A System for Real-Time Gesture Recognition and Classification of Coordinated Motion

by

Steven Daniel Lovell

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of Bachelor of Science in Electrical Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology

February 2005

Auth ....
Department of Electrical Engineering and Computer Science
uary 28, 2005

Certified by.............. ....
h A. Paradiso
Sony Career Development Prof. of Media Arts and Sciences
dia Lab
pervisor

Accepted by.......... ....
[illegible] C. Smith
Chairman, Department Committee on Graduate Students

BARKER

A System for Real-Time Gesture Recognition and Classification of Coordinated Motion

by

Steven Daniel Lovell

Submitted to the
Department of Electrical Engineering and Computer Science

January 28, 2005

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Electrical Science and Engineering and Master of Engineering
in Electrical Engineering and Computer Science

Abstract

This thesis describes the design and implementation of a wireless 6 degree-of-freedom inertial sensor system to be used for multiple-user, real-time gesture recognition and coordinated activity detection. Analysis is presented that shows that the data streams captured can be readily processed to detect gestures and coordinated activity. Finally, some pertinent research that can be pursued with these nodes in the areas of biomotion analysis and interactive entertainment are introduced.

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

Acknowledgments

Thanks to Joe Paradiso and the Responsive Environments Group for their guidance and advice during my time here at MIT.
Thanks to all my friends who kept me going during the long hard slog.
Thanks to MIT Project ORCA, the world's best AUV team ever.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Embedding inertial sensors in wearable systems has become a dominant theme in at-home medical monitoring and state-recognition systems for ubiquitous computing. These systems usually find the context or activity of the user, such as walking, walking up steps, or lying down through data logging and post processing. Recently, there has been significant work in using data from multiple points on a users body to determine user activity. However, little work has been done on detecting activity in real time from an ensemble of wireless sensors, either on a single person or on multiple people.

This thesis covers the design of the hardware for a many-node wireless inertial sensor-based real-time activity detection system and presents examples of its utility for coordinated activity detection. In this chapter, I will discuss the possible applications of this system, previous work in activity detection, and the goals of my work. In the following chapters are a description of the hardware, a description of the communications protocol and related firmware, data and results from the system in action, and finally some future directions of investigation.

1.1 Potential Applications

A simple example of the potential of coordinated activity detection was developed by Lester, Hanaford and Borriello in 2003[15]. Lester used a two node system that sensed acceleration in three axes. His goal was to determine if the two nodes were carried by the same person. This is perhaps the simplest form of coordinated activity detection: two signals are assessed for similarity. In his case, the magnitude of the acceleration vector was extracted from the node data and a coherence function was used to asses their similarity over frequencies from 0-10Hz. They restricted the sensed objects to be within the same belt-pack to simplify the analysis. Lester asserts that as the number of devices that people carry grow so will the effort necessary to coordinate them, and that therefore this type of detection is a necessary precursor to having the devices coordinate themselves by determining what other devices the person is carrying.

In 2003, Hinckley used coordinated activity to allow users to interact in new ways with tablet PCs[10]. This represents an interesting attempt at coordinated gestures: gestures that require two nodes to take complimentary actions for the gesture to be complete. Users could bump two tablet PCs together to tile their displays. The coordinated gesture is sensed through the accelerations (equal in magnitude and opposite in direction) of the two tablets. Hinckley argues that though we are developing new tools for computation, we are not developing new ways of communicating information between them. His synchronous bumping gesture is a first step at developing new ways for devices and people to interact and transfer information using physical gestures.

These two previous examples are solid efforts at coordinated activity detection. However, they do not allow for very expressive inputs. Lester's application does not explore the use of the 3-axis accelerometer as a control interface, and Hinckley's interface is limited to tilting and bumping the tablet in the plane in which it lies. These devices lack the expressive capability of devices with more degrees of freedom. Merrill developed a device with many more degrees of freedom and therefore much more expressive capability[17]. With this device he has shown real-time, single example gesture learning to be feasible for a single node system and created an adaptive musical interface with it. This thesis will explore such an interface with many nodes.

Some target applications of this system include interactive dance, performance analysis in teams sports, interactive physical therapy, and full-body biomotion analysis with networked wireless nodes. In prior work, a French group studying automated annotation of contemporary ballet moves used a vision system to determine the dance move performed by a dancer[6]. A multi-node wearable wireless sensing system has potential to make similar analysis easier to conduct by making the data analysis simpler: occlusions of the objects being tracked are not a factor and less infrastructure needs to be mounted on the wearer than for precision vision-based gesture systems[21]. A non-video data source would allow the data to be captured anywhere, rather than in a controlled environment set up for video capture. Also, a multi-node sensing system would more readily allow multi-dancer interactive performances. Paradiso's Dance Shoe was an early example of a rich multi-sensor wireless wearable node for interactive dance[19]. However, it only supported two nodes, one for each foot of a single dancer. This system will allow performances like those done with Paradiso's Dance Shoe to be multi-user, and could be used to create a laboratory for collaborative performances with nodes mounted at many places on the body and to begin to develop principles for collaborative interfaces.

Sensing coordinated activity in sports has begun to be explored by a group studying basketball teams[11]. Using a vision system for data capture, player movements were tracked to detect maneuvers. The data were analyzed for the probability that a maneuver occurred. They found that the probability that a maneuver occurred, as determined by the data, was a strong indicator of the success of the maneuver, if it did in fact occur. Thus, the probability that the maneuver occurred can be viewed as a measure of how well the maneuver was performed. This shows that team performance could be evaluated with such a system and that it could be used to assist teams in training. This system was able to track absolute position and velocity with reasonable accuracy (.6m and .2m/s respectively). However, whether more precise tracking could improve maneuver detection remains to be seen. Indeed, for sports such as rowing, where the environment is not suited to a fixed camera setup, absolute position is not as important, and the athlete motion is much more nuanced, inertial sensing would be far better.

These possible applications show the power that a wireless inertial sensing platform might have: such a system could detect an individual's body gestures in a variety of settings, simplifying current data collection or allowing data collection in new settings that current systems can't accommodate. This system could also detect rich coordinated activity such as the motions of interactive dance troupe with many dancers or team sports performance.

1.2 Prior Systems

In the past, most research in activity detection[1] systems has been on gesture recognition systems that used video input or magnetic trackers. Such systems detect hand gestures for sign language input[16], a pointing arm for commands[13], or the entire body for motion capture[6]. This task is sometimes facilitated by attaching markers to the body of the subject as reference points to allow the gesture to be tracked more precisely. Recently, however, wearable sensors are beginning to be used as the data source for these types of systems[8, 1, 12, 22, 23, 20, 5]. Technological innovations have enabled dense, multi-sensor systems to become truly wearable[2].

An oft cited example of an inertial-sensor-based system that replaces the need for a camera as the data source is the Acceleration Sensing Glove (ASG)[20]. The ASG is a device that uses many accelerometers to detect activity of a single user's hand. It uses an accelerometer on each finger tip on a glove to detect the orientation of the fingers. This gives information about the hand gesture being performed. This device was used as an input device, like a keyboard, with each hand gesture corresponding to a character.

Though this device does more than adequately replace the need for a camera as the data source, the manner in which it uses the sensors, to sense static pose, does not leverage the ability of inertial sensor based systems for dynamic gestures. Furthermore, the data the ASG captures could as readily be acquired with bend sensors on the fingers and a single accelerometer to sense palm orientation. The ASG

[1]For the purpose of this discussion, "activity detection" will mean a pattern recognition system that operates on data from human physical motion

[2]Wearable means the users' activity is not significantly changed by wearing the sensor system. This usually implies small and lightweight. It should also imply low power so that the energy source is also lightweight

is not sufficient as a starting point for the design proposed in this thesis for these reasons.

Bao used a multi-node activity detection system, where all the nodes were worn by the same individual to detect the individual's activity for context-aware systems[2]. His tests using this system were able to detect common activities, such as walking, bicycling, and typing, with high probability in real-world settings. The nodes collected and stored their data on on-board storage media for off-loading and post-processing. Bao's study only used 2 axes of acceleration data per node. Though the system is extensible through a daughter board connector, its large size (4"x2") makes it unattractive for a wearable activity detection platform.

These previously-described systems all used accelerometers on one or more axes to detect user activity. Work has also been done using other sensing modalities . Benbasat recently developed an extensible and compact sensing platform called 'The Stack[4],' named so because of its stackable architecture, where boards with different functionality could be stacked on top of each other to add sensing capabilities. This platform was specifically designed for wearable dense sensing. It has cards designed to give it many sensing modalities, including a 6 degree of freedom (DOF)[3] inertial measurement unit (IMU) card and a tactile sensing card that provides bend sensing and pressure sensing capabilities. Morris used this system, modified to be mounted on a shoe, for her work in detecting gait pathologies in Parkinson's Disease patients[18]. This same system was used by Merrill in his adaptive musical interface[17]. The Stack is a possible candidate for the hardware necessary for the sensing capabilities desired. Its data link, however, is too slow to have many nodes streaming data at once and it is somewhat large and heavy to be worn on locations other than the feet. It is however, a good starting point to evaluate what is necessary in a many-node wireless system.

[3]3 axes of linear acceleration sensing and 3 axes of rotational rate sensing

1.3 Goals

The purpose of this system is to be a wearable, wireless, multi-node data capture and gesture recognition system. Accordingly, the design goals are low power, high bandwidth communications, small size, and lightweight realization, with the requirement that the data captured be sufficient to distinguish gestures. High bandwidth communications is required to ensure that many nodes can transfer their data simultaneously, facilitating a real time system response. Small and lightweight are required to make the device wearable so that it does not impede the normal activity of the wearer. Low power is required to give the device a sufficient battery life with a small and therefore lightweight battery.

Because the system is to be not noticed when worn, some of the physical constraints can be quantified by making them comparable to an item worn by many people every day: a watch. Thus small form factor means less than 1in^3, about the size of, or slightly less than many watch faces. Lightweight means less than 150g, the weight of an average digital sports watch.

Sufficiency of the data for gesture recognition is in the context of Benbasat's Stack, and its use by Merrill and Morris, so the design goal will be a 6-DOF IMU and several tactile inputs. The communications link should be able to support the communications requirement of the nodes worn by several people fully outfitted. Thus, at least 25 nodes should be able to stream data simultaneously to a basestation so that data can be collected from 5 people wearing 5 nodes. From these requirements, we start with some initial goals of the communications system: communications link capable of transmitting sensor data (packets of 20 bytes) from 25 nodes at 100 Hz (400kbps total). The battery life should be sufficient such that the frequency of battery changes does not detract from the devices wearability, allowing 4 or 5 hours of continuous use.

Chapter 2

Hardware

2.1 Architecture Overview

In order to meet the aforementioned goals, a compact wireless sensing unit was designed, taking Benbasat's Stack as a starting point. This sensing unit consists of: sensors, signal conditioning, processor, wireless communications link, power supply, and mounting fixture.

2.2 Sensor Selection

Merrill[17] and Morris[18] have both shown 6-DOF IMUs with additional tactile inputs to be sufficient for different levels of gesture recognition when mounted at the hands and feet. Both Merrill and Morris used the sensor stack designed by Benbasat and Morris[4]. Sensors similar to those on Benbasat and Morris' sensor stack were used for this project: a full 6-DOF set of accelerometers and rate gyros with accommodation for pressure sensors. The number of tactile sensors supported on-card has been reduced to make the board smaller, although provisions are made for easy expansion.

There has been some progress in MEMS based inertial sensors since Benbasat's initial work on his 6-DOF IMU and his subsequent sensor stack based IMU, so a reevaluation of the available sensors was carried out. The table 2.1 shows some of the

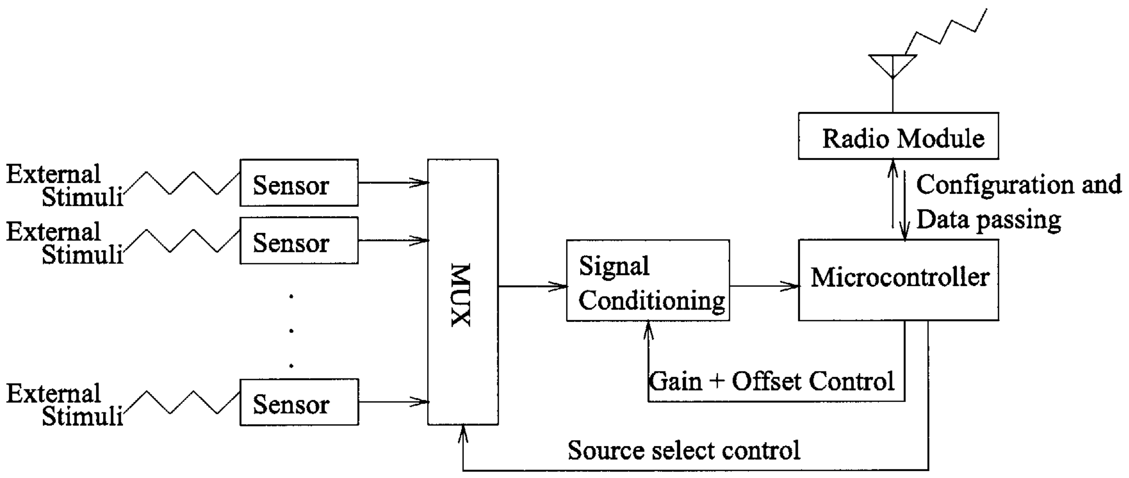


Figure 2-1: Overall System Block Diagram

key performance characteristics of the low cost MEMS accelerometers and gyros that were considered for this design.

2.2.1 Sensor Performance Characteristics

The critical factors of sensor selection for this application are power consumption, bandwidth, sensitivity, number of bits of information, and size. Power consumption is determined by the supply voltage, current draw and turn on time. If the device turn on time is small enough, then power cycling can be used to reduce the power consumption by turning the device off when not in use, as long as the device can be turned on in time to acquire the next sample. For example, if a device has a 10ms turn on time and the desired sampling frequency is 20Hz then the device need only be on 20(10ms + δms) every second, where δ is the time required to take a reading from the sensor.

The bandwidth of the device must be greater than desired bandwidth of the

	Part and Type	Supply Voltage	Current per axis	Axes	Turn on time	Sensitivity	Range	Noise floor	Bandwidth[a]	Bits @ 50 Hz[b]	Size (mm^3)	Cost per axis (US $)
Accelerometers	Analog Devices ADXL202[c]	3 – 5.25 V	0.3 mA	2	17 ms	$\frac{312\,\text{mV}}{\text{g}}$	±2 g[d]	$\frac{200\mu\text{g}}{\sqrt{\text{Hz}}}$	6 kHz	8.5	5x5x2	$7
	Analog Devices ADXL203	3 – 5.25 V	.35 mA	2	20 ms	$\frac{1000\,\text{mV}}{\text{g}}$	±1.7 g	$\frac{110\mu\text{g}}{\sqrt{\text{Hz}}}$	2.5 kHz	11	5x5x2	$10
	MEMSIC MXR2312E	3 – 5.25 V	1.6 mA@5 V	2	100 ms@5 V	$\frac{312\,\text{mV}}{\text{g}}$	±2 g	$\frac{200\mu\text{g}}{\sqrt{Hz}}$	17 Hz	10	5x5x2	$5
	Silicon Devices 1210	5 V	6 mA	1	UNK[d]	$\frac{800\,\text{mV}}{\text{g}}$	±5 g	$\frac{32\mu\text{g}}{\sqrt{\text{Hz}}}$	400 Hz	11	9x9x3	$120
	ST Microelectronics LIS3L02AQ	2.4 – 3.6 V	.3 mA	3	18.5 ms	$\frac{600\,\text{mV}}{\text{g}}$@3 V	±2, ±6	$\frac{50\mu\text{g}}{\sqrt{\text{Hz}}}$	4.5 kHz	Bits	7x7x1.8	$6
	ST Microelectronics LIS3L02DQ	2.7 – 3.6 V	.35 mA	3	50 ms	N/A[e]	±2, ±6	N/A[e]	4.5 kHz	8[f]	7x7x1.8	UNK[g]
Gyroscopes	Murata ENC-03M	2.7 – 5.25 V	4.5 mA	1	UNK[h]	$\frac{.67\,\text{mV}}{°/\text{sec}}$	±300°/sec	0.5°/sec[i] (50 Hz)	50 Hz	10.2	12.2x7x2.6	$80
	Analog Devices ADXRS150	4.75 – 5.25 V	6 mA	1	35*ms*	$\frac{12.5\,\text{mV}}{°/\text{sec}}$	±150°/sec	$\frac{.1°/\text{sec}}{\sqrt{\text{Hz}}}$	40 Hz	7.5	7x7x3.2	$40
	Analog Devices ADXRS300	4.75 – 5.25 V	6 mA	1	35*ms*	$\frac{5\,\text{mV}}{°/\text{sec}}$	±300°/sec	$\frac{.05°/\text{sec}}{\sqrt{\text{Hz}}}$	40 Hz	7.5	7x7x3.2	$40

[a] $3dB$ point
[b] $\log_2(\frac{2.5\times\text{Noise floor}\times\sqrt{50}}{Range})$
[c] Digital and analog outputs
[d] ±10 g version also available
[d] Value not given in datasheet
[e] Sensor has digital output
[f] assumes device has sufficiently low noise floor so all 8 output bits are valid
[g] Device going through redesign as of 12/04, price not given
[h] Value not given in datasheet
[i] Not given on datasheet; value as measured by Benbasat[3]

Table 2.1: Overview of available inertial components.

phenomenon to be sensed. Sensitivity and number of bits of information together determine how much information can be obtained from a device. The number of bits is related to the logarithm of the the dynamic range, $\#Bits = log_2(DR) = log_2(\frac{Range}{NoiseFloor})$. Though the dynamic range determines the theoretical limit of the information content achievable for a device, the sensitivity will effect how much of that information you can actually extract. For example, if all the bits of information in the sensor range span only one bit of the A/D because of low sensitivity, you would not be able to extract any information from the sensor. The limitation of the sensitivity can be overcome by using analog signal conditioning, though the dynamic range, and therefore number of bits of information, achievable from the device. Also, the sensor must be small enough so that the overall design can fit within your size constraints.

Benbasat has previously used the Analog Devices ADXL202 for accelerometer sensors and the Murata ENC-O3J for rate gyro sensors in his initial 6-DOF IMU[3]. For his microcontroller stack IMU he used a combination of two ENC-O3Js and one ADXRS150 so that all three gyros could be mounted on a single plane for fabrication simplicity. The Murata performs comparably, perhaps even better than the ADXRS150 (Its datasheet does not contain the information necessary to calculate this information, however, Benbasat has measured the devices by hand and determined it to perform comparably). However, its major limitation is its size. Even the ENC-03M, another rate gyro from Murata that performs comparably but comes in a smaller package, would be too large for this design due to its length. Even the ADXRS150 and ADXRS300 are a bit large; though they are the smallest rate gyro devices that could be found. The largest negative aspect of the ADXRS150/300 is its power consumption. They draw 6mA-8mA at 5V each. And because of their long turn on time, 35ms, they cannot be power cycled to reduce their current draw. The desired sampling rate of 100Hz requires a sample every 10ms. The ADXRS150/300's turn on time of 35ms makes turning it off between samples impossible.

It could be said that choosing the ADXRS over the Murata gyro does not avoid the size issue because the power draw of the ADXRS will cause the device to require a larger energy source. However, the difference between the 40mW which is required by three Murata gyros and the 90mW which is required by three Analog Devices gyros will not make the energy source decision significantly easier, as will be seen in the energy budget section. This fact, along with the fact that the Murata gyros require external signal conditioning circuitry, which will itself draw power and require more board space, makes the Analog Devices gyros the preferred choice for this design.

