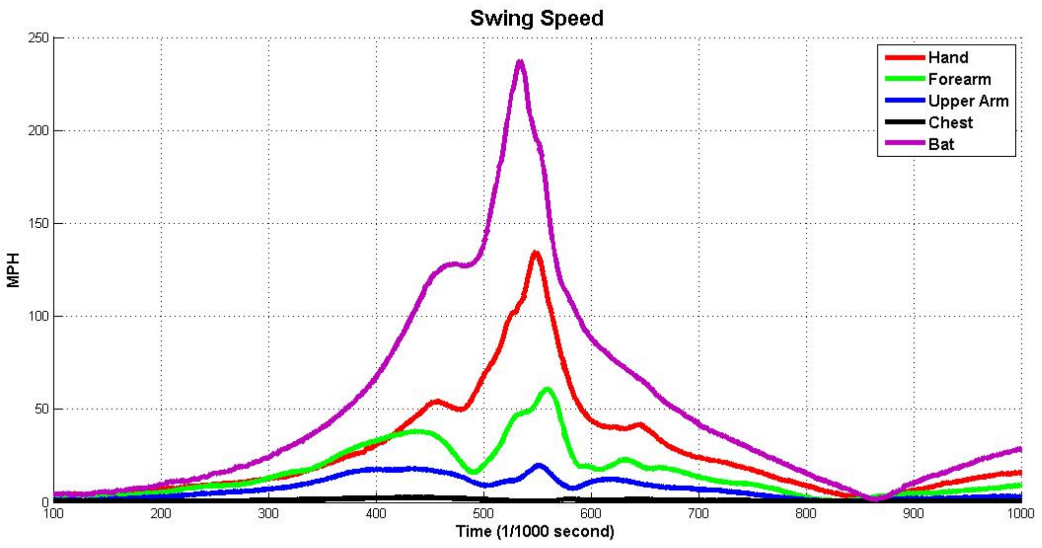


Figure 6-7: A single swing showing different segment speeds over time

much higher then hand speed, the bat's angular velocity is, in fact, lower (Figure 6-2). The idea that is employed in order to do bat speed estimation is to, first, find the peak speeds of each body segment during the swing, and then to look at the ratio between each body segment's speed to this peak bat speed. A graph of this data is presented in Figure 6-3, along with a graph of the standard deviations in Figure 6-4. Before looking at the actual data it is important to note how closely to their average each player's data is clustered. The highest deviation is 5% (forearm of Player 1 Figure 6-4), with the next highest being 3% (Hand, Player 3). This indicates that individual player's swings are very similar in terms of bat and body segment speeds. This is a positive indicator, which brings the author to the natural conclusion that if there were a set of data for each individual player, it would be possible to have a very good estimate($\pm 5\%$) of bat speed for the player without having to instrument the bat. This is a step closer for a system such as this to be used during real game play, as any modifications to player equipment are prohibited by most sports sanctioning bodies. With this in mind, we look at the actual data. The values between players are more variable. For example, hand speed ranges from 38 to 65 percent of bat speed. Considering that bat speeds are around 200MPH this is quite a large range. It is necessary to take the entire available dataset and create similar graphs, as shown in Figure 6-5 and Figure 6-6. Here again, the actual data of the hand being 54% and the forearm 40% of the bat's speed is not as important as how variable this data is. The standard deviations of this data set, at the hand, are below (±2.7%) that of the maximum for individual players and 3 times(±15%) at the forearm. While the data is more variable, it is not wildly so, leading to the conclusion that, with error bounds of ±2.7%, sportSemble is capable of estimating bat speed for a generic player, not requiring any player specific data, without having to instrument the bat. For individual players, sportSemble looks to be capable of doing the same with an accuracy up to ±1.7% with an uninstrumented bat. Please note that for individual players a series of swings would need to be measured in order to have a dataset to estimate from.

6.1.2 Batter Peak Time Differences

The second set of questions about batting that will be addressed has to do with the timing of a players swing, most importantly which body segment peaks occur when (in relation to each other)? The time differences between each individual batter's peaks are shown in Figure 6-8. Before discussing the data, there are two important things to note. First, certain players are missing data for given comparisons. This is due to failure in the gyroscopes on those body segments which resulted in unreliable data. Second, an explanation of the difference between a negative and positive value is necessary. A negative value is one indicating an event occurring before another (I.e. *Chest Peak Minus Bat Peak* being negative indicates that the chest reached its peak speed before the bat) A positive value indicates that the peak occurred after. For example, *Hand Peak Minus Chest Peak* values are positive for all players, which means that the hand reached its peak velocity after the chest.

An examination of the data shows that the the chest always reaches its peak velocity well before (100 ms) the bat does, while all other body segments reached peak velocity after the bat has hit its peak. What this means in terms of swing quality is unknown and bears further investigation and experimentation. In general, the data is very similar per player, except for *Player 4*. This may be indicative of a different batting style. With a larger population of batters, it will be possible to make these distinctions.

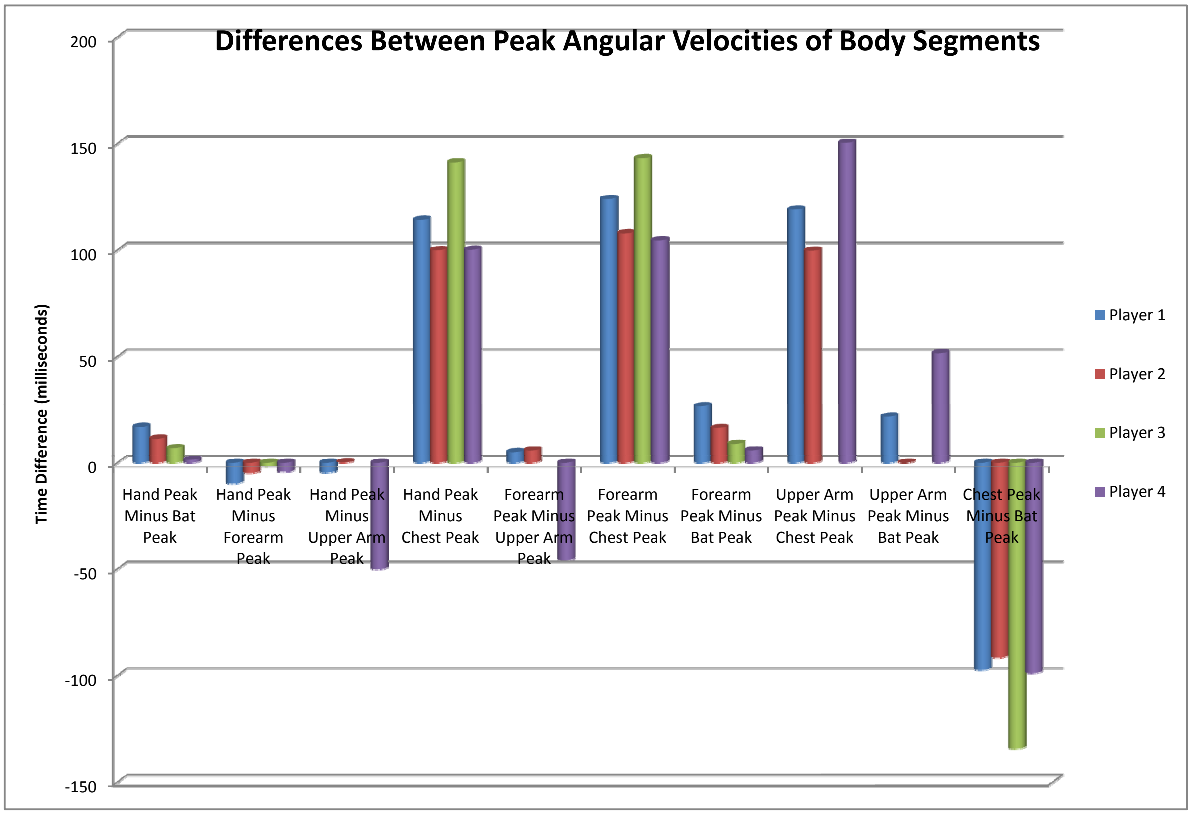


Figure 6-8: Peak Body Segment Time Differences Per Player. NOTE: Negative values indicate peak was reached before, Positive after.

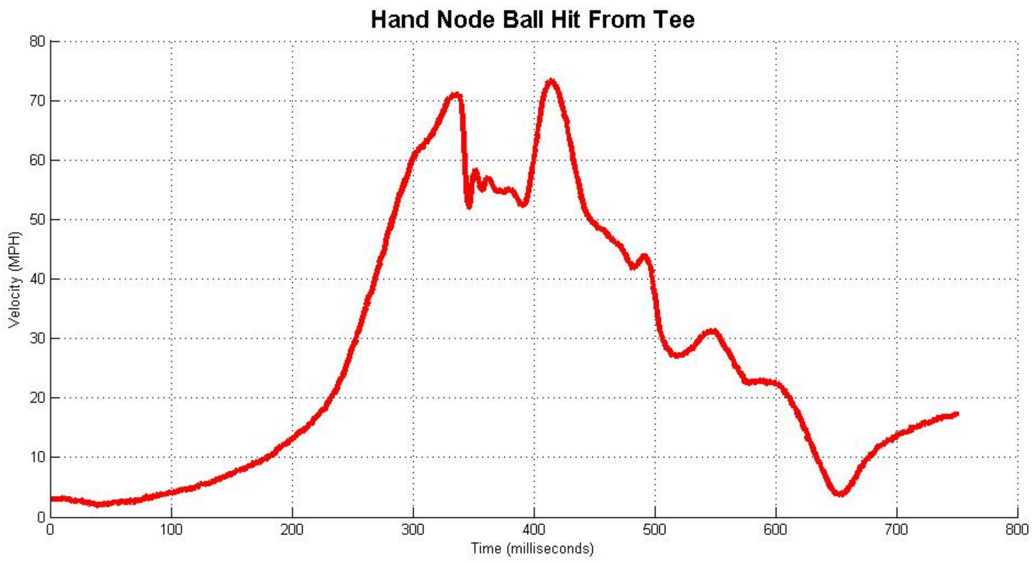


Figure 6-9: Swing with ball impact, no pre or post impact peaks present. (Gyroscope Data)

6.1.3 Ball Impact Detection

The last of the batting questions has to do with being able to detect the point of the swing at which the ball was hit with the bat. The caveat here is that the bat cannot be instrumented with a node because the impact forces would be so large that the MEMS gyroscopes and accelerometers could be destroyed. The first and most important question is whether or not this is possible. To answer the question, a comparison was made between a batter's free swing with no ball impact and swings of the bat when the ball was hit off of a tee. An examination of about 50 swings showed a very clear distinction between the two, as is visible by comparing Figure 6-11, Figure 6-9 and Figure 6-10.

Figure 6-10 shows a swing that appears to have ball impact before peak velocity is reached. The impact is denoted by the deep 'valley' before or, the middle of where the dominant peak should be. This in comparison with the free swing (Figure 6-11). Please note that the free swing also has a small 'valley', as most free swings do. However, these valleys are not nearly as pronounced as the ones in the swings with impact. Such significant 'valleys' are not present in any free swing data, but are present in all tee hit data. Some unanswered

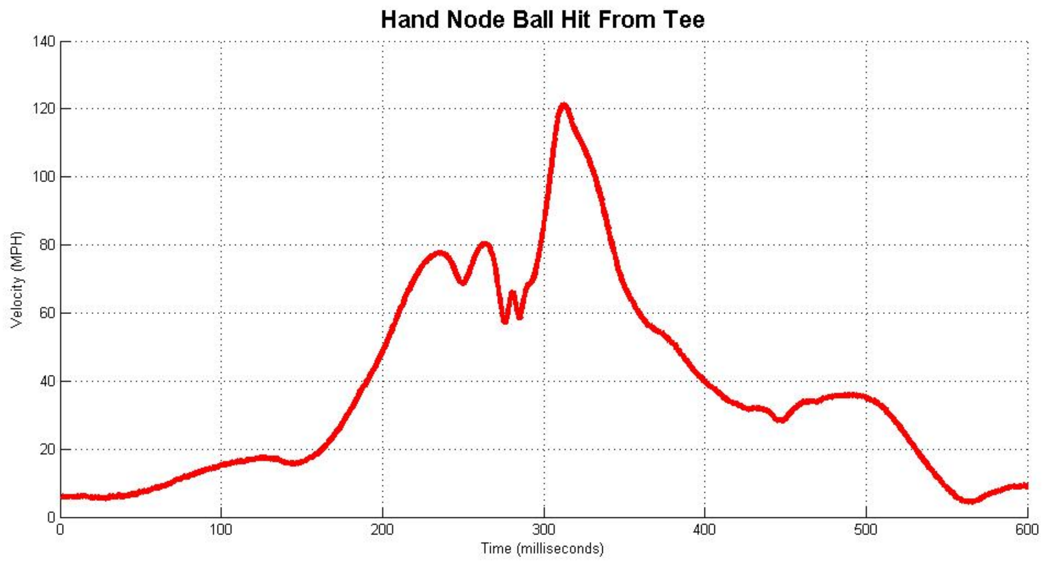


Figure 6-10: Swing with ball impact. The impact appears to occur before the peak bat speed is reached. (Gyroscope Data)

questions remain regarding why the 'valley' in Figure 6-9 is so wide, but the answer to that is out of the scope of this thesis.

An additional piece of validation that was performed was to compare the same impact swings gyroscope data with some uncalibrated accelerometer data. The purpose here was to see if there existed similar 'valleys' in the accelerometer data. Figure 6-9 and Figure 6-12 are the same hit of a ball off of a tee, one set of data being from gyroscopes and the other from accelerometers, respectively. Figure 6-11 and Figure 6-13 are free swings, again one set of data if from gyroscopes and one from accelerometers, respectively. Examining the swing with impact a similar valley is present in the accelerometer data as in the gyroscope data. As the hit looks to be much sharper in the accelerometer data, it's probably a more reliable means of determining the impact time. The repeated peaks in Figure 6-12 are probably due to the dynamic structure of the impact and its effect on sensor mounting; more investigation is necessary. Note also the large qualitative differences in the overall contour of the swings with (Figure 6-10, Figure 6-12) and without (Figure 6-11, Figure 6-13) ball hits. This indicates, perhaps, that the player dynamically adjusts their swing to hit the ball.

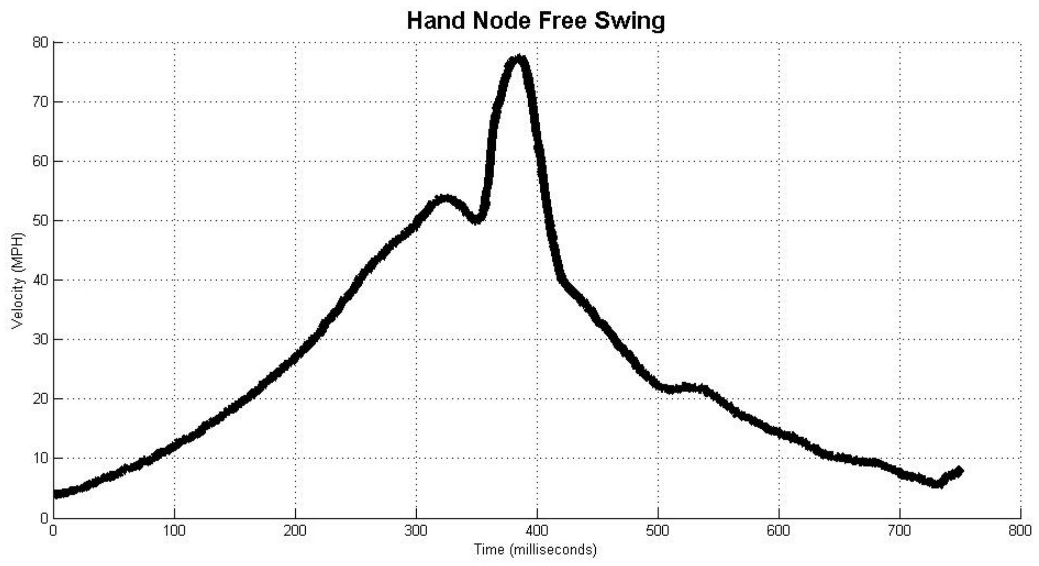


Figure 6-11: Free swing with no ball impact. (Gyroscope Data)

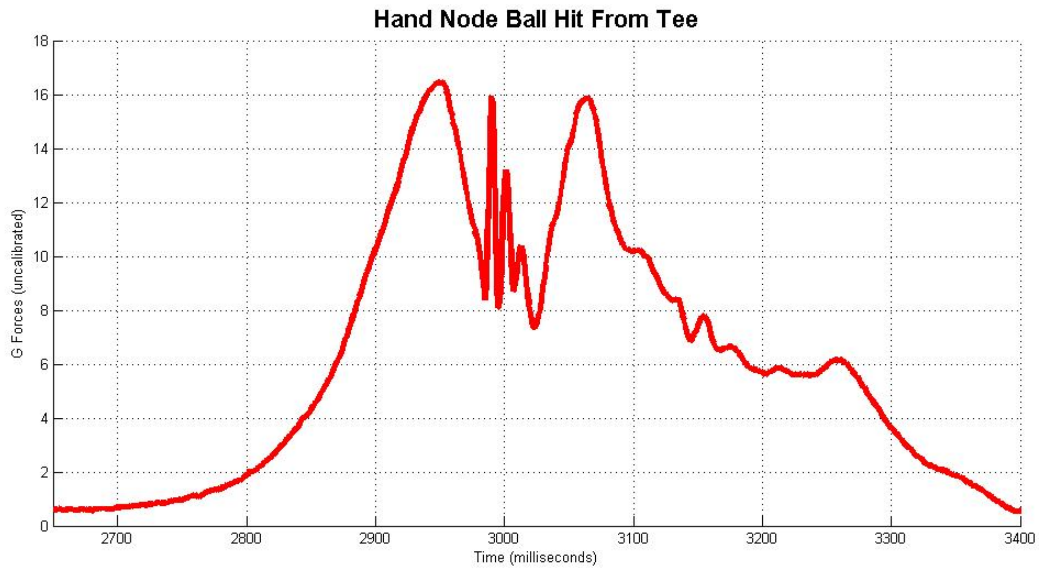


Figure 6-12: Swing with ball impact, no pre or post impact peaks present. (Uncalibrated Accelerometer Data)

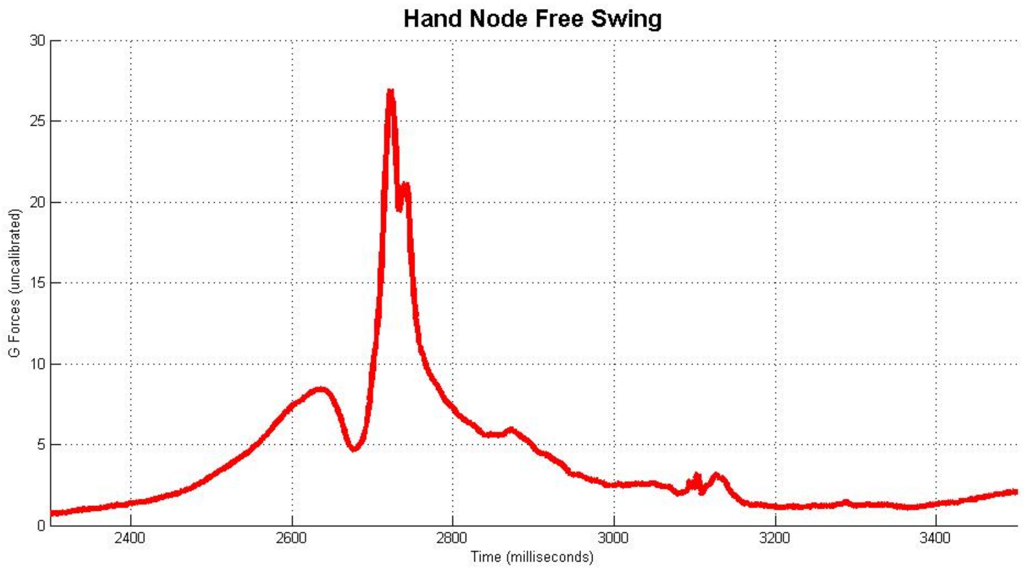


Figure 6-13: Swing with no ball impact. (Uncalibrated Accelerometer Data)

The ability to detect this interesting because it has not been possible with any other system[1]. Having this piece of data opens the doors for studying how the point of impact (pre-peak, at peak, post-peak) effects where the ball goes and how far it travels. In order to do this, further experimentation is necessary because initial experiments were held in batting cages and it was impossible to determine where the ball flew and how far.