The main difference between the two Analog Devices gyros is the sensitivity and range. The ADXRS300 has a larger range but a proportionally lower sensitivity[2], giving the ADXRS150 and ADXRS300 the same dynamic range. The ADXRS300 was chosen for its greater range. Though the devices' range can be increased by adding external passive components to the layout, the range can be increased to a maximum of 4x the original range and as the range increases, so does the initial offset and offset drift over temperature. Because these effects are not well documented, it is preferable to use the ADXRS300 if sensing of rotations greater than or equal to $\pm 300^\circ/\sec$ is desired.

For the accelerometers there is a significantly increased range of choices as they represent an established technology with wider incorporation into consumer devices such as cameras and PDAs. Analog Devices, MEMSIC and ST Microelectronics all have compelling, low-cost solutions. Silicon Devices also makes a precision accelerometer, though the per axis cost is an order of magnitude higher than the other solutions listed. The MEMSIC device is very similar to the ADXL202, having the same noise floor, range and sensitivity. However, the ADXL202 has much lower power consumption and turn on time, as well as higher maximum bandwidth. The MEMSIC chip has a lower bandwidth because it exploits a technology based on a relatively slow convection process, whereas Analog Devices accelerometers use MEMS technology. The one advantage of the MEMSIC device is its slightly lower cost.

The Analog Devices ADXL203 is functionally similar to the their ADXL202 : they are both 2 axis accelerometers in a 8-pin LCC package. However, they differ in their performance characteristics. Though slightly higher cost than the ADXL202,

[2]See table 2.1 for specifications

the ADXL203 has greater sensitivity and lower noise floor. This gives the ADXL203 a greater effective number of bits of information. Its main disadvantages over the ADXL202 are its slightly smaller range and slightly higher cost. It has lower maximum bandwidth as well, but its maximum bandwidth is still greater than the signals this design must sense so this is not a problem.

A newer, but less well known device by ST Microelectronics, is the LIS3L02AQ. This device has comparable sensitivity compared to the ADXL203 and a lower noise floor. In addition, it has user selectable range and three axes on one chip. The user selectable range is very attractive for the varying degree of sensitivity possible, as well as the increased range. Most attractive about the device, however, is the three axes of sensing on one chip. This eliminates the need to have a pair of two-axis devices. For a design that did not require rate gyros, this would make it possible to use only one board, eliminating perpendicular daughter boards to sense all 3 axes. Even more promising than the LIS3L02AQ is the more recent LIS3L02DQ. The DQ is a version of the AQ with an A/D built into the device. Communications with the device can occur either through an SPI or an I^2C port (these are both microcontroller serial communications protocols). Again, for a design that did not require rate gyros, this could significantly reduce the overall cost, allowing the use of less expensive microcontrollers without A/D's converters. Unfortunately, the LIS3L02AQ was pre-production throughout most of this project, hence was not not a viable option. The LIS3L02DQ is currently undergoing redesign and is no longer in production. These devices, however, have tremendous potential for future low cost, low power, small form factor designs.

Most pressure sensors are two terminal devices that vary their resistance with applied pressure. Because they are wired to the board through a connector and are not an integral part of the circuit board design, their are not discussed here. For a discussion of pressure sensors, see [18].

Given the above analysis, the selection of sensors chosen was 2x ADXL203 and 3x ADXRS300 and 2x resistive pressure sensors. An expansion port has been provided to allow the user to add circuitry to accommodate additional sensors. The inertial sensors must be mounted orthogonally to each other to sense all three axes. To achieve this, two daughter boards were mounted perpendicular to the main board. This configuration is shown in figure A-7. Considerations for imperfections in mounting orthogonality is discussed in the PC host software chapter.

2.2.2 Communications

To a large degree, the success of a wireless sensor design depends on its communications link. In particular, this design requires that many nodes be able to transmit periodically to a basestation that sends the information to a computer. This requirement implies that the communications link must have high bandwidth and/or a mechanism to assist in arbitrating the communications to efficiently use its bandwidth.

Initially, the radio module from Benbasat's microcontroller stack, the RFM DR3000-1 radio module, was considered for this system. The RFM DR3000-1 module is based on the RFM TR1000 transceiver and comes conveniently populated onto a daughter board with all the associated components necessary for the module. The module's data rate, 115.2kbps, is still adequate for many applications. However, this data rate did not seem quite adequate for a many-node system sampling at the rate needed to recognize human gestures. For example, assume you want to transmit 10 byte-long sensor readings at 100Hz per node. Also assume that because of inefficiencies in your communications protocol, you can only get 50% efficient use of the bandwidth. Then the total number of nodes that you can have transmitting on a 115.2kbps communications link is $\frac{115.2kpbs}{100Hz*10*8bits/(50\%)} = 7.2$. Thus 7.2 nodes can transmit simultaneously on a 115.2kbps link. We would like to have something on the order of 30 or more nodes transmitting simultaneously. An application that might use this many nodes is coordinated activity detection on a group of people, e.g. a dance troupe. Assume one node is placed on a person's hands, feet and chest for a total of 5 nodes per person. Thus fitting 6 people like this would require 30 nodes and require a bandwidth (assuming 50% efficiency) of at least 500kbps. A target communications rate of 1Mbps would comfortably fit this many nodes and allow for future increases in the number of nodes running simultaneously.

There is a wide range of data radio modules available. Laibowitz and Paradiso have surveyed the recent options and tabulated a representative sampling of RF solutions[14]. Currently, devices in the Mbps data rate range are outnumbered by those that are more aptly rated in kbps. 115.2kbps is the max speed of most commodity data links. The devices that are 100's of kbps or more are either rather new or rather expensive. Bluetooth modules have a data rate of 723kbps and have a fully developed communications protocol[9]. However, these devices have higher power consumption and tend to be physically larger than non-Bluetooth data radio modules due to their flexibility. Additionally, the Bluetooth protocol only allows 7 slave devices to be connected which would severely limit this design. ZigBee is a relatively new, moderate data rate protocol that uses a standardized physical layer with a few modules commercially available[7]. Zigbee was designed for mesh networks, where nodes communicate long distances by relaying data through other nodes, and is not suited for the applications desired here. Of the non-Bluetooth devices, the 1Mbps capable devices include the Nordic semiconductor nRF2401, the RF Monolithics TR1100 and the RF Waves RFW102-M. Of these, the nRF2401 was selected for its unique ShockBurst mode, selectable channel frequency, dual receiver design and low cost and small solution footprint.

The nRF2401 has a state machine inside the device that allows the microcontroller to clock data in and out of the device at a rate determined by the microcontroller. This state machine is utilized in ShockBurst mode. It allows the microcontroller to preload data for transmit as well as hold received data until the microcontroller is ready to read it. Both of these options are useful for minimizing power consumption and microcontroller utilization by minimizing communication protocol constraints. The nRF2401 is also capable of direct TX and RX which is the standard mode for radio communications modules, similar to a serial port : bits are be processed as they come out of and go into the device at the communications data rate. While this mode has the disadvantage that received bytes must be processed as they come in or they will be overwritten, it has the advantage that there is no protocol overhead and any required parsing or processing of the raw data could be handled by the host computer.

The nRF2401 has 128 different frequency channels that it can operate at. Thus, the device can allow more than one channel of 1Mbps communication to occur simultaneously, whereas the other 1Mbps devices have no such option. The nRF2401 also has two receivers on the same chip. This allows one receiver to receive data from two devices, such as a wireless mouse and keyboard. This allows multiple nodes to transmit at the same time to the same receiver, allowing the total data received by a single receiver basestation to be 2Mbps, provided that the basestation microcontroller can process the incoming data.

Nordic also makes a device called the nRF24E1. This is an nRF2401 integrated with an 8051 microcontroller core with an A/D. This is an attractive option since the it simplifies and reduces the size of the design by not needing any additional microcontroller interconnect routing is required. Tapia and Intille have successfully used the device to make a very compact 3 axis acceleration sensing device for wearable medical monitoring[24]. The nRF24E1 was an attractive solution, however, the decision was made to go with a stand alone microcontroller for greater versatility and capability in the design.

Additionally, the nRF2401 is the least expensive of all the solutions at approximately $8 for all parts. The solution footprint for the nRF2401 is not set, since they are not sold as modules, but must be laid out by the designer instead. However, the overall solution size is comparable to that of the RFM and RFW modules.

2.3 Microcontroller

The microcontroller chosen for the design was the Texas Instruments MSP430F147. The MSP430 family of microcontrollers from Texas Instruments are 16-bit RISC mixed-signal microcontrollers specifically designed for low power embedded applications such as this one. Its list of advantages include numerous low power sleep modes, hardware peripherals, ample RAM and Flash memory, and a plethora of development tools.

The MSP430 is well suited to this application for a variety of reasons. To minimize power consumption the microcontroller will often go into low power sleep mode. The microcontroller should be able to quickly wake up from external or internal events such as timer overflows (for periodic wake-ups) and external signals (for wakeup from radio upon data packet reception). The processor should also be able to do some data processing for future development in distributed gesture recognition. The MSP430 has a hardware 16 bit multiplier as well as an efficient instruction set where most operations take only 1 clock cycle to execute. The MSP430's main limitation is its 8MHz maximum clock frequency. Most 8051 based microcontrollers, though usually having higher maximum clock rates, usually require multiple clock cycles to execute common instructions. This levels the playing field a bit, but still places the MSP430 at a slight disadvantage unless 16 bit calculations are required, in which case the MSP430 is far superior to 8 bit microcontrollers.

Additionally, the MSP430F147 has a 12-bit A/D converter with internal 2.5V and 1.5V references. Using the 2.5V reference as the upper limit of the voltage conversion (V_{REF+}), one bit of the A/D output value represents .6mV.

2.4 Signal Processing

The analog signal processing section of the board is comprised mainly of a variable offset and gain op-amp stage. All low pass filtering of inertial sensor signals is done using capacitors external to the sensor ICs, as specified on their data sheets. No low pass filtering is done to the pressure sensors.

No external reference was used to reduce the number of components required (and therefore reduce board space and cost of the design). Also, this would limit how low the supply voltage for the microcontroller could go since AV_{CC}, the analog supply voltage, must be at least .15V greater than V_{REF+} for proper operation.

In connecting the inertial sensors to the A/D inputs of the microcontroller, it was possible to simply connect all sensor outputs to the A/D inputs through a resistor divider. The resistor divider is needed to scale the output voltage of the sensors to something in the range of the A/D (0-2.5V). However, this would create two nuisances: 1) all A/D inputs would be occupied and 2) setting the voltage divider would require 16 resistors.

While this design's purpose is not to be an extensible platform, as it is with Benbasat's microcontroller stack[4], having the ability to affix sensors for future development is a desired feature. Therefore occupying all A/D inputs is not preferred. Requiring 16 resistors in the signal path to the A/D inputs would be difficult to lay out and time consuming to populate and change if need be. The resistors might have to be changed if the internal reference need to be switched from 2.5V to 1.5V or vice versa. An alternative to using a fixed resistor divider for each A/D channel is to use a multiplexer, followed by a programmable gain op-amp stage. This way, only one A/D channel is occupied and changing the offset or scale factor requires only changing one resistor. Using an op-amp to buffer the sensor signal before it goes into the A/D is a common practice because sensor outputs tend to be high impedance signals.

Instead of using fixed resistors for the op-amp offset and gain stage, a digital potentiometer was used. This enables the microcontroller to control the offset and gain of the sensor signal. This would be most useful if such a variable offset and gain could be applied to signals going into the A/D ports left open for expansion, however, all inputs to the multiplexer that is fed through the variable offset/gain op-amp are used up. Using a larger multiplexer, or simply another multiplexer, so that more inputs could be passed through the variable offset/gain stage was considered. However, that was left for future consideration and not implemented in this design. The proper conditioning of auxiliary signals is left up to the designer. The conditioned signals can be routed through the terminals labeled 'A2' through 'A4' on the board and are passed straight to A/D inputs of the microcontroller. See figure A-8 for a diagram of the terminals.

2.5 Power Budget

To conserve power, some devices can be shut down when not in use. Thus, when not sampling, the microcontroller and data radio can go into lower power mode and the signal processing electronics and sensors could be shut down. What limits this is the turn-on time of each of the components. Table 2.2 shows a power budget with each of the devices' power consumption, turn-on time and possible % of time spent power cycled/shut down. The long turn on times of the gyros and accelerometers preclude power cycling when sampling continuous human activity at greater than 50Hz.

2.6 Power Supply

The power supply section of the design is constrained by three factors : input voltage, output voltage, and range of current draw. The input voltage is set by the selected battery and will determine what kind of voltage converter must be used (step-up or step-down) as well as limit which voltage converters can be used. Voltage converters typically can only accept a range of input voltages depending on what kind of applications they were designed for. The output voltage and current are requirements of the circuits being powered and also constrain which voltage converters can be used. Some batteries may not be able to provide the required peak current draw. Coin cells, for example, are an attractive source of energy for their high energy density and slim form factor, but have a low maximum current draw, making them insufficient for an application that uses rate gyros (because of their large power consumption).

Linear regulators can usually supply ample current, but waste power because of the ohmic nature of the pass element used to regulate the output voltage. Linear regulators also require that the battery voltage be higher than the required supply voltage. For these reasons, linear regulators are not considered. Switching regulators can not always supply high currents, so attention must be paid that the regulator used can supply the current required by the design's power budget. Also, if a switching regulator is improperly laid out, the sharp switching signals can interfere with on-

Device	Input Voltage Range : Voltage Used	Current Draw	Turn-on Time	% power cycled[a]	Power consumption per device
Texas Instruments MSP430F1497	1.9V-3.6V : 3V	200uA per MHz @ 8MHz	6us	70	1.5mW
Texas Instruments MSP430F1497 ADC	1.9V-3.6V : 3V	.8mA	17ms	0	2.5mW
Nordic nRF2401	1.9V-3.6V : 3V	20mA Rx, 13mA Tx	200us	90	6mW
Texas Instruments TLV2475	2.7V-6V : 5V	600uA/Channel	5us	0	12mW
Analog Devices AD5162	2.7V-5.5V : 3V	6uA	5us	0	18uW
Analog Devices ADG608	3V/5V : 3V	.2uA	100ns	0	1uW
Analog Devices ADXL203	3V-6V : 5V	.7mA	20ms	0	3.5mW
Analog Devices ADXRS300	4.75V-5.25V : 5V	6mA	30ms	0	30mW

[a]X percent power cycled means the device is off X% of the time

Table 2.2: Power Budget.

board radio or sensor signals. For this application, a properly selected and carefully designed switching regulator is ideal.

This design will require at least two different supply voltages because the operating voltage ranges for the data radio and the rate gyros do not overlap. The components can be divided into three classes : components that work in the 1.8V-3.6V range (data radio, microcontroller), components that work at 5V (rate gyros), and components that work within either range (accelerometers, op-amps, mux, digital pot). Therefore, it must be possible to generate these two voltage ranges from whatever battery that is used.

Again, the power consumption of the devices and the desired battery life will determine what battery must be used. Given the rather large power draw of the rate gyros and the data radio, the battery must have considerable capacity. A common choice for batteries is an alkaline cell. They have fairly high capacity and are cheap. Another option would be more exotic, rechargeable cells, such as NiMH and Lithium Polymer cells. The Li-Po cells were considered more attractive because of their flat, thin form factor, very similar to the design's target board size. However, all Li-Po cells that were smaller than the target design size did not have sufficient capacity to be used. The eventual decision was between using a standard AAA alkaline cell, using two small capacity Li-Po cells in parallel, or using a single medium capacity Li-Po cell that was slightly larger than the eventual design size. The larger Li-Po cell was ruled out since it would make the design larger and add to its complexity by requiring a charging circuit and making the power section require two different voltage converters : a step-up converter and a step-down converter. The reason that two different voltage converters would be required is that the cell voltage was between the two desired voltage rails for the 5V sensors and the low voltage data radio and microcontroller. Using two smaller Li-Po cells in parallel was attractive, since they would allow the design to nearly reach the target size of 1" cube. However, they would also still require the two different voltage converters and battery chargers. A design with a single alkaline cell (1.5V) could use two of the same step-up converters with different output voltages. This design has the advantage of not requiring a charging circuit, though it is also not as environmentally friendly since the batteries are one-use. The smallest alkaline cell of adequate capacity available is the AAA cell. The next size larger is the AA cell. The AAA cell was chosen for its smaller size. Though it has lower capacity than the AA, its capacity is sufficient for the design goal of several hours of continuous use.

Given that a AAA cell is to be used as the battery, a boost converter must be selected that can provide sufficient current given the range of output voltages the AAA has. Most boost converters are not specifically designed for single alkaline cell input voltages, and therefore have cutoff voltages higher than 1.5V, the nominal alkaline cell output voltage. Therefore, a boost converter must be specifically selected for its single cell design. For this, the Linear Technology LT3400 was chosen. It can operate with input voltages down to .7V and supply sufficient current at high efficiency (>80% over the range of output voltages required).

2.7 Mounting Fixture

The device must be firmly attached to the user and should be readily mounted on the arms and legs. An adjustable strap threaded through a Plexiglass plate was used to secure the device to the body. The device was mounted to the Plexiglass plate through its two mounting holes on opposite corners. The strap was secured with a friction fit buckle, like those used on backpacks. This method of mounting the device performed well during tests that included Tae Kwan Do moves. See figures 2-2, 2-3 for pictures of the device mounted during use.

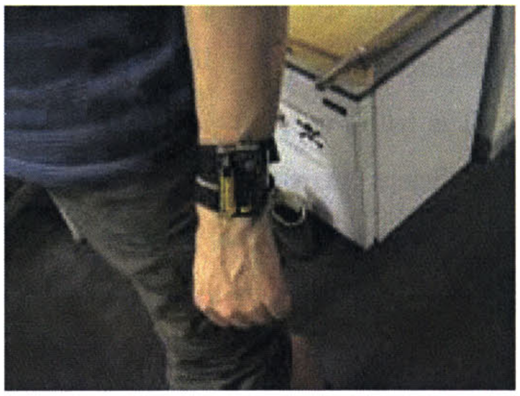


Figure 2-2: The device mounted on a users arm

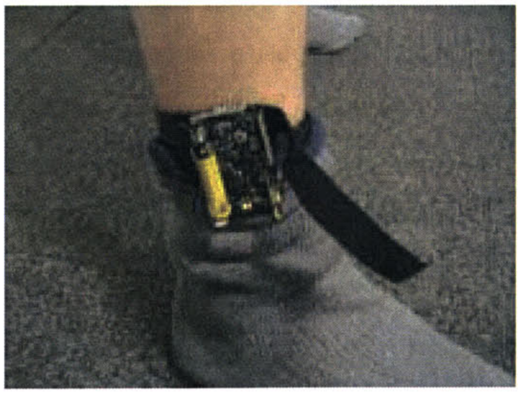


Figure 2-3: The device mounted on a users leg

Chapter 3

Firmware

This chapter discusses the firmware of the nodes and the basestation as well as the communications protocol between the nodes and the basestation. Examples of the capabilities of the nodes are given, but discussion of the overall capabilities of the devices is left to the Future Work chapter of this thesis.

As wireless wearable sensors, this system's main task is to transmit data from the nodes to the host computer. This data must be relayed to the computer through a receiving basestation that communicates with the nodes. The basestation is implemented using a node re-purposed for communications with the host computer: the basestation uses a support board that allows the node to interface to the computer using serial communications over RS-232 or using USB. Currently, the nodes only collect and transmit sensor data and the basestation only relays commands from the computer to the nodes and data from the nodes to the computer. Future implementations may have the nodes compress or do preliminary analysis on the data or have the basestation actively manage the communications protocol.

3.1 Communications Protocol

The communications protocol is a TDMA scheme that uses the basestation broadcasts as the time reference. Each node is assigned a time slot which is determined by its programmed timer interrupt value. The nRF2401 is set to ShockBurst mode and each packet is sent with an address. The receiving nRF2401 must be configured with the same address or else the microcontroller is not notified of the packet and the packet is dropped. All nodes are set to the same address, making all packets sent by the basestation to this address a broadcast packet. It is possible to have the nRF2401's second receiver set to a unique address and use this to send messages to specific nodes, but the messaging happens with such low frequency that it is easier and quicker to send messages in the broadcast packet's payload for this application. The TDMA scheme is shown in figure 3.1.

3.2 Node Firmware

The node's main task is to collect sensor readings and transmit them to the basestation. On power-up, the node microcontroller must initialize its internal peripherals, its analog signal processing circuits, and its data radio. After initialization, the node turns on its data radio to listen for basestation broadcast messages and puts the processor into a low-power sleep mode. Upon receiving a basestation broadcast message, the radio pulls an I/O line high to wake the microcontroller. The node reads the received data, collects its sensor data, parses the received data for commands, taking action if necessary, then turns its radio off for power savings and then goes back to sleep. This time, an internal timer will trigger an interrupt and wake the node up when it is supposed to transmit its sensor data. Upon waking up the node powers up its data radio, transmits it sensor data, sets its radio to listen for basestation broadcast messages and goes back into low power sleep mode. This process continues indefinitely. A listing of the node firmware is given in appendix B

3.2.1 Initialization

The microcontroller's internal peripherals that must be initialized are its I/O ports; its internal timer; its communications ports which include an SPI port for the data radio and an SPI port to control the digital potentiometer in the analog signal processing section; and the microcontroller's internal A/D converter. The nRF2401,

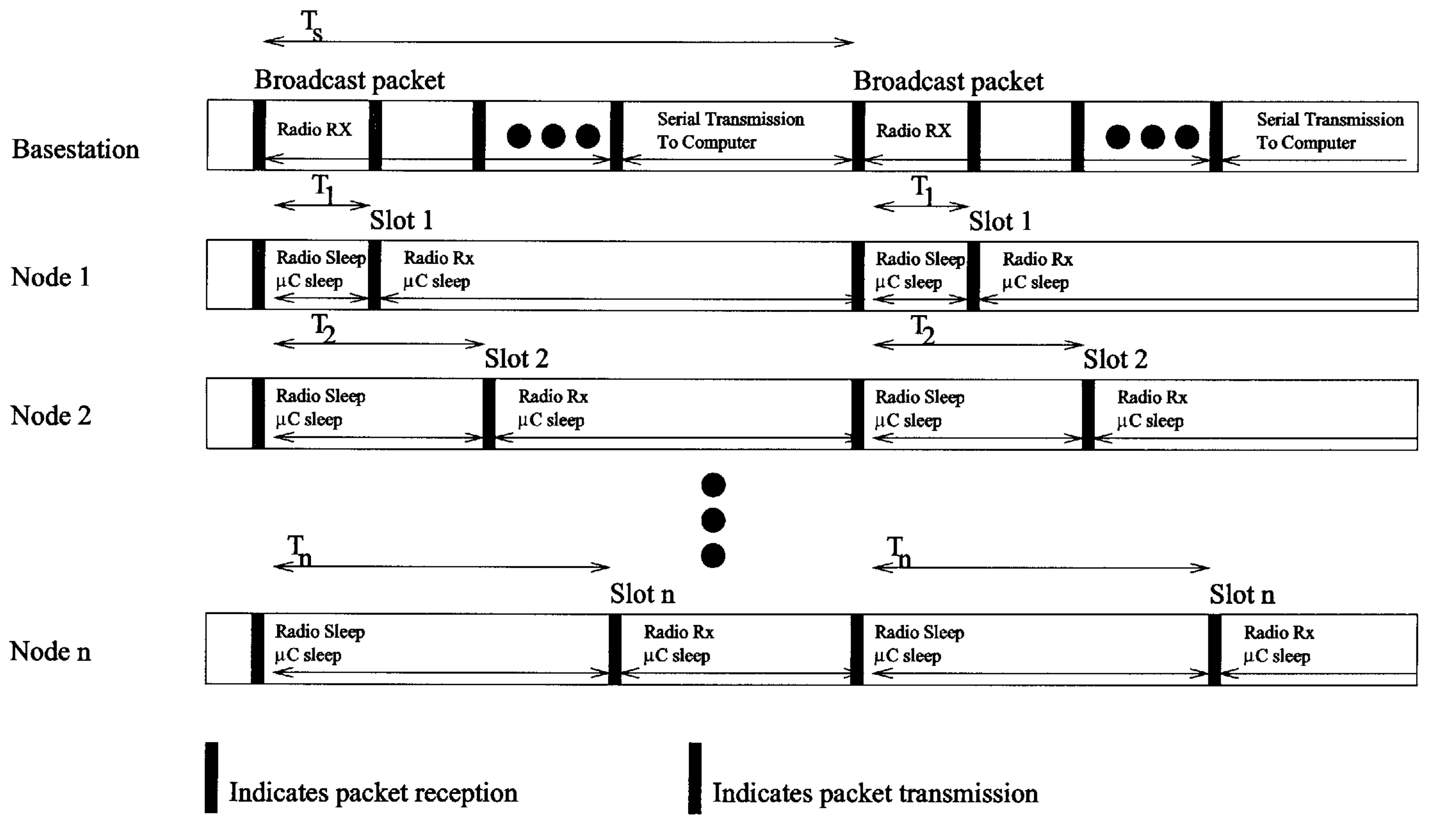


Figure 3-1: Depiction of TDMA scheme: T_1 to T_n determine the nodes' time slot. T_s is the sampling period.

when set to ShockBurst mode[1], and the digital pot have no special requirements for communications rates so their SPI port settings are straightforward. In this design, the MSP430's A/D converter uses its internal voltage reference (V_{REF}) for conversions so special care must be paid to ensure it functions properly with the microcontroller's analog supply voltage, AV_{CC}. The analog supply voltage must be 0.15V greater than the internal voltage reference. This design provides 3V for AV_{CC} so either of its 1.5V and 2.5V internal references can be used without concern. For ultra-low power operation, AV_{CC} can be lowered to 1.8V, in which case the 1.5V reference must be used. The internal timer is set to count down from a specified value and halt[2] and is used for timing intervals between reception of broadcast packets and transmission of its own packets.

The analog signal processing circuitry requires setting both of the digital potentiometer's taps to control the offset and gain of the op-amp circuit used to bring the sensor output into the range of the A/D. The configuration of the op-amp circuit is such that the circuit outputs a safe voltage at turn-on due to the power up settings of the potentiometer[3].

The data radio must be initialized with a sequence of bits to set the receiver's address, the data width in bits, the communication mode (ShockBurst/Direct), the frequency channel, the data rate, the output power and the mode of operation (TX/RX). Currently, all these parameters are programmed into each node's flash memory and are identical. These settings could be changed dynamically to avoid noisy channels, to move to unoccupied channels when there is no more room or to conserve power.

[1]The data radio has some special requirements in its direct mode. In its direct mode, the communications rate must be either 1Mbps or 250Kbps and the bit rate must be within ±200ppm for both rates for the data to be recognized by the radio and transmitted.

[2]Not auto-reloading

[3]The potentiometer powers up with its wiper at half-scale. The op-amp is configured as a subtractor with the scaled sensor value subtracted from the potentiometer's offset resistor divider output. Because the potentiometer has DV_{CC}, the digital supply voltage, as the voltage at the top of the resistor divider it forms, it outputs $\frac{DV_{CC}}{2}$ at power up and the circuit can provide at most $\frac{DV_{CC}}{2}$ to the microcontroller.

3.2.2 Main Loop

Though the node's main loop simply puts it to sleep, it actually goes through a series of states by coming out of sleep through interrupts. I use the term main loop to describe the repeating series of actions that the node goes through. The node starts in a low power sleep mode with its data radio turned on and set to receive. Upon reception of a basestation broadcast, the data radio signals the microcontroller by pulling a data line high. The microcontroller exits its low power sleep mode and executes its external interrupt handler. In the external interrupt handler, the microcontroller immediately starts a timer that will later wake it to transmits its sensor data. The received data is then read into the microcontroller's memory for later processing and the radio is set to a low power mode. The microcontroller then samples its sensor data. All steps up to this point are the same for all nodes to ensure that all sensor data is sampled concurrently. This was tested by toggling an I/O pin just prior to the sampling and noting that the toggling occurred within approximately 200ns of each other (about the span of one clock cycle).

After sampling the sensor data, each node checks the basestation broadcast for a message directed to it. Each node is programmed with a unique ID, which it compares to the ID in the received broadcast. If the IDs do not match, then the node proceeds to low power sleep mode. If the ID does match then the microcontroller checks the message for a command and executes it. Currently the only command is a command to change the delay between receiving a broadcast message and transmitting sensor data. This is implemented as a single byte for the command ID followed by two bytes of data to set the timer register which controls the interrupt interval. The microcontroller then proceeds to low power sleep mode.

The microcontroller stays in low power sleep mode until its timer interrupt occurs and executes its timer interrupt handler. The timer resets its counter for the next timing interval, writes its data to the radio, waits a short period of time and then sets the radio to RX mode and goes into low power sleep. The microcontroller must wait for a short period of time because its takes some finite time for the radio to create a radio packet and then for the data to be transmitted[4]. If the radio is configured before the data is done transmitting then the radio packet is not sent or is incomplete. The receiving radio cannot parse such a packet because its CRC is not correct, hence the packet is dropped.

The microcontroller remains in sleep mode until the data radio receives a valid packet, then pulls its data line to the microcontroller high. This causes the microcontroller to enter its interrupt service routine, bringing it to the state that it initially started in. This loop continues ad infinitum.

3.3 Basestation Firmware

The basestation's main task is to relay communications between the nodes and the computer. Similar to the node, the basestation must initialize its internal peripherals and its data radio. The basestation is not outfitted with sensors so it does not need to initialize any. After initialization, the basestation must send periodic broadcast messages, receive all incoming data from the nodes, and transmit the node data to the computer. The basestation does not go into a low power sleep mode like the nodes because it is simpler not to do so, but could readily be made to do so. The basestation differs from the nodes in that its timer interrupt is set to auto reload so its period is always the same, regardless of how many incoming node data packets it must handle. The node follows this pattern: broadcast message to nodes, set radio to listen, receive and store all incoming node packets, trigger timer interrupt, transmit all data to the computer, broadcast message to nodes, etc. Interspersed with this activity, the basestation handles serial messages from the computer as they come in. Note that the basestation does not need to know how many nodes are broadcasting, it merely saves all data packets and transmits them to the computer. The nodes must be preprogrammed with the proper time slots by the user to prevent collisions. A

[4]This all happens in the radio hardware and is part of the nRF2401's ShockBurst mode. The wait between the of writing the data to the nRF2401 and the beginning of transmission, ***Tsby2txSB*** in the nRF2401 datasheet, is specified as 200us. The transmission length depends on the amount of data to be sent and the data rate. Transmission length = $\frac{\#\textbf{bits to be sent}}{\textbf{data rate}}$

listing of the basestation firmware is given in appendix B

3.3.1 Initialization

The microcontroller's internal peripherals which must be initialized are its I/O ports; its internal timer and counter; its communications ports including an SPI port for the data radio and a UART. The I/O ports and the SPI port serve the same functions as on the node. The internal timer is set to be auto reloading and is used to trigger the sensor packet requests. The internal counter is used to measure the time difference between transmission of the broadcast packet and reception of the node packets. This difference can be used to identify which node has sent the data (and is also useful for debugging). The UART is set to 115.2kbps for high speed communications to the computer through an RS-232 transceiver. The RS-232 transceiver is connected to through the programming header on the node board.

Data radio initialization for the basestation is similar to that of the node.

3.3.2 Main Loop

Though the basestation's main loop just toggles an LED at 1Hz, it also goes through a series of states via interrupt service routines. On a periodic timer interrupt, the basestation checks to see how many data packets it has received from nodes during the previous timer interval. It transmits a header over RS-232 to the computer to signify the start of a new data record, followed by the number of packets it has received, followed by its data buffer. The data buffer contains all reported data prepended with the timer value at which it was received. The basestation configures the radio for transmission and then starts a counter to time when the data comes in[5]. The basestation then broadcasts a message to all nodes, to signify the start of a new data collection cycle. This broadcast message is primarily used for time synchronization, so its contents are not usually that important. If the basestation has received a message from the computer via RS-232, it inserts this message into the broadcast

[5]This is the timer value that is prepended to node data in the data buffer

message, otherwise the broadcast message is filled with *0xFF*. After the transmission, the basestation waits for the transmit to finish, configures the radio to receive mode and leaves the timer interrupt routine.

Every time a packet comes in the data radio raises a data line high, causing an external interrupt to occur. The first thing that occurs in the external interrupt handler is the basestation captures the counter value to determine when the packet arrived and writes this value to its data queue. The basestation reads the data from the radio, writes the data to the data queue, increments its message counter, and returns from the interrupt.

Message reception from the host computer occurs via serial communications over RS-232. The basestation uses its serial reception interrupt service routine to read incoming messages byte by byte to a buffer. The computer sends its messages sandwiched between a header byte and a footer byte. Upon reception of a complete message, the interrupt checks to see if the message is intended for the basestation or the nodes. Currently the only message for the basestation is to set the rate at which broadcast messages are sent. If the message is not for the basestation then it waits till the next broadcast message and sends it to the nodes.

Because transmission of the received node data uses a blocking serial write, the bandwidth of a single basestation system is currently limited to an average of 57.6kbps, which can only support 7 nodes at 50Hz sampling rate. This could be raised to 115.2kbps, the maximum standard serial rate, by creating a non-blocking serial transmission routine, and further raised to 1Mbps by implementing a USB interface through SPI.

Chapter 4

Analysis

This chapter analyzes the system's performance and discusses the tests run for correlated activity detection.

4.1 System Performance

To test the range of the system, data was collected in an indoor setting with a node at varying distances. The node was placed on a stool and the basestation on a table. From ranges inside 10 feet, 100% of packets sent by nodes were received by the basestation. From 20 feet, 94% of packets were received. From 30 feet 81% of packets were received, with most of the dropped packets occurring in groups of about 10 packets spread in 30 packets.

As a test of the systems ability to capture data, a subject wore 4 nodes, one on each hand and foot, while performing a Tae Kwan Do move : a punch, a punch, and then a kick. This was performed twice in a row. This test evaluates both how well the sensors capture a dynamic human gesture as well as how the data radio performs in a dynamic environment. The basestation was approximately 5 feet from the nodes during the test. Data from the experiment and pictures showing the body positions gone through are in figures 4-1, 4-2, 4-3, 4-4, 4-5. These graphs are representations of the motion that the node went through during data capture. Notice that at some points the readings saturate. This is caused by rotation or acceleration outside the range of the sensors. Dropped packet rate varied from 1% for the nodes on the feet to 4% for the nodes on the hands. Points where packets were lost are marked on the the graph by circles with x's inside them.

4.2 Host Software

The PC host software is implemented in Matlab. It captures a specified amount of data from the serial port, parses the raw serial data and extracts packets, performs linear interpolation between dropped packets for each node, plots the interpolated data noting where interpolation occurred, and tests for correlated activity. No action was taken to correct for misalignment of sensor axes. A calibration script was written but axis error was found to be low enough to not impact activity detection. A listing of the host software is given in appendix C.

For tests of correlated activity, 4 subjects wore a node on their left hand and did 'The Wave'[1] twice in a row.

To find areas of interest, the magnitude of the windowed variance was used as in Benbasat[3]. A threshold between activity and non-activity was selected by observing the range of the variance of the sensor data between when the notes are moving through gestures and sitting stationary. Once an area of activity is detected, the cross-covariance between it and the same sensor data stream on other nodes is taken[2]. The peaks of the cross-covariance are found and noted as possible correlated activity. The time lag between the highest cross covariance between the region of interest and the other data stream is returned. From this you can determine the relative degree of synchronicity of action between nodes. This test was performed on the data captured from the second accelerometer axis while performing 'The Wave' and the data and results are shown in figures 4-7, 4-8, 4-9, 4-10. Only one axis was used to simplify

[1]The wave is a move often done by spectator at sporting events. Spectators raise and lower their hands in succession creating a rippling effect across the crowd. The wave is depicted in figure 4-6

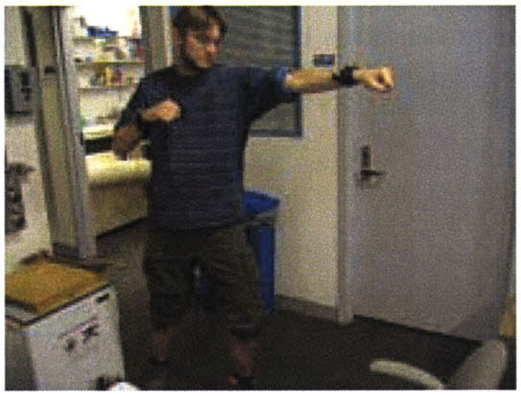
[2]If the cross-correlation were used, this would be a matched filter. The cross-correlation was not used to try to remove the effects of the mean on the output. Each sensor has a different offset voltage. Using the cross-correlation would make the quality of the match appear better for higher mean sensors.

Node #	Time lag of maximum covariance	un-normalized covariance value
1	0	2.62E7
2	22	2.25E7
3	36	1.53E7
4	52	2.62E7

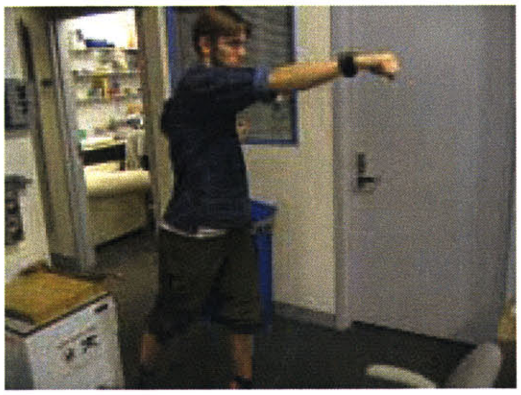
Table 4.1: Table of time lag of maximum covariance and un-normalized covariance value for each node. Notice that node 1, the template node, has a time lag of 0.

the presentation of results.

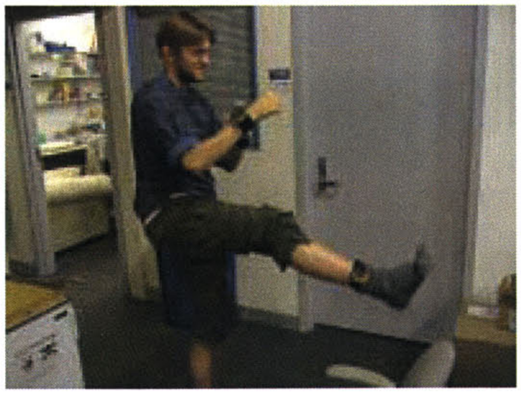
(a) Initial stance



(b) Punch!

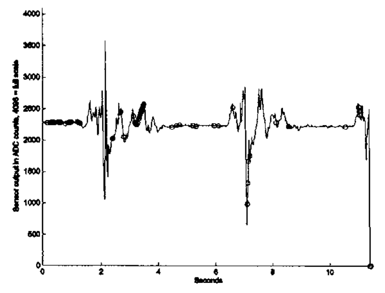


(c) Punch!