6.2 Pitchers

Discussions with our physician collaborators on what to look for in pitcher data were very far ranging, and some triage was made accommodate the timeframe of this thesis. The highest priority went to calculating shoulder internal rotation, which is defined as the rotation along the axis of the humerus during a pitch. As with batting, it is of interest to determine the consistency between pitches and between pitchers and to compare to known velocities published in the literature.

[1] according to discussions with medical collaborators

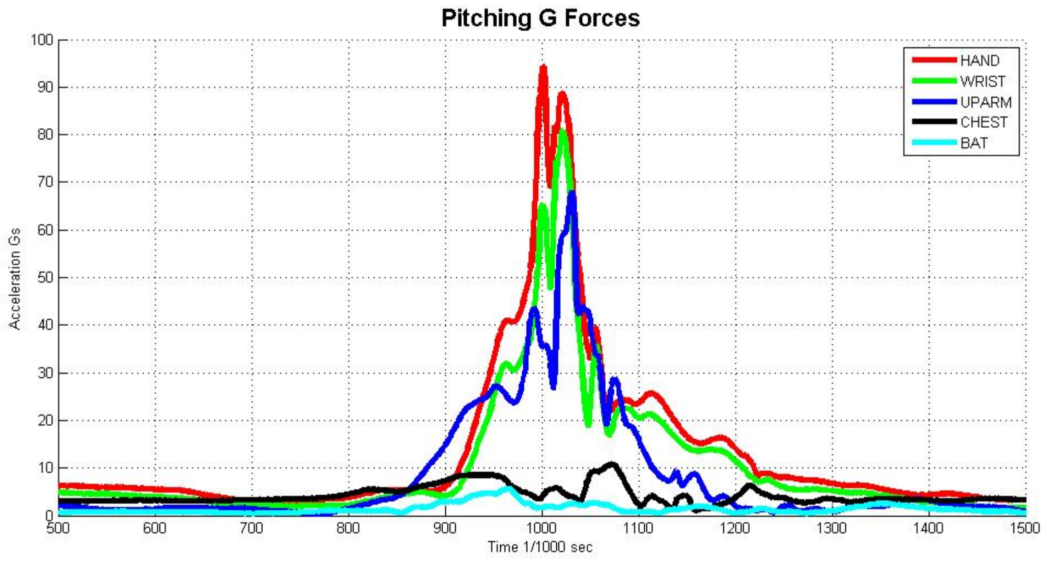


Figure 6-14: Accelerometer vector magnitudes for a single pitch.

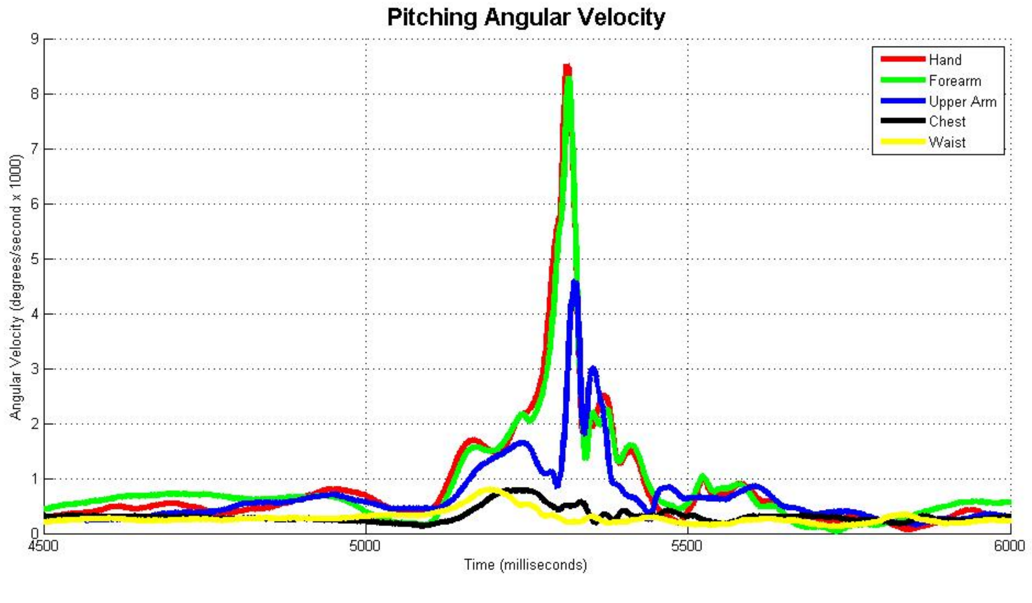


Figure 6-15: Gyroscope vector magnitudes for a single pitch.

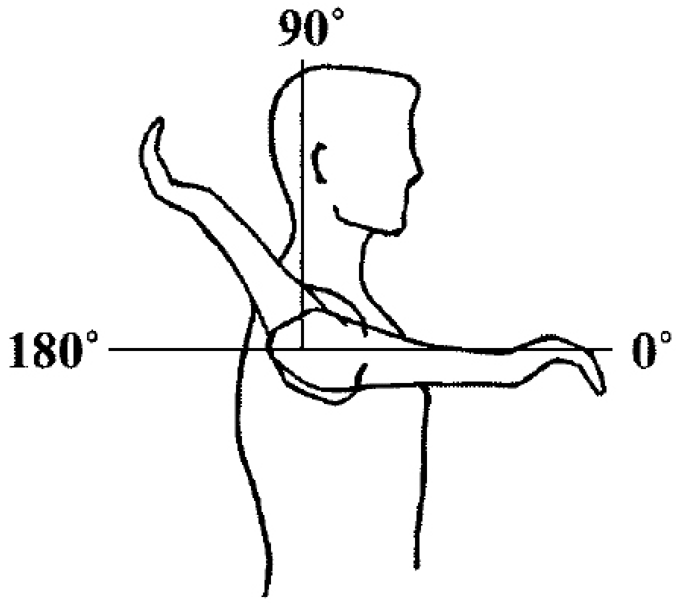


Figure 6-16: Shoulder Internal Rotation as defined by Werner et al.[1]

6.2.1 Shoulder Internal Rotation

Shoulder internal rotation is the rate at which the humerus bone rotates along an axis parallel to it during a pitch, and is a commonly used metric in sports medicine. Peak shoulder internal rotation occurs during the acceleration phase of the pitch. During the typically 30ms of this phase, shoulder internal rotation has been calculated between 6,400 to 10,000 degrees/second [5, 1, 64]. Camera-based systems attempt to determine this metric however the error bounds that they produce render the data not very useful. For example, published numbers from a study [8] give peak shoulder internal rotation as *7,536±5634* degrees/second. In other words the margin of error is ±75%. SportSemble's margin is no more then ±5%, in calibration data. Other errors, such as slight mistakes in body segment measurement may add to this, however it is nowhere near 75%. This is because sportSemble can measure this rate directly, opposed to a camera systems estimation, that require a derivative to be taken across optical position measurements. This was discussed in Section 1.2.

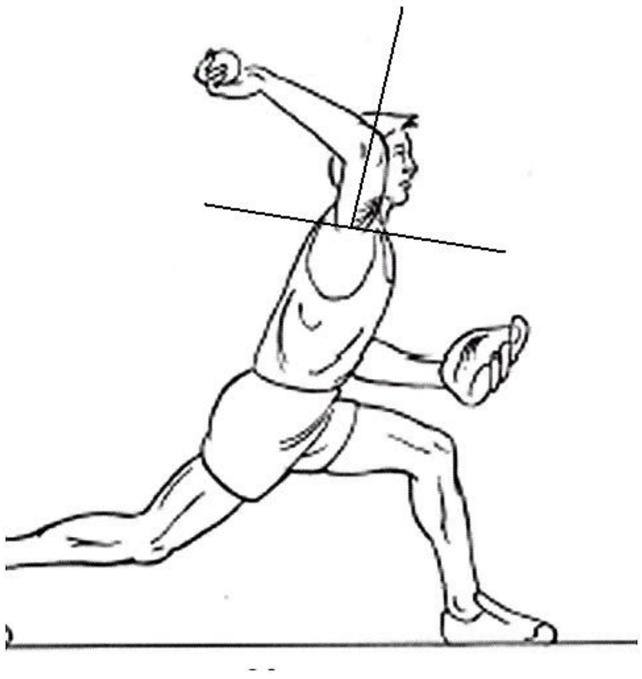


Figure 6-17: Shoulder Internal Rotation Vector Method as defined in [2]

Before discussing data, it is necessary to define the term *Shoulder Internal Rotation* as there are two different approaches to calculating it. In general, it is one of the metric's used in the biomechanical analysis of pitching. The first as utilized by Werner, [1] is depicted in Figure 6-16. This measures along one axis, which is parallel to the humerus bone. The second approach is a combination of the first (the axis that is illustrated in Figure 6-17) and the axis that is orthogonal to the humerus bone. The second approach uses all 3 possible axes of rotation in order to calculate a vector representation of the angular velocity. These calculations are almost trivial to perform with data from sportSemble because the *pitch axis* of the sportSemble node that is located on the upper arm (Section 4.2) is exactly parallel to that of of the axis defined in the *Werner* method. Further, the two other orthogonal axes defined in the second method are measured via the *roll* and *pitch axes* of the upper arm sportSemble node. We will examine data using both techniques, as neither is has been defined as a de-facto standard.

The metrics that were investigated were per pitcher averages and standard deviations. We

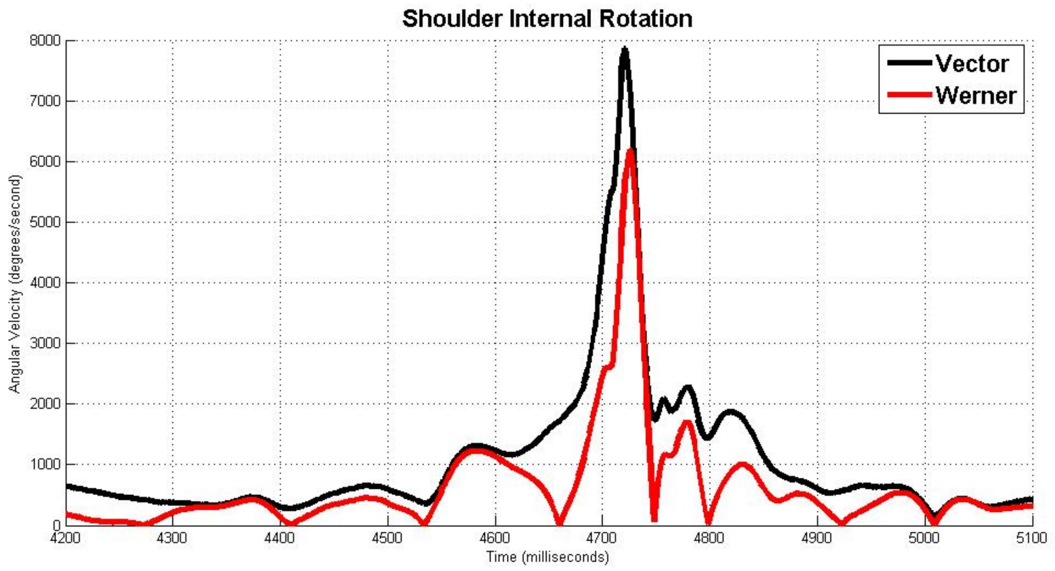


Figure 6-18: Shoulder internal rotation data for one pitch. The Werner data is a portion of the Vector data

will also compare the two methods of measurement by taking their ratio. Looking at per-pitcher averaged data for the vector method, the standard deviations (Figure 6-20) have a minimum of *108 degrees/second* and a maximum of *743 degrees/second* with a minimum value of *1706 degrees/second* and a maximum of *7861 degrees/second*. The minima and maxima are taken from the entire data set for all players, not on a per player basis.

The Werner method has a high standard deviation (Figure 6-19) of *454 degrees/second* and low of *159 degrees/second* with a maximum value of *5233 degrees/second* and a minimum of *229 degrees/second*. Looking at the minima and maxima, the numbers appear to be very variable, which may be attributed to different pitching styles or the variability in the different recorded ball speeds of the pitches. Pitch speeds ranged from 66-83 MPH, as measured by radar discussed in Section 4.2. However there does not appear to be a direct correlation between the two methods of measurement. The Werner method graphed as a percentage of the vector method (Figure 6-21), with the exclusion of *Players 1* and *Player 6*, shows that the the single axis is about 47% of the 3-axis measurement. It is interesting to note that *Player 1* had 99.6% of their internal rotation on the one axis, whereas *Player 6*

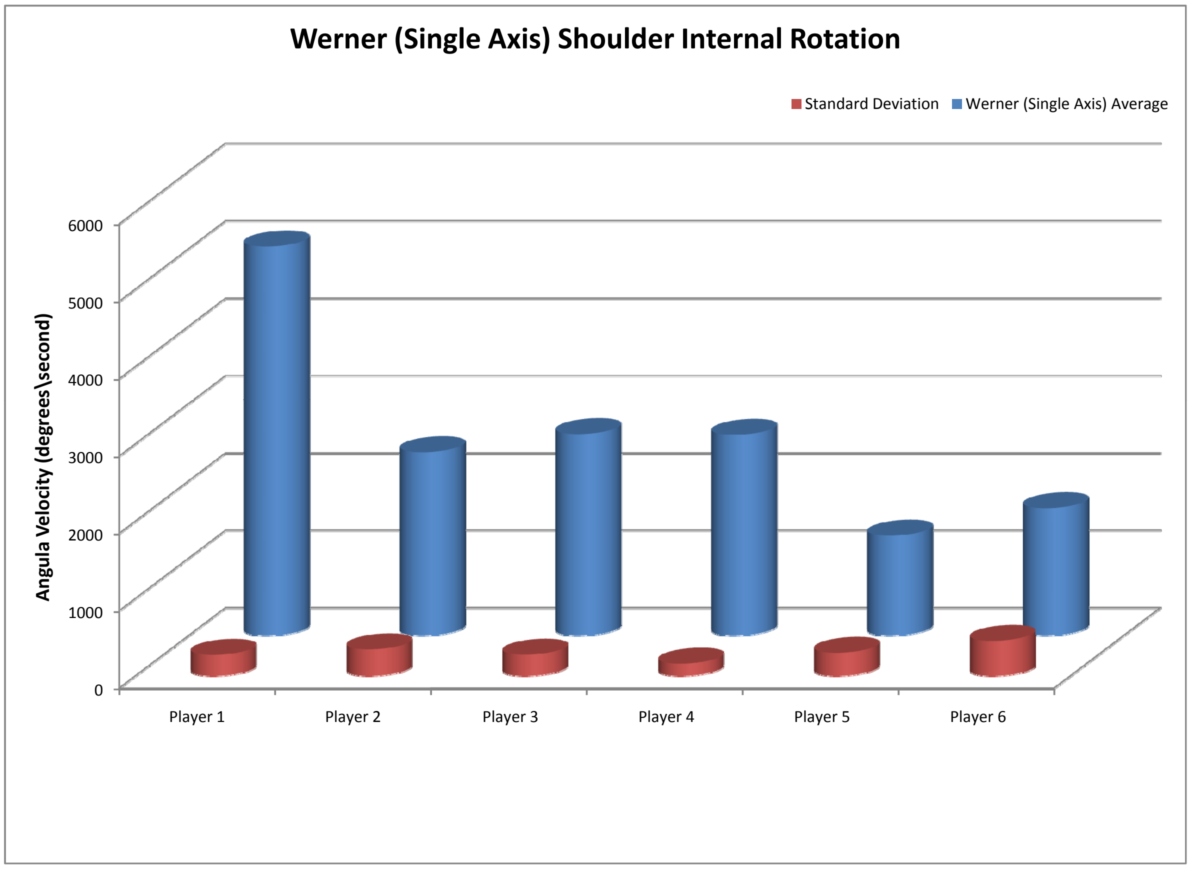


Figure 6-19: Pitcher Shoulder Internal Rotation as Measured Using the Werner Method and Standard Deviations

only had 35%. This also may be indicative of different pitching styles, although the average ball speeds were consistent, namely 79 MPH and 80 MPH, respectively, for each pitcher.