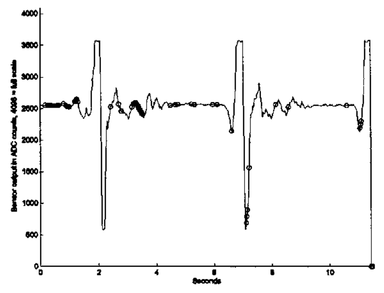


(d) Kick!

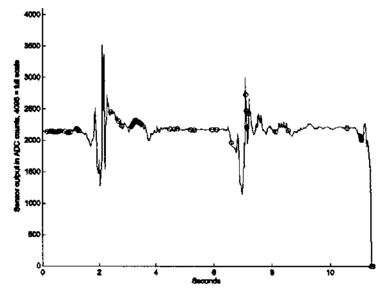
Figure 4-1: Stages of executed Tae Kwan Do move



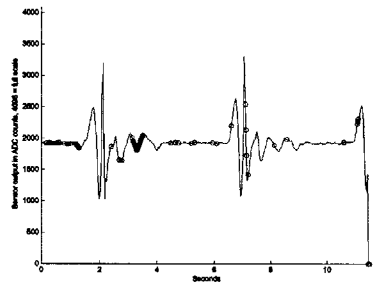
(a) Accelerometer 1



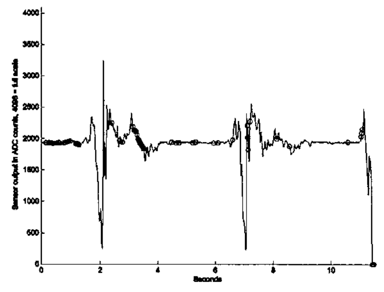
(b) Accelerometer 2



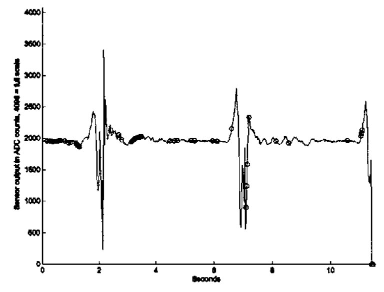
(c) Accelerometer 3



(d) Gyro 1

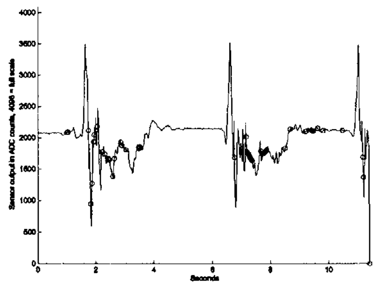


(e) Gyro 2

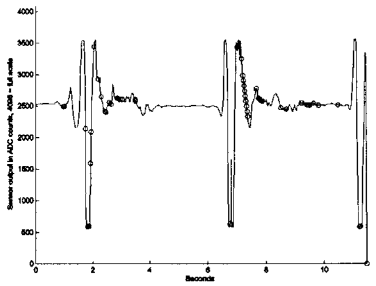


(f) Gyro 3

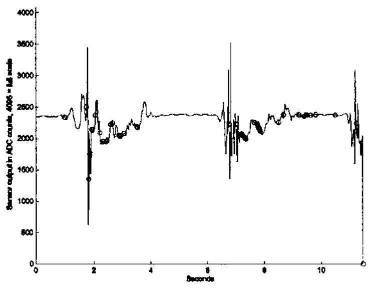
Figure 4-2: Data captured from right hand during a Tae Kwan Do move. Circles in data represent dropped packets.



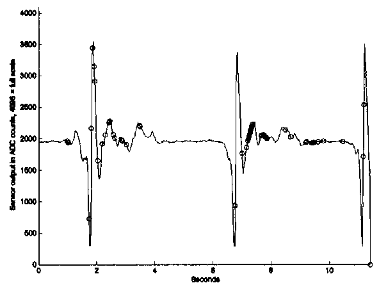
(a) Accelerometer 1



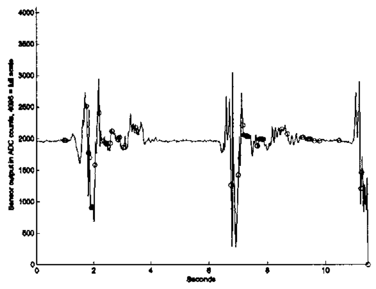
(b) Accelerometer 2



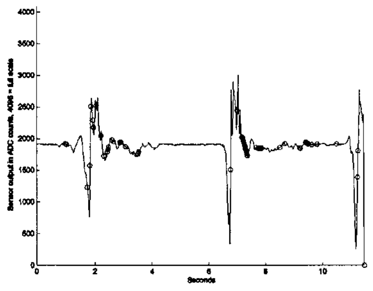
(c) Accelerometer 3



(d) Gyro 1

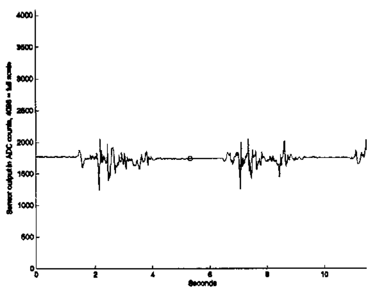


(e) Gyro 2

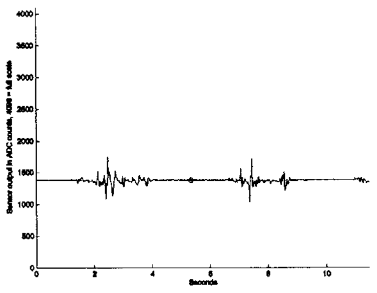


(f) Gyro 3

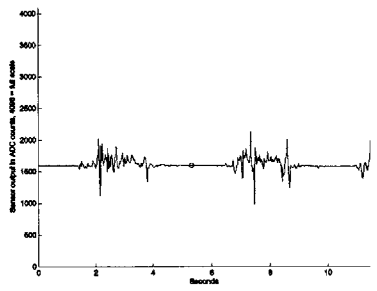
Figure 4-3: Data captured from left hand during a Tae Kwan Do move. Circles in data represent dropped packets.



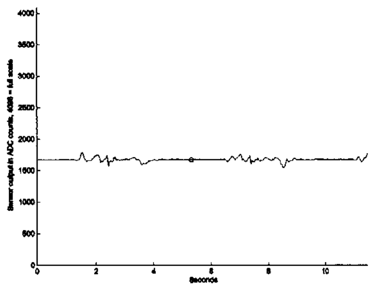
(a) Accelerometer 1



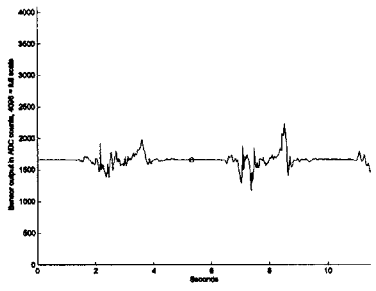
(b) Accelerometer 2



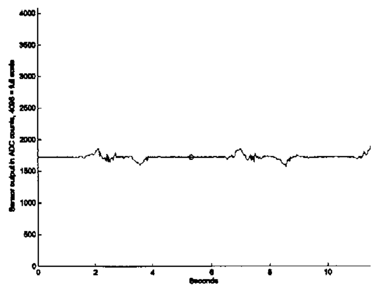
(c) Accelerometer 3



(d) Gyro 1

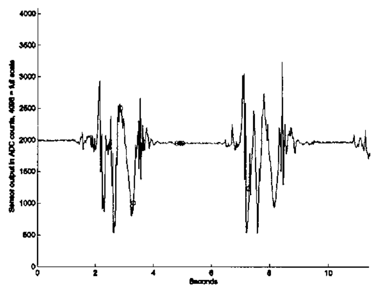


(e) Gyro 2

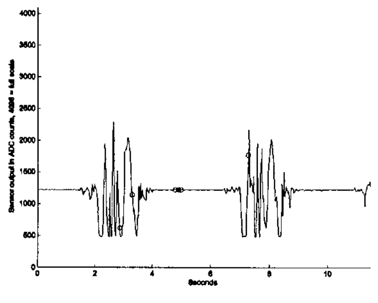


(f) Gyro 3

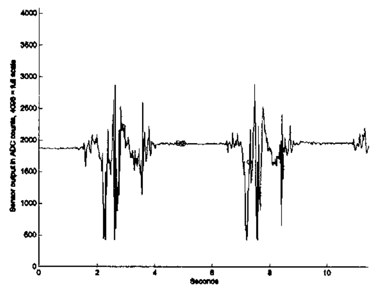
Figure 4-4: Data captured from left foot during a Tae Kwan Do move. Circles in data represent dropped packets.



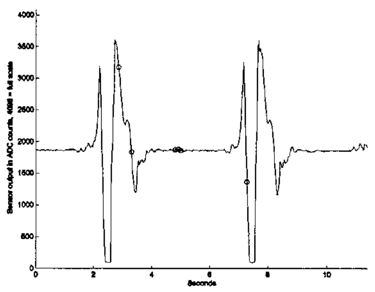
(a) Accelerometer 1



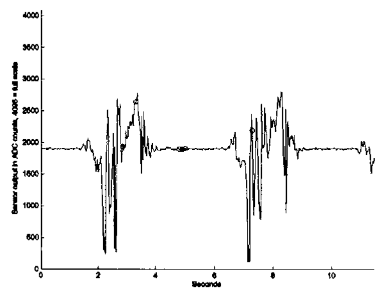
(b) Accelerometer 2



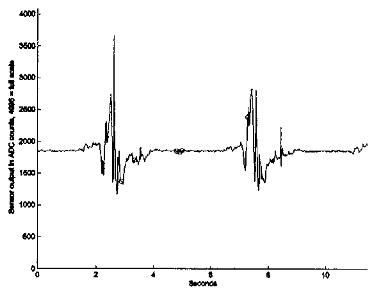
(c) Accelerometer 3



(d) Gyro 1

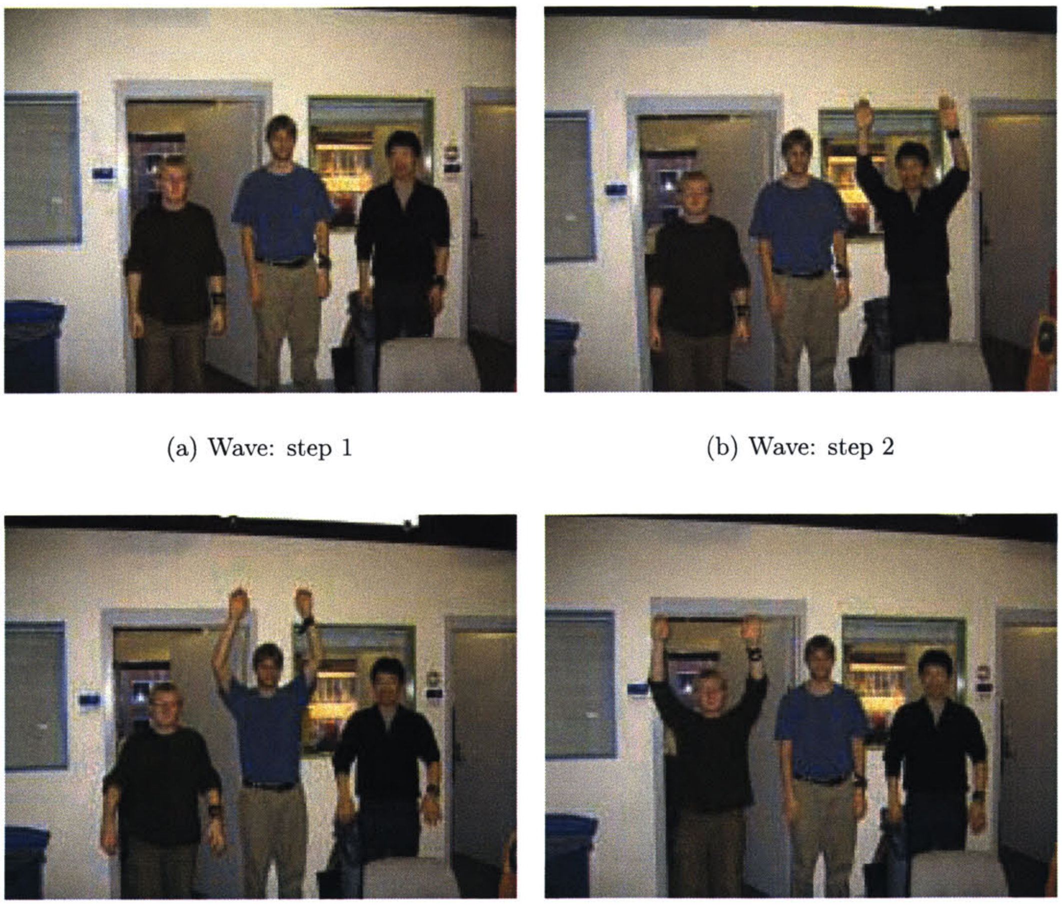


(e) Gyro 2



(f) Gyro 3

Figure 4-5: Data captured from right foot during a Tae Kwan Do move. Circles in data represent dropped packets.



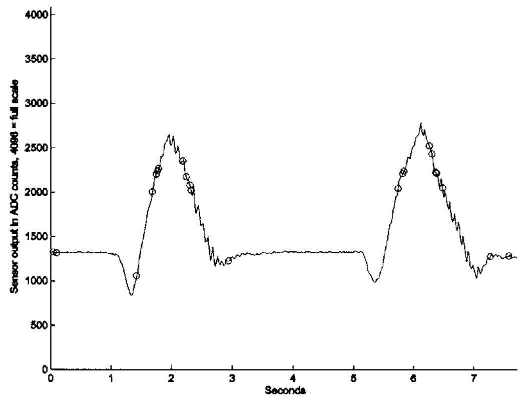
(a) Wave: step 1

(b) Wave: step 2

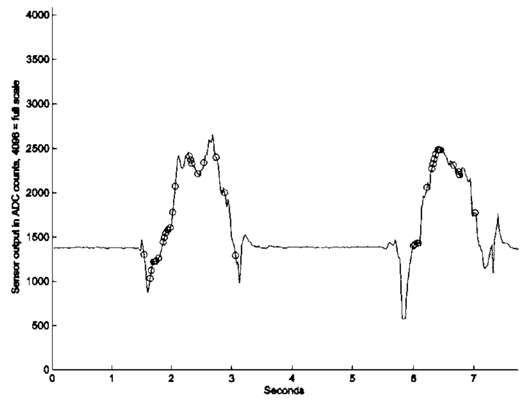
(c) Wave: step 3

(d) Wave: step 4

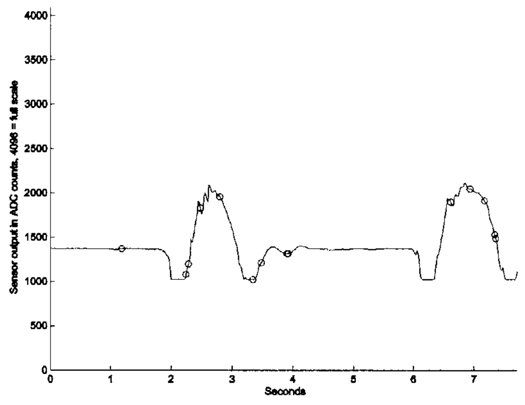
Figure 4-6: Pictures of the progression of the wave



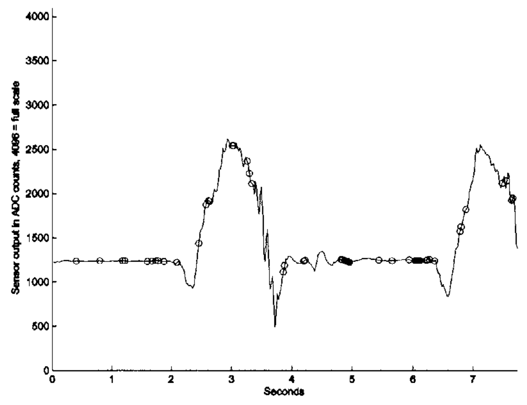
(a) Wave: Node 1



(b) Wave: Node 2

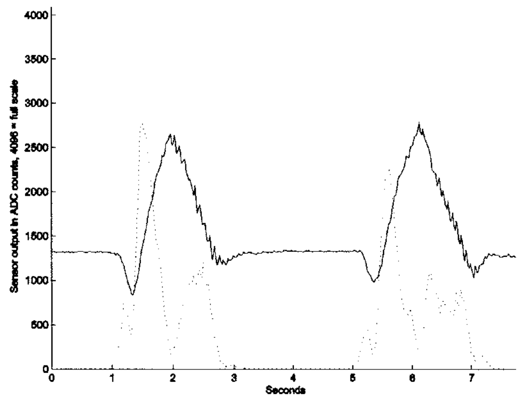


(c) Wave: Node 3

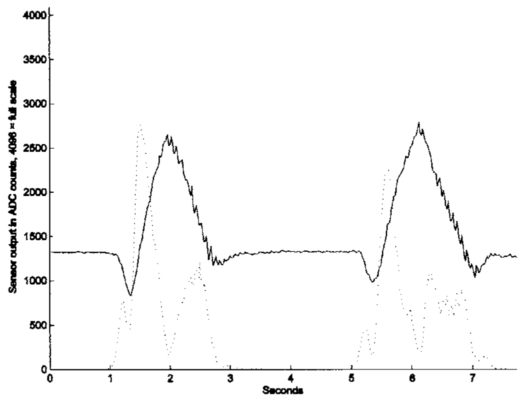


(d) Wave: Node 4

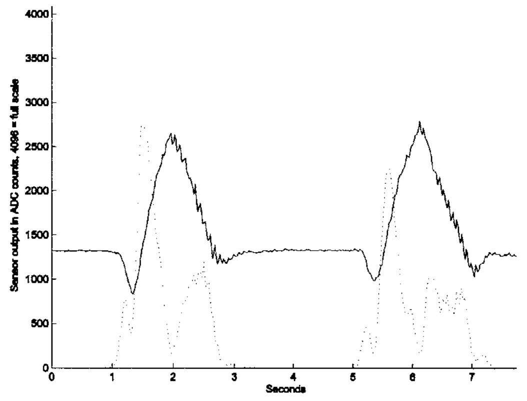
Figure 4-7: Raw data from the wave for the second accelerometer axis of each node. Circles in data represent dropped packets.