In the published literature [5, 1, 64], shoulder internal rotation has been calculated between 6,400 to 10,000 degrees/second. The values provided by sportSemble (Figure 6-20, Figure 6-19) fall within these known bounds while having much smaller margins of error.

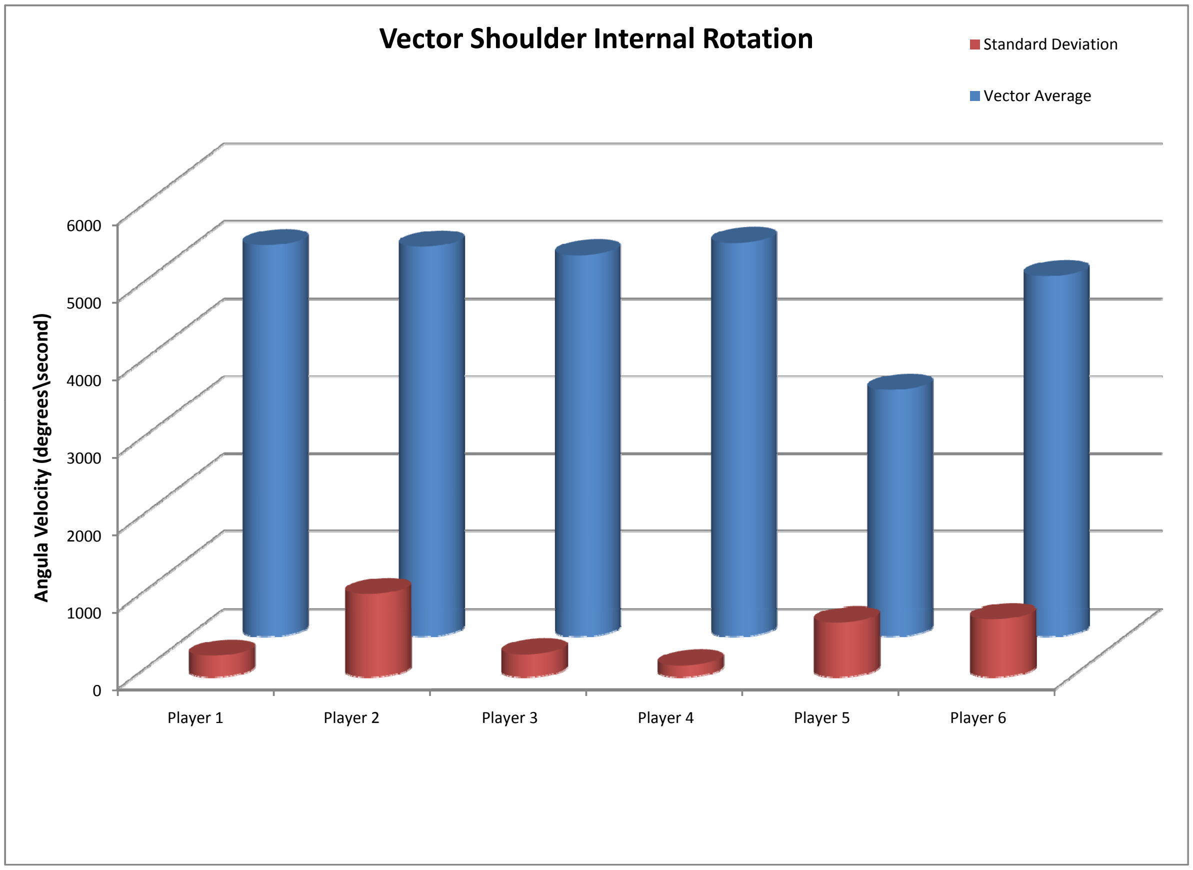


Figure 6-20: Average Pitcher Shoulder Internal Rotation Measured as a Vector and Standard Deviations

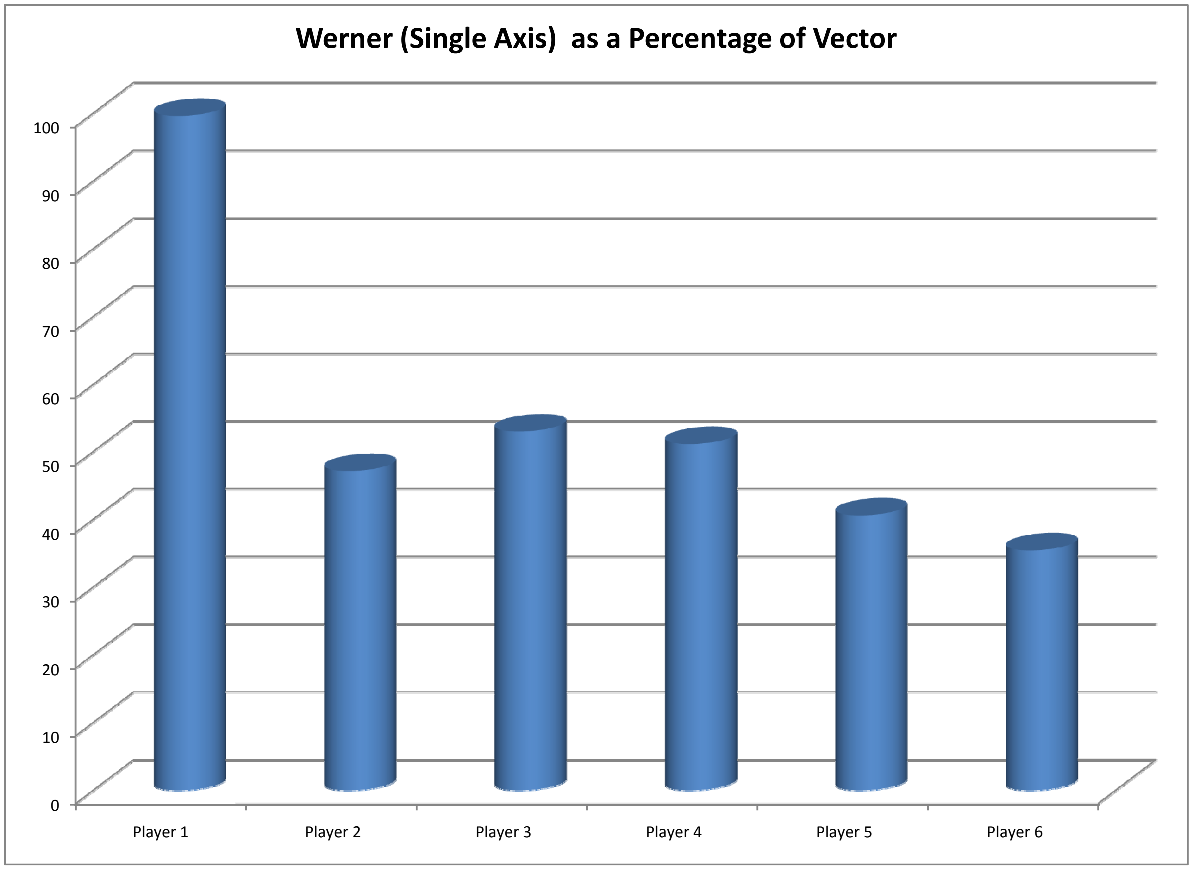


Figure 6-21: Werner Shoulder Internal Rotation as a Percentage of Vector Shoulder Internal Rotation

Chapter 7

Conclusions

7.1 Summary

This thesis presents and documents a wireless inertial sensor network, its associated processing infrastructure, and how it can be applied in measuring biomechanical data for the purpose of understanding the phenomena the body experiences during high-speed activity. The system satisfies the requirements of measuring and recording a high range of angular velocity and acceleration at a sampling rate of 1000Hz and has been calibrated in order to compare its measurements with those published by the sports medicine community. Additionally, the system is not limited to a single sport or activity, and is flexible enough to handle almost any motion of the human body.

The network is made up of small *55x50x13mm* nodes that are worn on the body; the experiments performed were baseball pitching and batting related, and nodes were placed on the waist, chest, upper arm, forearm, hand and bat. Each node contains 3 axes of high range gyroscope and accelerometer in addition to 3 axes of low range accelerometer and a digital compass with tilt compensation. A user-customizable, frequency-hopping TDMA wireless protocol running at 200Hz was developed to ensure that wireless data would be delivered reliably to a basestation that is connected to a PC or laptop via USB. A processing infrastructure records all data to files, and a post processing infrastructure processes gathered data with calibration data in order to provide precise inertial data to users.

The system has been tested with amateur collegiate pitchers, and experiments were performed with several professional-level baseball batters and pitchers. Processed data has been provided in the form of metrics that are relevant to current research in the sports medicine community.

7.2 Evaluation and Future Work

The experience that was gleaned in fielding sportSemble gave valuable lessons in designing and using subsequent systems. This section is entitled *Future Work* also, as evaluation points directly at what should be with the system next to improve it.

7.2.1 Hardware Evaluation

The fact that the essential system functioned for 3 days of data gathering during the spring training experiments and produced usable data shows that it met basic design goals. However, not everything went as smoothly as was desired. The first problem that comes to mind is the wireless protocol's inability to cope with single lost nodes. For example, if node 3 is lost and data cannot be gathered, nodes 4 and 5 are also lost. This is due to the iterative nature of the wireless protocol, and can be fixed in software. The second learning experience for the author was to calibrate hardware before bringing it out into the field, especially in such extreme kinematic environments. Even with professional soldering and X-Ray inspection, some of the solder joints lifted (or the chips themselves were damaged), rendering them useless for calibration, which occurred after experimentation. This was not a major problem, as the sensors are linear, and fairly consistent, hence calibration can be inferred with quiescent data before and after motion. Next, the hardware platform needs to be augmented with a different low range accelerometer (or the software protocol interrogating the current device needs revision), as the one that was used had very poor resolution and provided data that was not usable. There is no low-range gyroscope on the current hardware platform, and one is necessary to obtain useful data for slower movements (necessary for any kind of tracking). This became obvious when a 3D animation that was driven by the gyroscope data was created and viewed. High speed motions were accurately represented in the animation, however when looking at low speed motion, it was very obvious that not all data was being properly captured. The appropriate range of the low-rate gyroscope will need to be studied. Inertial components with logarithmic output would nicely address the issue of needing low and high rate sensors, however none are in existence that are known of. The usage of low-rate and low-G inertial components would also require study of what the effects of saturation are - a key issue here is the time it takes for the low-rate/low-G sensor to return to linear output after being saturated for a period of time. Overlapping information from the high range systems can be leveraged to smooth such transitions.

While the wireless protocol handled download of data from all nodes in about 35 seconds, this is still considered too long to wait. There are several approaches in order to solve this. The simplest would be to use the existing wireless radio for time synchronization, event demarcation, and add additional on board storage (in the form of flash memory or an SD Card) in order to store a long session of data and download it to a PC once the session was over. The second would be to use a faster radio than the 1MBps Nordic radio that was used. Another issue with the radio was that it sometimes could not communicate when occluded by the body, which would be solved by using a radio with a higher output power or lower carrier frequency. The third and most ideal method to gather data would be to stream the it in real time with a fast radio for the purposes of quick examination, while simultaneously storing data to onboard flash. The compass that was on each node also did not function, which requires rework of the protocol used to interrogate it. The best test for a complete set of hardware would be to do a full 3D joint angle reconstruction and compare it to the output of a camera system. Finally, if sportSemble could be built in a smaller form factor, with less weight, it would be less intrusive to an athlete's motion.

7.2.2 Software Evaluation

During the course of data analysis, several (14, not counting support functions) Matlab scripts were created in order to process raw data and transform it into data that could be input into Excel spreadsheets (which, unlike Matlab, are universally used by collaborators) to get what could be considered a finished product. In the process of using these, it became very apparent that a single unified infrastructure is necessary to perform data acquisition and analysis.

Further, it is also desirable to compare results with the existing camera-based motion capture systems and possibly augment their 3D animations with direct inertial data to create a better hybrid system. SportSemble has all of the features in hardware and software needed to do this and synchronization with the XOS Optical system was performed during experimentation (See Appendix C). However the way that both systems represent their data did not allow for a tractable (in terms of time-to-develop) way to use this synchronization. Ideally both systems should output some standard file format, such as C3D [65], that could be input into commercial biomechanical analysis software packages.

7.2.3 High Level Evaluation

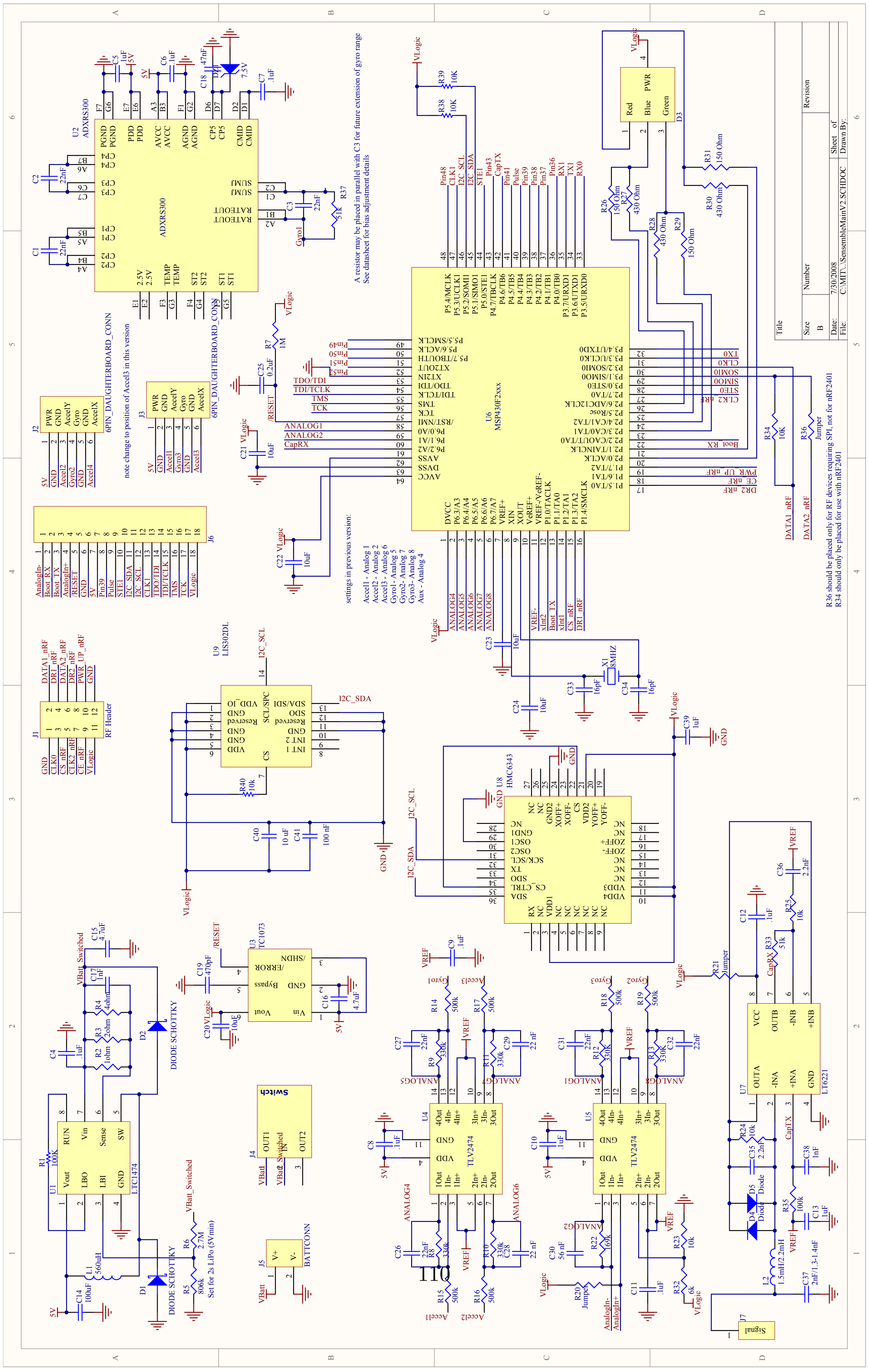
Here we examine what sportSemble provided and did that has not yet been done and look at what other data sportSemble can provide. First and foremost, sportSemble provides dynamic precision that no other system can offer because it is directly measuring acceleration and angular velocity, which are more closely related to forces and torques than position measurements, which need to be doubly differentiated.

The shoulder internal rotation data that was presented falls within the error ranges that have been published, although our data has much smaller margins of error. Bat speed estimation offers a promising avenue for estimating bat speed with an uninstrumented bat, but requires a larger sample population in order to generalize to a generic bat speed equation. Analyzing time differences between peak speeds of body segments during the course of a swing is something that had never been done before, and also offers the first quantitative suggestion of analyzing the timing of a batter's swing to determine the differences between good and bad swings. SportSemble is the first system to provide ball impact information in relation to swing timing. In addition to these metrics, many others, such as shoulder distractive force and torques in the elbow and wrist joints, are available from the data that sportSemble provides. All that is needed are more sophisticated models and processing software, now being developed, that will provide joint angles over time.

One piece of data that is hard to quantify, but is missing from the performed experiments, is the input of professional coaches that would be indicative of the quality of a given pitch or a given swing. This data would be invaluable in extracting what features a good or bad pitch/swing have. In terms of batting, a quantitative metric that is missing is how far the ball that was hit tracked and what part (left, right, center) of the field is landed in. This data would aid in testing the (hypothesized) relationship between the swing impact point and where the ball goes. The gathered data and presented metrics are only a representation of the potential of sportSemble, and may lead to the development of new metrics and new protocols that may aid in technique, training, and rehabilitation of athletes at all levels.

Appendix A

Schematics and PCB Layouts


Title
Size B
Number
Revision
Date: 7/30/2008
File: C:\MIT\..\SensembleMainV2.SCHDOC
Sheet of
Drawn By:
U6 MSP430F2xxx
U2 ADXRS300
U9 LIS302DL
U8 HMC6343
U1 LTC1474
U3 TC1073
U4 TLV2474
U5 TLV2474
U7 LT6221
J1 RF Header
J2 6PIN_DAUGHTERBOARD_CONN
J3 6PIN_DAUGHTERBOARD_CONN
J4 Switch
J5 BATTCONN
J6
J7 Signal
A resistor may be placed in parallel with C3 for future extension of gyro range
See datasheet for bias adjustment details
note change to position of Accel3 in this version
settings in previous version:
Accel1 - Analog 1
Accel2 - Analog 2
Accel3 - Analog 6
Gyro1 - Analog 5
Gyro2 - Analog 7
Gyro3 - Analog 8
Aux - Analog 4
R36 should be placed only for RF devices requiring SPI, not for nRF2401
R34 should only be placed for use with nRF2401
Set for 2s LiPo (5Vmin)
DIODE SCHOTTKY

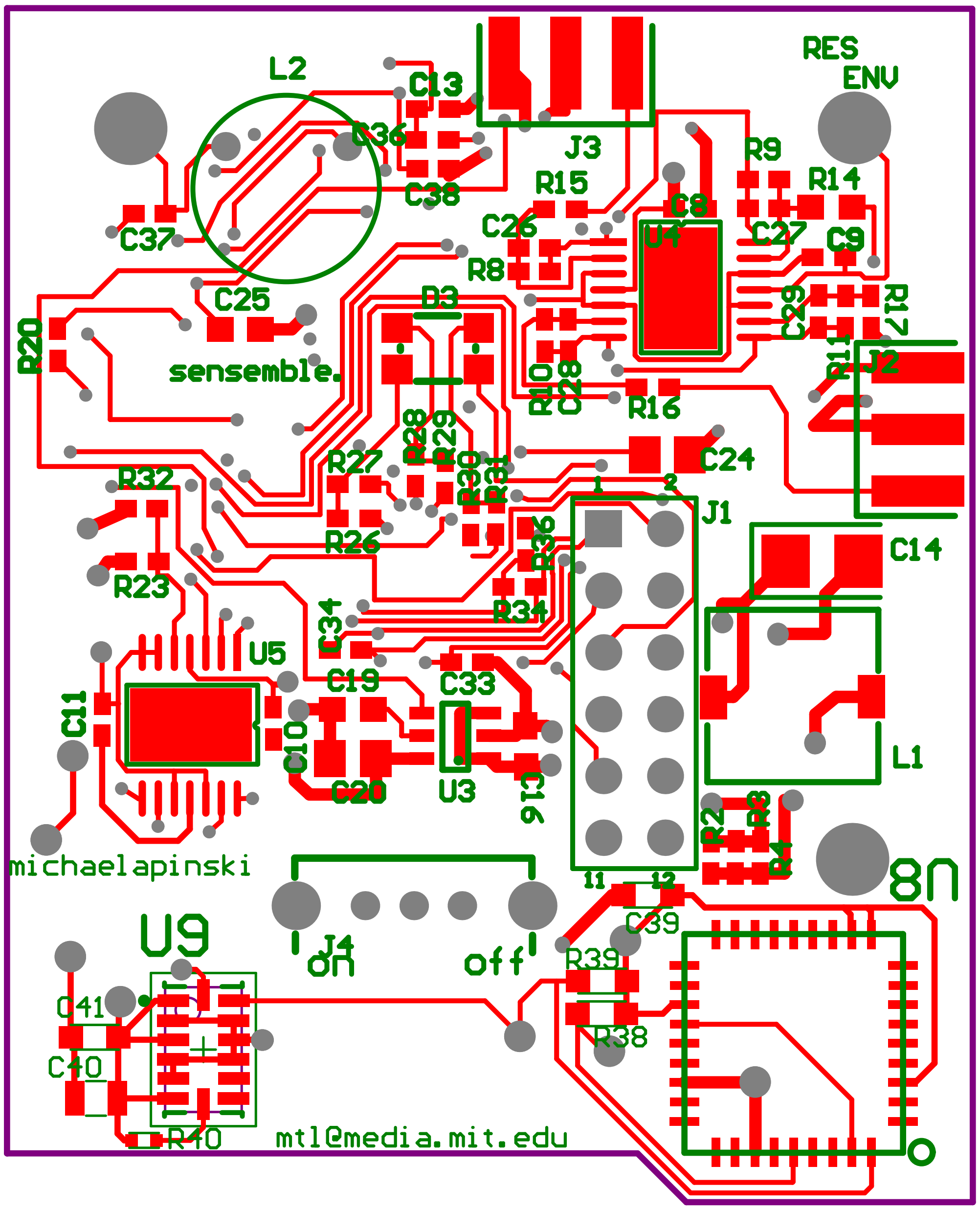


Figure A-2: sportSemble Main Board Top Layer PCB Layout

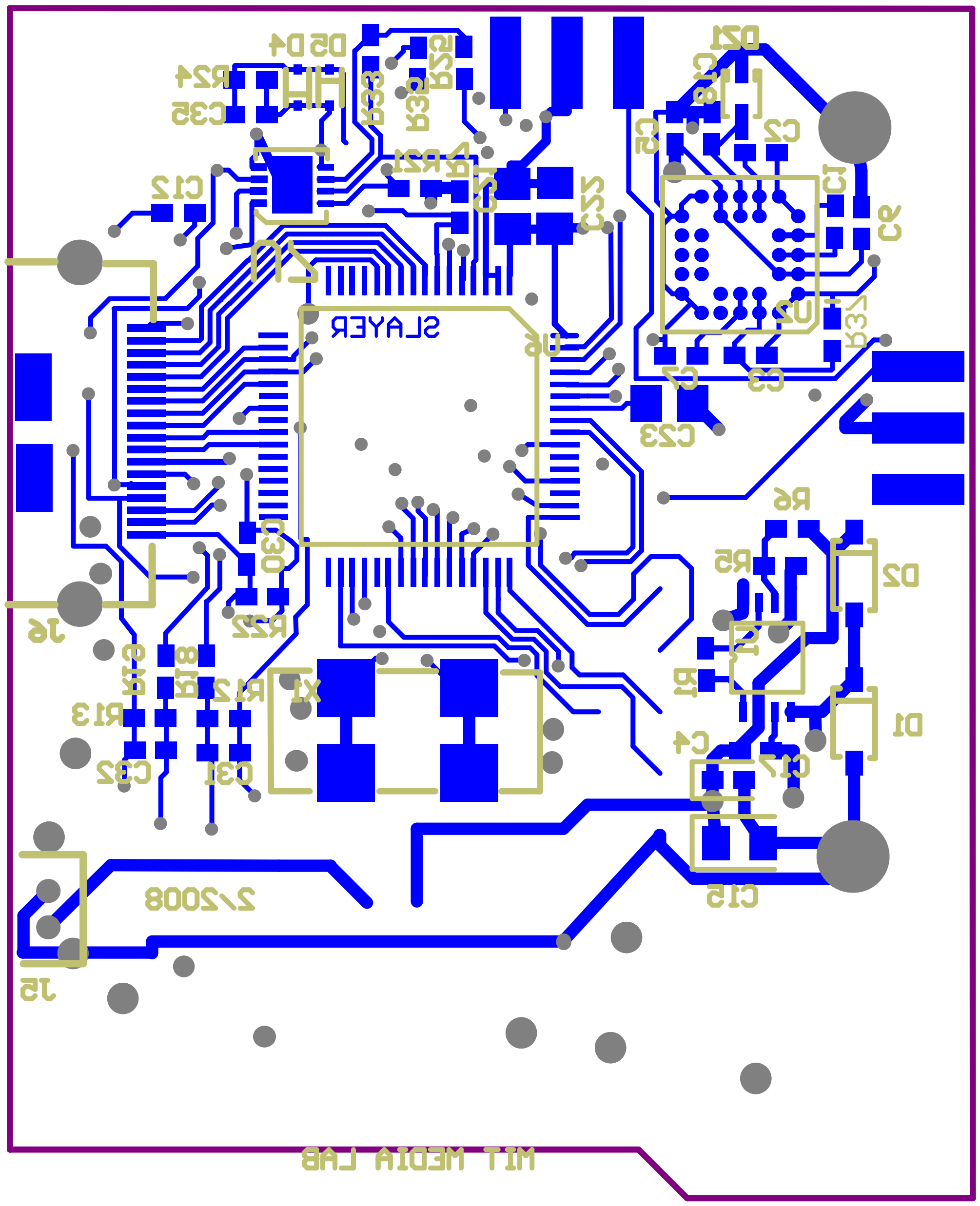


Figure A-3: sportSemble Main Board Bottom Layer PCB Layout

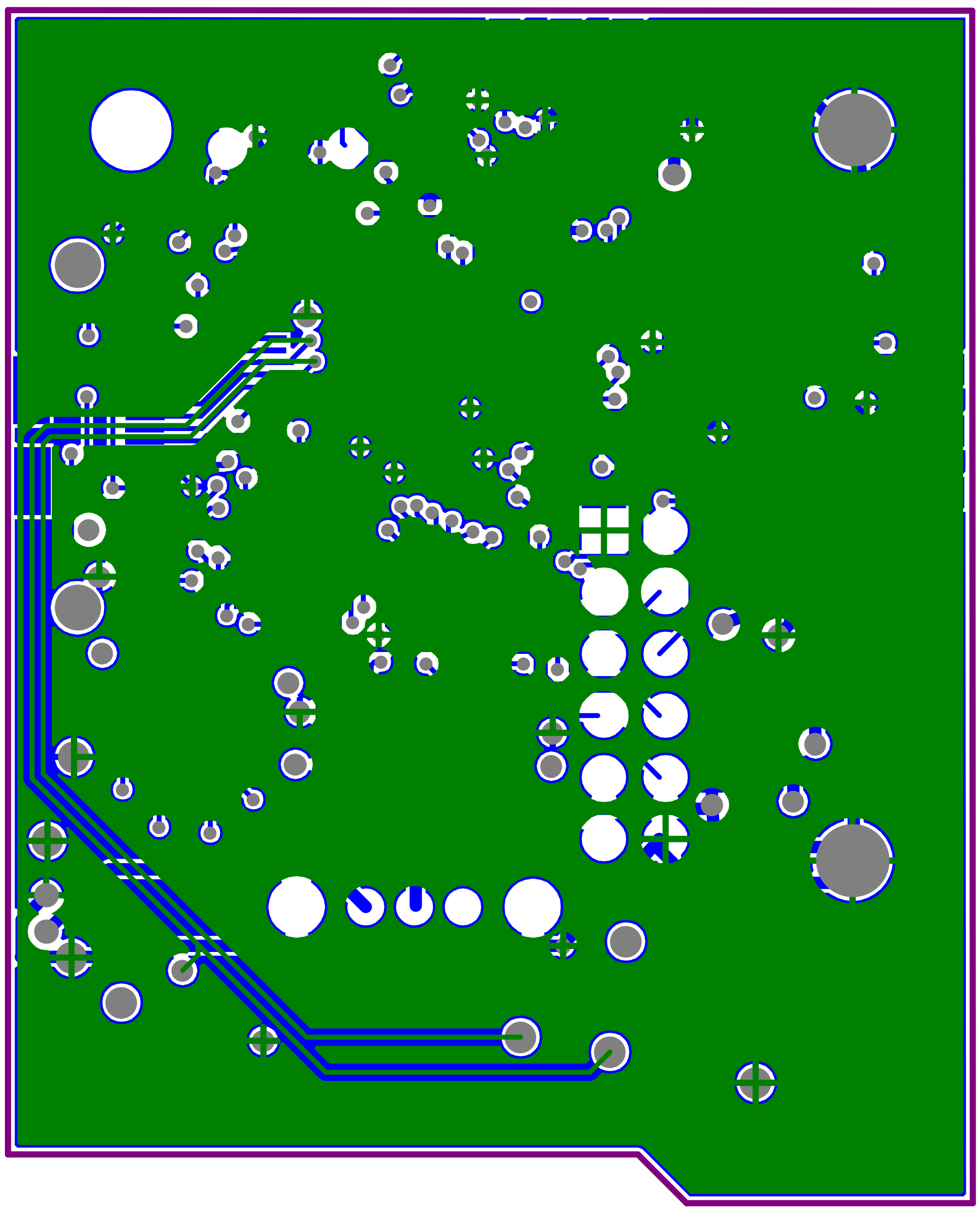


Figure A-4: sportSemble Main Board Ground Layer PCB Layout

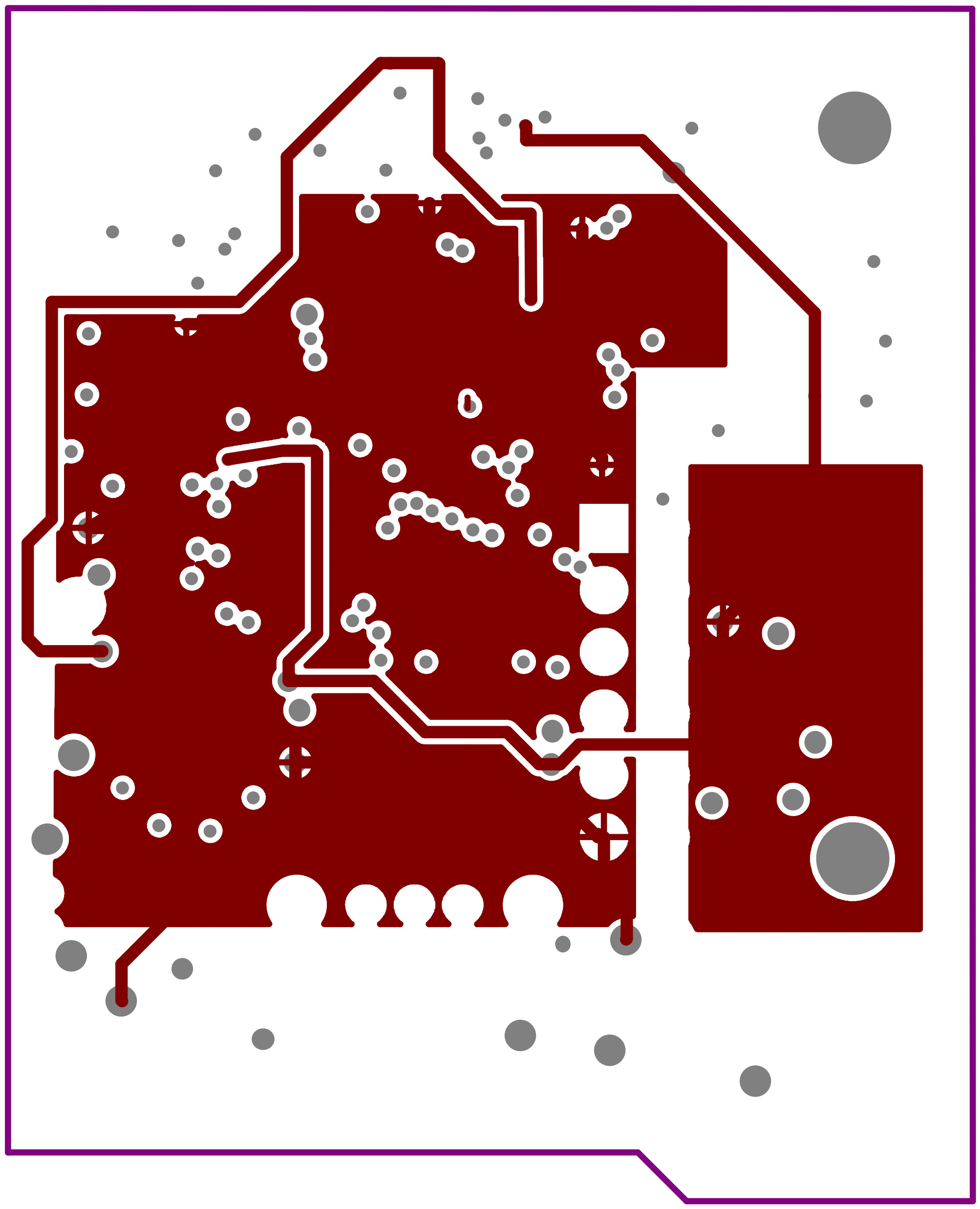
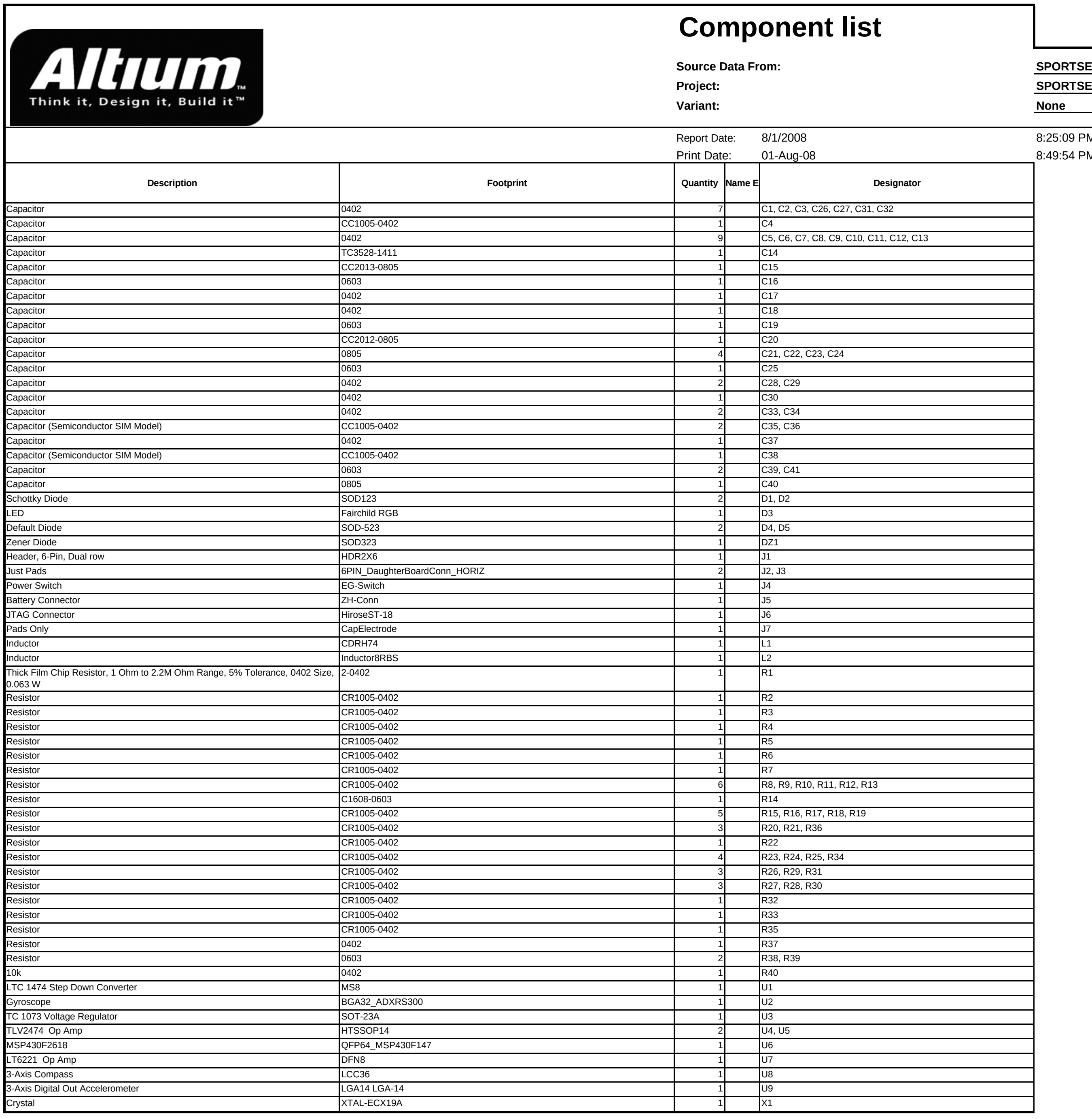