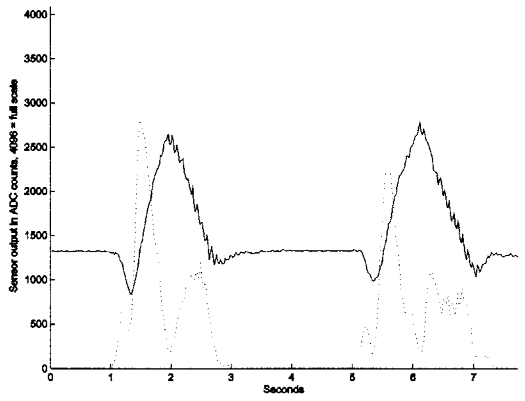
(a) Wave: Node 1



(b) Wave: Node 2



(c) Wave: Node 3



(d) Wave: Node 4

Figure 4-8: Variance of raw data from the wave. Solid line is the original signal. Dotted line is the variance of the signal

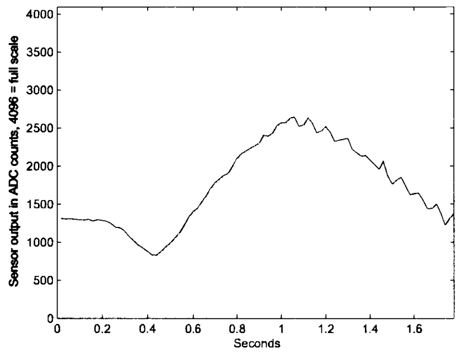
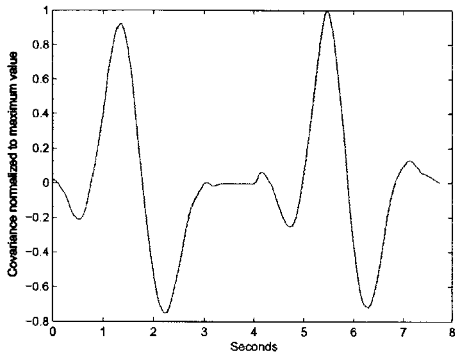
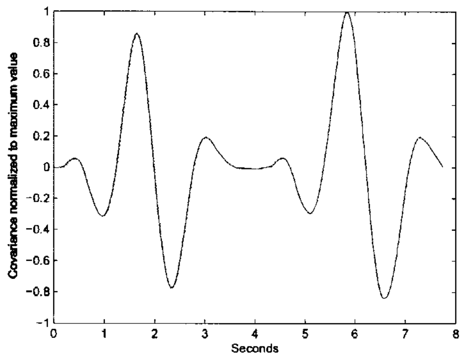


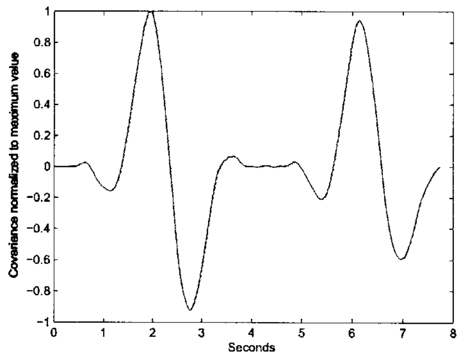
Figure 4-9: Region of interest selected from node 1's second accelerometer axis data. Node 1 was worn by the initiator of the wave, hence it is considered the gesture template.



(a) Wave: covariance of node 1 with node 2



(b) Wave: covariance of node 1 with node 3



(c) Wave: covariance of node1 with node 4

Figure 4-10: The Wave: Covariance of first area of interest from node 1 with data from other nodes. Data has been normalized by its maximum. There are two peaks because the wave was performed twice. Peaks are rough measures of similarity between the gesture template and the node data.

Chapter 5

Conclusions

Most of the design goals of the design as stated in chapter 1 were met. The nodes successfully detected coordinated activity from multiple nodes using algorithms that could be executed in real time. Some aspects could be improved upon. These include improving the communications link's reliability, finding a higher bandwidth replacement for serial communications between the basestation and the computer, and developing a robust and accurate gesture recognition scheme that can fully utilize the node's capabilities.

5.1 Future Work

5.1.1 Platform Improvements

Performance of the data radio was adequate at best, our tests indicated that the data radio solution needs to be re-evaluated. Levels of dropped packets when the nodes are in a static position are acceptable, but the dropped packet rate for a given distance increased when the nodes were moving. The current reliable transmission range is too short for a dance stage, making it unusable for one of its main applications. Several factors could be the cause. The antenna layout and design could require redesign. Currently, the antennae is a 2.4GHz quarter-wave dipole made out of a single piece of 26AWG single conductor wire. This could be replaced with a 50 Ohm printed circuit antenna. The particular frequency channel used could be a noisy one, though this is doubtful since some nodes transmitted with low dropped packet rates while others transmitted with high dropped packet rates during the same short time duration. It is possible that other data radio solutions may need to be considered. During the process of this design, several new data radio modules have come out with similar capabilities as the nRF2401: products in this field are evolving rapidly and a new evaluation of these modules is in order.

The wearability of the module was sufficient for the short duration tests performed, indicating they will be sufficient for target applications like dance performances and gestural input. Some redesign may be necessary for activities like coordinated activity detection in sports. The next revision of the board should increase clearance for the mounting screws and add mounting holes on all four corners of the board to increase stability of the board on the mount.

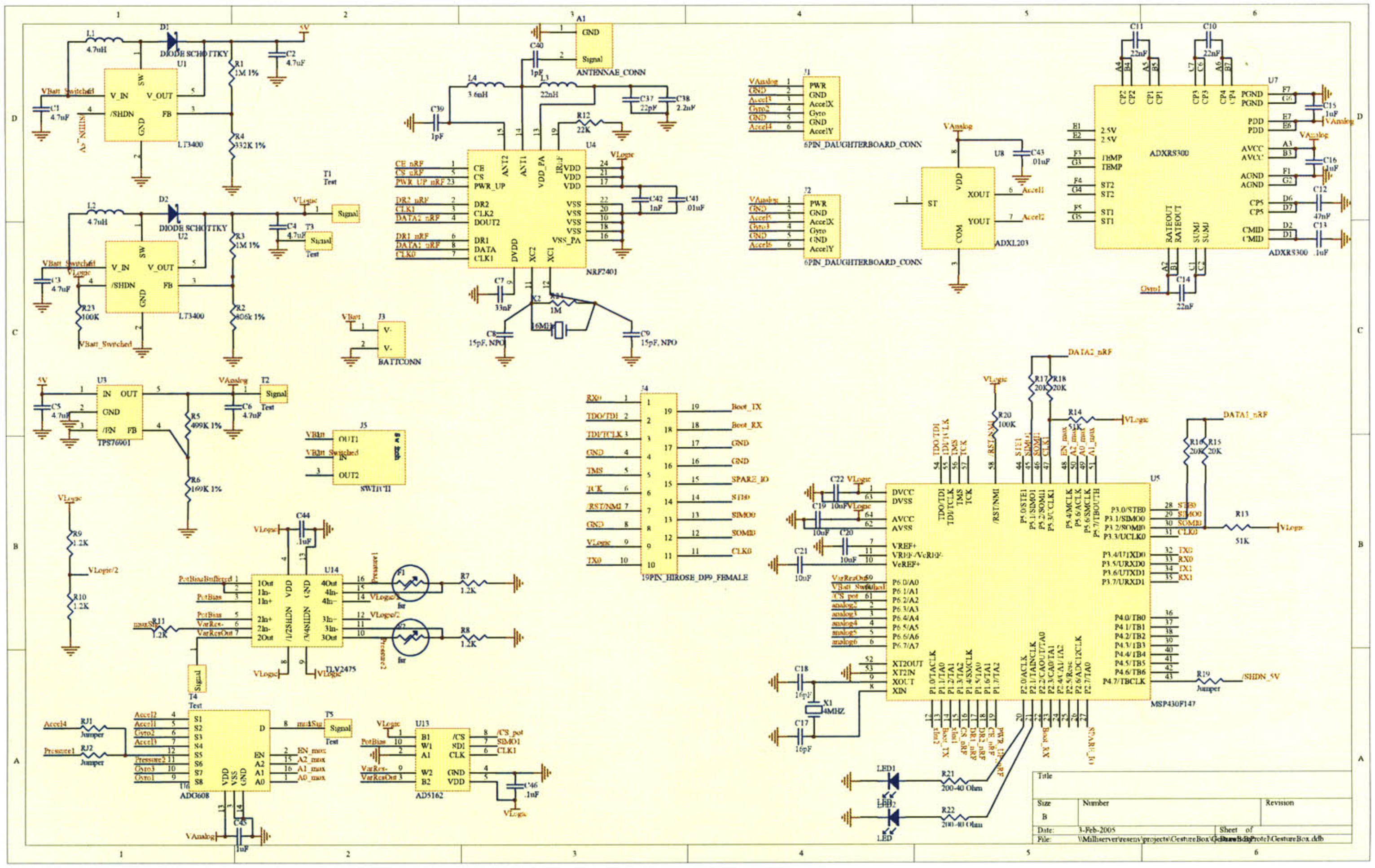
To capitalize on the communication link's high bandwidth, a fast and efficient method to get the data from the MCU to the computer must be found. Serial data communications will allow a maximum of 115.2kbps, whereas USB or Ethernet will allow higher rates. An attempt at USB communications was made where the MSP430 talked to a USB enabled MCU over SPI, with the USB enabled MCU as the slave. This method was not sufficient since the USB MCU's USB interrupts would prevent it from receiving SPI data. Future attempts could implement the MSP430 as the slave SPI device or use nested interrupts. Both these methods might prove fruitful in getting the data from the MSP430 to the USB MCU.

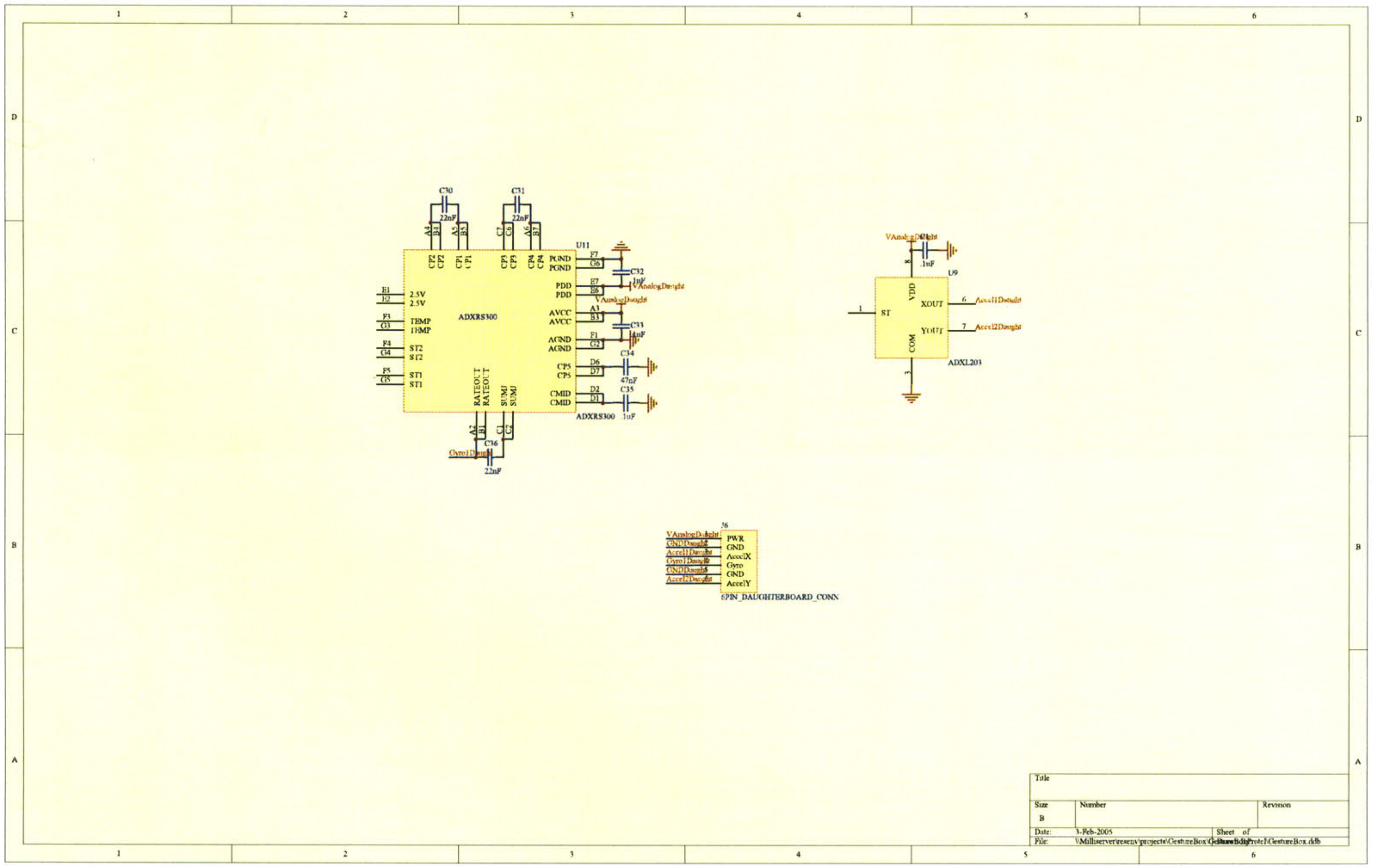
5.1.2 Detection Improvements

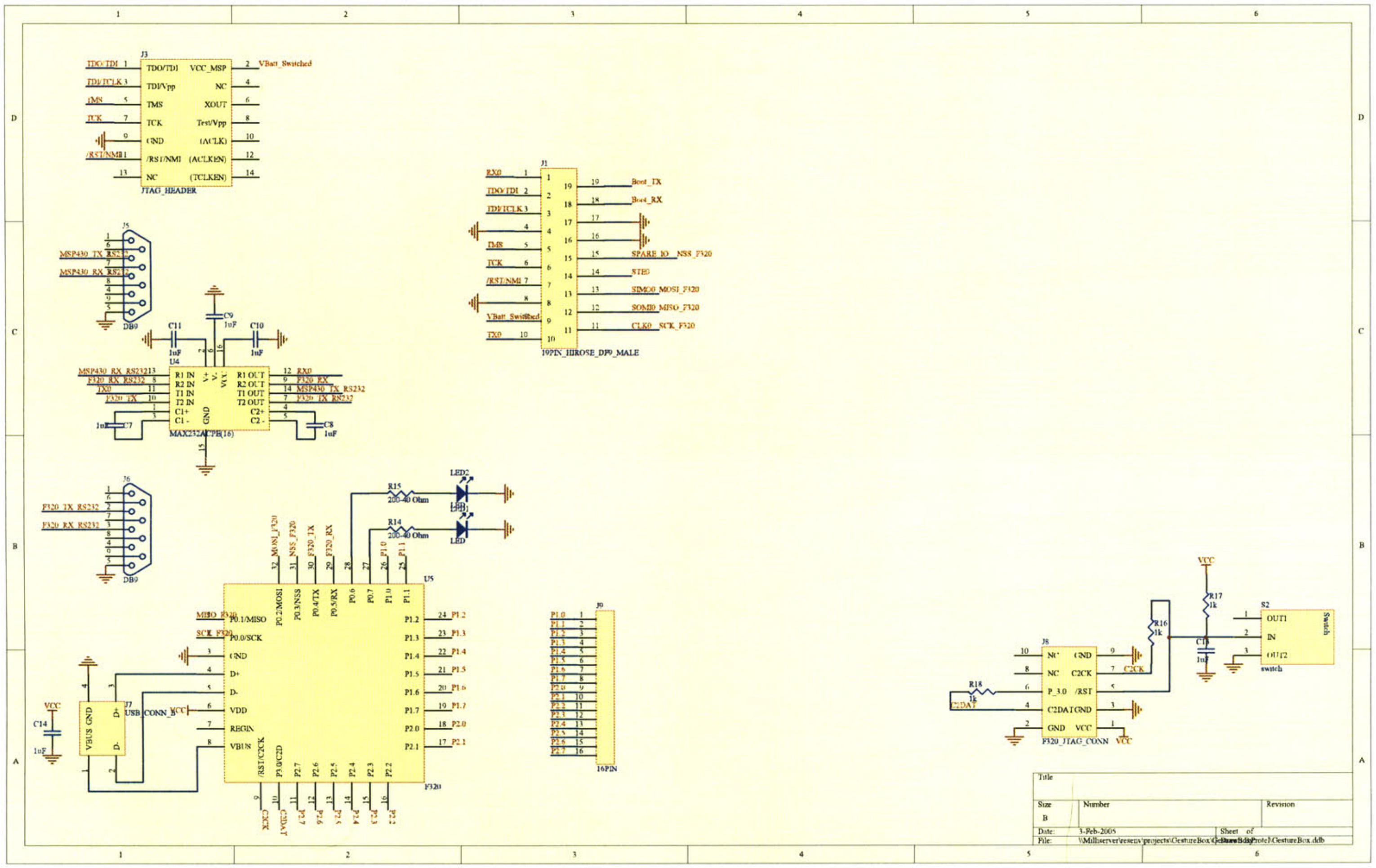
The detection methods discussed in the Analysis chapter are sufficient for activity detection, but does not discriminate gestures well. For gestural inputs, DTW techniques like those used by Merrill could be used. These techniques were too computationally intense for real-time recognition, but could be modified by pre-processing on the node. In particular, non-uniform sub sampling of the data stream could be performed on the nodes. This would reduce the size of the input to the DTW algorithm run on the host computer.

Appendix A

Schematics and Pictures of Boards


LT3400
DIODE SCHOTTKY
TPS76901
BATTCONN
SWITCH
TLV2475
ADG608
AD5162
ANTENNAE_CONN
NRF2401
19PIN_HIROSE_DF9_FEMALE
6PIN_DAUGHTERBOARD_CONN
ADXL203
ADXRS300
MSP430F147
VBatt_Switched
VLogic
VAnalog
5V
4MHZ
16MHz
Title
Size
B
Number
Revision
Date: 3-Feb-2005
Sheet of
File: \\Millhserver\resenz\projects\GestureBox\GestureBox.ddb


U11
ADXRS300
C30
22nF
C31
22nF
C32
.1uF
C33
.1uF
C34
47nF
C35
.1uF
C36
22nF
VAnalogDaught
Gyro1Daught
U9
ADXL203
VDD
XOUT
YOUT
ST
COM
.1uF
Accel1Daught
Accel2Daught
J6
PWR
GND
AccelX
Gyro
GND
AccelY
GNDDaught
6PIN_DAUGHTERBOARD_CONN
Title
Size
B
Number
Revision
Date: 3-Feb-2005
Sheet of
File: \\Millserver\resenv\projects\GestureBox\GestureBox\Protel\GestureBox.ddb


JTAG_HEADER
19PIN_HIROSE_DF9_MALE
MAX232ACPE(16)
DB9
F320
16PIN
F320_JTAG_CONN
USB_CONN_B
switch
Title
Size B
Number
Revision
Date: 3-Feb-2005
File: \\Milliserver\resenv\projects\GestureBox\GestureBox\Protel\GestureBox.ddb
Sheet of

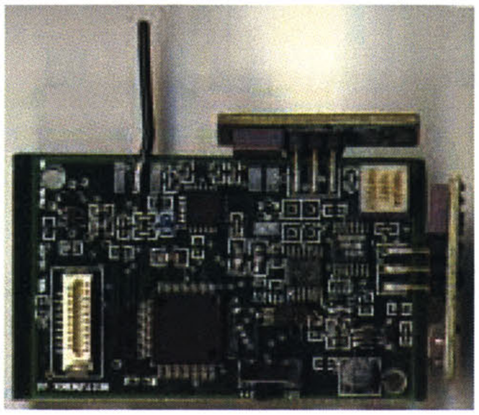


Figure A-4: Front side of main board with daughter boards mounted

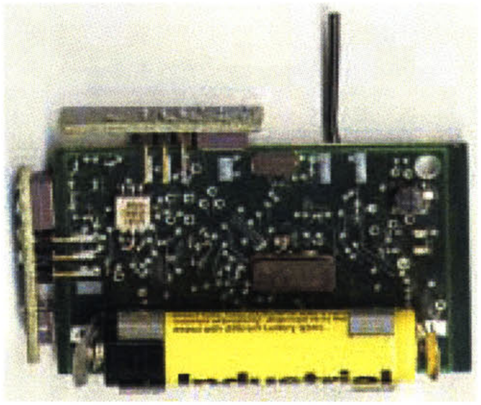


Figure A-5: Back side of main board with daughter boards mounted

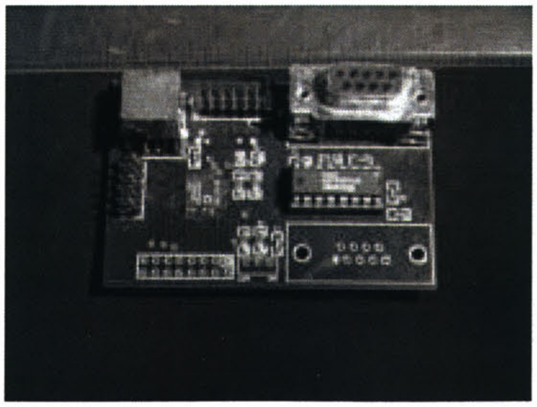


Figure A-6: Communications board populated

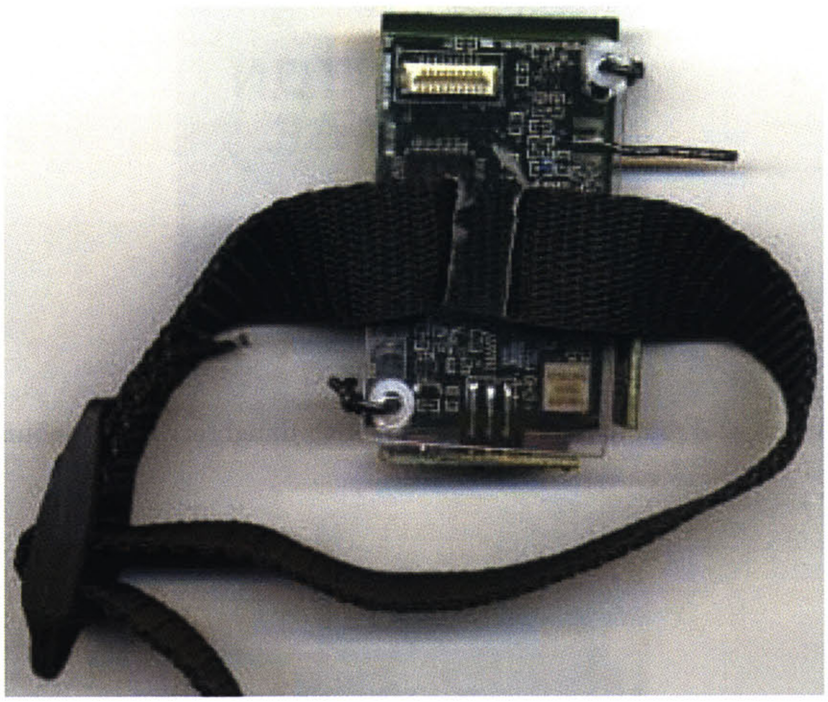


Figure A-7: Main board on mounting with strap

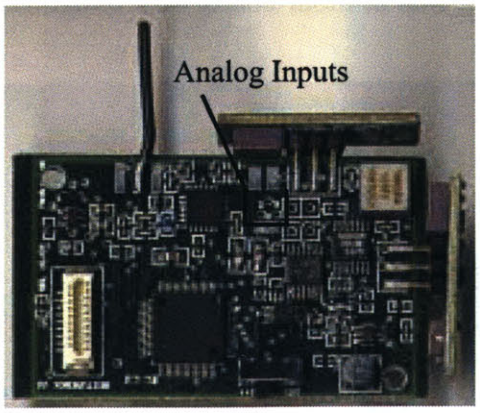


Figure A-8: Analog input terminals

Appendix B

MSP430 Embedded Code

```c
//mainBaseStation.c
//Written by Dan Lovell 9/8/04 for GestureBox Project, MIT Media Lab
//This file contains functions and variables specific to Basestation operation

#include <__cross_studio_io.h>
#include <msp430x14x.h>
#include <in430.h>      //has a _NOP()
#include "nRF2401.h"
#include "peripherals.h"
#include "config.h"
#include "mainBasestation.h"
#include "global.h"

const unsigned char ID = 0x00;
/*
      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?
 */

unsigned char TX_CONFIG_BYTES[15] = {INIT_RF_PACKET_BIT_LENGTH,INIT_RF_PACKET_BIT_LENGTH
  ,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xAA,0xA1,0x6F,0x04};
const int TX_CONFIG_BYTES_LENGTH = 15;
const unsigned char TX_SHORT_CONFIG_BYTES[1] = {0x04};
const int TX_SHORT_CONFIG_BYTES_LENGTH = 1;
unsigned char RX_CONFIG_BYTES[15] = {INIT_RF_PACKET_BIT_LENGTH,INIT_RF_PACKET_BIT_LENGTH
  ,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xAA,0xA1,0x6F,0x05};
const int RX_CONFIG_BYTES_LENGTH = 15;
const unsigned char RX_SHORT_CONFIG_BYTES[1] = {0x05};
const int RX_SHORT_CONFIG_BYTES_LENGTH = 1;
const unsigned char ADDR_BYTES[MAX_ADDR_BYTES_LENGTH] = {0xBB,0xBB,0xBB,0xBB,0xBB};
const int ADDR_BYTES_LENGTH = MAX_ADDR_BYTES_LENGTH;

extern const unsigned char HEADER[];
extern const int HEADER_LENGTH;

//used when the basestation has no message to send the nodes
const unsigned char FILLER_BYTE = 0xFF;

//used to store the message received from the F320
unsigned char RX_MESSAGE[16];
int RX_MESSAGE_LENGTH = 0;
//are you processing a message?
char RX_MESSAGE_FLAG = 0;

//used to store all the data received from the nodes during once TDMA cycle
unsigned char RX_DATA[256];
int RX_DATA_LENGTH = 0;

//used to temporarily store the data received from all the nodes during one
//TDMA cycle, therefore must have
union  {
  //two bytes allocated for ID and timestamp
  int intVal[MAX_NUM_NODES*(NUM_SENSOR_VALS+1)];
  //two char's per int
  unsigned char charVal[2*MAX_NUM_NODES*(NUM_SENSOR_VALS+1)];
} intChar;

int globalCounter=0;
int packetCounter=0;
int zeroLengthCounter=0;
char passMessage=0;

void main(void) {
  long i;

  //set up MSP430
  CONFIG();
  CONFIG_SPI0();
  CONFIG_UART1();
  CONFIG_UART1_RX();
  CONFIG_IO();
  nRF_RESET();
  nRF_SetStdByMode();

  //CONFIG_TIMER_A_INT(0x2700,1); //100Hz
  CONFIG_TIMER_A_INT(0x4E00,1); //~50Hz
  //CONFIG_TIMER_A_INT(0x3400,1);

  CONFIG_TIMER_B_COUNTER0();
  _EINT();

  //CONFIG_TIMER_B_INT();
  //give nRF time to go to PWR_UP mode before going to stand by
  //nRF_PWRUP has already been set high in CONFIG_IO()
  for(i=0;i<5000;i++) {}

  nRF_CONFIG(RX_CONFIG_BYTES,RX_CONFIG_BYTES_LENGTH);
  nRF_SetTXRXMode();

  while(1) {
    P2OUT ^= 0x01;    //toggle LED
    //for(i=0;i<30000;i++) {} // ~6.25ms to execute, ~5 clocks for one loop
  }
}

void processMessage(unsigned char * message, int messageLength) {
  /*
  *   Form of message is {HEADER0,DESTINATION_ID,COMMAND,...,HEADER1}
  *   indices               0          1            2         N
  */
  //check header to see if its for me
  char i;
  if(message[1]==ID) {
    switch(message[2]) {
      case(0x00):
        //change the poll rate of the sensors
        CCR0 = 256*message[3]+message[4];
        break;
      default:
        break;
    }
    //if so, process the message
  } else {
    //else set the passMessage flag so we know to send it to the Nodes
    passMessage = 1;
  }
}

void nRF_Data_Waiting(void) __interrupt[PORT1_VECTOR] {
  int i;
  intChar.intVal[(NUM_SENSOR_VALS+1)*packetCounter] = littleFromBig(captureTimerB());

  if(P1IFG & BIT4){   //clear the interrupt flag
    P1IFG &= ~BIT4;
  }

  //could probably just read into RX_DATA+2*packetCounter*NUM_SENSOR_VALS
  //but I'll try that later
  nRF_SHOCKBURST_BLOCKING_READ(RX_DATA,&RX_DATA_LENGTH);   //read the received data

  if(RX_DATA_LENGTH != 0) {
    //copy all the data bytes
    //advantage of not copying straight into RX_DATA is that you could process
    //any received data here, though you could still process in place in RX_DATA
    for(i=0;i<2*NUM_SENSOR_VALS;i++) {
      intChar.charVal[(2*NUM_SENSOR_VALS+2)*packetCounter+(i+2)] = RX_DATA[i];
    }
    packetCounter++;
  } else {
    zeroLengthCounter++;
  }
}

void Timer_A (void) __interrupt[TIMERA0_VECTOR] {
  int i;
  int temp=0;
  globalCounter++;

      P2OUT |= 0x02;      //toggle LED to indicate reception

  //this is just for testing to see if its going into timer interrupt with a
  //packet waiting

  if(globalCounter > 0) {
    globalCounter = 0;
    if(packetCounter>0) {

#ifdef SERIAL
      //configure the USART for serial and transmit the received data

      //CONFIG_UART0();
      //serial at this speed seems to always have trailing garbage
      usart1Write(HEADER,2);
      usart1Write(&packetCounter,2);
      //usart0Write(intChar.charVal,2*(NUM_SENSOR_VALS+1)*packetCounter);
      usart1Write(intChar.charVal,2*(NUM_SENSOR_VALS+1)*packetCounter);
      //usart0Write(&zeroLengthCounter,2);
      //write the timestamps and ID's of nodes
      /*
      for(i=0;i<packetCounter;i++){
        usart0Write(intChar.charVal+i*(2*(NUM_SENSOR_VALS+1)),4);
      }
      */
      //configure the USART for SPI, write RX configuration to nRF and enter RX mode
      //CONFIG_SPI0();
#endif

#ifdef USB
      //write data to cygnal USB via SPI
      f320Write(intChar.charVal,2*(NUM_SENSOR_VALS+1)*packetCounter);
      f320Write(HEADER,2);
#endif

    }
    packetCounter = 0;

    //broadcast to all nodes
    nRF_CONFIG(TX_SHORT_CONFIG_BYTES,TX_SHORT_CONFIG_BYTES_LENGTH);
    nRF_SetTXRXMode();

    //reset timer for counter
    CONFIG_TIMER_B_COUNTER0();
    if(passMessage) {
      if(RX_MESSAGE_LENGTH<=RF_PACKET_BYTE_LENGTH) {
        //need to write RX_MESSAGE_LENGTH BYTES OF RX_MESSAGE!!!
        nRF_SHOCKBURST_BLOCKING_WRITE(RX_MESSAGE,RF_PACKET_BYTE_LENGTH,ADDR_BYTES,ADDR_BYTES_LENGTH);
        RX_MESSAGE_LENGTH = 0;
        passMessage = 0;
      } else {
        //shouldn't be here!
        iShouldntBeHere();
      }
    } else {
      nRF_SHOCKBURST_BLOCKING_WRITE_REPEATED(FILLER_BYTE,RF_PACKET_BYTE_LENGTH,ADDR_BYTES,ADDR_BYTES_LE
    }


    //return to receive mode

    //probably need to make sure that this config doesn't occur too close to
    //the previous TX since you can't config while TX is occuring
    for(i=0;i<1000;i++) {
    //this wait might need to change
      //_NOP();
    }

    nRF_CONFIG(RX_SHORT_CONFIG_BYTES,RX_SHORT_CONFIG_BYTES_LENGTH);

    nRF_SetTXRXMode();

      P2OUT &= ~0x02;      //toggle LED to indicate reception

  }
}

void usart1_rx (void) __interrupt[UART1RX_VECTOR]
{
  char temp = RXBUF1;
  switch(temp) {
    case(HEADER0) :
      RX_MESSAGE_FLAG = 1;
      RX_MESSAGE[0] = temp;
      RX_MESSAGE_LENGTH = 1;
      break;
    case(HEADER1) :
      if (RX_MESSAGE_FLAG ==1) {
        //process the message
        RX_MESSAGE[RX_MESSAGE_LENGTH++] = temp;
        RX_MESSAGE_FLAG = 0;
        processMessage(RX_MESSAGE,RX_MESSAGE_LENGTH);
      } else {
        RX_MESSAGE_FLAG = 0;
        RX_MESSAGE_LENGTH = 0;
      }
      break;
    default :
      if (RX_MESSAGE_FLAG==1) {
        RX_MESSAGE[RX_MESSAGE_LENGTH++] = temp;
      } else {}
      break;
  }
}

/*
void Timer_B (void) __interrupt[TIMERB0_VECTOR] {
  while(1){}
}
*/
```

```
//mainBaseStation.h
//Written by Dan Lovell 9/8/04 for GestureBox Project, MIT Media Lab

#define SERIAL
//#define USB
#define BASESTATION
void processMessage(unsigned char * message, int messageLength);
```

```
//mainNode.c
//Written by Dan Lovell 9/8/04 for GestureBox Project, MIT Media Lab

#include <__cross_studio_io.h>
#include <msp430x14x.h>
#include <in430.h>       //has a _NOP()
#include "nRF2401.h"
#include "peripherals.h"
#include "config.h"
#include "mainNode.h"
#include "global.h"

#define NODE
//For nodes, ID should be one of the following 0x11,0x22,0x33,0x44,0x55
//For basestation, ID should be 0x00
const unsigned char ID = 0x11;
/*
      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?
 */
const unsigned char TX_CONFIG_BYTES[15] = {0x60,0x60,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xA1,0x6F,0x04};
const int TX_CONFIG_BYTES_LENGTH = 15;
const unsigned char TX_SHORT_CONFIG_BYTES[1] = {0x04};
const int TX_SHORT_CONFIG_BYTES_LENGTH = 1;
const unsigned char RX_CONFIG_BYTES[15] = {0x60,0x60,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xBB,0xA1,0x6F,0x05};
const int RX_CONFIG_BYTES_LENGTH = 15;
const unsigned char RX_SHORT_CONFIG_BYTES[1] = {0x05};
const int RX_SHORT_CONFIG_BYTES_LENGTH = 1;
const unsigned char ADDR_BYTES[5] = {0xBB,0xBB,0xBB,0xBB,0xAA};
const int ADDR_BYTES_LENGTH = 5;

//used to store the message received from the basestation
unsigned char RX_MESSAGE[16];
int RX_MESSAGE_LENGTH = 0;

extern const unsigned char HEADER[];
extern const int HEADER_LENGTH;

const unsigned char ACCEL1 = 1; //the input number on ADG608
const unsigned char ACCEL2 = 0; //the input number on ADG608
const unsigned char ACCEL3 = 3; //the input number on ADG608
const unsigned char PRESSURE1 = 4; //the input number on ADG608
const unsigned char PRESSURE2 = 5; //the input number on ADG608
const unsigned char GYRO1 = 7;  //the input number on ADG608
const unsigned char GYRO2 = 2;  //the input number on ADG608
const unsigned char GYRO3 = 6;  //the input number on ADG608

//node that pressure1 is also accel4, its the one we want
const unsigned char muxArray[8] = {1/*ACCEL1*/,0/*ACCEL2*/,4/*ACCEL4-PRESSURE1*/,7/*GYRO1*/,2/*GYRO2*/,6/*GYRO3*/,3/*ACCEL3*/,5/*PRESSURE2*/};

const unsigned char muxArrayLength = 8;

//used to store the sensor data from this node for transmission
union  {
  int intVal[NUM_SENSOR_VALS];
  unsigned char charVal[2*NUM_SENSOR_VALS];
} intChar;

int eventCounter=0;

//set node here

void main(void) {
  int i,j;
  unsigned char potSetting = 0x00;
  unsigned char muxIndex = 0;

  for(i=0;i<20;i++) {
    intChar.charVal[i] = ID;
  }

  //set up MSP430
  CONFIG();
  CONFIG_SPI0();
  CONFIG_SPI1();

  CONFIG_IO();
  CONFIG_ADC_AVCC();
  nRF_SetStdByMode();

  //change the interval to be timed
  switch(ID) {
    case 0x11 :
      CONFIG_TIMER_A_INT(0x0600,0);
      break;
    case 0x22 :
      CONFIG_TIMER_A_INT(0x0800,0);
      break;
    case 0x33 :
      CONFIG_TIMER_A_INT(0x0C00,0);
      break;
    case 0x44 :
      CONFIG_TIMER_A_INT(0x1000,0);
      break;
    case 0x55 :
      CONFIG_TIMER_A_INT(0x1400,0);
      break;
    default :
      //hopefully you're never here so lets freeze the MCU
      iShouldntBeHere();
  }
  _EINT();


  //give nRF time to go to PWR_UP mode before going to stand by
  //nRF_PWRUP has already been set high in CONFIG_IO()
  for(i=0;i<5000;i++) {}

  nRF_CONFIG(RX_CONFIG_BYTES,RX_CONFIG_BYTES_LENGTH);
  nRF_SetRXMode();

  //setPot(1,0x0C);   //0x0C gives gain of .5 when Rset is 1.2K
  setPot(1,0xFF);
  setPot(0,0x69);    //0x69, with gain of .5 gives 1.5V quisecent output
                     //when Vin is centered on 2.5V and Vlogic = 3.128V

  setMux(0);
#ifdef PWR_LED
  P2OUT |= 0x01;
#endif
  while(1) {
/*
    for(i=0;i<30000;i++) {
      for(j=0;j<100;j++) {
      }
    } // ~6.25ms to execute, ~5 clocks for one loop
*/
    LPM0;
  }
}

void sampleData(void) {
  char i;
  int j;

  for(i=0;i<6;i++) {
    setMux(muxArray[i]);
    intChar.intVal[i] = adcBlockingConversion();
  }


  /*
  intChar.charVal[0] = ID;
  intChar.charVal[1] = ID;
  for(i=1;i<6;i++) {
    intChar.intVal[i] = 0xBBBB;
  }
  */

}

void processMessage(unsigned char * message, int messageLength) {
  /*
  *   Form of message is {HEADER0,DESTINATION_ID,COMMAND,...,HEADER1}
  *   indices              0         1         2        N
  */

  //check header to see if its for me
  if(message[1]==ID) {
    //the message is for me
    switch(message[2]) {
      case(0x10):
        //change the delay before reporting
        CCR0 = 256*message[3]+message[4];
        break;
      default:
        break;
    }
  }
}

void nRF_Data_Waiting(void) __interrupt[PORT1_VECTOR] {
  long i;
```

```
#ifdef TX_LED
  P2OUT |= 0x02;      //toggle LED to indicate reception
#endif

  TACTL |= MC0;       // Start timing the interval
#ifdef TX_LED
  P2OUT ^= 0x01;    //toggle LED
#endif
  if(P1IFG & BIT4){   //clear the interrupt flag
    P1IFG &= ~BIT4;
  }

  nRF_SHOCKBURST_BLOCKING_READ(RX_MESSAGE,&RX_MESSAGE_LENGTH);   //read the received data
  nRF_CONFIG(TX_SHORT_CONFIG_BYTES,TX_SHORT_CONFIG_BYTES_LENGTH);
  nRF_SetTXRXMode();
#ifdef TX_LED
  P2OUT ^= 0x01;    //toggle LED
#endif

  //fill up intChar with the sampled data
  sampleData();
  _NOP();

  if(RX_MESSAGE[0] == HEADER[0]) {
    //process the received message
    processMessage(RX_MESSAGE,RX_MESSAGE_LENGTH);
  }
}

void Timer_A (void) __interrupt[TIMERA0_VECTOR] {
  int i;
  //send packet to basestation
  //turn off/reset timer A
  //return to receive mode
#ifdef TX_LED
  P2OUT &= ~0x02;      //toggle LED to indicate reception
#endif
  //for counter to look for lost packets
  //intChar.intVal[0]++;

  //reset timer A
  TACTL &= ~MC0;
  TACTL |= TACLR;                   // Stop timer A, clear TAR
  TACTL = TASSEL0 + ID1 + ID0;  // ACLK/8
  //just need to start timer with TACTL |= MC0 to count off interval

  //broadcast to all nodes, assume already in TX mode
  nRF_SHOCKBURST_BLOCKING_WRITE(intChar.charVal,12,ADDR_BYTES,ADDR_BYTES_LENGTH);

  //return to receive mode
  //probably need to make sure that this config doesn't occur too close to
  //the previous TX since you can't config while TX is occuring
  for(i=0;i<500;i++) {}
  nRF_CONFIG(RX_SHORT_CONFIG_BYTES,RX_SHORT_CONFIG_BYTES_LENGTH);
  nRF_SetTXRXMode();

}

/*
void Timer_B (void) __interrupt[TIMERB0_VECTOR] {
}
```

```
//mainNode.h
//Written by Dan Lovell 9/8/04 for GestureBox Project, MIT Media Lab
void processMessage(unsigned char * message, int messageLength);
```

```
//nRF2401.c
//This file used for functions and variables related to interaction with the nRF2401

#include <msp430x14x.h>
#include "peripherals.h"

#ifndef nRF2401
#include "nRF2401.h"
#define nRF2401

void nRF_SetTXRXMode(void) {
//Make sure the pins used are set for output
//Clear bits then set bits to make sure you're not in CE=CS=PWR_UP = 1
  P1OUT &= ~nRF_CS;
  P1OUT |= nRF_CE | nRF_PWRUP;
}

void nRF_SetConfigMode(void) {
//Make sure the pins used are set for output
//Clear bits then set bits to make sure you're not in CE=CS=PWR_UP = 1
  P1OUT &= ~nRF_CE;
  P1OUT |= nRF_CS | nRF_PWRUP;
}

void nRF_SetStdByMode(void) {
//Make sure the pins used are set for output
//Clear bits then set bits to make sure you're not in CE=CS=PWR_UP = 1
  P1OUT &= ~(nRF_CE | nRF_CS) ;
  P1OUT |= nRF_PWRUP;
}

void nRF_SetPwrDownMode(void) {
//Make sure the pins used are set for output
//Clear bits then set bits to make sure you're not in CE=CS=PWR_UP = 1
  P1OUT &= ~nRF_PWRUP;
}

void nRF_BLOCKING_WRITE(unsigned char * data, int length) {
  //write via SPI to the nRF
  //does not use SPI interrupt
  //configuration of PWR_UP, CE, CS pins assumed
  char temp;
  int counter = 0;
  while (length - counter > 0 ) {
    while ((IFG1 & UTXIFG0) == 0);        // USART0 TX buffer ready?
    TXBUF0 = data[counter];
    counter++;
  }

  //make sure the last byte has been received
  temp = RXBUF0;                          //clear the URXIFG0 flag
  while ((IFG1 & URXIFG0) == 0){};        // transmission completed?
}

void nRF_BLOCKING_WRITE_REPEATED(unsigned char data, int length) {
  //write via SPI to the nRF
  //does not use SPI interrupt
  //configuration of PWR_UP, CE, CS pins assumed
  int counter = 0;
  while (counter < length ) {
    while ((IFG1 & UTXIFG0) == 0);        // USART0 TX buffer ready?
    TXBUF0 = data;
    counter++;
  }
  //make sure the last byte has been received before you pull CS low
  data = RXBUF0;                          //clear the URXIFG0 flag
  while ((IFG1 & URXIFG0) == 0){};        // transmission completed?
}

void nRF_SHOCKBURST_BLOCKING_WRITE(unsigned char * data, int dataLength,unsigned char * addr,int addrLe

  //NOTICE that the nRF is left in Stand By Mode when done//
  int i;

  //prepare nRF for data write
  nRF_SetTXRXMode();

  //5us delay required between CE high and beginning of data
  for(i=0;i<40;i++) {}

  //write dest address
  nRF_BLOCKING_WRITE(addr,addrLength);
  //write data
  nRF_BLOCKING_WRITE(data,dataLength);

  //if this wait is only 10 then transmission comes back as UUUS (when its supposed
  //to be UUUU or doesn't come across at all
  for(i=0;i<20;i++) {}

  //take CE low to indicate data is ready
  nRF_SetStdByMode();
}

void nRF_SHOCKBURST_BLOCKING_WRITE_REPEATED(unsigned char data, int dataLength,unsigned char * addr,int

  //NOTICE that the nRF is left in Stand By Mode when done//
  int i;

  //prepare nRF for data write
  nRF_SetTXRXMode();

  //5us delay required between CE high and beginning of data
  for(i=0;i<40;i++) {}

  //write dest address
  nRF_BLOCKING_WRITE(addr,addrLength);
  //write data
  nRF_BLOCKING_WRITE_REPEATED(data,dataLength);