Figure A-5: sportSemble Main Board Power Layer PCB Layout



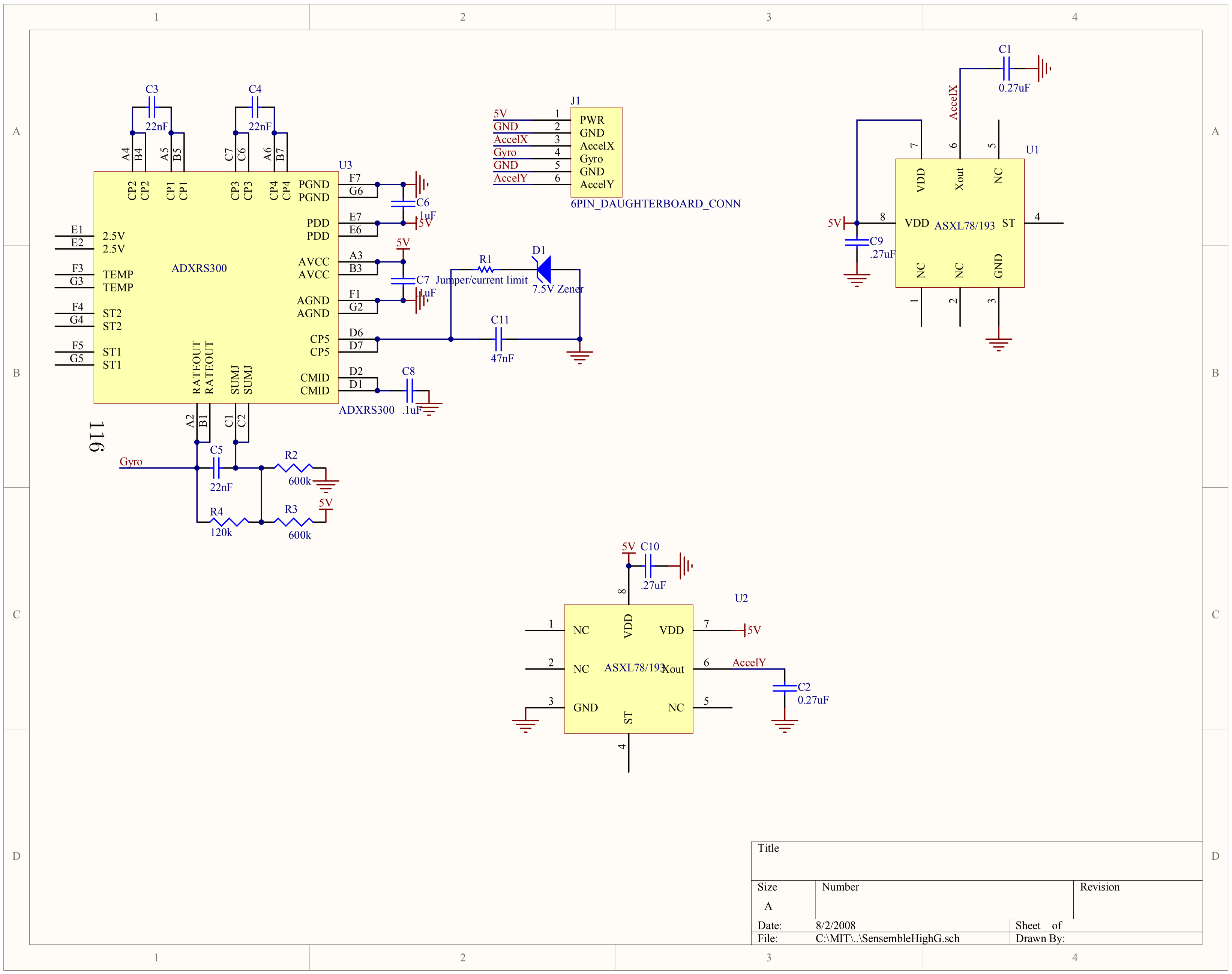
Component list

Source Data From: **SPORTSE**
Project: **SPORTSE**
Variant: **None**

Report Date: 8/1/2008 8:25:09 PM
Print Date: 01-Aug-08 8:49:54 PM

Description	Footprint	Quantity	Name E	Designator
Capacitor	0402	7		C1, C2, C3, C26, C27, C31, C32
Capacitor	CC1005-0402	1		C4
Capacitor	0402	9		C5, C6, C7, C8, C9, C10, C11, C12, C13
Capacitor	TC3528-1411	1		C14
Capacitor	CC2013-0805	1		C15
Capacitor	0603	1		C16
Capacitor	0402	1		C17
Capacitor	0402	1		C18
Capacitor	0603	1		C19
Capacitor	CC2012-0805	1		C20
Capacitor	0805	4		C21, C22, C23, C24
Capacitor	0603	1		C25
Capacitor	0402	2		C28, C29
Capacitor	0402	1		C30
Capacitor	0402	2		C33, C34
Capacitor (Semiconductor SIM Model)	CC1005-0402	2		C35, C36
Capacitor	0402	1		C37
Capacitor (Semiconductor SIM Model)	CC1005-0402	1		C38
Capacitor	0603	2		C39, C41
Capacitor	0805	1		C40
Schottky Diode	SOD123	2		D1, D2
LED	Fairchild RGB	1		D3
Default Diode	SOD-523	2		D4, D5
Zener Diode	SOD323	1		DZ1
Header, 6-Pin, Dual row	HDR2X6	1		J1
Just Pads	6PIN_DaughterBoardConn_HORIZ	2		J2, J3
Power Switch	EG-Switch	1		J4
Battery Connector	ZH-Conn	1		J5
JTAG Connector	HiroseST-18	1		J6
Pads Only	CapElectrode	1		J7
Inductor	CDRH74	1		L1
Inductor	Inductor8RBS	1		L2
Thick Film Chip Resistor, 1 Ohm to 2.2M Ohm Range, 5% Tolerance, 0402 Size, 0.063 W	2-0402	1		R1
Resistor	CR1005-0402	1		R2
Resistor	CR1005-0402	1		R3
Resistor	CR1005-0402	1		R4
Resistor	CR1005-0402	1		R5
Resistor	CR1005-0402	1		R6
Resistor	CR1005-0402	1		R7
Resistor	CR1005-0402	6		R8, R9, R10, R11, R12, R13
Resistor	C1608-0603	1		R14
Resistor	CR1005-0402	5		R15, R16, R17, R18, R19
Resistor	CR1005-0402	3		R20, R21, R36
Resistor	CR1005-0402	1		R22
Resistor	CR1005-0402	4		R23, R24, R25, R34
Resistor	CR1005-0402	3		R26, R29, R31
Resistor	CR1005-0402	3		R27, R28, R30
Resistor	CR1005-0402	1		R32
Resistor	CR1005-0402	1		R33
Resistor	CR1005-0402	1		R35
Resistor	0402	1		R37
Resistor	0603	2		R38, R39
10k	0402	1		R40
LTC 1474 Step Down Converter	MS8	1		U1
Gyroscope	BGA32_ADXRS300	1		U2
TC 1073 Voltage Regulator	SOT-23A	1		U3
TLV2474 Op Amp	HTSSOP14	2		U4, U5
MSP430F2618	QFP64_MSP430F147	1		U6
LT6221 Op Amp	DFN8	1		U7
3-Axis Compass	LCC36	1		U8
3-Axis Digital Out Accelerometer	LGA14 LGA-14	1		U9
Crystal	XTAL-ECX19A	1		X1

Figure A-6: sportSemble Main Board Bill of Materials



U3
ADXRS300
C3 22nF
C4 22nF
A4 B4 A5 B5 C7 C6 A6 B7
CP2 CP2 CP1 CP1 CP3 CP3 CP4 CP4
PGND PGND PDD PDD AVCC AVCC AGND AGND CP5 CP5 CMID CMID
F7 G6 E7 E6 A3 B3 F1 G2 D6 D7 D2 D1
2.5V 2.5V TEMP TEMP ST2 ST2 ST1 ST1
E1 E2 F3 G3 F4 G4 F5 G5
RATEOUT RATEOUT SUMJ SUMJ
A2 B1 C1 C2
C6 .1uF
C7 .1uF
C8 .1uF
5V
R1 Jumper/current limit
D1 7.5V Zener
C11 47nF
Gyro
C5 22nF
R2 600k
R3 600k
R4 120k
J1
6PIN_DAUGHTERBOARD_CONN
5V GND AccelX Gyro GND AccelY
PWR GND AccelX Gyro GND AccelY
U1
ASXL78/193
VDD Xout NC ST NC NC GND
C1 0.27uF
C9 .27uF
AccelX
U2
ASXL78/193
NC NC GND VDD Xout NC ST
C10 .27uF
C2 0.27uF
AccelY
Title
Size A
Number
Revision
Date: 8/2/2008
Sheet of
File: C:\MIT\..\SensembleHighG.sch
Drawn By:

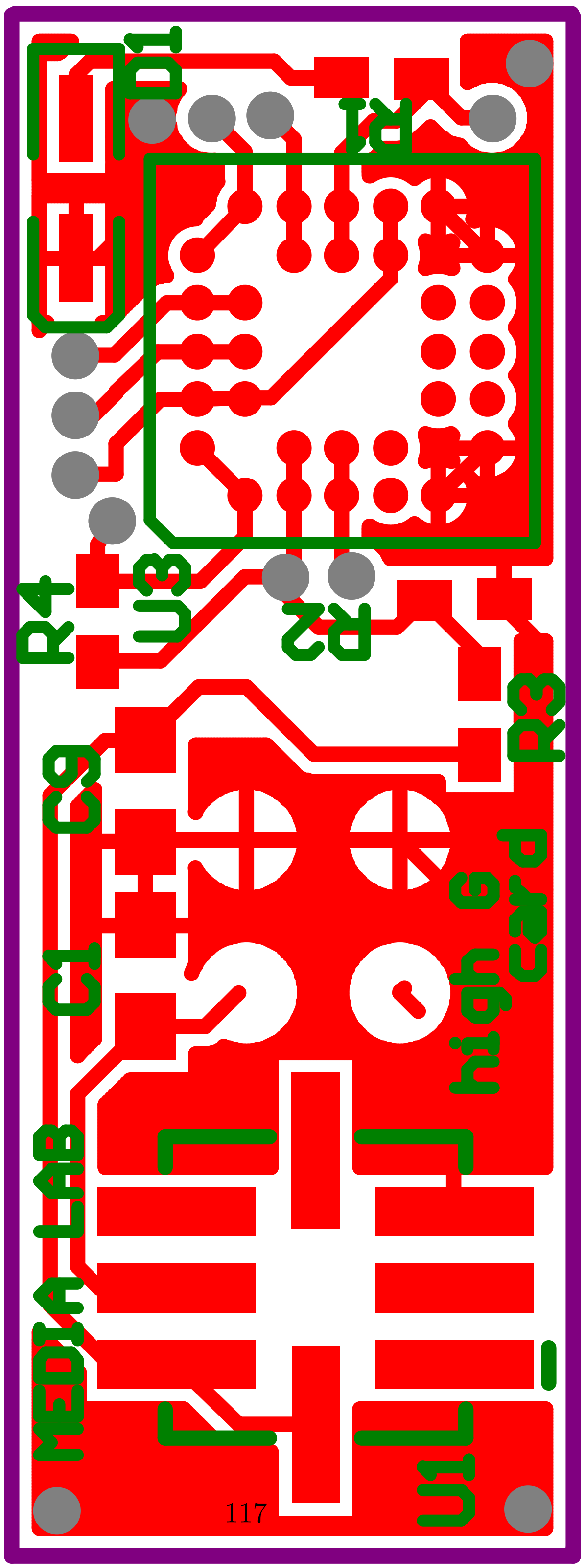


Figure A-8: sportSemble High G Board Top Layer PCB Layout

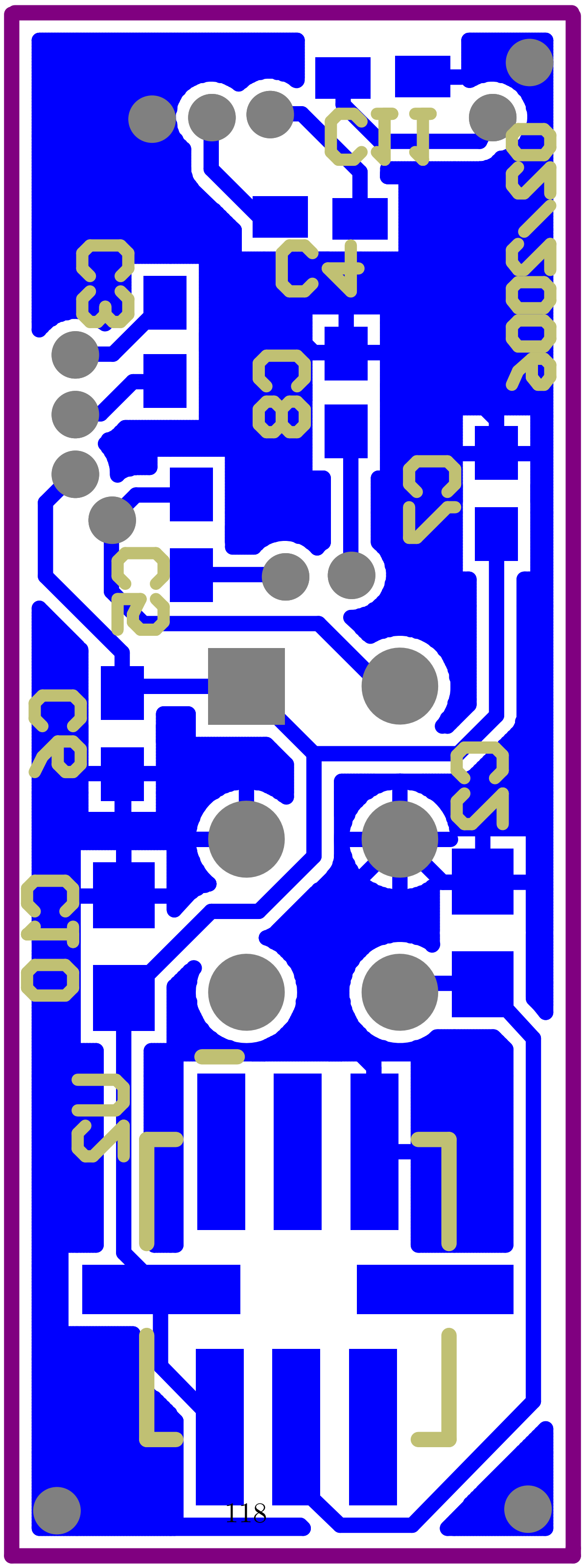
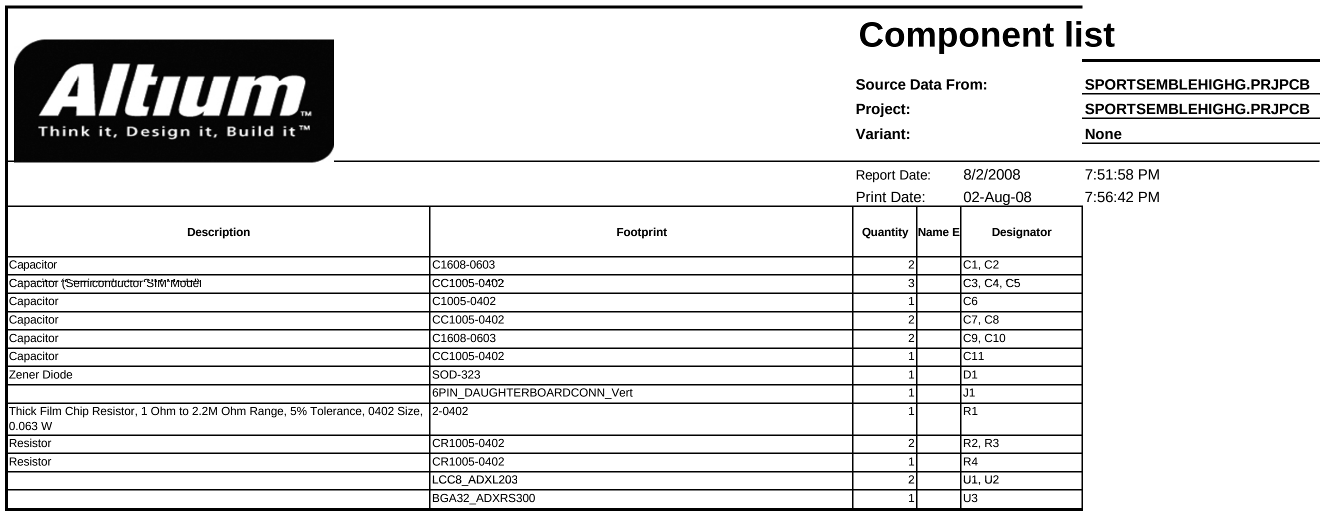