  //if this wait is only 10 then transmission comes back as UUUS (when its supposed
  //to be UUUU or doesn't come across at all
  for(i=0;i<20;i++) {}

  //take CE low to indicate data is ready
  nRF_SetStdByMode();
}

void nRF_SHOCKBURST_BLOCKING_READ(unsigned char * data, int * dataLength) {
  //THIS IS ONLY FOR SPI SINCE IT WRITES TO TXBUF
  //clear the URXIFG0 flag so you know when reception complete
  //char temp;
  IFG1 &= ~URXIFG0;
  *dataLength = 0;
  while(P1IN & 0x10) {
    //should be checking URXIFG0 instead of TX since you need to make sure you've
    //received the info you want
    TXBUF0 = 0xBB;                        //write to receive
    while ((IFG1 & URXIFG0) == 0);        // RX Complete?
    //temp = RXBUF0;
    data[(*dataLength)++] = RXBUF0;
  }

  //usart0Read(data,dataLength);

}

void nRF_CONFIG(unsigned char * DATA,int DATA_LENGTH) {
  //why does i need to be a long?
  long i;
/*
      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?
 */

  //during configuration: PWR_UP = 1, CE = 0, CS = 1
  nRF_SetConfigMode();


  //5us delay required between CS high and beginning of data
  for(i=0;i<40;i++) {
  }

  nRF_BLOCKING_WRITE(DATA,DATA_LENGTH);


  //I don't know why this wait is necessary
  for(i=0;i<8;i++) {}

  //return nRF to Stand by when done
  nRF_SetStdByMode();
}

void nRF_RESET(void) {
  //reset the nRF to bring it into a known state
  int i;
  //power down the nRF
  nRF_SetPwrDownMode();
  //wait some amount of time to make sure nRF powers down
  for(i=0;i<50;i++) {}
  //power the nRF back up
  nRF_SetConfigMode();
  //don't release control till the nRF is ready to go
  for(i=0;i<5000;i++) {}
}
```

```
#endif
```

```
//nRF2401.c
//This file used for functions and variables related to interaction with the nRF2401

#define nRF_CS 0x08
#define nRF_CE 0x40
#define nRF_PWRUP 0x80

/*
//DEFAULT TO XTAL_FREQ = 8MHz IF NOT ALREADY DEFINED
#ifndef XTAL_FREQ
#define XTAL_FREQ 8000000;
#endif
*/
void nRF_SetTXRXMode(void);
void nRF_SetConfigMode(void);
void nRF_SetStdByMode(void);
void nRF_SetPwrDownMode(void);
void nRF_BLOCKING_WRITE(unsigned char *, int);
void nRF_BLOCKING_WRITE_REPEATED(unsigned char, int);
void nRF_SHOCKBURST_BLOCKING_WRITE(unsigned char *, int,unsigned char *,int);
void nRF_SHOCKBURST_BLOCKING_WRITE_REPEATED(unsigned char, int,unsigned char *,int);
void nRF_SHOCKBURST_BLOCKING_READ(unsigned char *, int *);
void nRF_CONFIG(unsigned char *,int);
void nRF_RESET(void);
```

```
//config.c
//This file used for all configuration related functions and variables

#include <msp430x14x.h>
#include "nRF2401.h"

#ifndef config

#include "config.h"
#define config

void CONFIG_IO(void) {
  //I/O SETUP

  //make sure nRF pins are set to a valid state when msp430 pins become outputs
  //enable the nRF control pins and put the nRF into standby mode
  nRF_SetStdByMode();
  P1DIR |= nRF_CS | nRF_CE | nRF_PWRUP;

  P1IES &= ~(BIT4 + BIT5);       //make interrupts occur on low to high transitions
  P1IFG &= ~(BIT4 + BIT5);       //clear the interrupt flag before enabling them

#ifdef BASESTATION
  P1IES |= 0x04;       //external signal from F320 for SPI handling interrupts high to low
  P1IFG &= ~0x04;      //clear external interrupt signal
  //don't enable interrupts for P1.2, SPI ISR will poll the P1IFG flag
#endif

  P1IE  |= BIT4+BIT5;            //enable interrupts on P1.4,P1.5 for DR{1,2} from nRF

  P2OUT = 0x80;  //make sure F320 /NSS is high
  P2DIR |= 0x83;  //Two LEDs and F320 /NSS are on port 2


  P5OUT &= ~0x10; //make sure the mux isn't enabled on power up
  P5DIR |= 0xF0;

  P6OUT |= 0x04;
  P6DIR |= 0x04;
}

void CONFIG(void) {
  unsigned int i;
  WDTCTL = WDTPW + WDTHOLD;                   // Stop WDT
  BCSCTL1 |= XTS;                             // ACLK = LFXT1 = HF XTAL

  do {
    IFG1 &= ~OFIFG;                           // Clear OSCFault flag
    for (i = 0xFF; i > 0; i--);               // Time for flag to set
  } while ((IFG1 & OFIFG) != 0);              // OSCFault flag still set?

  BCSCTL2 |= SELM1+SELM0;                     // MCLK = LFXT1 (safe)
}

void CONFIG_SPI0(void) {
  //SPI SETUP
  UCTL0 |= SWRST;

  UTCTL0 = CKPH+SSEL0+STC;                    // ACLK, 3-pin mode
  UCTL0 = CHAR+SYNC+MM;                       // 8-bit SPI Master **SWRST**
  UBR00 = 0x09;                               // CLK0 = ACLK/9
```

```
  UBR10 = 0x00;                               //
  UMCTL0 = 0x00;                              // no modulation
  P3SEL |= 0x0E;                              // P3.1-3 SPI option select
  P3SEL &= ~0x30;                             // Disable UART0
  ME1 = USPIE0;                              // Enable USART0 SPI mode

  UCTL0 &= ~SWRST;
  //enable interrupts outside this function to prevent uninteded nesting of interrupts
  //_EINT();                                    // Enable interrupts
}

void CONFIG_UART0(void) {
  UCTL0 |= SWRST;

  UCTL0 = CHAR;                               // 8-bit character
  UTCTL0 = SSEL0;                             // UCLK = ACLK
  UBR00 = 0x45;                               // 8MHz 115200
  UBR10 = 0x00;                               //
  UMCTL0 = 0x2C;                              // no modulation
  P3SEL |= 0x30;                              // P3.4,5 = USART0 TXD/RXD
  P3SEL &= ~0x0E;                             // Disable SPI0
  ME1 = UTXE0 + URXE0;                       // Enable USART0 TXD/RXD

  UCTL0 &= ~SWRST;

  //IE1 |= URXIE0 + UTXIE0;                     // Enable USART0 TX/RX interrupt
}

void CONFIG_SPI1(void) {
  UCTL1 |= SWRST;

  UTCTL1 = CKPL+SSEL0+STC;            // ACLK, 3-pin mode
  UCTL1 = CHAR+SYNC+MM;                       // 8-bit SPI Master **SWRST**
  UBR01 = 0x09;                               // CLK1 = ACLK/9
  UBR11 = 0x00;                               //
  UMCTL1 = 0x00;                              // no modulation
  P5SEL |= 0x0A;                              // P5.1,5.3 SPI option select
  P3SEL &= ~0xC0;                             // Disable UART1
  ME2 = USPIE1;                              // Enable USART1 SPI mode

  UCTL1 &= ~SWRST;

  //enable interrupts outside this function to prevent uninteded nesting of interrupts
  //_EINT();                                    // Enable interrupts
}

void CONFIG_UART1(void) {
  UCTL1 |= SWRST;

  UCTL1 = CHAR;                               // 8-bit character
  UTCTL1 = SSEL0 ;                             // UCLK = ACLK
  UBR01 = 0x45;                               // 8MHz 115200
  UBR11 = 0x00;                               //
  UMCTL1 = 0x2C;                              // no modulation
  ME2 = UTXE1 + URXE1;                       // Enable USART1 TXD/RXD
  P3SEL |= 0xC0;                              // P3.6,7 = USART1 option select
  P3DIR |= 0x20;                              // P3.6 = output direction
  P5SEL &= ~0x0A;                             // Disable SPI1

  UCTL1 &= ~SWRST;
}
```

```
void CONFIG_UART1_RX(void) {
  //clear the interrupt flag
  IFG2 &= ~URXIFG1;
  //RX interrupt set!!!! make sure to handle it
  IE2 |= URXIE1;                          // Enable USART1 RX interrupt
}

void CONFIG_TIMER_A_INT(int timerVal,char startNow) {
  TACTL = TASSEL0 + TACLR + ID1 + ID0;  // ACLK/8, clear TAR
  CCTL0 = CCIE;                           // CCR0 interrupt enabled
/* Need the following two lines to set up an interrupt
  CCR0 = 20000;                           // Wait 20000 counts (with %8)
  TACTL |= MC0;                           // Start Timer_a in upmode
 */

  CCR0 = timerVal;
  if(startNow) {
    TACTL |= MC0;                           // Start Timer_a in upmode
  }
  //enable interrupts outside this function to prevent uninteded nesting of interrupts
}

void CONFIG_TIMER_B_INT(void) {
  TBCTL = TBSSEL0 + TBCLR + ID1 + ID0;  // ACLK/8, clear TAR
  TBCCTL0 = CCIE;                         // TBCCR0 interrupt enabled
/* Need the following two lines to set up an interrupt
  TBCCR0 = 20000;                         // Wait 20000 counts (with %8)
  TBCTL |= MC0;                           // Start Timer_b in upmode
 */
  //enable interrupts outside this function to prevent uninteded nesting of interrupts
  //_EINT();                                // Enable interrupts
}

void CONFIG_TIMER_B_COUNTER0(void) {
  TBCCTL0 = CM_3 + CCIS1 + SCS + CAP;         //capture on both edges, synchronously
                                              //interrupts are disabled

  TBCTL |= TBCLR;                       //clear TBR, mode and divider settings
  TBCTL |= TBSSEL0 + ID1 + ID0 + MC1; //ACLK/8, start timer_b in cont. mode
}

void CONFIG_ADC_AVCC(void) {
  ADC12CTL0 = ADC12ON+SHT0_2;                    // ADC12ON
  P6SEL |= 0x01;                          // P6.0 ADC option select
  ADC12CTL1 = SHP;                        // Use sampling timer
  ADC12CTL0 |= ENC;                       // Enable conversions
}

#endif
```

```
//config.h
//This file used for all configuration related functions and variables

void CONFIG_IO(void);
void CONFIG(void);
void CONFIG_SPI0(void);
void CONFIG_UART0(void);
void CONFIG_SPI1(void);
void CONFIG_UART1(void);
void CONFIG_UART1_RX(void);
void CONFIG_TIMER_A_INT(int,char);
void CONFIG_TIMER_B_INT(void);
void CONFIG_TIMER_B_COUNTER0(void);
void CONFIG_ADC_AVCC(void);
```

```
//peripherals.c
//This file used for all functions and variables related to the peripherals on
//the MSP430

#ifndef peripherals

#include <msp430x14x.h>
#include "mainBaseStation.h"
#include "peripherals.h"
#define peripherals


#ifdef BASESTATION
extern unsigned char RX_MESSAGE[];
//message length can be used as state
//length = 0 -> not receiving, else receiving
//length = # bytes received already
extern int RX_MESSAGE_LENGTH;
extern const unsigned char SPI_RX_HEADER[];

#endif

extern const unsigned char ACCEL1,ACCEL2,ACCEL3,GYRO1,GYRO2,GYRO3,PRESSURE1,PRESSURE2;

char waiting = 1;

void disableMux(int state) {
  P5OUT &= ~0x10;
}

void setMux(unsigned char sensor) {
//set pins on ADG608 to output the chosen sensor
//this function sets the mux enable line (P5.4) high
  if (sensor==ACCEL1) {
    P5OUT &= 0x0F;
    P5OUT |= 0x30;
  } else if(sensor==ACCEL2) {
    P5OUT &= 0x0F;
    P5OUT |= 0x10;
  } else if(sensor==ACCEL3) {
    P5OUT &= 0x0F;
    P5OUT |= 0xB0;
  } else if(sensor==GYRO1) {
    P5OUT |= 0xF0;
  } else if(sensor==GYRO2) {
    P5OUT &= 0x0F;
    P5OUT |= 0x90;
  } else if(sensor==GYRO3) {
    P5OUT &= 0x0F;
    P5OUT |= 0xD0;
  } else if(sensor==PRESSURE1) {
    P5OUT &= 0x0F;
    P5OUT |= 0x50;
  } else if(sensor==PRESSURE2) {
    P5OUT &= 0x0F;
    P5OUT |= 0x70;
  }
}

void setPot(char addr, unsigned char setting) {
  //requires that USART1 be set properly to SPI
  int i;

  if(addr != 1 && addr != 0) {
    //invalid addr
    return;
  }

  P6OUT &= ~0x04;   //pull CS low
  while ((IFG2 & UTXIFG1) == 0);          // USART1 TX buffer ready?
  TXBUF1 = addr;

  for(i=0;i<400;i++) {
  //delay some time to allow byte to finish writing
  }

  while ((IFG2 & UTXIFG1) == 0);          // USART1 TX buffer ready?

  //clear the URXIFG0 flag so you know when reception complete
  IFG2 &= ~URXIFG1;

  TXBUF1 = setting;

  for(i=0;i<400;i++) {
  //delay some time to allow byte to finish writing
  }


  //use the following line once the shiite is working
  //while ((IFG2 & URXIFG1) == 0);
  for(i=0;i<400;i++) {
  //delay some time to allow byte to finish writing
  }

  P6OUT |= 0x04;    //set CS high
}

void usart0Read(unsigned char * data,int * dataLength) {
  //THIS IS ONLY FOR SPI SINCE IT WRITES TO TXBUF
  //clear the URXIFG0 flag so you know when reception complete
  IFG1 &= ~URXIFG0;
  *dataLength = 0;
  while(P1IN & 0x10) {
    //should be checking URXIFG0 instead of TX since you need to make sure you've
    //received the info you want
    TXBUF0 = 0xBB;                          //write to receive
    while ((IFG1 & URXIFG0) == 0);          // RX Complete?
    data[(*dataLength)++] = RXBUF0;
  }
}

void usart0Write(unsigned char * data,int dataLength) {
  //forget what the wait in the loop is for
  //maybe to seperate the characters so they come across cleanly?
  int waitCounter,i;
  for(i=0;i<dataLength;i++) {
    while((IFG1&UTXIFG0) == 0) {}
    TXBUF0=data[i];
    for(waitCounter=0;waitCounter<300;waitCounter++) {}
  }
}
```

```
void f320Write(unsigned char * data,int dataLength) {
  //forget what the wait in the loop is for
  //maybe to seperate the characters so they come across cleanly?
  int i;
  int j=0;
  unsigned char tempChar;
  while((IFG1&UTXIFG0) == 0) {}
  IFG1 &= ~URXIFG0;
  for(i=0;i<dataLength;i++) {
    P2OUT &= 0x7F;  //take /NSS low to signal byte coming
    TXBUF0=data[i];
    while((IFG1&URXIFG0) == 0) {}
    tempChar = RXBUF0;  //same as "IFG1 &= ~URXIFG0;"
    P2OUT |= 0x80;  //take /NSS high to signal end of byte

    //process the received data
    if(tempChar == SPI_RX_HEADER[0]) {
      RX_MESSAGE_LENGTH = 1;
      RX_MESSAGE[0] = tempChar;
    } else if(RX_MESSAGE_LENGTH != 0) {
      //make sure not to overrun bounds!
      if(tempChar == SPI_RX_HEADER[1]) {
        //you have received the whole message, process it
        //indexing here is funky, make sure you understand it
        RX_MESSAGE[RX_MESSAGE_LENGTH] = tempChar;
        processMessage(RX_MESSAGE,RX_MESSAGE_LENGTH);
      } else if(RX_MESSAGE_LENGTH==16) {
        //about to write the 16th byte but it isn't the final header!
        //message has overrun bounds! so just reset it
        RX_MESSAGE_LENGTH = 0;
      } else {
        RX_MESSAGE[RX_MESSAGE_LENGTH++] = tempChar;
      }
    }
    //don't release until the F320 tells you its read the SPI character
    //WARNING: could hang here if F320 does not respond!!!!
    j=0;
    while(!(P1IFG & 0x04) && j<1000) {
      //While F320 hasn't caused high to low transition on P1.2
      j++;
    }
    if(j==1000) {
      //if you hang here then MSP430 hangs whenever F320 isn't responding
      //this means on bootup too!
      //iShouldntBeHere();
      _NOP();
    }
  }
}

void usart1Read(unsigned char * data,int * dataLength) {
  //THIS IS ONLY FOR SPI SINCE IT WRITES TO TXBUF
//this should be verified operational the first time used
  //clear the URXIFG0 flag so you know when reception complete
  IFG2 &= ~URXIFG1;
  *dataLength = 0;
  while(P1IN & 0x10) {
    //should be checking URXIFG0 instead of TX since you need to make sure you've
    //received the info you want
    TXBUF0 = 0xBB;                        //write to receive
    while ((IFG2 & URXIFG1) == 0);        // RX Complete?
    data[(*dataLength)++] = RXBUF1;
  }
}

void usart1Write(unsigned char * data,int dataLength) {
  //forget what the wait in the loop is for
  //maybe to seperate the characters so they come across cleanly?
//this should be verified operational the first time used

  int waitCounter,i;
  for(i=0;i<dataLength;i++) {
    while((IFG2&UTXIFG1) == 0) {}
    TXBUF1=data[i];
    for(waitCounter=0;waitCounter<300;waitCounter++) {}
  }
}

void delayUSec(int delay) {
  waiting = 1;
  TBCCR0 = delay;
  TBCTL |= MC0;     // Start Timer_b in upmode
  while(waiting) {}
}

int captureTimerB(void) {
//the capture requires a few clock cycles for the timer value to be
//written to the register TBCCR0, so must have a few NOP's
  TBCCTL0 ^= CCIS0;
  _NOP();
  _NOP();
  _NOP();
  _NOP();
  _NOP();
  _NOP();
  return TBCCR0;
}

int adcBlockingConversion(void) {
  ADC12CTL0 |= ADC12SC;               // Start conversion
  while ((ADC12IFG & BIT0)==0);
  return ADC12MEM0;
}
#endif
```

```
//peripherals.h
//This file used for all functions and variables related to the peripherals on
//the MSP430

#ifdef BASESTATION
extern unsigned char RX_MESSAGE[];
//message length can be used as state
//length = 0 -> not receiving, else receiving
//length = # bytes received already
extern int RX_MESSAGE_LENGTH;
extern const unsigned char HEADER[];
#endif
extern const unsigned char ACCEL1,ACCEL2,ACCEL3,GYRO1,GYRO2,GYRO3,PRESSURE1,PRESSURE2;

void disableMux(int);
void setMux(unsigned char);
void setPot(char, unsigned char);
void usart0Read(unsigned char *,int *);
void usart0Write(unsigned char *,int);
void f320Write(unsigned char *,int);
void usart1Read(unsigned char *,int *);
void usart1Write(unsigned char *,int);
void delayUSec(int);
int captureTimerB(void);
int adcBlockingConversion(void);
```

```
//global.h
//This file used for variables that need to effect both mainNode and mainBasestation

#define XTAL_FREQ 8000000;
#define INIT_RF_PACKET_BYTE_LENGTH  12
#define INIT_RF_PACKET_BIT_LENGTH 0x60
#define MAX_ADDR_BYTES_LENGTH 5
#define NUM_SENSOR_VALS 6
#define MAX_NUM_NODES 15

#define PWR_LED
//#define TX_LED

unsigned char RF_PACKET_BYTE_LENGTH = INIT_RF_PACKET_BYTE_LENGTH;
unsigned char RF_PACKET_BIT_LENGTH = INIT_RF_PACKET_BIT_LENGTH;

//used to delimit the start and end of all packets
#define  HEADER0 0x96
#define  HEADER1 0x69

const unsigned char HEADER[4] = {HEADER0,HEADER1,HEADER0,HEADER1};
const int HEADER_LENGTH = 4;
const unsigned char SPI_RX_HEADER[2] = {0x62,0x6D};
const int SPI_RX_HEADER_LENGTH = 2;
```

```
void iShouldntBeHere(void) {
  int i,j;
  while(1) {
    for(i=0;i<30000;i++) {
      for(j=0;j<100;j++) {
      }
    } // ~6.25ms to execute, ~5 clocks for one loop
    P2OUT ^= 0x03;
  }
}

int littleFromBig(int inVal) {
  union {
    int intVal;
    unsigned char charVal[2];
    unsigned char tempChar;
  } intChar;
  intChar.intVal = inVal;
  intChar.tempChar = intChar.charVal[0];
  intChar.charVal[0] = intChar.charVal[1];
  intChar.charVal[1] = intChar.tempChar;
  return intChar.intVal;
}
```