Figure A-9: sportSemble High G Board Bottom Layer PCB Layout



Component list

Source Data From: **SPORTSEMBLEHIGHG.PRJPCB**
Project: **SPORTSEMBLEHIGHG.PRJPCB**
Variant: **None**

Report Date: 8/2/2008 7:51:58 PM
Print Date: 02-Aug-08 7:56:42 PM

Description	Footprint	Quantity	Name E	Designator
Capacitor	C1608-0603	2		C1, C2
Capacitor (Semiconductor SIM Model	CC1005-0402	3		C3, C4, C5
Capacitor	C1005-0402	1		C6
Capacitor	CC1005-0402	2		C7, C8
Capacitor	C1608-0603	2		C9, C10
Capacitor	CC1005-0402	1		C11
Zener Diode	SOD-323	1		D1
	6PIN_DAUGHTERBOARDCONN_Vert	1		J1
Thick Film Chip Resistor, 1 Ohm to 2.2M Ohm Range, 5% Tolerance, 0402 Size, 0.063 W	2-0402	1		R1
Resistor	CR1005-0402	2		R2, R3
Resistor	CR1005-0402	1		R4
	LCC8_ADXL203	2		U1, U2
	BGA32_ADXRS300	1		U3

Figure A-10: sportSemble High G Board Bill of Materials

Appendix B

MSP430F2618 Firmware

```
/************************************main.c*********************************

**************************************************************************/
#include <msp430x26x.h>
#include "mainRedSoxNode.h"
#include "nRF2401.h"
#include "flash.h"
#include "initRedSoxNode.h"
#include "i2c_devices.h"

//ID should start from 1 and should not exceed MAX_NUM_NODES
const unsigned char ID = 1;

/**************** RADIO CONFIG **************/
/*
      nRF Configuration Bytes:
      Byte 0: 16 Bit payload length for RX channel 2
      Byte 1: 16 Bit payload length for RX channel 1
      Byte 3-5: Address for RX channel 2
      Byte 6-10: Address for RX channel 1
      Byte 11, upper 6 bits: Address width in # bits
      Byte 11, lower 2 bits: CRC settings
      Byte 12: Various settings including Two Channel Receive, RF power output
      Byte 13, upper 7 bits: Frequency channel selection
      Byte 13, lower 1 bit: RX?
 */
//CHANNEL 99
unsigned char TX_CONFIG_BYTES[15] = {RF_PACKET_BIT_LENGTH,RF_PACKET_BIT_LENGTH,
 0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xA1,0x6F,0xC6};
const int TX_CONFIG_BYTES_LENGTH = 15;

//unsigned char TX_SHORT_CONFIG_BYTES[1] = {0x42};
unsigned char TX_SHORT_CONFIG_BYTES[1] = {0xC6};
const int TX_SHORT_CONFIG_BYTES_LENGTH = 1;

unsigned char RX_CONFIG_BYTES[15] = {RF_PACKET_BIT_LENGTH,RF_PACKET_BIT_LENGTH,
 0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xA1,0x6F,0xC7};
const int RX_CONFIG_BYTES_LENGTH = 15;


//unsigned char RX_SHORT_CONFIG_BYTES[1] = {0x43};
unsigned char RX_SHORT_CONFIG_BYTES[1] = {0xC7};
const int RX_SHORT_CONFIG_BYTES_LENGTH = 1;

unsigned char ADDR_BYTES[5] = {0xBB,0xBB,0xBB,0xBB,0xAA};//basestation address
const int ADDR_BYTES_LENGTH = 5;

//used to store the message received from the basestation
unsigned char RX_MESSAGE[RF_PACKET_BYTE_LENGTH];
int RX_MESSAGE_LENGTH = 0;
/****************** END RADIO CONFIG **************/


//data packet structure (packed into 10 bytes plus 6 dummy bytes when
//TX for compatability with 16B radio comm)
//ID,AccX,AccY,AccZ,GyrP,GyrR,GyrY
unsigned char DATA_PACKET[RF_PACKET_BYTE_LENGTH];




unsigned int FlashReadIdx = MAX_ADDR;
unsigned char flash_state = 0;
unsigned char flash_cycle = 0;


//mtl Radio Code
unsigned char current_channel;
unsigned int TS_BYTE0;
unsigned int TS_BYTE1;
unsigned int rxcnt;
unsigned int configRXCnt;
unsigned int txcnt;

unsigned char nRF_BASE_CHANNEL = 99;
unsigned char nRF_CHANNEL_COUNT  = 2;

//denotes state of the system
//states are in mainRedSoxNode.h
unsigned char SYS_STATE;


//external variable pointing to flash
//extern long * flashPtr;
extern unsigned long flashPtr;

//try to figure out radio state
#define RX 0
#define TX 1
extern int RADIO_STATE;
int RADIO_CONFIG;

//unsigned int sample_count = 0;
//unsigned int sentSample0Count = 0;

//Button click variables
//unsigned int isButtonClicked = 0;
//unsigned int btnClickedSentCnt = 0;

//variable to keep track of when to sample compass and low g accell
unsigned long compassLowGCtr = 0;

unsigned char LOW_G_DATA[3];//low g accel data
unsigned int I2CAccelStage = 0;
unsigned char I2CAccelREG;

unsigned char COMPASS_DATA[6];
unsigned int compassByteCtr = 0;
unsigned int compassStage = 0;
unsigned int compassIdleCtr = 0;

void main(void) {
  int i,temp,delay;

  //set up MSP430
  CONFIG();
  CONFIG_SPI0();
  CONFIG_I2C();
  CONFIG_SPI1();
  CONFIG_IO();
  CONFIG_ADC();
  CONFIG_FLASH();
  nRF_RESET();
  nRF_SetStdByMode();

  //configI2CAccel();
  //configI2CCompass();


  CONFIG_TIMER_B0();

  //sampleI2CCompass();

    //mtl
  current_channel = nRF_BASE_CHANNEL;
  rxcnt = 0;
  configRXCnt = 0;
  SYS_STATE = 0x03; //start in unknown state
  txcnt = 0;
  compassLowGCtr = 0;



  //initialize the interval to be timed - DEPENDS ON RF PACKET
  //SIZE AND TIME SPENT IN DATA WAITING
  //large interval temp is between RX of broadcast and preparation
  //for next broadcast
  //should be somewhat less than BROADCAST_INTERVAL
  //tradeoff between power consumption and tolerance to node/basestation
  //clock skew
  //second arg of CONFIG_TIMER_A_INT must be less than temp
  temp = BROADCAST_INTERVAL - 0x0200; //maximum timer arg is 9487 - max
  //num nodes should be 26

  // added 100 to the original 330 value to make it 430 because 4 bytes
  //added to payload.
  if(ID > 0 && ID <= MAX_NUM_NODES){
    delay = 225+(ID*430); //225 and 330 are offset based on a minimum
 //overhead time of 1095 usec
    CONFIG_TIMER_A_INT(temp,delay,0);
    }

  //give nRF time to go to PWR_UP mode before going to stand by
  for(i=0;i<5000;i++) {}
  nRF_CONFIG(RX_CONFIG_BYTES,RX_CONFIG_BYTES_LENGTH);
  RADIO_CONFIG = RX;

  for(i=0;i<100;i++) {} //debug
  //HACK the nRF_SetTXRX will flip this state
  RADIO_STATE = TX;
  nRF_SetTXRXMode();


  //initialize header for RF packets
  DATA_PACKET[0] = ID;

  for(i = 0; i < 6; i++)
   COMPASS_DATA[i] = 0;

  LedDriver(1,0,0); //dim red led

  //enable interrupts
  _EINT();
```

```
  while(1) {
    //enter low power mode
    //LPM0;
  }
} // end main()


void ERROR(void){
  while(1);
}

void sampleLowGAccel(void){
 //WRITES TO BYTES 10, 11, 12 of DATA_PACKET
 DATA_PACKET[4] = 0;
 DATA_PACKET[5] = 0;
 DATA_PACKET[6] = 0;

   DATA_PACKET[1] = 0;
   DATA_PACKET[2] = 0;
   DATA_PACKET[3] = 0;
 DATA_PACKET[4] = readI2CAccelRegister(OUT_X);
 DATA_PACKET[5] = readI2CAccelRegister(OUT_Y);
 DATA_PACKET[6] = readI2CAccelRegister(OUT_Z);
   DATA_PACKET[7] = 0;
   DATA_PACKET[8] = 0;
}

void sampleCompass(void) {
 //WRITES TO BYTE 13,14 of DATA_PACKET
 sampleI2CCompass();
 DATA_PACKET[13] = COMPASS_DATA[0];
 DATA_PACKET[14] = COMPASS_DATA[1];
}

void sample_data(void) {

  unsigned int adc1,adc2;

  //50us to complete conversions
  ADC12CTL0 |= ADC12SC; // Start conversion
  while(!(ADC12IFG & 0x40));
  adc1 = ADC12MEM2;
  adc2 = ADC12MEM4;
  DATA_PACKET[1] = adc1>>4; //analog 4 - AccX high byte
  DATA_PACKET[2] = (adc1<<4)|(adc2>>8); //AccX low nibble and AccY high nibble
  DATA_PACKET[3] = adc2; //analog 6 - AccY low byte
  adc1 = ADC12MEM5;
  adc2 = ADC12MEM6;
  DATA_PACKET[4] = adc1>>4;  //analog 7 - AccZ high byte
  DATA_PACKET[5] = (adc1<<4)|(adc2>>8); //AccZ low nibble and GyrP high nibble
  DATA_PACKET[6] = adc2; //analog 8 - GyrP low byte
  adc1 = ADC12MEM0;
  adc2 = ADC12MEM3;
  DATA_PACKET[7] = adc1>>4; //analog 1 - GyrR high byte
  DATA_PACKET[8] = (adc1<<4)|(adc2>>8); //GyrR low nibble and GyrY high nibble
  DATA_PACKET[9] = adc2; //analog 5 - GyrY low byte
  //adc1 = ADC12MEM1;
  //DATA_PACKET[10] = adc1>>4; //analog2 - Aux high byte
  //DATA_PACKET[11] = adc1<<4; //Aux low nibble
```

```
}

/*****
* RX_MESSAGE[0] = command value
* RX_MESSAGE[1] = Timestamp byte 1
* RX_MESSAGE[2] = Timstamp byte 2
* RX_MESSAGE[3-15] = command specific data
********/
void nRF_data_waiting(void) {


  _DINT();
  //LedDriver(0,4,4);//Red LED off
  //clear the interrupt flag
  P1IFG &= ~BIT4;

  //Start timers for standard mode right away, reset timers when entering
  //new mode
  TACTL |= MC0;      // Start timing the RF interval

  //COMMENTED NO SUCH FUNCTION EXISTS????
  //nRF_SHOCKBURST_BLOCKING_READ(RX_MESSAGE,&RX_MESSAGE_LENGTH);
  //read the received data
  nRF_READ_FROM_EXPECTED_PACKET(RX_MESSAGE,8);

  //if the system is in the middle of gathering data to flash record
  //new timestamp, enable interrupts& ret
  //NEED A STOP FOR THIS
  if(SYS_STATE == SAMPLE_2_FLASH && RX_MESSAGE[0] == SAMPLE_2_FLASH){
   TS_BYTE0 = RX_MESSAGE[1];
    TS_BYTE1 = RX_MESSAGE[2];
    _EINT();
    return;
  }

  //when we hear a packet we hop up to the next channel and reply on that
  //channel. if its a network config packet we do not increase the current
  channel (may not need this)
  if(RX_MESSAGE[0] == RF_CONFIG_CMD )
    nRFSetupChannelBytes(0);
  else
    nRFSetupChannelBytes(1);
   //network reconfig packet
  if (RX_MESSAGE[0] == RF_CONFIG_CMD ){//&& configRXCnt == 0)
 //network configuration packet == 0xAA
    //TACTL &= MC_0;    //stop Timer_A because we will not be replying
    handleAAPkt();
  } //sample data and send it back packet
  else if(RX_MESSAGE[0] == 0xAF){
    handleAFPkt();
  } else if(RX_MESSAGE[0] == STOP_SAMPLE_2_FLASH){
    SYS_STATE = STOP_SAMPLE_2_FLASH;
  } else if(RX_MESSAGE[0] == SAMPLE_2_FLASH){
    if(SYS_STATE == FLASH_FULL){
      //set the data packet up to reply FLASH_FULL as status and keep
   //sending that
      //once the host PC has heard back from all the nodes that flash is
   //full it
      //will start getting the data from the nodes.
      //get ready for TX
      nRF_CONFIG(TX_SHORT_CONFIG_BYTES,TX_SHORT_CONFIG_BYTES_LENGTH);
      RADIO_CONFIG = TX;
      DATA_PACKET[0] = FLASH_FULL;
      DATA_PACKET[1] = ID;
      DATA_PACKET[2] = 0;
      DATA_PACKET[3] = 0;
      DATA_PACKET[4] = 0;
      DATA_PACKET[5] = 0;

    } else {
      TACTL &= ~MC0; //for the duration of sample & write to flash remain
   //in rx mode
//    P1IE &=  ~BIT4; //disable interrupts for the NRF
//    P1IFG &= ~BIT4; //clear any pending

      nRF_CONFIG(TX_SHORT_CONFIG_BYTES,TX_SHORT_CONFIG_BYTES_LENGTH);
      RADIO_CONFIG = TX;

      handleSample2FlashPkt();
      //TACTL |= MC0;
    }
  }
  else if(RX_MESSAGE[0] == ERASE_FLASH){
    TACTL &= ~MC0;    //stop Timer_A because we will not be replying during
       //flash erase
    handleEraseFlashPkt();
    TACTL |= MC0;
  }
  else if(RX_MESSAGE[0] ==  SYS_IDLE) {
    //Node Idle Mode
    //get ready for next TX
    SYS_STATE = SYS_IDLE;
    nRF_CONFIG(TX_SHORT_CONFIG_BYTES,TX_SHORT_CONFIG_BYTES_LENGTH);
    RADIO_CONFIG = TX;
    handleSysIdlePkt();
  } else if(RX_MESSAGE[0] == NEXT_PACKET){
    //TX MODE
    nRF_CONFIG(TX_SHORT_CONFIG_BYTES,TX_SHORT_CONFIG_BYTES_LENGTH);
    RADIO_CONFIG = TX;
    handleNextPacketPkt();
  } else
    ERROR();

  //LedDriver(1,4,4);
  _EINT();
}




/**
* Use Timer A1 to set radio to TX mode and send data
* Use Timer A0 to set radio back to RX mode
*****/

//CrossWorks
//void Timer_A1 (void) __interrupt[TIMERA1_VECTOR]{
__interrupt void Timer_A1(void);
TIMERA1_ISR(Timer_A1)
__interrupt void Timer_A1(void) {
  int i, flag;
  _DINT();
  if(SYS_STATE == FLASH_FULL)
    LedDriver(1, 1 , 0);
  else
    LedDriver(0, 1, 0);


  txcnt++;
  DATA_PACKET[18] = txcnt >> 8;
  DATA_PACKET[19] = txcnt;

  flag = TAIV; //have to manually reset A1
  //set TX mode
  nRF_SetTXRXMode();

  //write packet to nRF buffer
  nRF_SHOCKBURST_BLOCKING_WRITE(DATA_PACKET,RF_PACKET_BYTE_LENGTH,ADDR_BYTES,
   ADDR_BYTES_LENGTH);
  //take CE low to start TX, then enter standby
  nRF_SetStdByMode();

  //prepare return to receive mode
  for(i=0;i<950;i++) {}//wait for TX to occur 195us + TOA
  //LedDriver(4,4,0);


  nRF_CONFIG(RX_SHORT_CONFIG_BYTES,RX_SHORT_CONFIG_BYTES_LENGTH);
  RADIO_CONFIG = RX;
  //config leaves nRF in standby
  LedDriver(0, 0, 0);
  _EINT();
}

/** set radio to RX mode after a TX **/
__interrupt void Timer_A0(void);
//Crossworks void Timer_A0 (void) __interrupt[TIMERA0_VECTOR] {
TIMERA0_ISR(Timer_A0)
__interrupt void Timer_A0(void) {
  _DINT();

    if(SYS_STATE == FLASH_FULL)
    LedDriver(1, 0, 1);
   else
    LedDriver(0, 0, 1);

  //turn off/reset timer A0
  TACTL &= ~MC0;
  //Return to receive mode
  nRF_SetTXRXMode();
  _EINT();
}


/**
* Running at 1000Hz
***/
__interrupt void Timer_B0(void);
// Crossworks void Timer_B0 (void) __interrupt[TIMERB0_VECTOR] {
TIMERB0_ISR(Timer_B0)
__interrupt void Timer_B0(void){
  unsigned char status = 0;
```

```
 int i;
 //_DINT();
 //P1OUT |= BIT1;   //TESTING TIMEING
 status = TBIV;
 //LedDriver (1, 0, 0);


//this is annoyign code that does I2C commands as other sensors are being
   //sampled
  if(compassLowGCtr%215 == 0 && compassStage == 0){

DATA_PACKET[1] = 0xFF;
  DATA_PACKET[2] = 0xFF;
  DATA_PACKET[3] = 0xFF;
  DATA_PACKET[4] = COMPASS_DATA[0];
  DATA_PACKET[5] = COMPASS_DATA[1];
  DATA_PACKET[6] = COMPASS_DATA[2];
  DATA_PACKET[7] = COMPASS_DATA[3];
  DATA_PACKET[8] = COMPASS_DATA[4];
  DATA_PACKET[9] = COMPASS_DATA[5];
DATA_PACKET[10] = 0xFF;
DATA_PACKET[11] = 0xFF;

   compassStage = 1;


  }else if(I2CAccelStage == 0 && compassStage == 0
   && compassLowGCtr < 83 && 0){

   //83 is hand tuned to make sure a low g accel
   //read doesnt start when the compass is due to start
   //low g accel read take 16 cycles of a 1000Hz timer
  //record previous data
  //for(i = 1; i <= 11; i++)
  // DATA_PACKET[i] = 0;
  DATA_PACKET[1] = 0;
  DATA_PACKET[2] = 0;
  DATA_PACKET[3] = 0;
  DATA_PACKET[4] = LOW_G_DATA[0];
  DATA_PACKET[5] = LOW_G_DATA[1];
  DATA_PACKET[6] = LOW_G_DATA[2];
  DATA_PACKET[7] = 0;
  DATA_PACKET[8] = 0;

  //compassLowGCtr = 0;
  I2CAccelREG = OUT_X;
  I2CAccelStage = 1;

 } else
  sample_data();

 compassLowGCtr++;

 if(compassStage > 0){
  switch (compassStage){
   case 1:
    readCompassSendStart();
    compassStage = 2;
    break;
   case 2:
    if(readCompassCheckTX() == 0x01)
    compassStage = 3;
   break;
  case 3:
   if(readCompassCheckSTPTX() == 0x01)
    compassStage = 4;
   break;
  case 4:
   if(readCompassCheckStart2TX() == 0x01)
    compassStage = 5;
   break;
  case 5:
   if(compassIdleCtr==1) {
    compassIdleCtr = 0;
    compassStage = 6;

   } else {
    compassIdleCtr++;
    P1OUT |= BIT1;
   }
   break;
  case 6:
   if(readCompassCheckByteRX() == 0x01){
    COMPASS_DATA[compassByteCtr] = UCB1RXBUF;
    compassByteCtr++;
    if(compassByteCtr == 6) {
     UCB1CTL1 |= UCTXSTP;  //send stop
     P1OUT &= ~BIT1;
     compassByteCtr = 0;
     compassStage = 7;
    }
   }
   break;
  case 7:
   if(readCompassCheckSTPTX2() == 0x01)
    compassStage = 0;
   break;
 }

}else if(I2CAccelStage > 0){
  switch(I2CAccelStage) {
case 1:
   readI2CAccelRegSendStart(I2CAccelREG);
   I2CAccelStage = 2;
   break;
  case 2:
   if(readI2CAccelRegCheckTX() == 0x01)
    I2CAccelStage = 3;
   break;
  case 3:
   if(readI2CAccelRegCheckStart() == 0x01)
    I2CAccelStage = 4;
   break;
  case 4:
   if(readI2CAccelRegCheckStop() == 0x01){
    if(I2CAccelREG == OUT_X){
     LOW_G_DATA[0] = UCB1RXBUF;
     I2CAccelREG = OUT_Y;
     I2CAccelStage = 1;
    } else if(I2CAccelREG == OUT_Y){
     LOW_G_DATA[2] = UCB1RXBUF;
     I2CAccelREG = OUT_Z;
       I2CAccelStage = 1;
      } else {
       LOW_G_DATA[2] = UCB1RXBUF;
       I2CAccelStage = 0;
      }
     }
     break;
    }
  }

  status = mtl_write_sample_to_flash(DATA_PACKET, 1, 1);

  if(status == FLASH_FULL){

    TBCTL &= ~BIT4; //Stop timer b
    TBCTL &= ~BIT0; //clear interrupt flag...
    SYS_STATE = FLASH_FULL; //let everything else know the flash is full.

    DATA_PACKET[0] = FLASH_FULL;
    DATA_PACKET[1] = ID;

    TACTL |= MC0;
    P1IE |=  BIT4;    //reenable interrupts for radio.
    LedDriver(0, 1, 0); //back to red led
  }

   //P1OUT &= ~BIT1;
   //_EINT();
}

void handleSysIdlePkt(void){

      LedDriver(0,0,0);
      //Process Message
      if(RX_MESSAGE[1] == 0xFF || RX_MESSAGE[1] == ID){
        //we have a message
        if(RX_MESSAGE[2] == 0x01){
          LedDriver(RX_MESSAGE[3],RX_MESSAGE[4],RX_MESSAGE[5]);
        }
      }
      DATA_PACKET[0] = RX_MESSAGE[0];
      DATA_PACKET[1] = ID;
      //DATA_PACKET[2] = isButtonClicked;
      //if(isButtonClicked == 1)
      //  btnClickedSentCnt++;

      //if(btnClickedSentCnt >= 10){
      //  isButtonClicked = 0;
      //  btnClickedSentCnt = 0;
      //}
}

void handleNextPacketPkt(void){
  int po = 0;
  int i;
  unsigned long addr;

  i = RX_MESSAGE[4];
  i = i << 8;
  i += RX_MESSAGE[5];
```

```
      //if(RX_MESSAGE[4] == 0x00 && RX_MESSAGE[5] == 0x01)
      //  sentSample0Count++;

      //if(sentSample0Count == 10)
      //  sentSample0Count++;

      if(RX_MESSAGE[4] == 0)
        LedDriver(0,0,1);

      DATA_PACKET[0] = RX_MESSAGE[0];
      DATA_PACKET[1] = ID;
      DATA_PACKET[2] = RX_MESSAGE[4];
      DATA_PACKET[3] = RX_MESSAGE[5];

      addr = (unsigned long) ((unsigned long)UPPER_FLASH_TOP -
    ((unsigned long)SAMPLE_SIZE * (unsigned long)i));
   //standard memory arch, get rid of UPPER and make LOWER to use

      if(addr > UPPER_FLASH_TOP)
       po++;

      //for msp430f2618 need to dick around with the interrupt vector being in
   //the middle of flash
      if(addr-SAMPLE_SIZE < (unsigned long)UPPER_FLASH_BASE) { //addr is the
  //top of the sample, if entire sample doesnt fit start at the top of
  //lower mem subtract the number of whole samples that are in the upper
  //memory then start reading samples from LOWER_FLASH_TOP
       i = i - (UPPER_FLASH_TOP - UPPER_FLASH_BASE)/SAMPLE_SIZE;

       addr = (unsigned long) (LOWER_FLASH_TOP - (SAMPLE_SIZE * i));
      }

      i = mtl_read_sample_from_flash(addr);

      if(i == FLASH_END){
        DATA_PACKET[4] = FLASH_END;
        DATA_PACKET[5] = FLASH_END;
        DATA_PACKET[6] = FLASH_END;
      }
}

/***** handle command to sample to flash ***/
void handleSample2FlashPkt(void){

  SYS_STATE = SAMPLE_2_FLASH;
  //starting timer b will start flash gathering process
  start_timer_B();

}


void handleEraseFlashPkt(void){
  int i;
  char * addr;
  unsigned long longAddr;

  _DINT();//just in case

  LedDriver(1, 1, 0);
  //reset flash ptr
  //erase segment by segment

  //this erases upper flash memory
  longAddr = (unsigned long) UPPER_FLASH_TOP;
  for(i = 0; i < UPPER_SEGMENT_COUNT+1; i++){
     mtl_erase_upper_flash_segment(longAddr);
     longAddr -= 512;
  }

  //this erases lower flash memory
  addr = (char *) LOWER_FLASH_TOP;
  for(i = 0; i < LOWER_SEGMENT_COUNT+1; i++){
     mtl_erase_flash_segment(addr);
     addr -= 512;
  }
  flashPtr = (unsigned long) UPPER_FLASH_TOP;

  LedDriver(1, 0, 0);
  _EINT();//just in case
}


/*** Network Configuration Packet ***/
void handleAAPkt(void){
  configRXCnt++;
  nRF_BASE_CHANNEL = RX_MESSAGE[5];
  //bytes 3 & 4 becase USB packet is shifted to insert timestamp
  nRF_CHANNEL_COUNT  = RX_MESSAGE[4];
  current_channel = nRF_BASE_CHANNEL;

  /****
   * byte 5 has the base channel because somethign was fucking it up along the
   *way...
  ****/
}


/*** Sample and Respond withing RF interval **/
void handleAFPkt(void){
    //mtl
    //save timestamp to tx back to base
    TS_BYTE0 = RX_MESSAGE[1];
    TS_BYTE1 = RX_MESSAGE[2];
    rxcnt++;

    // Standard Low Latency Mode
    //get ready for next TX
    nRF_CONFIG(TX_SHORT_CONFIG_BYTES,TX_SHORT_CONFIG_BYTES_LENGTH);
    //config leaves nRF in standby
    DATA_PACKET[0]=ID; //set packet header to node ID
    //sample IMU and pressure sensors
    sample_data(); //100us to complete conversions with capacitive sensing on
    LedDriver(2,2,0);
    //Process Message
    if(RX_MESSAGE[1] == 0xFF || RX_MESSAGE[1] == ID){
      //we have a message
      if(RX_MESSAGE[2] == 0x01){
        LedDriver(RX_MESSAGE[3],RX_MESSAGE[4],RX_MESSAGE[5]);
      }
    }
```

```
}


/**
* Increase channel, deal with wraparound, and set all configuration bytes for
* the radio properly.
**/
void nRFSetupChannelBytes(int increaseChannel){
  unsigned char configByte;
  if(increaseChannel) {
    if(current_channel == nRF_BASE_CHANNEL + nRF_CHANNEL_COUNT)
      current_channel = nRF_BASE_CHANNEL;
    else
     current_channel++;
  }

  configByte = current_channel << 1;

  //assume that we are padded with 0's
  TX_CONFIG_BYTES[14] = configByte;
  TX_SHORT_CONFIG_BYTES[0] = configByte;

  //flip bit for RX mode
  configByte |= 0x01;
  RX_CONFIG_BYTES[14] = configByte;
  RX_SHORT_CONFIG_BYTES[0] = configByte;
}

__interrupt void P1_interrupt(void);
PORT1_ISR(P1_interrupt)
//Crossworksvoid P1_interrupt(void) __interrupt[PORT1_VECTOR] {
__interrupt void P1_interrupt(void) {
  _DINT();


  if((P1IFG & BIT4) != 0)//{
    nRF_data_waiting();
  //} else if((P1IFG & BIT1) != 0){ //Button pressed
  //    P1IFG &= ~BIT1; //clear interrupt
  //    isButtonClicked = 1;
  //}
  else
    ERROR();

  _EINT();
}



void start_timer_B(void){
  TBCTL |= BIT4;

  //disable radio interrupts.
  P1IE &=  ~BIT4;     //disable interrupts for the NRF
  P1IFG &= ~BIT4;     //clear any pending

}
```

```
; ****************************************************************************
; File:     Ext_Intrinsics.asm
; Author:   Bhargavi Nisarga, Texas Instruments Inc
; Date:     July 2007
;
; Routines to access the upper memory above 64-KB data area
; Accessed out of C as standard extern Calls.
;
; These assembler functions are mainly to used to access the upper memory above
; 64 KB as a data area or access the 20-bit SFRs in lower 64 KB.
;
;   Note: This assembler file should be included in the project while exceuting
;   C programs which interface the global functions listed in this
;   assembler file.
;
; ****************************************************************************
   .global __data20_read_char
   .global __data20_read_short
   .global __data20_write_char
   .global __data20_write_short
            .global __data16_write_addr
            .global __data16_read_addr

;============================================================================
; "unsigned char __data20_read_char(unsigned long __addr)"
; This function is used to access the upper memory above 64-KB as a data area.
; An unsigned char present at the upper memory above 64 KB is read.
; Passing parameter : unsigned long __addr - the 20-bit address is stored in long
;     integer as two words; lower word in R12 and upper word in R13
; Return parameter  : unsigned char - the 8-bit char return value is the data
;                   present at the 20-bit address location is returned in R12.
;============================================================================
      .text
__data20_read_char:
      push.w  R13   ;upper word
      push.w R12   ;lower word
      popx.a R13   ;20-bit address
      ;Move 8-bit char value at the 20-bit addr locn to R12
      movx.b 0x0(R13), R12 ;R12 contains the return value
      reta    ; return

;============================================================================
; "unsigned short __data20_read_short(unsigned long __addr)"
; This function is used to access the upper memory above 64-KB as a data area.
; An unsigned short interger present at the upper memory above 64 KB is read.
; Passng parameter : unsigned long __addr - the 20-bit address is stored in long
;       integer as two words; lower word in R12 and upper word in R13
; Return parameter : unsigned short - the 16-bit return value is the data
;                  present at the 20-bit address location is returned in R12.
;============================================================================
      .text
__data20_read_short:
      push.w  R13   ;upper word
      push.w R12   ;lower word
      popx.a R13   ;20-bit address
      ;Move 16-bit short value present at the 20-bit address
      ;(in R13) location to R12
      movx.w @R13, R12 ;R12 contains the return value
      reta    ; return

;============================================================================
; "void __data20_write_char(unsigned long __addr, unsigned char __value)"
; This function is used to access the upper memory above 64-KB as a data area in
; address order to write a 8-bit char value into the extended memory whose 20-bit
; is passed to the function as a 32-bit long integer.
; Passing parameter : unsigned long __addr - the 20-bit address location to which
;      a char value is to be written is stored in long integer as
;      two words; lower word in R12 and upper word in R13;
;      unsigned short __value - the 8-bit char value stored in
;                   R14, is the data that is to be written into the 20-bit
;      address location.
; Return parameter  : Void
;============================================================================
      .text
__data20_write_char:
      push.w R13   ;upper word
      push.w R12   ;lower word
      popx.a R13   ;20-bit address
      ;Write the 8-bit char value in R14 onto the 20-bit
      ;address (in R13) location
      movx.b R14,0x0(R13)
      reta    ; return

;============================================================================
; "void __data20_write_short(unsigned long __addr, unsigned short __value)"
; This function is used to access the upper memory above 64-KB as a data area in
; order to write a 16-bit short integer into the extended memory whose 20-bit
; address is passed to the function as a 32-bit long integer.
; Passing parameter : unsigned long __addr - the 20-bit address location to which
;      a char value is to be written is stored in long integer as
;      two words; lower word in R12 and upper word in R13;
;      unsigned char __value - the 16-bit char value stored in R14,
;                   is the data that is to be written into the 20-bit addr locn.
; Return parameter  : Void
;============================================================================
      .text
__data20_write_short:
      push.w R13   ;upper word
      push.w R12   ;lower word
      popx.a R13   ;20-bit address
      ;Write the 16-bit short value in R14 onto the 20-bit
      ;address (in R13) location
      movx.w R14,0x0(R13)
      reta    ; return
;============================================================================
; "void __data16_write_addr(unsigned short __addr, unsigned long __value)"
; This function is used to access 20-bit SFRs in lower 64 KB. An unsigned long
; value having 20-bit data can be written into the lower 64-KB memory location
; that can store a 20-bit value.
; Passing parameter : unsigned short __addr - the 16-bit address location, in
;                   the lower 64-KB memory space that can store a 20-bit value
;     is stored in register R12;
;     unsigned long __value - the long integer contains the 20-bit
;     value in two words; lower word in R13 and upper word in R14.
; Return parameter  : Void
;============================================================================
      .text
__data16_write_addr:
      push.w  R14   ;upper word
      push.w R13   ;lower word
      popx.a R14   ;20-bit address
      ;Move the 20-bit address in R14 to a 16-bit address
      ;(in R12) location, i.e. move a 20-bit value to a
      ;20-bit SFR in the lower 64-KB memory location
      movx.a R14, 0x0(R12)
      reta    ; return
;===============================================================================
; "unsigned long __data16_read_addr(unsigned short __addr)"
; This function is used to access 20-bit SFRs in lower 64 KB. A 16-bit unsigned
; short address that contains 20-bit address value is read.
; Passing parameter : unsigned short __addr - the 16-bit address location of the
;                   20-bit SFR , which can stores a 20-bit value, in the lower
;     64-KB memory space is stored in register R12
; Return parameter  : unsigned long __value - the 20-bit value that was present
;     at the 16-bit address location in the lower 64 KB memory, is
;     returned an a long integer contained in two words; lower
;     word in R12 and upper word in R13.
;===============================================================================
      .text
__data16_read_addr:
      ;R12 has the 16-bit address of a 20-bit SFR
      pushx.a @R12   ;push 20-bit addr to stack
      ;pop as two words
      pop.w R12   ;lower word in R12
      pop.w R13   ;upper word in R13
      reta    ; return

      .end    ; End of assembler code
```

```
//flash.c
//Handle the on chip flash memory
//Configuration is handled in initNode

#include <msp430x26x.h>

#ifndef flash
#include "flash.h"
#define flash



//start mtl flash code
unsigned long flashPtr = (unsigned long) UPPER_FLASH_TOP;


//extern functions to do extended memory stuff on msp4302618
unsigned char __data20_read_char(unsigned long __addr);
void __data20_write_char(unsigned long __addr, unsigned char __value);

/**
* DATA_PACKET[0] == node id, dont write
* DATA_PACKET[1-9] == A/D data from sensors
* DATA_PACKET[10-12] == low g accell X, Y, Z
* DATA_PACKET[13-14] == Compass heading
* OLD - OLD --DATA_PACKET[10-11] == Timestamp RX'd at 100Hz from basestaion
**/
unsigned char write_sample_to_flash(unsigned char packet[], int writeLowG,
          int writeCompass){
  //ASSUME INTERRUPTS ARE DISABLED
  //dont write node id
  char * addr;
  int i;
  if(flashPtr-SAMPLE_SIZE > (unsigned long)UPPER_FLASH_BASE){
 //OK to write sample to upper flash
 FCTL3 = FWKEY; //unlock
    FCTL1 = FWKEY + WRT;

    for(i = 1; i <= SAMPLE_SIZE; i++){
      __data20_write_char(flashPtr, packet[i]);
      while(FCTL3 & BUSY);
      flashPtr--;
    }
    FCTL1 = FWKEY;
    FCTL3 = FWKEY + LOCK; //lock

  } else if(flashPtr-SAMPLE_SIZE < (unsigned long)UPPER_FLASH_BASE &&
   flashPtr-SAMPLE_SIZE > (unsigned long)LOWER_FLASH_TOP)
   flashPtr = (unsigned long)LOWER_FLASH_TOP;


  //ensure we wont overwrite the bottom of our alowed flash
  if((flashPtr-SAMPLE_SIZE) > ((unsigned long)LOWER_FLASH_BASE) &&
   flashPtr-SAMPLE_SIZE < (unsigned long)LOWER_FLASH_TOP) {
   FCTL3 = FWKEY; //unlock
    FCTL1 = FWKEY + WRT;

    for(i = 1; i <= SAMPLE_SIZE; i++){
      addr = (char *) flashPtr;
      *addr = packet[i];
      while(FCTL3 & BUSY);
      flashPtr--;
    }
    FCTL1 = FWKEY;
    FCTL3 = FWKEY + LOCK; //lock
  } else if(flashPtr < LOWER_FLASH_TOP)
    return FLASH_FULL;

  return 0;
}

extern char DATA_PACKET[];

unsigned char read_sample_from_flash(unsigned long addr){
  int i;
  char * shortAddr;

  if(addr-SAMPLE_SIZE > (unsigned long) UPPER_FLASH_BASE){
 //this sample is in the upper flash area.
 for(i = 0; i < SAMPLE_SIZE; i++)
  DATA_PACKET[i+4] = __data20_read_char(addr-i);
  } else if((addr-SAMPLE_SIZE) > (unsigned long)LOWER_FLASH_BASE) {
 //lower flash area
   shortAddr = (char *) addr;
   for(i =0; i < SAMPLE_SIZE; i++)
    DATA_PACKET[i+4] = *(shortAddr - i);
  } else
   return FLASH_END;

  return (unsigned char) 0x00;
}


void erase_flash_segment(char * addr){
  if(LOWER_FLASH_BASE <= (unsigned int)addr && (unsigned int)addr <= LOWER_FLASH_TOP){
    FCTL1 = FWKEY + ERASE;
    FCTL3 = FWKEY;
    *addr = 0;
    //Ryans code didnt do this....
    while(FCTL3 & BUSY);
    FCTL3 = FWKEY + LOCK;
  }
 }

void erase_upper_flash_segment(unsigned long longAddr){
 if(UPPER_FLASH_BASE <= longAddr && longAddr <= UPPER_FLASH_TOP){
  FCTL1 = FWKEY + ERASE;
  FCTL3 = FWKEY;
     __data20_write_char(longAddr, (unsigned char) 0);
     //Ryans code didnt do this....
     while(FCTL3 & BUSY);
     FCTL3 = FWKEY + LOCK;
 }
}

#endif
```

```
//flash.h

void erase_flash_segment(char *);
void _erase_upper_flash_segment(unsigned long);
unsigned char write_sample_to_flash(unsigned char [], int, int);
unsigned charread_sample_from_flash(unsigned long);

//if changing this change mainRedSoxNode.h too!!
#define FLASH_FULL 0xBD
//if changing this change flash.h too
#define FLASH_END 0xB1

#define LOWER_FLASH_TOP  0xFDFF
#define LOWER_FLASH_BASE  0x5000     //MSP430F148
//#define FLASH_BASE 0x2000     //MSP430F149

//MSP430F2618 has its main memory divided by the interrupt vector in the middle, lame
#define UPPER_FLASH_TOP 0x1FFFF
#define UPPER_FLASH_BASE 0x10000

#define UPPER_SEGMENT_COUNT ((UPPER_FLASH_TOP-UPPER_FLASH_BASE)/512)
#define LOWER_SEGMENT_COUNT ((LOWER_FLASH_TOP-LOWER_FLASH_BASE)/512)
#define SAMPLE_SIZE 11              //11 byte samples will be written to flash

/**** Flash structure  F149
see output of compiler to see where to place flash data and where te code is writen
****/
```