Appendix C

MATLAB Code

```
%this script captures data from the serial port and prints it up
%the data capture occurs from typing the command till you type r-e-t-u-r-n
%[nodeData,out,numNodesArr] = serialReader;
%plotData(nodeData)

[nodeData,out,numNodesArr] = serialReader;
[fixedData,lostPacket] = interpolateLostPackets(nodeData);
plotData(nodeData)
```

```
function nodeData = findNodeData(packetData, headerVal, dataLength)
%given some data, a header to look for, and the length of substring wanted, find
%the substring
for i = 1:length(packetData)-1,
    if(packetData(i) == headerVal && packetData(i+1) == headerVal)
        if(length(packetData) >= i+2+dataLength-1)
            nodeData = packetData((i+2):(i+2+dataLength-1));
            return
        else
            nodeData = -1;
            return
        end
    end
end
nodeData = -1;
return
```

```
function nodeData = packNodeData(data,numSensors)
if(length(data)==1 && data == -1)
    nodeData = zeros(1,numSensors);
    return
else
    for i=1:numSensors,
        nodeData(i) = data(2*i-1)+256*data(2*i);
    end
    return
end
return
```

```
function plotData(data)

numNodes = size(data,1);
numSensors = size(data,2);
numDataPoints = size(data,3);

[data,lostPacket,hugeData] = interpolateLostPackets(data);

for i = 1:numNodes
    figure(i);
    clf;
    plotNodeData(reshape(data(i,:,:),numSensors,numDataPoints));
    badIndices = find(lostPacket(i,:));
    plot(badIndices,lostPacket(i,badIndices),'bx')
    moreBadIndices = find(hugeData(i,:));
    plot(moreBadIndices,hugeData(i,moreBadIndices),'ro');
end

return
```

```
function plotNodeData(data)

colors = ['r','g','b','c','m','y','k'];
if(size(data,1)>7)
    display('Only plotting first seven rows of data')
end

hold
for i = 1:min(size(data,1),7),
    plot(data(i,:),colors(i))
end

return
```

```
function [out,lostPacket,hugeData] = interpolateLostPackets(in)
%function out = interpolateLostPackets(in)
%format of in is (numNodes,numSensors,numDatums)
[numNodes,numSensors,numDatums] = size(in);

out = in;

lostPacket = zeros(size(in));
hugeData = zeros(size(in));
%first pass to determine where lost packets are
for i=1:numNodes
    for j=1:numDatums
        if isequal(in(i,:,j) == 0,ones(1,numSensors))
            lostPacket(i,j) = 1;
        end
        if ~isequal(in(i,:,j) > 2^12,zeros(1,numSensors))
            lostPacket(i,j) = 1;
            hugeData(i,j) = 1;
        end
    end
end

%second pass to determine where first valid packet is
for i=1:numNodes
    for j=1:numDatums
        if lostPacket(i,j) == 0
            validBounds(i,1) = j;
            break;
        end
    end
end

%third pass to determine where last valid packet is
for i=1:numNodes
    skip(i) = 1;
    for j=1:numDatums
        if lostPacket(i,numDatums + 1 - j) == 0
            validBounds(i,2) = numDatums + 1 - j;
            skip(i) = 0;
            break
        end
    end
end

%currently doesn't work when one of the nodes are off
%how to fix, figure out what valid bounds are

gaps=zeros(1,numNodes);
%fourth pass to determine gaps to interpolate over
for i=1:numNodes
    if skip(i) == 1
        %if there is no valid data for this node then don't process it
        continue
    end
    j = validBounds(i,1);
    while j<=validBounds(i,2)
        if lostPacket(i,j) == 1
            gaps(i) = gaps(i) + 1;
            gapBounds(i,2*gaps(i)-1) = j;
            while lostPacket(i,j) == 1
                j = j+1;
            end
            gapBounds(i,2*gaps(i)) = j-1;
        end
        j=j+1;
    end
end


%fifth pass to fill in gaps
for i=1:numNodes
    if skip(i) == 1
        %if there is no valid data for this node then don't process it
        continue
    end
    for j=1:gaps(i)
        tempStart = gapBounds(i,2*j-1); %index of first missing datum
        tempEnd = gapBounds(i,2*j);     %index of last missing datum
        startBound = reshape(in(i,:,tempStart-1),1,numSensors);  %value of last valid datum before gap
        endBound = reshape(in(i,:,tempEnd+1),1,numSensors);      %value of first valid datum after gap
        boundsDiff = endBound - startBound;
        numSteps = tempEnd - tempStart + 1;     %number of steps to interpolate over
        boundsDelta = boundsDiff/(numSteps+1);  %size of each step to be added
        for k = 1:numSensors,
            for m = 1:numSteps
               out(i,k,tempStart+m-1) = startBound(k) + boundsDelta(k)*m;
            end
        end
    end
end

return
```

```
function [matchIndex,value] = bestMatch(data,gestureNode,gestureSensor,dataNode,dataSensor)
%function index = bestMatch(data)

indices = regionsOfInterest(findWindowVariance(data(gestureNode,gestureSensor,:),15));
if isequal(size(indices),[0,0])
    matchIndex = -inf;
    value = -inf;
    return
end
%testing only first sensor on first node right now
gesture = data(1,1,indices(1,1,1):indices(1,1,2));
[indexes,values] = bestMatchIndex(gesture,data(dataNode,dataSensor,:));
indexes = indexes - size(data,3) - indices(1,1,1) + 1;

%matchIndex = indexes(end);
%value = values(end);

i = 0;
maxIndex = length(values);
while i<maxIndex
    if values(end-i) > 100
        if abs(indexes(end-i)) > 100
            i=i+1;
        else
            matchIndex = indexes(end-i);
            value = values(end-i);
            return
        end
    else
        matchIndex = -inf;
        value = -inf;
        return
    end
end
```

```
function [index,value] = bestMatchIndex(gesture,data)
%function index = bestMatchIndex(gesture,data)

xcovDat = xcovFunc(gesture,data);
[value,index] = sort(xcovDat);
```

```
function winVar = findWindowVariance(data,windowSize)
%function findWindowVariance(data,windowSize)

if mod(windowSize,2) == 0
    windowSize = windowSize + 1;
end
windowShift = (windowSize-1)/2;

for i=1:size(data,1)
    for j=1:size(data,2)
        for k=1:(size(data,3) - windowSize + 1)
            winVar(i,j,k) = var(data(i,j,k:k+windowSize-1));
        end
    end
end
```

```
function plotWinVar(data,nodes,sensors)
%function plotWinVar(data,nodes,sensors)

windowSize = 15;
if(mod(windowSize,2)==0)
    windowSize = windowSize+1;
end
windowShift = (windowSize-1)/2;

winVar = findWindowVariance(data,windowSize);

numNodes = length(nodes);
numSensors = length(sensors);
numDatums = size(data,3);
%winVar data needs to be shifted by (N+1)/2 when plotted
for i=1:numNodes
    for j=1:numSensors
        %figure( (i-1)*numNodes + j);
        clf;hold;
        tempData = reshape(data(nodes(i),sensors(j),:),1,numDatums);
        plot(tempData,'r');
        tempVarData = reshape(winVar(nodes(i),sensors(j),:),1,size(winVar,3));
        plot((1+windowShift):(numDatums-windowShift),tempVarData)
        title(strcat('Node: ',num2str(i),' Sensor: ',num2str(j)));
    end
end
```

```
function indices = regionsOfInterest(data)
%function indices = regionsOfInterest(data)

threshold = 35*1000;

numNodes = size(data,1);
numSensors = size(data,2);
numDatums = size(data,3);

threshData = data > threshold;

indices = [];
for i=1:numNodes
    for j=1:numSensors
        k=1;
        currNumIndices = 0;
        while k<=numDatums
            if threshData(i,j,k) == 1
                indices(i,j,currNumIndices+1) = k;
                while k<=numDatums && threshData(i,j,min(k,numDatums)) == 1
                %while threshData(i,j,k) == 1
                    k = k + 1;
                end
                indices(i,j,currNumIndices+2) = k-1;
                currNumIndices = currNumIndices + 2;
            end
            k = k+1;
        end
    end
end
```

```
function saveFigs(data)

numNodes = size(data,1);
numSensors = size(data,2);
numDataPoints = size(data,3);

[data,lostPacket,hugeData] = interpolateLostPackets(data);

figure(1);
for i = 1:numNodes
    %badIndices are same for all sensors of a given node
    badIndices = find(lostPacket(i,:));
    moreBadIndices = find(hugeData(i,:));
    for j = 1:numSensors
        clf;hold;
        currData = reshape(data(i,j,:),1,numDataPoints);
        plot(currData);
        %plot(badIndices,lostPacket(i,badIndices),'bx');
        %plot(moreBadIndices,hugeData(i,moreBadIndices),'ro');
        plot(badIndices,currData(badIndices),'bo');
        plot(moreBadIndices,currData(moreBadIndices),'rx');
        print('-f1','-deps',strcat('node',num2str(i),'sensor',num2str(j)));
    end
end

return
```

```
function output = xcovFunc(gesture,data)
%function output = xcovFunc(gesture,data)

gesture = reshape(gesture,1,size(gesture,3));
data = reshape(data,1,size(data,3));

output = xcov(data,gesture);
```

Bibliography

[1] Bao, L. and Intille, S. S. Activity recognition from user-annotated acceleration data. In *Proceedings of Pervasive 2004: the Second International Conference on Pervasive Computing*, April 2004.

[2] Bao,Ling. *Physical Activity Recognition from Acceleration Data under Semi-Naturalistic Conditions.* M.Eng., EECS Department, MIT, 2003.

[3] Benbasat, A.Y. *An Inertial Measurement Unit for User Interfaces.* M.S., MIT Media Lab, September 2000.

[4] Benbasat, A.Y., Morris, S.J., and Paradiso, J.A. A wireless modular sensor architecture and its application in on-shoe gait analysis. In *Sensors, 2003. Proceedings of IEEE*, October 2003.

[5] Graeme S. Chambers, Svetha Venkatesh, Geoff A. W. West, and Hung Hai Bui. Segmentation of intentional human gestures for sports video annotation. In *10th International Multimedia Modelling Conference*, pages 124–129, 2004.

[6] Frédéric Chenevière and Samia Boukir. Deformable model based data compression for gesture recognition. In *International Conference on Pattern Recognition*, volume 4, pages 541–544, 2004.

[7] Cross, P. Zeroing in on zigbee (part 1). *Circuit Cellar*, (175):16–23, February 2005.

[8] Fontaine, D., David, D., and Caritu, Y. Sourceless human body motion capture. In *Smart Objects Conference Proceedings*, Grenoble, France, May 2003.

Available on the Internet: http://www.grenoble-soc.com/proceedings03/Pdf/30-fontaine.pdf.

[9] Haartsen, J. The bluetooth radio system. *IEEE Personal Communications*, 7(1):28–36, 2000.

[10] Ken Hinckley. Synchronous gestures for multiple persons and computers. In *UIST '03: Proceedings of the 16th annual ACM symposium on User interface software and technology*, pages 149–158. ACM Press, 2003.

[11] Marko Jug, Janez Pers, Branko Dezman, and Stanislav Kovacic. Trajectory based assessment of coordinated human activity. In *Computer Vision Systems, Third International Conference, ICVS 2003, Graz, Austria, April 1-3, 2003, Proceedings*, pages 534–543, 2003.

[12] Holger Junker, Paul Lukowicz, and Gerhard Tröster. Continuous recognition of arm activities with body-worn inertial sensors. In *8th International Symposium on Wearable Computers (ISWC 2004), 31 October - 3 November 2004, Arlington, VA, USA*, pages 188–189, 2004.

[13] R. Kahn, M. Swain, P. Prokopowicz, and R. Firby. Technical report tr-96-04: Real-time gesture recognition with the Perseus system. Technical report, University of Chicago, 1996.

[14] Laibowitz, M. and Paradiso, J. Wireless wearable transceivers. *Circuit Cellar*, (163):28–29, February 2004.

[15] Jonathan Lester, Blake Hannaford, and Gaetano Borriello. "Are you with me?" - using accelerometers to determine if two devices are carried by the same person. In *Proceedings of the 2003 Conference on Pervasive Computing*, pages 33 50, 2004.

[16] R. Lockton and A. W. Fitzgibbon. Real-time gesture recognition using deterministic boosting. In *Proceedings of the British Machine Vision Conference*, 2002.

[17] Merrill, David. *FlexiGesture: An sensor-rich real-time adaptive gesture and affordance learning platform for electronic music control.* M.S., MIT Media Lab, June 2004.

[18] Morris, S.J. *A Shoe-Integrated Sensor System for Wireless Gait Analysis and Real-Time Therapeutic Feedback.* Sc.D., Harvard-MIT Division of Health Sciences and Technology, June 2004.

[19] J.A. Paradiso. Design and implementation of expressive footwear. *IBM Systems Journal*, 39:511–529, 2000.

[20] John Kangchun Perng, Brian Fisher, Seth Hollar, and Kristofer S. J. Pister. Acceleration sensing glove (ASG). In *Fifth International Symposium on Wearable Computers (ISWC)*, pages 178–180, 1999.

[21] Qian, G., Guo, F., Ingalls, T., Olson, L., James, J., and Rikakis, T. A gesture-driven multimodal interactive dance system. In *Proceedings of the International Conference on Multimedia and Expo*, June 2004. Available on Internet: http://ame2.asu.edu/faculty/qian/Publications/icme04-ame.pdf.

[22] Cliff Randell and Henk Muller. Context awareness by analysing accelerometer data. In Blair MacIntyre and Bob Iannucci, editors, *The Fourth International Symposium on Wearable Computers*, pages 175–176. IEEE Computer Society, 2000.

[23] J. Rekimoto. Gesturewrist and gesturepad: Unobtrusive wearable interaction devices. Fifth International Symposium on Wearable Computers, 2001.

[24] Tapia, E.M., Marmasse, N., Intille, S.S., and Larson, K. MITes: Wireless portable sensors for studying behavior. In *Proceedings of Extended Abstracts Ubicomp 2004*, 2004.