Appendix C

Camera Synchronization

In order to facilitate validation of future sportSemble work, namley joint angle calculations and any 3D representation driven by sportSemble data, it was necessary to synch sportSembles data stream with a data stream from a known system, XOS Technologies Sport Motion was the chosen system. XOS was kind enough to augment on of thier cameras with a BNC adapter that carried a digital signal out of thier system. The signal was synchronous with the shutters on their high speed cameras, it would be raised high for the duration of the open shutter and would be held low when the shutter was closed. Counting these pulses is what enables synchronization.

A standard coaxial cable was run from the BNC connector on the camera to a second BNC connector on the sportSemble basestaion. The shielding of the cable was connected to ground, and the center wire was connected to a digital I/O pin on the basestation's microcontroller.

The basestation's firmware used a counter to count the pulses from XOS's camera, it also managed resetting, recording, and sending counter values to the host PC. The counter behaved as follows; with any *FLASH_ERASE* command it was reset, remember once it was reset digital pulse that came in was recorded and the counter was incremented by 1. These are analagous to individual frames in video data that XOS can provide. When a

SAMPLE_2_FLASH command is sent the value of the counter is stored in memory, this is the frame number in an XOS video file that is equal to the start time of sportSembles data recording. The counter continues to be incremented while bothe systems record data. When a *FLASH_FULL* packet is recieved from a sportSemble node the counter value is also stored in memory. This second counter value denotes what frame number in the XOS video marks the end of sportSembles data stream. Once this occurs the basestation sends both of the counter values to the PC host application and they are recorded in a file.

Knowing how many frames into the video sportSemble started recording, and how many frames into the video sportSemble stopped recording allows for synchronization of the tho data sets.

There is on caveat, before a new video is started in the XOS system the cameras shutters are opened and closed several times with no frames being recorded to the video file. This problem is simply worked around as the number of unrecorded frames is always the same, 226. When doing any data synchronization it is necessary to subtract 226 from both recorded counter values.

Bibliography

[1] Werner SL, Murray TA, Hawkins RJ, et al. Relationships between throwing mechanics and shoulder distraction in professional baseball pitchers. In *American Journal of Sports Medicine*, 2001. 12, 24, 25, 96, 97, 99

[2] Fleisig G, Escamilla R, Andrews J. Biomechanics of Throwing. In: Zachazewski J, Magee D, Quillen W, eds. Athletic injuries and rehabilitation. In *W.B. Saunders Company*, 1996. 12, 97

[3] Tullos HS, King J. Throwing mechanism in sports. In *Orthopedic Clinic of North America; 4*, pages 709–720, 1973. 18

[4] Conte S, Requa RK, Garrick JG. Disability days in major league baseball 1989-1999. In *American Journal of Sports Medicine; 29(4)*, pages 431–436, 2001. 18

[5] Fleisig GS, Kingsley DS, Loftice JW, et al. Kinetic comparison among the fastball, curveball, change-up, and slider in collegiate baseball pitchers. In *American Journal of Sports Medicine*, 2005. 18, 24, 96, 99

[6] XOS Technologies. Xos technologies. http://www.xostech.com/. 20

[7] Motion Analysis Inc. Motion analysis. http://www.motionanalysis.com/. 20

[8] Berkson E., Aylward R., Zachazewski J., Paradiso J., Gill T.J. Imu arrays: The biomechanics of baseball pitching. In *The Orthopaedic Journal at Harvard Medical School, Vol. 8*, pages 90–94, June 2007. 20, 22, 96

[9] M.S. Grewal A.P. Andrews. *Kalman Filtering Theory and Prectice Using MATLAB, Second Edition*. McGraw-Hill, 2001. 20, 22

[10] Qualisys AB. Qualisys motion capture systems. http://www.qualisys.com/. 21

[11] Ramesh Raskar, Hideaki Nii, Bert deDecker, Yuki Hashimoto, Jay Summet, Dylan Moore, Yong Zhao, Jonathan Westhues, Paul Dietz, John Barnwell, Shree Nayar, Masahiko Inami, Philippe Bekaert, Michael Noland, Vlad Branzoi, Erich Bruns . Prakash: lighting aware motion capture using photosensing markers and multiplexed illuminators. In *ACM SIGGRAPH 2007: International Conference on Computer Graphics and Interactive Techniques*, 2007. 21

[12] Ryan Aylward. Sensemble: A Wireless Inertial Sensor System for Interactive Dance and Collective Motion Analysis. Master's thesis, Media Laboratory, Massachusetts Institute of Technology, 2006. 22, 27, 29, 30, 39, 55

[13] Bamberg, S.J.M., Benbasat A.Y., Scarborough D.M., Krebs D.E., Paradiso J.A. Wearable wireless sensor network to assess clinical status in patients with neurological disorders. In *Proceedings of the 6th international conference on Information processing in sensor networks*, pages 563–564, 2007. 22

[14] Stacy J. Morris, Joseph A. Paradiso. Shoe-integrated sensor system for wireless gait analysis and real-time feedback. In *Proceedings of the 2nd Joint IEEE EMBS (Engineering in Medicine and Biology Society) and BMES (the Biomedical Engineering Society) Conference*, pages 2468–2469, October 2002. 22, 70

[15] Ari Y. Benbasat. An Inertial Measurement Unit for User Interfaces. Master's thesis, Media Laboratory, Massachusetts Institute of Technology, 2000. 22

[16] A. Benbasat, J. Paradiso. Compact, configurable inerital gesture recognition. In *Proceedings of the ACM CHI 2001 Conference - Extended Abstracts*, pages 183–184, 2001. 22

[17] Daniel Vlasic, Rolf Adelsberger, Giovanni Vannucci, John Barnwell, Markus Gross, Wojciech Matusik, Jovan Popovic. Practical motion capture in everyday surroundings. In *ACM Transactions on Graphics 26(3), Article 35.*, page Article 35, 2007. 22

[18] Daniel Vlasic, Jovan Popovic. Mit computer science and artificial intelligence labratory. In *CSAIL Research Abstracts 2007*, page http://publications.csail.mit.edu/abstracts/abstracts07/drdaniel_2007/drdaniel.html, 2007. 22

[19] Buddhika de Silva, Anirudh Natarajan, Mehul Motani, Kee-Chiang Chua. A real-time feedback utility with body sensor networks. In *5th International Workshop on Wearable and Implantable Body Sensor Networks*, pages 49–53, June 2008. 22

[20] Kwang Yon Lim, Wei Dong, Francis Young Koon Goh, Kim Doang Nguyen, I-Ming Chen, Song Huat Yeo, Henry Been Lirn Duh. A preliminary study on the accuracy of wireless sensor fusion for biomotion capture. In *5th International Workshop on Wearable and Implantable Body Sensor Networks*, pages 99–102, June 2008. 22

[21] W. Yeoh, J. Wu, I. Pek, Y. Yong, X. Chen, A.B. Waluyo . Real-time trackign of flexionangle by using wearable accelerometer sensors. In *5th International Workshop on Wearable and Implantable Body Sensor Networks*, pages 125–128, June 2008. 22

[22] A.D. Young, M.J. Ling, D.K. Arvind. Orient-2: A realtime wireless posture tracking system using local orientation estimation. In *4th Workshop on Embedded Networked Sensors: EmNets 2007*, pages 53–57, November 2006. 23

[23] E.R. Bachman. Inertial and magnetic trackign of limb segment orientation for insertign humans into synthetic environments. In *PhD disertation, Naval Postgraduate School, Monteray, California*, 2000. 23

[24] D. Fontaine, D. David, Y. Caritu. Sourceless human body motion capture. In *Smart Objects Conference*, May 2003. 23

[25] MiniSun LLC . Intelligent device for energy expenditure and physical activity. http://www.minisun.com/. 23

[26] Lorincz K., Kuris B., Ayer S.M., Patel S., Bonato P., Welsh M. Wearable wireless sensor network to assess clinical status in patients with neurological disorders, cambridge ma. In *International Conference on Information Processing in Sensor Networks*, pages 413–423, April 25-27, 2007. 23

[27] Measurand Inc. http://www.motion-capture-system.com/shapewrap.php. 23

[28] Polhemus. http://www.polhemus.com/. 23

[29] Ascension Technology Corporation. http://www.ascension-tech.com/. 23

[30] Roetenberg, D. Slycke, P. J. Veltink, P. H. Ambulatory position and orientation tracking fusing magnetic and inertial sensing. In *Biomedical Engineering, IEEE Transactions on*, pages 883 – 890, May 2007. 24

[31] Xsens Technologies B.V. Xsens motion technologies. http://www.xsens.com. 24

[32] Biometrics Ltd UK. http://www.biometricsltd.com/. 24

[33] Koon Kiat Teu, Wangdo Kim, Franz Konstantin Fuss. Using dual number method for motion analysis of left arm in a golf swing. In *Proceedings of the 2004 ACM SIGGRAPH international conference on Virtual Reality continuum and its applications in industry*, pages 217–220, June 2004. 24

[34] T.W. Calvert, Jo Chapman, A. Patla. The integration of subjective and objective data in the animation of human movement. In *ACM SIGGRAPH Computer Graphics, Volume 14, Issue 3*, pages 198–203, 1980. 24

[35] Werner SL, Gill TJ, Murray TA, et al. Relationship between throwing mechanics and elbow valgus in professional baseball pitchers. In *Journal of Shoulder and Elbow Surgery*, 2001. 25

[36] Jacob Millman and Arvin Grabel. *Microelectronics; 2nd Edition*. McGraw-Hill, 1987. 28, 47

[37] NXP Semiconductors Inc. http://www.standardics.nxp.com/support/i2c/usage/. 28, 31, 45, 49

[38] Institute of Electrical Engineers. http://ieee802.org/15/pub/TG4.html. 28

[39] Jose Gutierrez, Ed Callaway and Raymond Barrret. *Low-Rate Wireless Personal Area Networks: Enabling Wireless Sensors with IEEE 802.15.4*. IEEE Press, 2004. 28

[40] David Kalinsky and Roee Kalinsky. Introduction to serial peripheral interface. 28, 45

[41] Jenifer Bray and Charles F Sturman. *Bluetooth: Connect Without Cables*. Prentice Hall, 2001. 28

[42] Eric Berkson. Memo from eric berkson 2/20/2006. 29

[43] Analog Devices Inc. http://www.analog.com/en/mems-and-sensors/imems-gyroscopes/adxrs300/products/product.html. 30, 69

[44] Harvey Weinberg. AN-625: Modifying the Range of the ADXRS150 and ADXRS300 Rate Gyro. http://www.analog.com/static/imported-files/application_notes/5913552204290679037672688l9AN625_0.pdf. 30, 69

[45] Analog Devices Inc. ADXL193: Single-Axis, High-g, iMEMS Accelerometers. http://www.analog.com/en/mems-and-sensors/imems-accelerometers/ADXL193/products/product.html. 30

[46] Analog Devices Inc. http://www.analog.com/en/mems-and-sensors/imems-accelerometers/adxl210/products/product.html. 31

[47] STMicroelectronics NV. http://www.st.com/stonline/products/literature/ds/12726/lis302dl.htm. 31

[48] Honeywell. 3-Axis Digital Compass Module HMC6343. http://www.ssec.honeywell.com/magnetic/datasheets/HMC6343.pdf. 31, 32

[49] Nordic Semiconductor. nRF2401A Transceiver . http://www.nordicsemi.com/index.cfm?obj=product&act=display&pro=64#. 33, 34, 57

[50] Texas Instruments. Msp430f148 mixed signal microcontroller. http://focus.ti.com/docs/prod/folders/print/msp430f148.html. 35

[51] Chris Nagy. *Embedded Systems Design using the TI MSP430 Series*. Newnes, 2003. 35

[52] Jerry Luecke. *Analog and Digital Circuits for Electronic Control System Applications Using the TI MSP430 Microcontroller*. Newnes, 2005. 35

[53] Texas Instruments. Msp430f2618 mixed signal microcontroller. http://focus.ti.com/docs/prod/folders/print/msp430f2618.html. 35, 42, 45

[54] Texas Instruments. Plastic quad flatpack no pins. http://focus.ti.com.cn/cn/lit/ml/mpqf141a/mpqf141a.pdf. 35

[55] Texas Instruments. Plastic quad flatpack with pins. http://focus.ti.com.cn/cn/lit/ml/mtqf008a/mtqf008a.pdf. 35

[56] John Catsoulis. *Designing Embedded Hardware; 2nd Edition*. O'Reilly, 2005. 39, 42

[57] Silicon Laboratories Inc. Usbxpress. https://www.silabs.com/products/mcu/Pages/USBXpress.aspx. 52

[58] Microsoft. Dynamic-link libraries. http://msdn.microsoft.com/en-us/library/ms682589.aspx. 52

[59] Jenifer Bray and Charles F Sturman. *Bluetooth: Connect Without Cables*. Prentice Hall, 2001. 55

[60] Dassault Systmes SolidWorks Corp. http://www.3dsystems.com/products/invision/index.asp. 72

[61] 3D Systems. Invision 3d modeler. http://www.solidworks.com/. 72

[62] Richard T. Weidner and Robert L. Sells. *Elementary Classical Physics, Second Edition, volume 1*. Allyn and Bacon, 1973. 78

[63] Vadim Gerasimov. Swings That Think. http://vadim.oversigma.com/stt/bat.html. 83

[64] Dillman CJ, Fleisig GS, Andrews JR. Biomechanics of pitching with emphasis upon shoulder kinematics. In *Journal of Orthopedic Sports Physical Therapy*, pages 402–408, Aug 1993. 96, 99

[65] Motion Lab Systems. C3D File Format. http://www.c3d.org/. 106